Qubo Hu

# Hierarchical Memory Size Estimation for Loop Transformation and Data Memory Platform Optimization

PhD thesis
Norwegian University of Science and Technology

# Abstract

In today's embedded systems, the memory hierarchy is rapidly becoming a major bottleneck in terms of power, performance and area, due to the very large amount of (memory related) data need to be transferred and stored (temporarily). This is especially the case for portable multi-media applications systems. These applications are characterized by deep loop nests and multi-dimensional arrays at the high level. Due to the dramatically increasing size and complexity of system-on-a-chip (SoC) designs and stringent time-to-market requirement, the methodology and tools for chip design must be raised to the system level. Early analysis tools are particularly critical in enabling SoC designers to take full advantage of the many architectural options available. For memory optimization, the early high level techniques aim either to design an optimal memory platform for a given application or to optimize the application code in order to take advantage of the memory platform features, or even both. Loop transformation is such an important high level optimization technique. It modifies the execution order of loops and statements without changing the application functionality. Existing loop transformation algorithms are all performed based either on reduction of data access lifetime and on improvement in data locality and regularity to steer selection of loop transformations. These are, however, very abstract cost functions which do not represent the exact memory size requirement of the arrays and how the data will be mapped onto the memory platform later on. Existing algorithms all result in one final loop transformation solution. As different loop transformations may result in optimal utilization for different memory platform instances, ad-hoc decisions at this stage without estimating their impact on the actual hierarchy utilization can lead to a final sub-optimal solution. An evaluation of later design stages' effort is hence required. On the other hand, there usually exist a huge number of loop transformation possibilities, the estimation is required to be performed repeatedly and its computation time of the estimation technique also becomes critical to make it useful during the loop transformation search space exploration.

This dissertation proposes a memory footprint estimation methodology. An intra-array memory footprint estimation is performed first followed by an inter-array estimation. In order to achieve a fast estimate to make it useful repeatedly during the early high level search space exploration, several techniques have been introduced. A fast intra-array memory footprint estimation is performed at the iteration domain based on the maximal lifetime of data accesses, which is defined by the maximal dependency vector. Two approaches, an ILP formulation and vertexes approach, have been introduced for achieving a fast maximal dependency vector calculation. The fast inter-array estimation has been achieved based on several Hanoi tower based approaches.

A hierarchical memory size estimation methodology has also been proposed in this dissertation. It estimates the influence of any given sequence of loop transformation instances on the mapping of application data onto a hierarchical memory platform. As the exact memory platform instantiation is often not yet defined at this high level design stage, a platform independent estimation is introduced with a Pareto curve output for each loop transformation instance. It can steer the designer or an automatic steering tool to select all the interesting loop transformation instances that might later lead to low power data mapping for any of the many possible memory hierarchy instances. This is useful when the memory platform is not defined yet, or for a given memory hierarchy instance. It also allows to find the most appropriate low power memory hierarchy instance by performing an early power estimation of different memory hierarchy instances. Initially the source code is used as input for estimation, resulting in an initial approach. However, performing the estimation repeatedly from the source code is too slow for the large loop transformation search space exploration. An incremental approach, based on local updating of the previous result, is thus introduced to handle sequences of different loop transformations. Several advanced techniques have also been used on these two approaches in order to perform a fast estimation, such as bounding box geometrical model based data reuse analysis, platform independent memory hierarchy layer assignment estimation, fast intra- and inter-array memory footprint estimation.

The feasibility and usefulness of the methodologies are substantiated using several representative real-life application demonstrators. It shows for instance that the fast memory footprint estimation can be two order of magnitude faster than compared techniques while still achieving fairly accurate estimation result. For hierarchical memory size estimation methodology, the initial approach is two order of magnitude faster than the compared technique and the incremental approach is another two order of magnitude faster than the initial approach, which can just take a few milliseconds. The fast computation time of the incremental approach make it feasible to be used repeatedly during the loop transformation exploration over a very large number of possibilities. Furthermore, prototype CAD tools has been developed that includes mast parts of the methodologies.

# Preface

This dissertation is submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree of *doktor ingeniør* (corresponds to Ph.D).

The Ph.D study consisted of compulsory courses corresponding to one year full-time studies, one year of teaching assistantship and other duties, as well as the research activity, and conducted in the period from October 2002 to April 2007. The work herein was performed at and founded by the Department of Electronics and Telecommunications, NTNU, under the supervision of Associate Professor Per Gunnar Kjeldsberg. I have also performed many time visits, in total about one year, to IMEC for research cooperation under the supervision of Professor Francky Catthoor.

## Acknowledgments

When I am writing down these words at the quiet night, I cannot stop myself passing back through the days and nights of the last four years. This is a journey full of various experience and feeling, with no lack of joys and frustrations. I am so still when I get here and now look back at this journey. Fortunately, I am not lonely and I am accompanied with my advisors and colleges for guidance, idea inspiration, discussions, support and also having fun together. I would like to express my grateful thanks to all who have been helped me in this part of my life.

I would first like to thank Associate Professor Per Gunnar Kjeldsberg, who has been my advisor, for bringing me to this exciting research direction and introducing me for the cooperation with researcher at IMEC, Leuven, Belgium. This work would not have been so without his excellent encourage and guidance, valuable suggestions, and generous help and supports. I am so glad for his accompany and bringing me up.

I am extremely grateful to Professor Francky Catthoor, who has been my co-advisor at IMEC, for his excellent guidance, invaluable suggestions and idea inspiration, fruitful discussion and feedback. Francky Catthoor introduced me to the topic that I have focused on, and has been continuously available for my requests. My deepest gratitude also goes to Arnout Vandecappelle, Martin Palkovic, Erik Brockmeyer and Sven Verdoolaege for their significant supports, discussions and new inputs through my work.

I would also like to thank Dr. Benny Thörnberg from Mid-Sweden University, Sundsvall, Sweden, for a lot of valuable discussions and cooperation with him.

I would also like to thank my colleagues and friends at the Department of Electronics and Telecommunications, NTNU and especially those of the Circuit and System group. My special thanks go out to our group leader, Professor Einar J. Aas for his valuable discussion, friendly help and kindly barbecue organization. Likewise, I would like to thank the people and friends working at IMEC for their helps and making my life enjoyable and colorful during my many times visits over there. In addition to those already mentioned by name, I am in particular indebted to the following persons (in alphabetic order):

Bart Durinck
Eddy De Greef
Øystein Gjermundnes
Jing Guo
Tajeshwar Singh
Joar Sæther
Tanja Van Achteren
Sven Verdoolaege
Kaidong Xu
Peng Yang

Last but not least, I want to thank my wife Jing for all her support. Thanks for my eleven weeks old son for his inspiring smiles and big cooperation during the finalization stage of this work. My thanks also go to my uncle Zhou for his encourage to me for taking my PhD study. Thank my parents for their support throughout these years.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| ALAP | As Late As Possible |
| ASIC | Application Specific Integrated Circuit |
| CPU | Central Processing Unit |
| DAB | Digital Audio Broadcast |
| DMA | Direct memory access |
| DRAM | Dynamic Random Access Memory |
| DSA | Dynamic Single Assignment |
| DTSE | Data Transfer and Storage Exploration |
| DV | Dependency Vector |
| EDV | Extreme Dependency Vector |
| GM | Geometrical Model |
| HMSE | Hierarchical Memory Size Estimation |
| ILP | Integer Linear Programming |
| LBL | Linearly Bounded Lattices |
| LT | Loop Transformation |
| MDV | Maximal Dependency Vector |
| MFE | Memory Footprint Estimation |
| MHLA | Memory Hierarchy Layer Assignment |
| MPEG | Moving Picture Experts Group |
| QSDPCM | Quadtree Structured Difference Pulse Code Modulation |
| RAM | Random Access Memory |
| ROM | Read Only Memory |
| RTL | Register Transfer Level |
| SoC | System on Chip |
| SPM | Scratch-Pad Memory |
| SRAM | Static Random Access Memory |
| TLB | Translation Lookaside Buffer |

# Chapter 1

# Introduction

## 1.1 Background and Motivation

Recent multi-media systems such as video compression/decompression, (medical) image processing, advanced audio and speech coding, 3D gaming, typically require very large amounts of data to be transferred and stored (temporarily). For this class of data dominant applications, this memory related data transfer and storage largely determine the total system cost and performance parameters. This is especially a problem for portable embedded systems because the needed memories and bus transfers consume a lot of power. [160] has shown that between 50% to 80% of the power in embedded multi-media systems is consumed by data transfer and storage, as opposed to the computations which consume much less. This is the case for both parallel and single-processor systems. Figure 1.1 shows an example of digital audio broadcast (DAB) ASIC chip in which memory consumes over 66% of the total power [26]. Higher power consumption also means more costly packaging and cooling equipment, and lower reliability. Figure 1.2 shows the energy dissipation for external DRAM access compared to on-chip SRAM and also compared to selected execution units realized in hardware. If off-chip memories can be avoided or the access to them can be reduced by having most accesses on-chip in the hierarchical memory architecture, the average power consumption per overall access is dramatically reduced.

For these applications, the large amounts of data to be stored obviously require a large overall memory size (in the order of hundreds of Mbits to Gbits). Given the need for caching at least part of this huge data, this means that the area cost (both on-chip and off-chip, on the board) is usually for a large part domi-

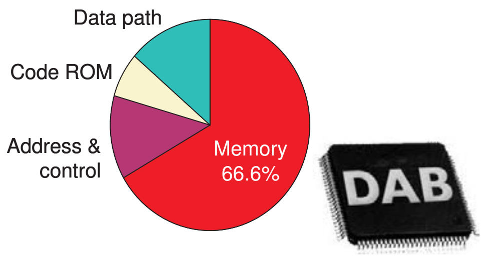


**Figure 1.1:** Power dissipation for DAB ASIC chip

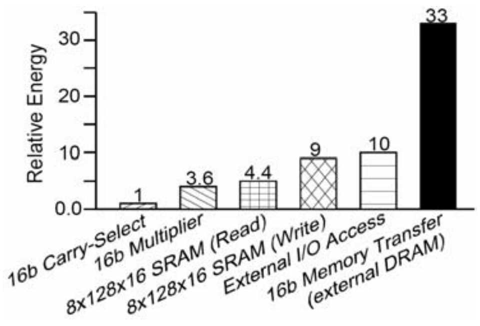


**Figure 1.2:** Energy dissipation for hardware realization of selected operations

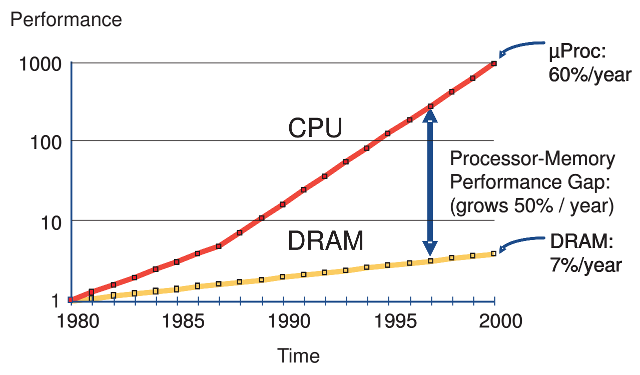


**Figure 1.3:** Performance gap between microprocessors and external DRAM by Moore's law

nated by the memories. For future systems-on-chips the memory is predicted to constitute 94% of the total chip size (in year 2016) [1]. Large memory size also means more power dissipation per access and longer access time. Consequently, the memory related data transfer and storage is also a determining factor for overall system performance. The exponential growth in processor execution speed according to Moore's law, as shown in as Figure 1.3, coupled with a much lower growth in the access time to main memories, have resulted in an ever growing processor-memory performance gap [110, 149]. This bottleneck can be alleviated by hardware controlled caching schemes, or by inserting faster smaller scratch-pad memories between the processor and the main memory to reduce/tolerate the memory latency.

It is commonly agreed that low power design requires optimizations at all levels of the design hierarchy, i.e., technology, device, circuit, logic, architecture, algorithm, and system level [20, 21, 119]. Larger freedom usually exists at the higher abstract levels so that larger gains can be achieved. Due to the dramatically increasing size and complexity of system-on-a-chip (SoC) designs, system-level design becomes crucial for rapidly performing design space exploration without resorting to detailed implementations. System level design can, with a global view, make proper high-level design decisions such as algorithm selection, hardware-software partition, and trade-off between various optimization techniques. It can also shorten the design time, thereby reducing the ever important time-to-market. Various memory related system level design and optimization techniques have recently been developed. A review of important work in this field is given in Chapter 2.

## 1.2 Problem Statement and Objective

System designers traditionally begin the design by evaluating different architectural configurations based on their intuition and experience. The performance and power consumption improvement of the memory subsystem relies mainly on technological advances. A randomly chosen memory architecture is often far from optimal and can usually not be evaluated and optimized until the very late design stage. At the same time, the memory subsystem offers tremendous opportunities for design optimization. A well-matched memory architecture, which highly depends on the characteristics of the target application, can improve the system dramatically.

In multi-media embedded systems, the application is typically described using languages such as C and C++ at the system level. The code is then dominated by large multi-dimensional arrays and loop nests. During system level design, a number of optimization techniques are used that substantially influence data transfer and storage size related issues. One very important example is the global loop transformations step. It modifies the execution order of loops and statements without changing the application functionality. Loop transformations can change the execution order so that the data write (production) and the data read (consumption) are moved closer together in time. The result is that the global lifetime of array elements are reduced and the data locality and regularity of data accesses are improved. Hence, more data can be saved in smaller on-chip memories, from where it can be accessed repeatedly. This is vital to the overall system power consumption, performance and area. Traditionally reductions in data element's lifetime or improvement in data locality and regularity are used to steer selection of loop transformations. These are, however, very abstract cost functions that do not represent how the data will be mapped onto the memory platform. On the other hand, there usually exist a huge number of (combined) loop transformation possibilities for real applications with multiple loop nests (up to dozens). [38] has proven that even performing loop fusion is an NP-complete problem. The task is even more complex when various other loop transformation techniques are considered at the same time, e.g., loop interchange, loop reverse, loop skewing, loop shifting, etc. Different loop transformations may result in optimal utilization for different memory platform instances as will be demonstrated on experiments in this dissertation. Ad-hoc loop transformation decisions without estimating their impact on the actual hierarchy utilization usually lead to final sub-optimal solutions.

Thus, it is critical to perform an estimation of data mapping onto memory platform during the early loop transformation exploration in order to find all interesting (including intermediate) loop transformation instances. The state-of-the-art loop transformation algorithms all result in a single final loop transformation solution. Their solution may be optimal for certain memory hierarchy instances, but typically not for all. As mentioned, the data memory platform is typically not defined at this early design stage. The estimation must hence be platform independent. A memory footprint estimation is also required to give an early feedback on the actual memory size requirement of all arrays as loop transformations change array's lifetime and hence size requirement.

At the same time it is not enough simply to estimate the actual size of a given array and of the given application, since the minimal size may still be too large to fit on-chip. In addition, if sufficient locality between read accesses is present, a local copy of part of the array to on-chip memory may already remove most of the off-chip accesses [157], making the actual size of that array less relevant. To select all the interesting loop transformation alternatives, it is therefore critical to identify the frequently accessed data and estimate their mapping on the hierarchical memory platform. Later, when the details of the memory platform is decided, the set of transformation solutions can help the designer or a steering tool to find the optimal version of code while trading off memory size and power (i.e., off-chip accesses). Such an estimation tool can be of great use to a system designer or an automatic steering tool by providing accurate and fast feedback for a given sequence of transformation instances during the huge loop transformation search space exploration. It is also useful to provide the designers with an early memory architecture exploration while taking into account the data access characteristics of the target application during the system level code transformation exploration. This allows the designer to transform the target application for optimal utilization either of a given memory architecture or for a customized memory architecture instance. The goal of this dissertation work has been to develop techniques and tools for the early global loop transformation and memory architecture exploration. The focus has mainly been on the early global loop transformation design step. But it can also be used to provide feedback to other early system-level transformations such as global data-flow transformations [31].

## 1.3 Application Domain and Targeted Architecture

The target application domain is multi-media applications, which are data access dominated. The main characteristics of this class of applications are many deep loop nests and multi-dimensional data. Due to the rectangular shape of images and video streams, the loop bounds are often constant, or at least manifest (i.e., only dependent on the other iterators). Typically the data being written or read during the execution of a program are grouped into sets of similar variables, which are called arrays. These arrays are arranged as multi-dimensional structures, in which each individual variable can be addressed by a unique set of indices. Most of the conditions and the array index expressions are manifest. They are functions of the surrounding loop iterators. However, more and more multimedia applications (e.g. MPEG-4) also contain data dependent conditions and index expressions.

This dissertation focuses on evaluating the later stage memory footprint requirement and data mapping onto the memory hierarchy. This is based on the fact that memory power consumption depends primarily on the access frequency and the size of the memory. Power can be saved by accessing heavily used data from smaller memories instead of from large background memories. Such optimizations either have to rely on hardware cache controllers which copy relevant data into the cache based on some local criteria, or they rely on scratch-pad memories (SPMs). SPM is also called software-controlled memory. For the data dominated applications in our target domains, SPM is normally preferred over cache due to the stringent low-power and real-time requirements of embedded systems. Caches incur a significant penalty in area, energy, and to hit latency and real-time guarantees. SPM does not need any extra hardware or the set-associative cache matrix. A detailed recent study [128] compares the tradeoffs of a cache as compared to an SPM. Their results show that an SPM has 34% smaller area and 40% lower power consumption than a cache memory of the same capacity. Further, the runtime with an SPM using a simple static knapsack-based [128] allocation algorithm, was measured to be 18% better as compared to a cache. Given the power, cost, performance and real time advantages of SPM, it is not surprising that SPM is the most common form of SRAM in embedded processors. Many embedded processors can thus contain normal SPMs and/or caches, some even leaving out the cache. [100] lists various types of embedded processor families having SPM (from ARM, Motorola, Analog Devices, Atmel, Philips and Hitachi). The recent IBM's CELL processor has, for example, 256KB SPM for each cell and no cache. Its architecture consists of eight "Synergistic Processing Elements" (SPE), each a powerful processor in its own right, together with a powerful 64-bit Dual-threaded IBM PowerPC core.

Data of an application has to be mapped efficiently on its memory hierarchy. SPM require source code transformations that exploit on-chip memory layers to which frequently used data will be stored or copied. Specifically, copies of data will be introduced from larger off-chip memories to smaller on-chip memories. As most of the data access patterns in our application domain are known at compile time, a global view can be taken during code optimization and data mapping [107, 27]. In this dissertation, we choose to focus on the SPM based memory platform. This platform consists of one or multiple SPM layers in addition to off-chip main memory.

## 1.4 Main Contributions

The main contributions of this dissertation can be summarized as follows:

- A complete fast memory footprint estimation (MFE) methodology for data intensive applications has been developed. This methodology consists of intra-array and inter-array memory footprint estimation. It uses a geometrical model (GM) and includes features such as
  - A fast intra-array memory footprint estimation on the iteration domains in the GM defined by the maximal lifetime window, which is constrained by the maximal dependency vector (MDV).
  - A ILP formulation for the MDV calculation.
  - A vertexes approach for the MDV calculation on the bounding box GM which is extremely fast making it feasible to perform the intra-array memory footprint estimation repeatedly during the system level design exploration.
  - A fast inter-array memory footprint estimation by investigating the largest group of arrays that are simultaneously alive. Fast grouping algorithms are introduced by solving Hanoi-tower puzzles.
- A complete fast hierarchical memory size estimation (HMSE) methodology for steering of loop transformation exploration at the system level has been developed. It consists of two parts: initial HMSE starting from the source code, and incremental HMSE for exploration of sequences of loop transformations. It includes features such as:
  - Fast bounding box based data reuse analysis.
  - Fast platform independent memory hierarchy layer assignment (MHLA) estimation with Pareto curve generation. Two heuristics are used: a

simple greedy MHLA estimation algorithm and an improved MHLA estimation algorithm.

  - The intra-array and inter-array memory footprint estimation are also included in the HMSE methodology.
  - Fast incremental data reuse analysis based on evaluation of matrix operations which correspond to the loop transformations performed. For sequences of loop transformations this can be extremely fast as most performed loop transformations usually just have local effects from the previous performed one.

- Methodology for early low power memory architecture exploration based on the HMSE Pareto curve (with both memory size and access information) including power estimation for possible memory architecture instances.
- Prototype tools, MFE and HMSE, have been developed based on major techniques mentioned above for both the fast memory footprint estimation methodology and the hierarchical memory size estimation methodology.
- The tools and the methodologies have been evaluated with successful results for several realistic test vehicles.

## 1.5 Outline

This dissertation is organized in the following manner:

In Chapter 2, related work will be reviewed. It includes a review of the memory related system level optimization techniques, particularly the data transfer and storage exploration (DTSE) methodology. It also presents previous work about loop transformations and memory footprint requirement estimation.

Chapter 3 gives an overview of the polyhedron GM and the simplified bounding box GM. The principle of incremental loop transformations is also presented.

The fast intra-array memory footprint estimation based on the MDV calculation and the two approaches on how to perform the MDV calculation are given in Chapter 4.

Chapter 5 presents the three approaches for fast inter-array memory footprint estimation: initial Hanoi-tower approach, multiple layer Hanoi tower approach and improved Hanoi tower approach.

In Chapter 6, the initial HMSE is presented which starts from the source code.

Chapter 7 presents the incremental HMSE.

In Chapter 8, early power estimation for different memory architectures of each loop transformation instance is presented that enables a low power memory architecture exploration among all the interesting versions of code found during loop transformation exploration.

The estimation methodologies have been applied on a number of real-life application vehicles. The results from these experiments are given in Chapter 9.

Finally, in Chapter 10, conclusions are given together with a discussion of future work.

# Chapter 2

# Related work

In the recent advanced embedded system design domain, the memory related data transfer and storage issues has been investigated in order to achieve a cost effective end-product in terms of power consumption, performance and area. The task is either to find an optimal customizable memory architecture for the given application(s) and/or to optimize the application code in order to maximally utilize the (given) memory architecture. Efforts have been spent at all levels of the design hierarchy, i.e., technology, device, circuit, logic, architecture, algorithm, and system level [119, 20]. However, due to the increasing complexity of the system-on-chip design with integrated applications and strict time-to-market requirements, system level design is becoming more increasingly important. It is commonly recognized that larger freedom usually exists at the higher abstract levels so that larger gains can be achieved.

Many researchers have studied memory related issues of system level design. Section 2.1 gives an overview of work related to the memory optimization in general. Section 2.2 reviews the *data transfer and storage exploration methodology* developed at IMEC, Leuven, Belgium. In this section, previous work done by other researchers have also been included and compared to the IMEC techniques. After that, previous work of loop transformations, storage requirement estimation and memory in-place mapping optimization are reviewed, respectively, in Section 2.3, Section 2.4 and Section 2.5.

## 2.1 System Level Memory Synthesis and Optimization

System level memory synthesis and optimization have been studied extensively in the past two decades by a number of research groups. Studies have been done in the domains of code compilation/optimization and high level architectural synthesis. Here we will try to highlight some of the important work. This section will present major groups that have performed research that are in particular relevant for the work described in this dissertation. This review focuses on the work of the SPM-based memory subsystem synthesis and optimization.

At IMEC, Leuven, Belgium, Catthoor et al. has developed the data transfer and storage exploration methodology for exploration of memory organization at the early system level and it contains different optimization techniques. This will be reviewed in detail in Section 2.2. At University of California, Irvine, Dutt et al. have also studied these topics extensively. Panda et al. [108, 106, 105] has highlighted the importance of this requirement and has presented MeMEplore, an exploration framework for optimizing an on-chip data memory organizations. Grun et al. [61] also proposed a memory architecture exploration framework.

At Pennsylvania State University, Kandemir et al. have also studied memory subsystem design and management extensively in [75, 120, 72, 73, 74]. They has mostly focused on the compiler-related software solutions for both the SPM-based or cache-based on-chip memory management. They have investigated in both compiler-time and dynamical memory management for uni-processor and multiprocessor cases. Benini et al. [20, 21, 23, 22] at University of Bologna, Italy, also studied the various system level power optimization techniques including optimization of the memory subsystem. In the Inria Compsys project at Institut National des Sciences Appliquees de Lyon, France, researchers also work on development of compilation techniques adapted to the design of embedded systems [58, 33, 59, 39, 37]. At Maryland University, USA, a number of papers has also been published about SPM-based memory allocation [136, 100, 137]. Macii et al at Politecnico di Torino, Italy, has been studied for efficient memory design for embedded system [94, 109, 124]. Marwedel et al. at Dortmund University, Germany, have also introduced various techniques for low power SPM assignment and optimization [128, 13, 95]. Featrier et al. at University of Pierre et Marie Curie, Paris, France, have focused on storage size optimization and management for parallel programs [51, 87, 88].

## 2.2 Data Transfer and Storage Exploration Methodology

The data transfer and storage exploration (DTSE) methodology is a systematic methodology for exploration of memory organizations at early system levels for embedded multimedia systems design. The motivation is that the initial specification heavily influences the outcome of architectural exploration and mapping tools, e.g., for data-path allocation, memory allocation, address generation. Therefore transforming the specification is one of the most prominent tasks during the early system-level exploration. The goal of DTSE methodology is to determine an optimal execution order for the data transfers together with an optimal memory architecture for storing the data of the given application. The cost functions are power, performance and area oriented. A detailed description of the complete DTSE methodology can be found in [30, 29]. In order to demonstrate the usefulness of the work presented in this dissertation within their methodology, a brief summary of its main steps is given here. Note that this section does not just contain the work done by IMEC researchers. For some steps which are considered within this dissertation, the work done by other researchers has also been reviewed.

The methodology is divided into constrained orthogonal steps with a fixed sequence that removes the need for iterative loops (the traditional phase coupling has been broken). This sequencing where specific constraints are propagated between the steps, facilitates the designers learning and usage while they also make the full methodology better automatable.

Figure 2.1 shows the design flow of the DTSE methodology. The flow is based on the idea of constrained orthogonalization, 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 next steps. The order of the steps is so that the most important decisions and the decisions that do no put too many restrictions on the other steps are taken earlier. It is based on a source-to-source code transformation approach that can be easily adopted on top of most existing design flows. The starting point is an executable system specification with accesses to multi-dimensional array elements. The output is a transformed source code specification which is an optimized input for the software compilation stage in the case of instruction-set processors or is ready for low power hardware realization.

The first steps are platform independent in the sense that they do not depend on

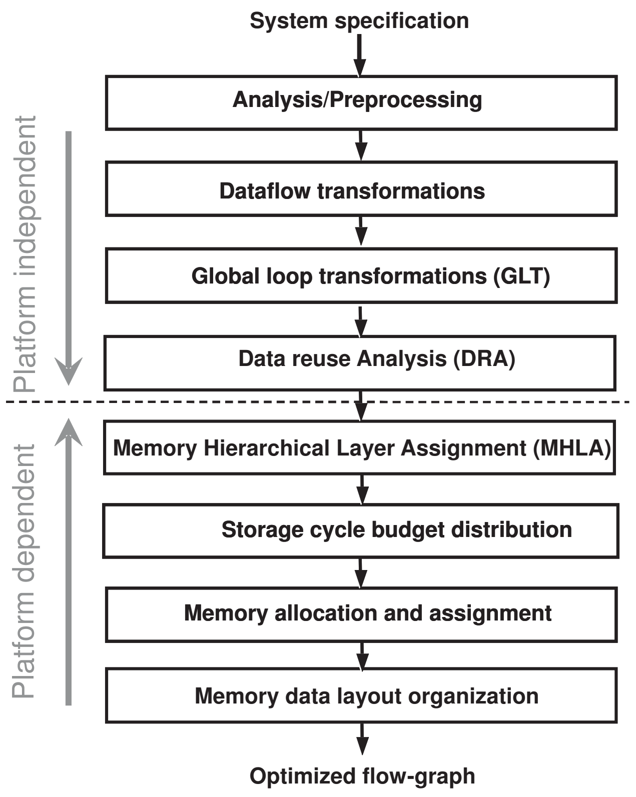

System specification
Analysis/Preprocessing
Dataflow transformations
Global loop transformations (GLT)
Data reuse Analysis (DRA)
Memory Hierarchical Layer Assignment (MHLA)
Storage cycle budget distribution
Memory allocation and assignment
Memory data layout organization
Optimized flow-graph
Platform independent
Platform dependent

**Figure 2.1:** DTSE methodology for data transfer and storage exploeration: global overview

the particular memory organization instance that has been selected [*]. They aim at removing redundant accesses in the data flow, optimizing the regularity and locality of data accesses in general, and exploiting the data reuse possibilities explicitly. The following steps are platform dependent, where physical properties of the specific target architecture instance are taken into account to map and schedule the data transfers in a cost- and power-efficient way. Note that the platform dependent steps can be used to explore the search space for an optimized memory platform or to map data onto a predefined but configurable (knob-enabled) platform in an optimized way. For each of the steps we describe briefly their general functionality and, when appropriate, their specific relation to this work.

### 2.2.1 Pruning and Related Preprocessing Step

This step precedes the actual DTSE optimizations; it is intended to isolate the data-dominant code which is relevant for DTSE, and to present this code in a way which is optimally suited for transformations [30]. All freedom is exposed explicitly, and the complexity of the exploration is reduced by hiding constructs that are not relevant. [104, 103] have recently presented systematic preprocessing techniques for data dependent constructs, making it possible to include more code for exploration at the subsequent steps. Also the transformation of the data references into a dynamic single assignment form [142] belongs to this step.

### 2.2.2 Global Data Flow Transformation

The original algorithm often contains bottlenecks preventing code restructuring transformations to be applied. The global data-flow transformations remove these bottlenecks and are as such enablers for the subsequent steps of the DTSE methodology [31]. Another goal of this step is the removal of access redundancy in the data-flow. Examples of global data flow transformations are advanced signal substitution avoiding unnecessary copies of data, and shifting of ”delay lines” through the algorithm to reduce the storage requirement. Details on this step can also be found in [70, 71].

The memory footprint estimation techniques from this dissertation can be used to give an early feedback on the transformation effect.

[*]Also these steps are still technology and library dependent though as the relative size of specific cost functions would influence their outcome

### 2.2.3 Global Loop Transformations

The loop transformations step within the DTSE methodology aim at globally improving the data access locality and regularity for multi-dimensional array signals. They are applied globally across the full code, not only on individual loop nests but also across function scopes because of the selectively inlining applied in the preprocessing step. An automatic compiler technique is currently being developed. The original work of [141, 54] has been used as a basis. That technique has been extended with more refined abstract cost estimators of the subsequent memory related optimization steps, specifically data reuse and locality reuse [35, 34]. [144, 48] has further presented an algorithm for memory optimization focusing on loop fusion and loop shifting.

The HMSE technique presented in this dissertation is very useful during global loop transformation exploration. It forms a significant extension of the early estimators in [35, 34]. A more complete overview of related work on loop transformation and size/reuse estimation techniques is therefore given in Section 2.3.

### 2.2.4 Data Reuse Analysis

The data reuse analysis (DRA) step in the DTSE methodology exploits temporal locality in the data accesses and identifies frequently accessed data. This allows these data to be be copied to the smaller memories closer to the data path and accessed multiple times from there. This both reduces the power consumption and improves performance as accessing smaller memory closer to the processor is faster and more power efficient than accessing the larger off-chip memory.

A basic systematic data reuse methodology [45, 157] has been developed. That methodology is however only manually applicable and not directly implementable in a fully automated tool. Moreover, it has restrictions on the actual data reuse behavior that can be handled. In [138, 140, 139] some vital cost parameters are introduced to describe a more complete search space. They explore the relation between these parameters and the cost function for power and memory size, and propose heuristics to steer the search for a good solution. They proposed an analytical model for the cost parameters as a function of the index expression and loop bounds. This leads to a fully automatable design technique for all loop-dominated applications to find optimal memory hierarchies and generate the corresponding optimized code. [139] attempts to explore tradeoffs between SPM size and power, assuming an optimal run-time placement of data in SPM. But no technique is presented yet for automatic tool implementation.

Besides the techniques developed at IMEC, there also exist other techniques [24, 69, 72]. [24] proposes a method to evaluate the lifetime of stencil elements. This means an exact analysis of reuse factor but it is high complexity for analysis and for generated code. [69] does the analysis by evaluating the reuse distance of the same set of data between the current and next iteration of a certain loop level. It ignores boundary effects and results in simpler generated code. However, all these methods work on geometrical model which is quite computation expensive. [72] presents another related work as also reviewed in Section 2.3. They proposes to have a given memory hierarchy and come up with the best loop transformations, or to use the current program to design the optimal memory hierarchy. This is achieved based on an abstract data reuse performed on transformation matrices† at loop transformation stage but they does not perform a full reuse exploration.

Our memory footprint estimation can be used at this step to find the memory footprint requirement for each array. The actual memory footprint requirement for one array may be much smaller than the declared size. This effect should be taken into account during the data reuse analysis. It is also an important step of the HMSE methodology presented in this dissertation. A novel data reuse analysis technique has also been presented in this dissertation. Further comparison among the different data reuse analysis techniques will be presented later in Chapter 6.

### 2.2.5 Memory Hierarchy Layer Assignment

Based on the data reuse exploration performed in the previous step, the memory hierarchy layer assignment (MHLA) step assigns the identified frequently accessed data (which is usually part of an array), together with the original arrays, onto the given memory platform instance. This is also the first platform dependent step. Usually it is combined as a sub-step in the subsequent storage cycle budget distribution step [30]. The mapping is done using cost functions of power consumption, performance and size requirement. The backtracking algorithm is introduced which can achieve an near-optimal mapping. This is based on bandwidth and high-level memory size estimation. The memory class of each of the memory layers is always given (e.g. ROM, SRAM or DRAM and other RAM ”flavors”). Further data assignment within each layer, such as memory/bank allocation, is still left to be optimized at a later step. Details on this step can be found in [27].

There also exist extensive other work on memory assignment. Panda et al.

†What a transformation matrix is will be discussed later in Chapter 3

present early work for efficient utilization of SPM in [107]. Steinke et al. also propose their algorithm for data and code assignment to SPM for energy reduction in [128]. Nguyen et al. in [100] proposes to take advantage of compiler analysis of data access patterns and make it dynamically portable across SPMs of any size at run-time. All these work use greedy algorithms and also does not offer trade-offs between power consumption, performance and size requirement as [27] does.

This is a step where the memory footprint estimation presented in this dissertation can be used to give high level memory footprint requirement feedback. Because this is also a step at where loop transformations have a significant effect on the data accesses in the memory hierarchy structure, it is hence essential to estimate the data mapping in this step during the evaluation of the effect of loop transformation. This step is hence considered in the HMSE methodology presented in this dissertation. Two MHLA estimation algorithms are introduced in this dissertation later in Chapter 6.

### 2.2.6 Storage Cycle Budget Distribution

Due to the additional power consumption and design cost of multi-port memories and excessively complex multi-memory architectures, it is necessary to determine in an optimal way which data is going to be accessible simultaneously. This must be done within a limited storage cycle budget given by stringent real-time constraints. The storage cycle budge distribution step mainly determines the bandwidth/latency requirements and the balancing of the available cycle budget over the different memory accesses. The cycle budget is distributed over different loop nests through loop transformations such as splitting, merging, and reordering of loops and loop bodies. Details on this step can be found in [159, 158, 161, 102].

The memory footprint estimation presented in this dissertation can be used at this step to find the storage requirement changes caused during the storage cycle budget optimization.

### 2.2.7 Memory Allocation and Assignment

The goal of the memory allocations and assignment step is to allocate memory units and ports (including their types) from a memory library and to assign the data to the best suited memory units, given the cycle budge and other timing constraints [12, 126]. It is performed based on the MHLA output. The combination of the tools for the storage cycle budget distribution step and memory allocation and assignment step allows to derive Pareto trade-off curves of the background memory related cost (e.g., power) versus the cycle budget. Details on this step can be found in [28].

The assignment of arrays to physical memories should be performed taking into account in-place mapping opportunities. At this point of the design trajectory, much of the execution ordering is fixed, so the final in-place optimization step can be used directly as feedback. Still, the memory footprint estimation technique presented in this dissertation can be employed for fast feedback during the first scan of the solution search space, when the number of implementation alternatives is still large.

### 2.2.8 Memory Data Layout Optimization

In the above 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. The physical address for every signal element still has to be determined. This transformation is the data layout decision. It is also called in-place optimization. This involves several sub-steps and focuses both on the on-chip memories (cache(s) or the SPM) 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 by performing the in-place mapping. The execution ordering is now fully fixed, but the storage order can still be optimized to determine the actual layout of arrays in memory. This is done through two sub-steps. First the intra-array in-place optimization is performed by fixing individual array storage order and it determines the internal organization of an array in memory (e.g. row-major or column-major layout). Next, the inter-array storage order fixes the relative positions of arrays in memory reusing the same memory locations through inter-array in-place mapping. Details on how to perform such in-place mapping can be found in [41, 44, 42, 43].

For hardware-controlled caches advanced main memory layout organization techniques have been developed, which allow for removal of most conflict misses due to the limited cache associativity [83, 98].

This step performs the in-place mapping that determines the actual storage requirement so estimation is not needed. To have high accuracy, the estimates at earlier system level design steps, e.g., during the loop transformation step and hierarchical memory layer assignment step, should be close to the implementation results achieved here. It is however even more important that the estimates have high fidelity, ensuring that the best solution is selected when alternatives are being compared [56]. The memory footprint estimation techniques presented in this dissertation fulfill this requirement, as will be demonstrated in later chapters.

## 2.3 Loop Transformations

Given the very large body of work performed w.r.t. loop transformations (LT), this overview cannot be complete. This section will give a brief review and the reader is referred to [16, 97] for more details.

Loop transformations are an essential part of modern optimizing and parallelizing compilers. They are mainly used to enhance the temporal and spatial locality for cache performance and to expose the inherent (asynchronous or synchronous) parallelism. A large body of work has been performed for this purpose for a long time, e.g. [7, 154, 16, 152, 8, 97]. Their main goal is, however, to reveal and exploit code and data parallelism for improving performance, so that multiple instantiations of (parts of) a loop nest can be executed simultaneously, and to improve cache hit ratio and execution time.

Loop transformations have also been studied in the embedded system design domain where memory size and energy consumption are important factors besides performance. Loop transformations can reduce the required buffer size and improve access behavior to optimize the embedded application(s). Improved data access locality in turn can improve the efficiency e.g. in terms of power consumption and memory footprint of applications [37, 127, 53, 144, 72, 73]. This is also a crucial step within the DTSE methodology as described in the previous section.

Initial research on loop transformations focused on optimizing locality and parallelism within one (perfectly) nested loop, e.g., [152]. This approach did not allow optimization of large buffers between loop nests. [39, 52] presented a generic approach for performing loop transformations by applying a loop transformation on every statement in the loop nest. Their research was mainly focused on parallelization, though.

Other groups have proposed to perform these transformations across all loop nests in one procedure or in the entire program [54, 33, 58]. Within the DTSE framework, researchers propose to divide the loop transformations into several sub-steps. Van Swaaij et al. [141] 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 et al. [35, 34] has split the placement step in a linear transformation step and a translation step. To steer these two steps, they also provide several heuristics based on abstract data reuse and size optimization cost functions. Different kind of linear transformation techniques are taken into account in their linear transformation step while loop fusion and loop shifting are considered in the translation step. Different loop transformations will be categorized and illustrated later in Chapter 3. This split further reduces overall algorithm complexity. Verdoolaege et al. [48] has further shown that it is possible to avoid the ordering step when only the memory organization itself is targeted on a platform that is not data-parallel. They present a greedy algorithm for the translation optimization [144]. Their global loop transformation step can be viewed as a pre-compilation phase, applied prior to conventional compiler loop transformations. This preprocessing also enables later memory customization steps such as memory hierarchy assignment, memory organization, and in-place mapping to arrive at the desire reduction in storage and transfers.

When transformations are to be performed globally across all loop nests, there exists a huge number, thousands or more, of loop transformation possibilities. It is hence crucial to perform an automatic loop transformation exploration in order to find the optimal one(s). [34, 144] and [33, 58] present frameworks that facilitate automatic search of a sequence of loop transformations for deep parallelism and memory hierarchies. However, up till now the focus has been on finding one optimal solution. For real-life applications, it has been shown that multiple optimal solutions often exist, depending on the memory platform considered. This is also demonstrated by experiments in Chapter 9. It is hence essential to find systematic techniques to identify all of these optimal solutions, especially when the memory architecture is not given at the early transformation stage. Kandemir et al. [72] present a method with these two goals. They either assume a given memory hierarchy and find good loop transformations for the current program, or design an optimized memory hierarchy for the current program, based on an abstract data reuse analysis performed on data access pattern of the code. Their approach only performs a limited number of loop transformations.

Most of the research in this field is based on the use of geometrical models where the set of transformations is limited to affine matrices [32, 39, 52, 76, 47]. The techniques that do not use a geometrical model, e.g. [6, 16, 113], can usually only apply a limited set of linear transformations. Typically affine transformations are loop interchange, loop reverse, loop skewing, loop fusion and loop shifting. Other important loop transformations are, e.g., loop tiling, strip mining and loop coalescing. How to perform the different loop transformations will be illustrated in Chapter 3.

## 2.4 Memory Size Estimation

Memory size estimation and/or computation has been tackled in the past both in register-transfer level (RTL) programs at scalar level [86, 135, 111, 57, 101] and in behavioral specifications at non-scalar level [9, 162, 60, 81, 121, 163]. The non-scalar level typically refers to multi-dimensional arrays. Different techniques in these two levels are reviewed below separately.

### 2.4.1 Scalar-based Estimation

Most work on memory size estimation was originally scalar-based, also called signals or variables. The register allocation/assignment problem in programs was initially formulated in the field of software compilers [5], aiming at a high-quality code generation. The problem of deciding which values in a program should reside in registers (allocation) and in which register each value should reside (assignment) has been solved by a graph coloring approach [25].

In the field of synthesis of digital systems, starting from a behavioral specification, the register allocation /assignment problem has been solved for non-repetitive schedules like the *left-edge* algorithm in polynomial time, when the life-time of all scalars is fully determined [86]. Techniques such as clique partitioning are also exploited to group variables than can be mapped together [135].

In [101], a lower bound for the register count is found without the fixation of a schedule, through the use of *As-soon-As-Possible* (ASAP) and *As-late-As-Possible* (ALAP) constraints on the operations. A lower bound on the register cost can also be found at any stage of the scheduling process using Force-Directed scheduling [111]. *Integer Linear Programming* (ILP) techniques are used in [57] to find the optimal number of memory locations during a simultaneous scheduling and allocation of functional units, registers and busses. Good overviews of the techniques can be found in [56].

### 2.4.2 Array-based Estimation

Common to all scalar-based techniques is that the number of scalars is limited. If multi-dimensional arrays are treated, the computation time increases dramatically and these techniques would break down if the arrays are flattened and each array element is considered a separate scalar. For high-level algorithmic specifications characterized by deep loop nests and multidimensional arrays, this problem must be overcome using array-based storage requirement estimation. Several research teams have tried to perform array-based storage requirement estimation. [10] has given an review of most of work.

Typically, the array is considered as an unit or split into suitable units before estimation is performed. The estimation approaches can be basically split into two categories: those requiring a fully-fixed execution ordering, and those assuming a non-procedural specification where the execution ordering is still not (completely) fixed. The execution ordering decides the order in which statements in the code are executed, and hence the order in which array elements are written and read. When it is unfixed, we can assume any ordering as long as no array element is read before it is written. The techniques falling in the second category will be addressed first.

Balasa et al. propose to estimate the memory footprint when the execution ordering is unfixed [9]. Data dependency analysis is performed and the total memory footprint is found through a greedy traversal of the relative graph. In order to deal with multiple read and write statements for the same array, arrays are partitioned into non-overlapping basic sets. Basic set is described as polytopes using linearly bounded lattices (LBLs). The basic set sizes, and the sizes of the dependencies, are found using an efficient lattice point counting technique. The dependency size is the number of elements from one basic set that is read while producing the depending basic set. This information is used to generate a data-flow graph where the basic sets are the nodes and the dependencies between them are the branches. The total storage requirement for the application is found through a traversal of this graph, where basic sets are selected for production by a greedy algorithm. A basic set is ready for production when all basic sets it depends on have been produced and is consumed when the last basic set depending on it has been produced. The maximal combined size of simultaneously alive basic sets gives the storage requirement. Basic set partitioning is however time consuming. Since Balasa's technique does not take into account the execution ordering, large over-estimation can occur.

Kjeldsberg et al. propose to estimate the size allowing a partially fixed execution ordering [81, 82, 79]. Their approach allows to estimate the size requirement while guiding the fixing of the unfixed execution ordering. They perform the estimate in two steps. First the storage requirement of individual data dependencies are estimated by taking into account the partially fixed execution ordering with lower bound and upper bound output. As the execution ordering is gradually fixed, the upper and lower bounds on the data dependencies converge. The size requirement is calculated by counting the number of nodes in the iteration domain at where an array is written while constrained by the Extreme Dependency Vector (EDV). The EDV is the maximal projection of all the dependency vectors on each iteration dimension. This concept will be illustrated later in Chapter 3. The estimate then continues between different data dependencies by grouping simultaneously alive data dependencies together. The group which requires the largest size requirement defines the memory requirement for the application. By taking into account the partially fixed execution ordering (based mainly on loop interchange), their approach avoids the possible overestimates due to the total ordering freedom. Their approach by itself is very fast, assuming the EDV is already given. However, calculating the EDV is still time consuming with the existing dependency analysis techniques, using the integer linear programming algorithm [146, 93, 92] and Fourier-Motzkin variable elimination algorithm [116, 96]. Further more, a projection of all dependencies at each loop dimension needs to be performed. Techniques presented later in this dissertation can help overcome this problem of calculating the EDV.

All other techniques assume that the execution ordering is already fixed. Verbauwhede et al. propose to build up a production axis for each array to model the relative production and consumption time of the individual array elements [143]. The difference between these two dates equals the number of array elements produced between them and the maximum difference defines the size requirement for the array. The maximum difference is calculated based on an integer linear programming model using the OMEGA calculator [116], which is an ILP solver based on the Fourier-Motzkin elimination method. ILP calculation can however be computational expensive.

Zhao and Malik describes a methodology for so-called exact memory size estimation for array computation [162]. It is based on live variable analysis and integer point counting for intersection/union of mappings of parameterized polytopes. In this context, a polytope is the intersection of a finite set of half-spaces and may be specified as the set of solutions to a system of linear inequalities. It is shown that it is only necessary to find the number of live variables for one statement in each innermost loop nest to get the minimum memory size estimate. The live variable analysis is performed for each iteration of the loops however, which makes it computationally hard for large multi-dimensional loop nests. This counting of live array elements is done by set operations (union, intersection) which are the whole or parts of the iteration domains. Grun et al. use the data dependency relations between the array references in the code to find bounds on the number of array elements produced or consumed by each assignment [60]. Then, a memory trace as a function of time is found. The peak memory trace contained within the bounding rectangles yield the total memory requirement. If the difference of boundaries for the critical rectangle is too large, the corresponding loop is split and the estimation is rerun in order to improve the estimation accuracy. In a worst case, a full loop unrolling is required to achieve a satisfactory estimate, which is unaffordable.

In [120], Ramanujam et al. propose to use a reference window for each array in a perfectly nested loop. At any point during execution, the window contains array elements that have already been referenced and will also be referenced in the future. These elements are hence stored in local memory. The maximal window size found gives the memory requirement for the array. If multiple arrays exist, the maximum reference window size equals the sum of the windows for individual arrays. Inter-array in-place is consequently not considered. They further present on how to reduce the size through loop level transformations.

Zhu et al. [163] propose to first decompose the array references into disjoint linearly bounded lattices. Then the memory size at the boundaries between the blocks of code is calculated. The maximum memory size inside each block is further estimated and the maximum found defines the overall memory footprint. Unfortunately, all the above listed techniques are still too computationally expensive to be performed frequently during system level design exploration. This is especially non-trivial when the applications become realistic and hence large.

Rydland et al. extends the work of Kjeldsberg et al. and presents a grouping algorithm for simultaneously alive dependencies considering inter-array size estimation [121]. They assume the execution ordering to be fully fixed. It reuses the technique of individual data dependency estimate used by Kjeldsberg et al.

At the early steps of the system level design trajectory, the memory footprint estimation usually needs to be performed repeatedly to give the designer/tool fast feedback during exploration. The above listed techniques are still too computationally expensive to be performed frequently during system level design exploration. This is especially non-trivial when the applications are large. On the other hand, the estimation accuracy varies between the techniques. [162, 60, 163] can find accurate size requirement but the computation complexity for it is also very high. On the other hand, it is either very computational expensive or not realistic to achieve their reported size with any existing memory in-place optimization techniques which is discussed in the following section.

## 2.5 Memory In-place Optimization

In the previous section, we have reviewed array-based memory in-place estimation techniques. This estimation is usually required at the early design stages. In contrast, the actual memory in-place optimization, reusing memory locations between different data, is usually performed at later design stage. At the later design stage, the execution ordering is already fixed and other optimizations, such as data allocation and assignment, have been performed. Memory optimization has been a research topic [49] for a long time as is the case for storage requirement estimation. Most previous work on storage requirement optimization is also scalar based and used for foreground memory. Many of the techniques described in Section 2.4.1 are also used for optimization purpose. Additional techniques are presented in [129, 105]. In this section we will review different array-oriented memory in-place optimization techniques.

De Greef et al. have presented a technique for background memory requirement optimization by taking advantage of storage ordering [41, 44, 42, 43]. Their techniques focus on one-dimensional linearized address generation for arrays. This is part of the DTSE methodology as described in Section 2.2. Some more details are given here since their techniques are focused on optimization through in-place mapping. It uses a pragmatic two-phase approach to optimize the storage ordering so as to maximize the possibility of in-place mapping considering linear address generation in an one-dimensional (denoted by 1-D) window. First the intra-array storage order determines the internal organization of an array in memory (e.g. row-major or column-major layout). Next, the inter-array storage order fixes the relative positions of arrays in memory reusing the same memory locations through inter-array in-place mapping. Heuristics have also been presented in order to speed up the storage ordering exploration.

Troncon et al. in [134] propose to compute in the original n-dimensional space of array indices, an n-dimensional bounding box (i.e. n modula computed separately as the maximal index address difference in each dimension, defining an n-dimensional rectangular window), instead of a one-dimensional window (i.e., the modulo b) in the linearized space of addresses.

The same problem of memory reuse analysis using a polyhedral model has been investigated in the context of the ALPHA language [147, 118]. Wilde et al. studied the problem of memory reuse for systems of recurrence equations, a computation model used to represent simple algorithms to be compiled into circuits. The principle for memory reuse is based on projections of multi-dimensional arrays. A tight bound on the number of linearly independent memory projections is found, giving rise to an optimal memory reduction.

In [87, 88], Lefebvre and Feautrier are concerned with automatic parallelization of static control programs, developed a technique of partial data expansion, which (even if not the original goal) can also be used for memory reduction. [62] performed the memory aware compilation through accurate timing extraction. The memory optimization is just one factor during their compilation process. [78] proposed to perform an array address translation method for increasing memory bandwidth and minimizing the number of overhead cycles in video applications.

Darte et al. in [37] have developed a mathematical framework based on critical lattices in a polyhedral model. They first consider the set of indices that conflict, i.e., those that can not be mapped to the same memory location. Then, they construct the set of different modular mappings and the set of conflicting index differences. The memory required by an optimal modular mapping is equal to the determinant of the corresponding lattices. Their approach provides alternative solutions of linearized spaces of addresses or affine spaces of addresses. However, it does not consider more advanced storage ordering which might further reduce the size required but with complex index generation. There is actually a tradeoff between index generation complexity and size requirement. [136, 137] have presented a framework for array allocation for both affine and non-affine code. They introduced a integrated solution which combines the specialized solution for affine program allocation with a general framework for other code. The difference between their allocation technique for an array from the above mentioned techniques are not clearly addressed.

Besides, the PHIDEO silicon compiler by Lippens et al. also focuses on optimal memory synthesis [90, 91]. It is tuned to fixed rate streaming data applications and is not intended for a general memory hierarchy. Scheduling is performed before memory synthesis. The HADES synthesis framework focuses on background memory management and uses graph matching techniques to determine if arrays need to be stored in background memory and to investigate in-place mapping opportunities [112]. It is used towards process control systems and consider only one and two dimensional input data. The ASSASYN tool [122] designs memory systems using array grouping techniques and dimensional transformations. The declared size of arrays is used and the lifetimes of different arrays are not taken into account, so neither intra-array nor inter-array in-place mapping are explored.

# Chapter 3

# Geometrical Model and Loop Transformations

The geometrical model (denoted by GM) has been widely used to model the data and control flow for the target class of data intensive applications. It has been used for performing (automatic) loop transformations [36, 145, 58] and is also what our work is based on. In this chapter, we first give an intuitive introduction to the polyhedral geometrical model in Section 3.1. In this model, objects can be of any convex polytope shape. Following in Section 3.2 a bounding box geometrical model is introduced. The bounding box geometrical model is a simplification of the general polyhedral geometrical model. Operations on bounding box objects are much simpler and less computationally expensive. The bounding box geometrical model is mostly used in this dissertation in order to achieve fast estimation. In Section 3.3, the different loop transformation techniques are briefly reviewed. Their representation as matrix operations for the purpose of automatic loop transformation exploration is also discussed.

## 3.1 Polyhedral Geometrical Model

For the target class of data intensive applications, data, operations, dependencies, and the order in which they are handled, are very important. It is therefore vital to be able to model data and operations and their relationship to each other accurately. There can be millions of operations and data elements, but it is usually possible to detect much regularity in how they are handled. This opens for grouping of elements that are treated the same way. It has been shown that a polyhedral geometrical modeling of groups of data and groups of operations is a viable model [141]. The polyhedral geometrical model, also called the polyhedral dependency graph model, is widely used for modeling data and control flow [41, 36, 145, 148]. It was originally used in systolic array synthesis and has been applied in diverse optimization techniques, including automatic loop parallelization [33, 58] and memory usage optimization [118, 141, 59]).

In such a model, the geometrical objects are typically polyhedra or related types of sets, since such representations are very compact and can be manipulated more efficiently than arbitrary sets. Other approaches, e.g., Pressburger formulas and linear bounded lattice (LBL), have also been used to represent the geometrical objects. The difference is just a manner of representation. This Section describes the important concepts in the polyhedral geometrical model. A thorough introduction with more mathematical description can also be found in [41, 145].

```
    for (i=0; i<=4; i++)
      for (j=0; j<=5; j++)
        for (k=0; k<=2; k++)  {
S1:       A[i][j][k] = ...;
          if (i>=1) && (j>=2)
S2:          ... = f(A[i-1][j-2][k]);
      }
```

**Figure 3.1:** Code example

## 3.1.1 Iteration Domain

An iteration domain (denoted by $I$) of a statement [51, 44, 145, 79] is a geometrical domain in which each node with integral coordinates represents exactly one execution of the statement. It is also called node space and computation space. Its description is derived from the constraints corresponding to the boundaries of the surrounding loops and conditions that restrict the execution of the statement. Each node within the iteration domain can be identified by its loop iterators (denoted by $\vec{i}$). $\vec{i} = [i_1, i_2, \ldots, i_m]$ counting from outermost iterator to innermost. The parameter $m$ is implicit and $i_g$ is the $g$th iterator in the loop nest. In this dissertation, loop dimension is also used referring to the loop iterator. Each statement has its own iteration domain, and the set of points in this domain represents all the executions of this statement.

Since the iterators are required to have linear constraints, one iteration domain can always be expressed as rational polyhedron[*], which is a subspace of $(\mathbb{Z}^m)$ bounded by a finite number of hyperplanes

$$I = \{[i_1, i_2, \ldots, i_m] \mid \sum_{g=1}^{m} C_g^L \le i_g \le C_g^U \wedge \bigwedge_{d=1}^{n} (\sum_{g=1}^{m} A_g^d i_g + a_d = 0) \wedge [i_1, i_2, \ldots, i_m] \in \mathbb{Z}^m\} \tag{3.1}$$

in which $C_g^L$ and $C_g^U$ means the constant lower and upper constraint value at the $g$th dimension. $\sum_{g=1}^{m} A_g^d i_g + a_d = 0$ refers to the constraints expressed in the if-condition clauses in which $A_g^d$ refers to the $g$th loop iterator in the $d$th constraint expression. There are always an explicit number of such constraints defined as $n$.

Figure 3.1 presents a simple code example in which array $A$ has been referenced (both written and read in this case) in two statements, $S1$ and $S2$, respectively. The iteration domain for the two statements, denoted as $I_{S1}$ and $I_{S2}$, are represented as[†]

$$\begin{aligned}
I_{S1} &= \{[i,j,k] \mid 0 \le i \le 4 \wedge 0 \le j \le 5 \wedge 0 \le k \le 2 \wedge [i,j,k] \in \mathbb{Z}^3\} \\
I_{S2} &= \{[i,j,k] \mid 0 \le i \le 4 \wedge 0 \le j \le 5 \wedge 0 \le k \le 2 \wedge i \ge 1 \wedge j \ge 2 \wedge [i,j,k] \in \mathbb{Z}^3\} \\
&= \{[i,j,k] \mid 1 \le i \le 4 \wedge 2 \le j \le 5 \wedge 0 \le k \le 2 \wedge [i,j,k] \in \mathbb{Z}^3\}
\end{aligned}$$

Figure 3.2 graphically illustrate the iteration domains of the two statements in this simple code. Each node in the figure corresponds to one iteration of the loops surrounding the statements at where the statements are executed. The iterator values are the values over which a dimension iterates. For the $i$-dimension in Figure 3.1 , the iterator $i$ has the values from 0 to 4. The iteration domain of statement $I_{S1}$ contains the iteration nodes at where the array $A$ is written. The iteration domain of statement $I_{S2}$ contains the iteration nodes at where the array $A$ is read, which is further constrained by the $if$-clause than statement $S1$. To unify the write and read accesses, it is said, in this dissertation, that array $A$ is referenced in the two statements. Since array $A$ is written and read in two statements respectively, the iteration domains are also represented as array write and read iteration domains (also symbolled as $I$ and $I'$ respectively as will be used later on). In this dissertation, a specific iteration within the iteration domain is contained by the small bracket symbols. For example, the first iteration node within the iteration domain at where the array is written is expressed as $(0,0,0)$.

[*]Since all the polyhedra in this text will be rational, we will usually omit this qualification.

[†]In the example illustration, we will represent the iteration domain in the parametric set instead as it is more compact representation.

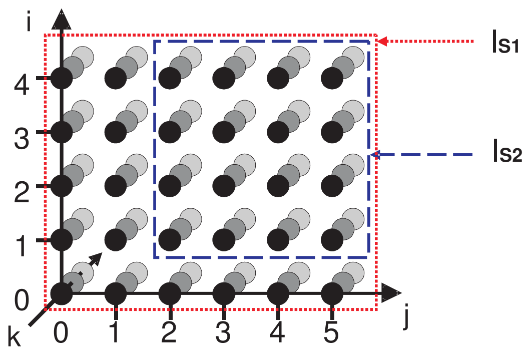


**Figure 3.2:** Iteration domains (I) of two statements (S1 and S2)

### 3.1.2 Data Domain

As we are interested in the memory related issues such as memory accesses and storage requirements, both the execution of the statements of a program (by means of iteration domains) and the accesses to the array elements are modeled. This has lead to the concepts of data domains (denoted by $D$), which is also called variable domains [41]. A data domain is a mathematical description of the variables in one array.

Each point with integer coordinates in this domain corresponds to exactly one variable in the array. It is not necessary that every statement accesses all array variables completely during the execution. Typically the executions of a statement only access part of or a few variables of an array. These accessed variables consist of the data domain for that specific array reference in that statement. The data domain for one array would therefore be the description of all the variables accessed for all its references. Data domains have further been distinguished as definition domains and operand domains, corresponding to array write references and array read references respectively [41, 79].

The relations between the executions of the statements and the arrays that are being referenced, are represented by means of mathematical mappings (denoted by $M$). Each reference has an associated index expression $E$ which maps the iterators of the statement to the accessed index. The index expression is required to be an affine function of the iterators in our work. They are expressed as

$$\begin{aligned} E = \{[i_1, i_2, \ldots, i_m] \mapsto [d_1, d_2, \ldots, d_n] \mid \bigwedge_{h=1}^{n} d_h = \sum_{g=1}^{m} c_g^h i_g + c_{m+1}^h \\ \wedge [i_1, i_2, \ldots, i_m] \in \mathbb{Z}^m \wedge [d_1, d_2, \ldots, d_n] \in \mathbb{Z}^n\} \end{aligned} \tag{3.2}$$

where $d_h$ is the $h$th index expression dimension for the array. $c_g^h$ is the index expression coefficient of the $g$th loop iterator at the $h$th index expression dimension, $c_{m+1}^h$ means the constant offset at the $h$th index expression dimension. Note that, in this dissertation, $g$ is always used as subscript for the the loop dimension or the iteration domain and $h$ is always used as as subscript for the array dimension or data domain.

The set of all indices encountered by an array reference over all iterations of the surrounding loops consists of the data domain for that reference. The number of dimensions for the index expression defines the number of array dimensions and hence its data domain. We will use $[d_1, d_2, \ldots, d_n]$ to represent the dimensions of a data domain, where the parameter $n$ is implicit. The data domain of one array reference hence is

$$\begin{aligned} D = E(I) = \{[d_1, \ldots, d_n] \mid \bigwedge_{h=1}^{n} d_h = \sum_{g=1}^{m} c_g^h i_g + c_{m+1}^h \\ \wedge [i_1, i_2, ..., i_m] \in I \wedge [d_1, d_2, ..., d_n] \in \mathbb{Z}^n\} \end{aligned} \tag{3.3}$$

where the comparison is applied componentwise.

For the example code, the mapping and the data domain for array $A$ referenced in statement $S2$ are

$$\begin{aligned} E_{A,S2} =& \{[i, j, k] \mapsto [d_1, d_2, d_3] \mid d_1 = i - 1 \wedge d_2 = j - 2 \wedge d_3 = k \\ & \wedge [i, j, k] \in I_{S2} \wedge [d_1, d_2, d_3] \in \mathbb{Z}^3\} \\ D_{A,S2} =& E_{A,S2}(I_{A,S2}) \\ =& \{[d_1, d_2, d_3] \mid d_1 = i - 1 \wedge d_2 = j - 2 \wedge d_3 = k \\ & \wedge [i, j, k] \in I_{S2} \wedge [d_1, d_2, d_3] \in \mathbb{Z}^3\} \\ =& \{[d_1, d_2, d_3] \mid 0 \leq d_1 \leq 3 \wedge 0 \leq d_2 \leq 3 \wedge 0 \leq d_3 \leq 2 \\ & \wedge [i, j, k] \in I_{S2} \wedge [d_1, d_2, d_3] \in \mathbb{Z}^3\} \end{aligned}$$

### 3.1.3 Execution Ordering and Common Iteration Space

In our work, the execution ordering of the program code is assumed to be fixed already. We need to be able to represent the exact execution ordering of operations within a given polyhedron and the relative execution order of the whole code. To order the elements of a given polyhedron, we typically use the lexicographical order corresponding to the sequential execution order.

**Definition (Lexicographic order $\prec$** Given two $m$-dimensional loop iterations $\mathbf{i} = (i_1, i_2, \ldots, i_m)$ and $\mathbf{j} = (j_1, j_2, \ldots, j_m)$ satisfy $\mathbf{i} \prec \mathbf{j}$ iff there exists an integer k, $1 < k < m$ such that $i_1 = j_1, \ldots, i_{k-1} = j_{k-1}$ and $i_k < j_k$. Note that this lexicographical order can be expressed using linear constraints.

Within one loop nest, the loop iterators $\vec{i}$ already define the order of all the loop iterators, starting from the outermost dimension. However, there often exist multiple statements within one loop nest and multiple loop iterators can be located in parallel at the same level within one loop nest but executed sequentially. The order of loop iterators can not represent their relative execution ordering. This is, for example, the case between the two statements for the code in Figure 3.1. The relative execution ordering of multiple loop nests in the program code also needs to be identified in the model. In order to uniquely identify the exact execution ordering of the program code, pseudo time dimensions (denoted by $t$) are added to represent their relative execution ordering of both iterators and statements at a given nest level. The one that is executed first should have a time dimension value lexicographically smaller than the others. We often use integer values to represent the relative execution ordering giving the first one the value 0 at its time dimension, the second one the value 1 and so on.

In this way, a *common iteration space* is generated for the analyzed program code. This is also discussed in [131] with the name *common node space*. A common iteration space can be regarded as representing one loop nest (or one iteration domain) surrounding the entire, or a given, specific part of the code. In such a way, the original code is also converted in the perfectly nested loop format, which is the prerequisite of performing unimodular transformations as will be explained in Section 3.3. The common iteration space is similarly required in [36] in order to allow performing global loop transformations cross multiple loop nests or procedures. It is also used in [79] for storage size requirement estimation though the execution ordering of the code can also be unfixed or partially fixed.

One simple way to generate a common iteration space is by adding one pseudo time dimension before each loop dimension and before each statement. The ones which are located in parallel at the same nest level but executed sequentially should have different values for that added time dimension in order to represent their lexicographical order. Otherwise, they have identical value for that added time dimensions. However, when the loop iterator(s) already identify the relative execution ordering uniquely, the pseudo time dimension(s) becomes redundant and is not really required. For the example code in Figure 3.1, only one time dimension is necessary in order to uniquely identify the relative execution ordering of the two statements while any other added time dimension are redundant. Figure 3.3 shows how the code looks when one time dimension is added before the two statements. The time dimensions should hence be represented also in the representation of the iteration domain. The loop iterators $\vec{i}$ now should be updated as $[i, j, k, t]$ and the polyhedra representation of the iteration domains for the two statements should also be modified accordingly. Efficient generation of the common iteration space including how to represent its statements is related to the code generation after loop transformations. These issues will not be covered here. Further information can be found in [7, 77, 19].

```
for (i=0; i<=4; i++)
  for (j=0; j<=5; j++)
    for (k=0; k<=2; k++)
      for (t=0; t<=1; t++) {
        if (t==0)
S1:       A[i][j][k] = ...;
        if (i>=1) && (j>=2) && (t==1)
S2:       ... = f(A[i-1][j-2][k]);
      }
```

**Figure 3.3:** Example of common iteration space representation

## 3.1.4 Data Dependencies

The domain and order models presented in the previous sections are sufficient to be able to perform data flow analysis of the program. An important concept in data and control flow analysis is that of a data dependency. Data dependency analysis also needs to be performed to check the transformation validity during loop transformations.

In general a data dependency denotes precedence constraint between operations. The basic type of dependency is the value-based flow dependency. A value-based flow dependency between two operations denotes that the first one produces a data value that is being read by the second one, so the first one has to be executed before the second one. Other types of dependencies (e.g. memory-based flow dependencies, anti dependencies, output dependencies, anti-dependencies) also correspond to precedence constraints, but these are purely storage related, i.e. they are due to the sharing of storage locations between different data values. This kind of dependencies is only important for the analysis of procedural non-single assignment code as explained below. Eventually, the goal of dependency analysis is to find all the value-based flow dependencies, as they are the only "real" dependencies. Given the value-based flow dependencies, it is (in theory) possible to convert any procedural non-single-assignment code to single assignment (SA) form [50, 142], where the only precedence constraints left are the value-based flow-dependencies. More information about different types of data dependencies can be found in [51, 85]. Hereafter, we use the term $flow\ dependency$ or $dependency$ to refer to the value-based flow dependency.

#### 3.1.4.1 Dependency

**Definition (Dependency)** An iteration $\mathbf{i} \in \vec{i}$ is dependent on iteration $\mathbf{j} \in \vec{j}$, denoted by $\mathbf{i}\delta\mathbf{j}$, iff there exists an element of an array written at $\mathbf{i}$ and read at $\mathbf{j}$ such that $\mathbf{i} \prec \mathbf{j}$.

**i** and **j** can hence belong to the same or different loop iteration domains. If they belong to the same iterators, a statement depends on itself and it is a self-dependency. Otherwise, the dependency relation between two statements constrains the order in which the statements may be executed. The dependency relation between two statements is the set of all pairs of iterations that exhibit a dependency.

#### 3.1.4.2 Dependency Relation

**Definition (Dependency relation)** Given iterations **i** and **j**, two references to an $n$-dimensional array are to the same element iff the two index expressions for the two array references at iterations **i** and **j** are equal. The dependency problem can then be stated as follows: do there exist iterations **i** and **j** in the two iteration domains such that

$$E(I) = E'(I') \tag{3.4}$$

in which $E_w$ means the index expression which maps the data for the array write reference and $E_r$ means the index expression which maps the data for the array read reference. The iteration domains of the loop iterators defines the dependency constraints.

The above problem is NP-complete considering the polyhedron constraints of the iteration domain and the data mapping from the iteration domain to data domain. In non-polyhedron related work, dependencies are modeled in the above format that retains only their existence and not the exact number of dependency instances which exist. With regard to the computational complexity of deciding the existence of a dependence, [14, 16, 17] serves as good references of early development in the area. Recent developments about dependency analysis can be found in [89, 96, 114, 156]. In the recent work, dependencies are modeled and there are different ways to represent the dependencies, e.g., dependency distance vector [114, 115, 99], dependency direction vector [155, 156], and distance vector polytope/cone [36, 145]. In this dissertation, we will use dependency distance vectors to describe the dependencies.

## 3.1.5 Dependency Vector

**Definition (Dependency vector)** For each pair of iterations **i** $\in \vec{i}$ and **j** $\in \vec{j}$ such that **iδj**, the difference **d** between **i** and **j**, i.e.

$$d = j - i \tag{3.5}$$

at each loop dimension, is called the dependency distance vector. We simply call it dependency vector (DV) hereafter. We can define the DV corresponding to this dependency as the vector pointing from the iteration node at where an array element is written to the iteration node at where it is read, i.e. in the direction of the data flow. In practice, usually multiple dependencies exist between two statements. The DVs of the dependency relations between two statements at where array is written and read are represented by

$$DV = I' - I \forall E'(I') = E(I) \tag{3.6}$$

A dependency is called uniform when all the DVs between two statements have the same length and direction. For example, for the code in Figure 3.1, the DVs are uniform and equal to $[1,2,0]$ corresponding to the ordered loop iterators. When the dependencies are not uniform, the dependencies can differ in both distance and direction. Figure 3.4 shows a code example for which the dependencies are not uniform as drawn in Figure 3.5 [80]. For readability, the figure is split into three, one for each value of the $k$-iterator.

```
      for (i=0; i<=4; i++)
        for (j=0; j<=7; j++)
          for (k=0; k<=2; k++) {
S1:         A[i][j][k] = ...;
            if (i>=k) && (j>=i-k)
S2:           ... = f(A[i-k][j-i+k][k]);
        }
```

**Figure 3.4:** Code example

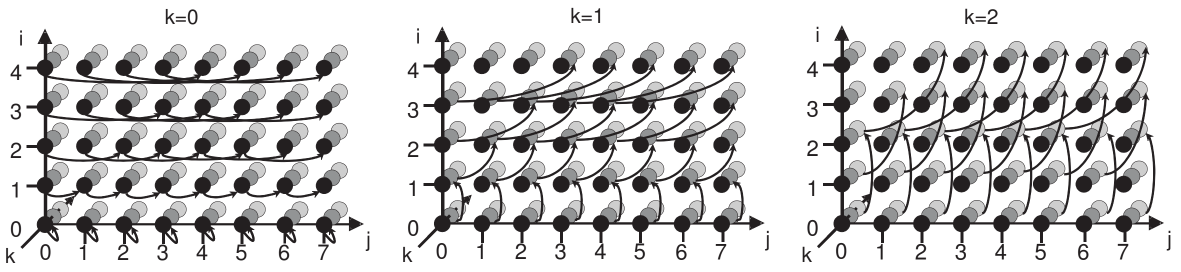


**Figure 3.5:** Dependencies for the example code in Figure 3.4

When drawing the actual DVs, we usually draw the same arrow, but translated to the origin. The end point of this arrow is then equal to the DV value. Figure 3.6 shows the DVs for the above example in this format. As there can be many non-uniform dependencies, it is not always convenient to mathematically represent all the dependencies or list all the DVs. An easy way is to express the dependencies as a dependency function [114, 155] or constraints of the dependency vector polytope/core [36, 145].

Various techniques have been proposed to perform the dependency analysis and hence to calculate the DVs. They can be classified in two types: integer programming algorithms [146, 93] and Fourier-Motzkin variable elimination (of an integer programming problem) techniques [116, 96]. Dependency analysis is required to be performed during each incremental sequence of loop transformation in order to check the validation of the transformation.

## 3.1.6 MDV and EDV

In this dissertation, two specific DVs are particularly interesting: the maximal DV (MDV) and the extreme DV (EDV).

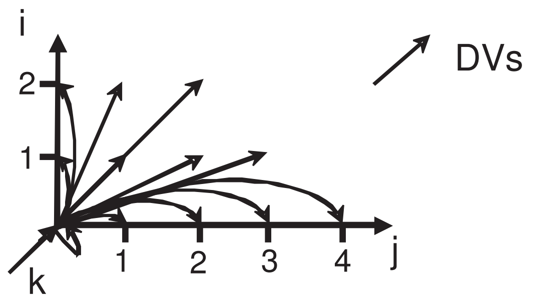


**Figure 3.6:** Dependency vectors for the example code in Figure 3.4

The MDV is the maximal DV among all DVs for the given execution ordering. The MDV will always be one of the existing DVs. The MDV must be one of the DVs with the largest dependency distance at the outermost dimension. Among these candidates, it must be one of the DVs with the largest dependency distance at the second outermost dimension. This continues until there is just one remaining DV candidate at one inner dimension. Figure 3.7 shows the MDV for the above example code, which exists among the DVs. It equals to $[2,2,0]$ corresponding to the ordered $i$-, $j$- and $k$- loop iterators respectively. We see that the MDV does not have the maximal dependency distance for dimension $j$ (which is 4). This is because the DV with length 4 in dimension $j$ has length 0 in the outer $i$ dimension.

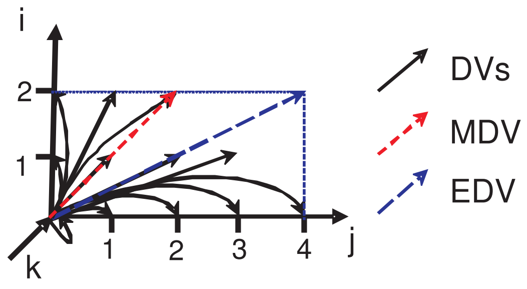


**Figure 3.7:** The MDV and EDV for the example code in Figure 3.4

The EDV refers to all the DVs in the worst case there is no execution order given for the dependency relations. It is calculated by projecting all the DVs at each iteration dimension and taking the maximal distance values. The EDV hence does not have to be one of the actual DVs. This is the case for the above example code as shown in Figure 3.7. Here the EDV equals to $[2,4,0]$.

At the iteration domain, the DVs define the lifetime window of each array element between its write reference and read reference. The MDV among all DVs hence defines the maximal lifetime window among all the array elements. Both the MDV and the EDV are used for the memory footprint requirement estimation as will be discussed in Chapter 4. The existing approaches to calculate the MDV first perform a dependency analysis, i.e., with tools like the OMEGA calculator [116] based on Pressburger formulas and Polylib library [92]. This is then followed by a selection of the MDV. For the EDV, extra projection operations of all DVs need to be performed at each loop dimension.

### 3.1.7 Dynamic Single Assignment Code

To enable the data dependency analysis, the input code must be written in single assignment form [142]. Any signal (scalar or array element) can be written (produced) only once throughout the code. For code that is not in this form, this is achievable for most cases by a proper array data flow analysis preprocessing [50, 51, 142]. Figure 3.8(a) gives an example of code which is not in a single assignment format. If necessary, it can be rewritten into this format as shown in Figure 3.8(b) . The rewriting of the code into single assignment in some cases adds an extra dimension to arrays resulting in an increased storage requirement. After optimizations, the subsequent in-place mapping will however remove this as shown in [43, 147]. In single assignment code, only value-based flow dependency exists. As will be discussed in later chapters, our estimation methodology can work for both single assignment code and non-single assignment code.

```
A = 1;                                  A[0] = 1
for (i=0; i<=7; i++)                    for (i=0; i<=8; i++)
  A = A + in[i];                          A[i] = A[i-1] + in[i];
         (a)                                     (b)
```

**Figure 3.8:** Code examples in (a) non-single assignment, (b) single assignment

### 3.1.8 Tools and Code Constraints

Many tools exist for converting the implicit representation of a polyhedron to its explicit representation and back. Example tools include CDD, PORTA, QHull and LRS. Some libraries such as PolyLib, polymake, polka and PPL also include support for performing other operations on these polyhedra, e.g., intersections and affine transformations. Other kind of libraries, such as Omega are also useful for dependency analysis. A more complete overview can be found in to [55, 145].

The geometrical model techniques used in this work are mainly restricted to static control programs with affine manifest indices, loop boundaries and conditions for modeling multi-dimensional data processing programs [141]. Programs not satisfying these requirement has to be modeled by taking worst-case assumptions, if possible. These limitations have not been severe since the programs to be optimized have usually satisfied these requirements. Due to the growing complexity of image and video processing algorithms, there now also exist other programs which can no longer be accurately modeled. This disables some potential optimizations or makes them less effective. More accurate models have been developed for dealing with non-affine indices, loop boundaries and conditions [117, 4, 103]. These techniques can also be used here.

## 3.2 Bounding Box Geometrical Model

In the polyhedron geometrical model, the polyhedron can have complex convex polytope shapes that makes the operations to be performed on them computationally complex. Typical operations used in this work are union or intersection of two polyhedra, and counting of the number of lattice points in a polyhedron. For speed reason, a simplified geometrical model, named the bounding box geometrical model, has hence been used instead. The bounding box geometrical model is a simplified version of the polyhedron geometrical model in which all the polyhedron representations are, if not already, approximated as bounding box. A bounding box iteration domain (denoted by $\overleftrightarrow{I}$) is a rectangular approximation of the original iteration domain which can have any convex shape. This approximation is also called orthogonalization in [79]. The bounding box domain can easily be defined by the lower bound ($L$) and upper bound ($U$) for each dimension as

$$\overleftrightarrow{I} = \{\vec{i} = [i_1, \ldots, i_m] \mid \bigwedge_{g=1}^{m} L_g \leq i_g \leq U_g \wedge [i_1, \ldots, i_m] \in \mathbb{Z}^m\} \tag{3.7}$$

in which $L_g$ and $U_g$ means the lower bound and the upper bound values at the $g$th loop dimension.

Figure 3.9 provides a code example of a two dimensional iteration domain

```
for (i=0; i<=4; i++)
  for (j=0; j<=4; j++)
    if (i<=j)
      A[i][j] = ...

        (a)
```

```
for (i=0; i<=4; i++)
  for (j=0; j<=4; j++)
    A[i][j] = ...

        (b)
```

**Figure 3.9:** Approximation of iteration domain (a) before , (b) after

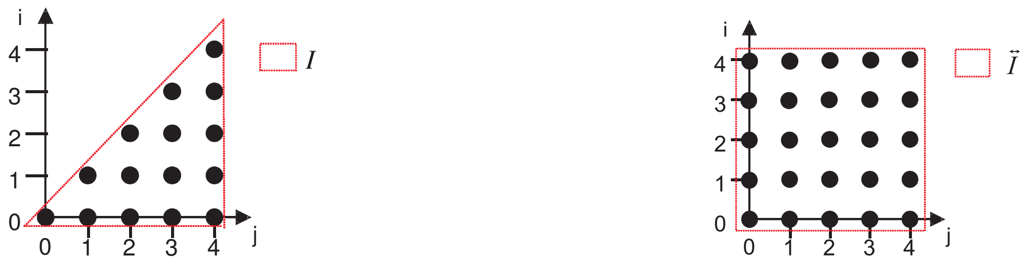


**Figure 3.10:** Approximation of iteration domain (a) before , (b) after

approximation. This is also illustrated in Figure 3.10. The bounding box iteration domain now is

$$\overleftrightarrow{I} = \{[i,j] | 0 \le i \le 4 \wedge 0 \le j \le 4 \wedge [i,j] \in \mathbb{Z}^2\}$$

The advantage of the bounding box geometrical model is that the operations performed on it is in general much simpler and faster than on the polyhedron geometrical model. For example, the counting of the lattice points of a bounding box can be performed by multiplying the distance of each dimension as defined by the lower and upper bounds of the bounding box in that dimension. This is much simpler and faster compared to counting the lattice points in a general polyhedron. Intersection and union operations on bounding boxes are also simple. In practice, the bounding box representation is an efficient and appropriate approximation since most convex polytopes in our targeted application domain are rectangular and have regular accesses, i.e., images, blocks, etc. The possible errors this approximation incurs are discussed in later chapters.

For the code example in Figure 3.4, the bounding box expression for statement $S1$ exactly represents the original polyhedron shape while for statement $S2$ it is approximated as

$$\overleftrightarrow{I}' = \{[i,j,k] | 0 \le i \le 4 \wedge 0 \le j \le 7 \wedge 0 \le k \le 2 \wedge [i,j,k] \in \mathbb{Z}^3\}$$

Given a bounding box iteration domain of a statement, the data domain for one array reference in that statement is also a bounding box since array indices are required to be affine functions of the surrounded loop iterators. For each array reference, the associated index expression maps the iterators of the statement to the accessed index for that array reference. The data domain of one array reference is now calculated by

$$\begin{aligned}\overleftrightarrow{D} = E(\overleftrightarrow{I}) =& \{[d_1,\ldots,d_n] \mid \bigwedge_{h=1}^{n} d_h = \sum_{g=1}^{m} c_g^h i_g + c_{m+1}^h \\ & \wedge [i_1,i_2,\ldots,i_m] \in \overleftrightarrow{I} \wedge [d_1,d_2,\ldots,d_n] \in \mathbb{Z}^n\} \\ =& \{[d_1,\ldots,d_n] \mid \bigwedge_{h=1}^{n} L_h \le d_h \le U_h \wedge [i_1,i_2,\ldots,i_m] \in \overleftrightarrow{I} \wedge [d_1,d_2,\ldots,d_n] \in \mathbb{Z}^n\},\end{aligned} \tag{3.8}$$

where the comparison is applied componentwise and $L_h$ and $U_h$ are the lower bound and upper bound values of the $h$th array dimension. For the example code in Figure 3.4, the bounding box data domain for array $A$ referenced in statement $S2$ is:

$$\begin{aligned}\overleftrightarrow{D}' = E'(\overleftrightarrow{I}') =& \{[d_1,d_2,d_3] \mid d_1 = i-k \wedge d_2 = j-i+k \wedge d_3 = k \\ & \wedge [i,j,k] \in \overleftrightarrow{I}_{S2} \wedge [d_1,d_2,d_3] \in \mathbb{Z}^3\} \\ =& \{[d_1,d_2,d_3] \mid 0 \le d_1 \le 4 \wedge 0 \le d_2 \le 7 \wedge 0 \le d_2 \le 2 \\ & \wedge [i,j,k] \in \overleftrightarrow{I}_{S2} \wedge [d_1,d_2,d_3] \in \mathbb{Z}^3\}\end{aligned}$$

Note $c_{m+1}^h$ in this code is zero at all dimensions of the index expression. $\overleftrightarrow{D}$ can also be an approximation of the original data domain.

The MDV/EDV can also be calculated based on the bounding box geometrical model. Due to the approximation of the polyhedron representation, pseudo dependencies and hence pseudo dependency vectors might be added. They may have effect on the calculation of MDV/EDV. In general, pseudo dependencies can occur for non-affine code at where array index expression is not affine function of surrounding loop dimensions. It only have effect when the pseudo dependency vector is larger than the actual MDV/EDV. This however does not occur so often for our target applications as most applications have regular data access with an affine index expression of the arrays. It is hence acceptable to ignore it for our estimation purpose. Figure 3.11 shows all the DVs for the bounding box representation of the code in Figure 3.4. The vectors in the dashed lines are the pseudo

DVs which do not exist in the polyhedron geometrical model representation. For this example, these pseudo DVs does not effect the MDV/EDV calculation. Hereafter, when we say DVs in the bounding box GM, it also include the pseudo DVs.

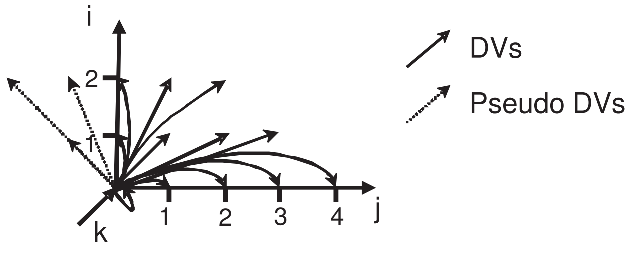


**Figure 3.11:** Dependency vectors in bounding box geometrical model

## 3.3 Loop Transformations in the Geometrical Model

The objective of a loop transformation is to change the execution order of the loops to exploit the architectural features of the hardware, while still computing the same result. Loop transformations are mostly used in program optimization for parallelism and performance but also for low power embedded system design as reviewed previously in Chapter 2. Loop transformations are typically divided into two classes: affine loop transformations and non-affine loop transformations. Typical affine transformations are loop interchange, loop reverse, loop skewing, loop fusion (also called merging), and loop shifting (also called bumping). Important non-affine transformations include strip mining, loop tiling and loop fission (here only loop body splitting is considered). Affine loop transformations can further be classified as linear loop transformations and translations [35, 144]. Loop interchange, loop reverse, and loop skewing are linear transformations which can be represented as unimodular transformations, while loop fusion and loop shifting are translations.

A loop transformation can be viewed as a reorganization of the loop iterations. It is characterized by a transformation of the iteration domain. In this dissertation we are interested in the unimodular transformations [15, 153, 46, 84]. Unimodular transformations can be represented as matrix operation with a unimodular matrix[‡] which determines the transformed loop and dependencies. The unimodularity of the transformation matrix ensures that the mapping is one to one and with unit stride. Unimodular transformations are linear transformations on an iteration domain that gives a unified view of all possible sequence of most of the existing transformations. In this dissertation, for linear transformation, we mean the unimodular linear transformations. Although the approach itself does not solve the problem of deriving the optimal transformation(s) for a given loop nest or for multiple loop nests in a program code, it allows to automatically perform a set of transformations incrementally. This makes loop transformation exploration feasible, even with a large set of transformation possibilities.

In this Section, we will first introduce the general technique of how linear and also affine loop transformations are performed as unimodular matrix operation. This is the fundament for the incremental HMSE methodology as presented later in Chapter 7. The various affine loop transformations as well as other loop transformations are studied afterwards.

## 3.3.1 Fundamental for Unimodular Loop Transformations

A unimodular loop transformation is defined by an $m$ by $m$ unimodular integer matrix $A$. $A$ maps the iterator $\vec{i} = (i_1, \ldots, i_m)^T$ into a new iteration vector $\vec{i}^* = (i_1^*, \ldots, i_m^*)^T$,

$$\vec{i}^* = A \cdot \vec{i} \tag{3.9}$$

and maps each dependency (denoted by $P$) into a new dependency (denoted by $P'$)

$$P^* = A \cdot P \tag{3.10}$$

The transformation is legal iff all dependency in $\underset{\cdot}{P}$ are lexicographically positive. The positivity of transformed dependencies is a strong condition, and ensures a valid sequential execution of the transformed statement in a loop. Determining the new loop bounds of the transformed iteration domain is nontrivial in the polyhedron geometrical model. For the bounding box geometrical model, it is much simpler since the new loop bounds of the bounding box iteration domain is still a bounding box.

When a statement in the loop is transformed, the order in which the array elements are referenced in the statement may also be transformed. The transformed index expression of the arrays referenced in the statement is calculated as

$$E^* = E \cdot A^{-1} \tag{3.11}$$

[‡]A matrix with determinant $\pm 1$.

The transformed data domain for one array reference in the statement is

$$D^* = E^* \cdot I^* = E \cdot A^{-1} \cdot A \cdot I = E \cdot I = D \tag{3.12}$$

This shows that, for one array reference, the transformed data domain is always equivalent with the original. This corresponds to the prerequisite that transformations do not changed the functionality of the original program code.

An affine loop transformation can be represented as a linear transformation with unimodular integer matrix $A$ and a translation vector $a$ as

$$\vec{i^*} = A \cdot \vec{i} + \vec{a} \tag{3.13}$$

The translation vector always refers to the constant part. Translation includes loop fusion and loop shifting which only have effect on the translation part.

Homogeneous coordinates [59, 34] are used in order to transform affine problems into linear problems, such that also translations can be represented as matrix multiplications. Originally, homogenous coordinates were used to prove certain problems in $N$-space. These problems have a corresponding problem in ($N$+1)-space, where a solution can be found more easily. This solution is then projected back to $N$-space. A point $(x_1,x_2,...,x_n)^T$ can be represented in homogeneous coordinates by the point $(wx_1,wx_2,...,wx_n)^{T+1}$, where $w$ is called the scale factor. It is also easy to project a point in homogeneous coordinates back to affine coordinates. In this case, $w$=1 is always chosen.

Equation 3.13 can hence be represented with homogeneous coordinates by adding an extra column corresponding to the translation vector and by adding an extra row as

$$\begin{bmatrix} \vec{i^*} \\ 1 \end{bmatrix} = \begin{bmatrix} A & \vec{a} \\ \vec{0}^A & 1 \end{bmatrix} \begin{bmatrix} \vec{i} \\ 1 \end{bmatrix} \tag{3.14}$$

this is simply written as

$$\tilde{\vec{i^*}} = \tilde{A} \cdot \tilde{\vec{i}} \tag{3.15}$$

$\vec{0}^A$ is a vector of as many zeros as there are columns in $A$. The index expression of an array reference can also be represented by homogeneous coordinates, since an index expression is in fact a mapping, namely from $m$-dimensional loop coordinates to $n$-dimensional array subscripts. We will denote the full index expression by $\tilde{E}$, and the linear part by $E$.

The transformed index expression of the array reference in a transformed statement is hence equal to

$$\tilde{E}^* = \tilde{E} \cdot \tilde{A}^{-1} \tag{3.16}$$

The main benefit of this homogeneous coordinate notation is that the composition of a series of loop transformations amounts to multiplication of the corresponding homogeneous transformation matrices. When a sequence of affine loop transformations are performed, the transformations can be represented as

$$\tilde{i}^* = \tilde{A}_N \cdot \tilde{A}_{N-1} \cdot \ldots \cdot \tilde{A}_1 \cdot \tilde{i} \tag{3.17}$$

in which $\tilde{A}_1$ to $\tilde{A}_N$ are the sequence of transformation matrices, corresponding to a sequence of N incremental loop transformations. The new iteration domain of the statement and the index expression of array references in the transformed statement also needs to be incrementally updated based on Equation 3.17 and Equation 3.16, respectively.

Loop interchange, loop reverse, and loop skewing on perfectly nested loop structure can all be captured by such unimodular transformation matrices. They are illustrated based on the example code in Figure 3.12(a). In order to allow to perform loop transformation globally among multiple loop nests, the program code first needs to be placed in a common iteration space. For this example, only one extra time dimension needs to be added as shown in Figure 3.12(b).

```
for (i=0; i<=4; i++)
  for (j=0; j<=5; j++) {
S1:    A[i][j] = ...;
       if (i>=1 and j>=2)
S2:       ... = f(A[i-1][j-2]);
  }

            (a)
```

```
for (i=0; i<=4; i++)
  for (j=0; j<=5; j++)
    for (t=0; t<=1; t++) {
      if (t==0)
S1:      A[i][j] = ...;
      if (i>=1 and j>=2 and t==1)
S2:      ... = f(A[i-1][j-2]);
    }

            (b)
```

**Figure 3.12:** Loop transformation code example (a) original (b) in the common iteration space

#### 3.3.1.1 Loop Interchange

Loop interchange changes the position of two loops in a loop nest. Any two loops in an $N$ dimensional loop nest can be interchanged as long as it does not introduce negative dependencies. Loop interchange is one of the most powerful transformations and can improve performance and access regularity. An example of a loop interchange transformation is illustrated in Figure 3.13 which interchange the outer $i$ and $j$ dimensions of the code in Figure 3.12(b). Note that, in this dissertation, the original iterator names are not changed but their corresponding loop bound values and their use in the index expressions are changed accordingly. The extra pseudo time dimension has also been inserted in order to uniquely identify the statements in the common iteration space.

```
for (j=0; j<=5; j++)
  for (i=0; i<=4; i++)
    for (t=0; t<=1; t++) {
      if (t==0)
S1:     A[j][i] = ...;
      if (j>=1 and i>=2 and t==1)
S2:     ... = f(A[j-1][i-2]);
    }
```

**Figure 3.13:** Loop interchange example

With loop interchange, the two loop dimensions $i$ and $j$ are interchanged for both statements. The transformation matrix in the homogeneous coordinate format for the two statements (in this case they are identical) is

$$\tilde{A}_{S_1} = \tilde{A}_{S_2} = \begin{bmatrix} 0 & \mathbf{1} & 0 & 0 \\ \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In the above transformation matrix, each row (except the bottom row) corresponds to a dimension ($i, j, t$ in that order) in the loop nest. The matrix identifies how each loop dimension is transformed including both linear loop transformation and translation.

#### 3.3.1.2 Loop Reversal

When loop reversal is performed, the iteration order of one loop dimension is reversed. The loop bounds are negated and the lower and the upper bounds are switched. The loop index in the array index expression is negated for the array(s) accessed in the body of the loop.

The unimodular transformation matrix

$$\tilde{A}_{S_1} = \tilde{A}_{S_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -\mathbf{1} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

will for instance transform the code of Figure 3.12(b) into the new code shown in Figure 3.14.

```
for (i=0; i<=4; i++)
   for (j=-5; j<=0; j++)
      for (t=0; t<=1; t++) {
         if (t==0)
S1:         A[i][-j] = ...;
         if (i>=1 and j<=-2 and t==1)
S2:         ... = f(A[i-1][-j-2]);
      }
```

**Figure 3.14:** loop reversal example

#### 3.3.1.3 Loop Skewing

Loop skewing is an enabling transformation that is primarily useful in combination with loop interchange. Skewing was invented to handle wavefront computations, in which the array propagate like a wave across the iteration domain.

```
for (i=0; i<=4; i++)
  for (j=i; j<=i+5; j++)
    for (t=0; t<=1; t++) {
      if (t==0)
S1:     A[i][j-i] = ...;
      if (i>=1 and j<=-2 and t==1)
S2:      ... = f(A[i-1][j-i-2]);
    }
```

**Figure 3.15:** Loop skewing example

Figure 3.15 shows the typical wavefront array accesses. Skewing is performed by adding the outer loop index multiplied by a skew factor to the bounds of the inner iteration variable, and then subtracting the same quantity from every use of the inner iteration variable inside the loop. The unimodular transformation matrix

$$\tilde{A}_{S_1} = \tilde{A}_{S_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \mathbf{1} & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

will for instance transform the code in Figure 3.12(b) to the new code in Figure 3.15.

#### 3.3.1.4 Loop Fusion

Loop fusion merges together two loops in one new loop. The bodies of the two original loops are used together as the new body. In case of non equal bounds of the two loops, one new loop can appear (in that case it is called body-merge-if loop transformation). This transformation can add conditions to the original loop nest bodies so that they are executed only for certain values. An alternative, instead of keeping it in one loop nest, three new loop nests can appear (in that case it is also called body-merge loop transformation). The middle new loop nest will contain the loop bounds for which both bodies have to be executed, the first and last ones will contain the just one body, executed for the still missing parts in the iteration space. In this dissertation the first representation approach is used. This actually does not affect the result of our estimation.

For instance, the unimodular transformation matrix

$$\tilde{A}_{S_2} = \begin{bmatrix} 1 & 0 & 0 & 0 & \mathbf{-1} \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

will transform statement $S_2$ in the code of Figure 3.16(a) to the code in Figure 3.16(b) which is equivalent to the code in Figure 3.12(b) but with the extra time dimension $t_0$ added at the outermost dimension. The extra time dimension $t_0$ is needed for the code of Figure 3.16(a) in order to uniquely identify the execution ordering of each iteration in the common iteration space. During the fusion, the statement $S_1$ is not changed.

In the above transformation matrix, each row (except the bottom row) corresponds to a dimension ($t_0, i, j, t$ in the order) in the loop nest. As shown, the last element in each row is the translation part, corresponding to effect of the loop fusion in this case. In this example, statement $S_2$ has been shifted at the two pseudo time dimensions, therefore merging the two loop nests together.

#### 3.3.1.5 Loop Shifting

Loop shifting, also called bumping, is a specific case of affine loop transformation. It adjusts the lower and upper bounds of a loop by adding a constant integer. Loop shifting is usually introduced when direct fusion is not legal because data dependencies in the fused loop exist for which statements of one loop iteration depend on results from statements of the following loop iterations.

For instance, the unimodular transformation matrix

```
for (t0=0;t0<=1; t0++) {
  for (i=0; i<=4; i++)
    for (j=0; j<=5; j++)
      for (t=0; t<=0; t++)
        if (t0==0 and t==0)
S1:       A[i][j] = ...;
  for (i=0; i<=4; i++)
    for (j=0; j<=5; j++)
      for (t=0; t<=0; t++)
        if (t0==1 and i>=1 and j>=2)
S2:       ... = f(A[i-1][j-2]);
}
```

(a)

```
for (t0=0;t0<=0; t0++)
  for (i=0; i<=4; i++)
    for (j=0; j<=5; j++)
      for (t=0; t<=1; t++) {
        if (t0==0 and t==0)
S1:       A[i][j] = ...;
        if (i>=1 and j>=2 and t==1)
S2:       ... = f(A[i-1][j-2]);
      }
```

(b)

**Figure 3.16:** Loop fusion example (a) before (b) after

$$\tilde{A}_{S_1} = \tilde{A}_{S_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \mathbf{3} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

will transform the code of Figure 3.12(b) into the new code shown in Figure 3.17.

```
    for (i=0; i<=4; i++)
      for (j=3; j<=8; j++)
        for (t=0; t<=1; t++) {
          if (t==0)
S1:          A[i][j-3] = ...;
          if (i>=1 and j>=2 and t==1)
S2:          ... = f(A[i-1][j-2-3]);
        }
```

**Figure 3.17:** Loop reversal example

Above we have illustrate how affine loop transformations are performed based on matrix manipulation. The unimodular matrix identifies how each loop dimension is transformed including both linear loop transformation and translation. If the diagonal element of one row is different from 1 and any of the other elements on that row are non-zero, the corresponding loop dimension is transformed for that statement. The last element in each row corresponds to the translation part for that dimension. If it has a value other than zero, it corresponds to loop fusion and/or loop shifting. Based on the evaluation of the elements in the matrix, we can identify the loop dimensions that are transformed. This is fundamental for our incremental HMSE methodology as will be discussed in Chapter 7. Note that it is easier to identify which loop dimensions that have been transformed than to identify what kind of loop transformations that have been performed based on the unimodular transformation matrix. This is especially true when the unimodular transformation matrix represents multiple loop transformations at a time.

Other kind of loop transformations include loop extend/reduce which goes to larger or smaller iterator range by adding manifest conditions, loop index split which produces two loops from iterator range and loop body split which produces two loops from statements in loop body. Further information can be found in [14, 17, 85].

## 3.3.2 Non-affine Loop Transformations

Non-affine loop transformations cannot directly be represented as unimodular matrix operations. Strip mining and loop tiling are the most useful among the non-affine loop transformations. Further details about non-affine loop transformations can be found [17, 85].

### 3.3.2.1 Strip Mining

Strip mining is used to adjust the granularity of an operation, especially for parallelizable operation. An example is shown in Figure 3.18.

```
for (i=0; i<=5; i++)              for (i1=0; i1<=1; i1+=2)
  A[i] = ...;                         for (i2=0; i2<=2; i2++)
                                        A[i1*3+i2] = ...;

        (a)                                    (b)
```

**Figure 3.18:** Strip mining example (a) before (b) after

### 3.3.2.2 Loop Tiling

Loop tiling is the multi-dimensional generalization of strip-mining. Tiling (also called blocking) is primarily used to improve cache reuse by dividing an iteration domain into tiles and transforming the loop nest to iterate over them. However, it can also be used to improve processor, register, translation look-aside buffer (TLB), or page locality.

```
for (i=0; i<=n; i++)
  for (j=0; j<=n; j++)
    A[i][j] = ...;
```

(a)

```
for (t_i=0; t_i<=n; t_i+=64)
  for (t_j=0; t_j<=n; t_j+=64)
    for (i=t_i; i<=min(t_i+63, n); i++)
      for (j=t_j; j<=min(t_j+63,n); j++)
        A[i][j] = ...;
```

(b)

**Figure 3.19:** Loop tiling example (a) before (b) after

There also exists other non affine loop transformations, such as loop unrolling which can reduce loop overhead by reducing the number of iterations and replicating the body of the loop.

# Chapter 4

# Fast Intra-Array Memory Footprint Estimation

An essential step during the high level design space exploration is to perform memory footprint estimation in order to give an early feedback on the memory requirement. As there usually exist a large number of design possibilities at the high level, e.g., to perform loop transformation, memory footprint estimation is required to be performed frequently. The required estimation time hence becomes critical, in addition to a sufficient level of estimation accuracy.

As discussed in Chapter 2, current techniques for memory footprint estimation are too slow and/or assume a single memory. In [64] an algorithm has been introduced for steering loop transformations, specifically loop fusion and loop shifting, for memory minimization based on an integrated memory footprint estimation technique. This technique is however slow and not scalable for large examples. This chapter presents a fast memory footprint estimation technique that is usable in a practical context with a distributed scratchpad-based memory organization. In this dissertation, the execution ordering of the application is assumed to be already fixed. The focus is on fast estimation of memory footprint requirements for arrays. Our estimation is performed in two phases:

1. intra-array memory footprint estimation

2. inter-array memory footprint estimation

During intra-array memory footprint estimation, the memory location reuse is exploited between elements of each individual array when their lifetimes are not overlapping. During inter-array memory footprint estimation, the reuse possibilities are exploited among multiple arrays. This chapter first presents how to perform a novel fast intra-array memory footprint estimation. The inter-array memory footprint estimation technique will be presented in Chapter 5.

This chapter is organized as follows. Section 4.1 presents how to perform a fast intra-array memory footprint estimation based on the MDV. Due to the fact that existing approaches to calculate the MDV is slow, two approaches on how to calculate the MDV are presented in Section 4.2: a general approach based on an ILP formulation and a novel vertexes approach when iteration domains are approximated as bounding boxes. Section 4.3 discusses errors which may occur within the estimation approaches. Finally a summary is drawn in Section 4.4.

## 4.1 Intra-array Memory Footprint Estimation

For an array, a straightforward way of estimating its memory footprint requirement is to multiply the size of each array dimension. This will normally result in a huge overestimate however, since not all array elements are alive at the same time and the elements with non-overlapping lifetime can reuse the same memory locations. One element's lifetime is defined from the moment it is written (produced) and until it is read for the last time (consumed). It can be depicted as a lifetime window in the iteration domain as illustrated later, which is constrained by the dependency relation for this array element's write and read references. In this dissertation, the array index expression is assumed to be affine function of the surrounding loop iterators. The estimation techniques presented in this chapter consider that the array index expression can be optimized later on as a linear function of the surrounding loop iterators, simply called linearized array. [43, 134, 37] have presented different approaches for memory optimization with linearized address generation as described in Chapter 2.

The intra-array memory footprint estimation is performed on the geometrical model based on data dependency analysis. Following let us first look at how to perform the estimation for one array when it just has one write reference and one read reference and there exist single pair dependencies between them. This approach is then extended to the general case where multiple write references and read references can exist for one array.

### 4.1.1 Estimation for Single Pair Dependencies

For single pair dependencies, all the array elements which are written within one array element's lifetime window are also simultaneously alive since array's index expression is affine function of the surrounded loop iterators. The maximal lifetime window among the lifetime windows of all array elements in turn defines the maximal number of simultaneously alive array elements. It hence constrains the size requirement for this dependencies pair, and also the array's memory footprint requirement.

The problem of detecting the memory footprint requirement is hence equivalent to detecting the maximal lifetime window of all array elements and of identifying the number of array elements written within this period. At the iteration domain, this problem is equivalent to finding the dependency that defines the array's maximal lifetime and to count the number of iteration nodes at where array elements are written within the corresponding lifetime window. Proposition 4-1 gives a definition of this dependency.

**Proposition 4-1:** In the iteration domain, the maximal dependency vector (MDV) defines the maximal lifetime window over all array element accesses for one pair dependencies.

The above proposition is always valid as long as the array index expressions are affine and manifest functions of the surrounding iterators. The MDV always contains the largest number of iteration nodes at where an array is being written before it is read. The MDV concpet has been presented already in Chapter 3. The problem is now how to calculate the MDV and how to perform the memory footprint estimation in the iteration domain. The MDV calculation will be presented in the Section 4.2. The size estimation is done by counting the number of iteration nodes constrained by the MDV at where array elements are written.

#### 4.1.1.1 Illustration on an Example

Let us first illustrate how to estimate the memory footprint on the example code in Figure 3.4. The memory footprint estimation is performed by counting the number of iteration nodes constrained by the MDV. For this code, there are one write reference and one read reference for array $A$ and data dependencies exist between them. Assuming the array is not used anywhere else, the array read reference consumes the data. Assume $i$, $j$ and $k$ refer to the values of the loop variables at the time the write is performed and $i'$, $j'$ and $k'$ refer to the values of the loop variables at the time the read is performed. The loop iterators defines the two iteration domains as introduced in 3.1.4.2, at where array is written and read.

$$I = \{[i,j,k] | 0 \leq i \leq 4 \wedge 0 \leq j \leq 7 \wedge 0 \leq k \leq 2 \wedge [i,j,k[ \in \mathbb{Z}^3\}$$
$$I' = \{[i',j',k'] | 0 \leq i' \leq 4 \wedge 0 \leq j' \leq 7 \wedge 0 \leq k' \leq 2 \wedge i' >= k' \wedge j' >= i' - k' \wedge [i',j',k'[ \in \mathbb{Z}^3\}$$

The dependencies between these two array references are illustrated previously in Figure 3.5 and the dependency vectors of the original code are drawn in Figure 3.6. In this dissertation, the memory footprint estimation is performed on the bounding box iteration domains. This is because the counting of iteration nodes in a bounding box iteration domain is simple and fast as discussed in Section 3.2. For this example, the bounding box expression of the iteration domain for the array write statement (denoted by $\overleftrightarrow{I}$) exactly represents the original polytope shape, while the bounding box of the iteration domain for the array read statement (denoted by $\overleftrightarrow{I}'$) is approximation of the original given by

$$\overleftrightarrow{I}' = \{[i',j',k'] | 0 \leq i' \leq 4 \wedge 0 \leq j' \leq 7 \wedge 0 \leq k' \leq 2 \wedge [i',j',k'[ \in \mathbb{Z}^3\}$$

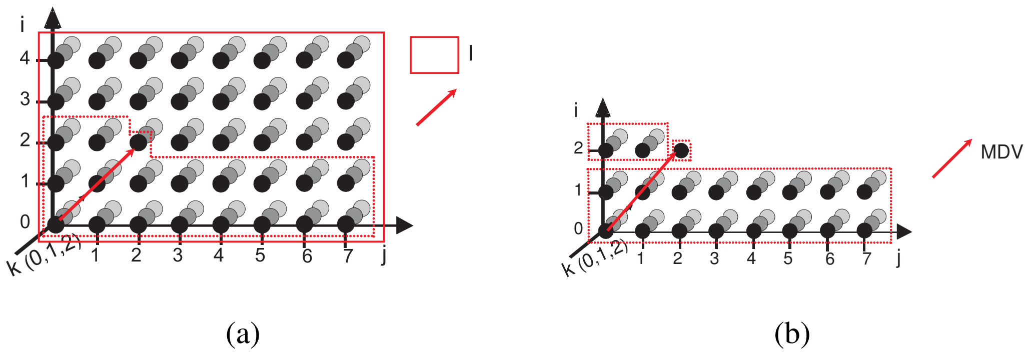


**Figure 4.1:** (a) The maximal lifetime window constrained by MDV, (b) memory footprint estimation on the MDV

As mentioned, the MDV defines the maximal lifetime window among all array element accesses. In the bounding box iteration domain, the MDV can always be drawn from the origin node to the node strided by the MDV, as shown in Figure 4.1(a) for this example. In this case, there is a one-to-one mapping between the data elements and iteration nodes, i.e., each iteration node writes one data element.

The size requirement can then simply be calculated by counting the number of iteration nodes within the array write iteration domain constrained by the MDV. An efficient counting approach is to count and sum up the iteration nodes constrained by the dependency distance of the MDV at each loop dimension from the outermost dimension to the innermost dimension.

When there is a one-to-one mapping between the data elements and iteration nodes, the counting can be formulated as

$$size = \sum_{g=1}^{m-1} \left( MDV[g] \cdot \prod_{l=g+1}^{m} (\mid U_l - L_l \mid +1) \right) + MDV[m] + 1 \tag{4.1}$$

in which $m$ is the number of loop dimensions of the array write iteration domain. In this formula, the counting is summed over each loop dimension starting from the outermost dimension. Let us illustrate how it works on Figure 4.1(b) at where the MDV equals to $[2,2,0]$. The distance of MDV at the outermost $i$-dimension is first multiplied with the upper and lower bound distance difference, $\mid U_l - L_l \mid +1$, of all inner dimensions. The resulting number of iteration nodes is 48 as the distance at the outermost $i$-dimension is 2 and the bound ranges of $j$ and $k$ are 8 and 3 respectively. This procedure is repeated at the $j$ dimension where the number of nodes is 6. The innermost dimension, $k$, is treated separately by the second part of the formula, since it is not going to be multiplied with anything. In this case, the MDV distance of the $k$-dimension is zero. The last constant one is added in order to count the iteration node reached by the MDV at where one array element is written before the first read starts. The size requirement for this example is hence 55.

If there is not a one-to-one mapping, e.g., data are accessed at every $N$ iterations, the end size requirement should be divided by the interval $N$ as the array index expressions are affine functions of the surrounding loop dimensions. This can easily be handled as a postprocessing.

Counting the number of iteration nodes in the iteration domain based on Equation 4.1 is very simple and fast. In contrast, counting integral points in general polyhedra is more computationally expensive [18].

#### 4.1.1.2 General Estimation Algorithm

Above we have illustrate how the memory footprint estimation is performed on an example. Equation 4.1 is always valid when the two iteration domains are overlapping in each loop dimension. However, it would give incorrect results

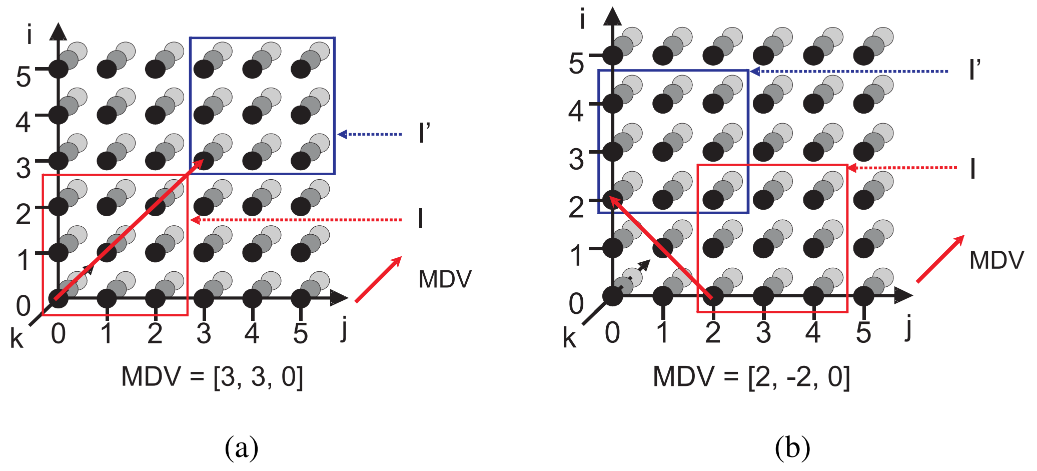


**Figure 4.2:** Other MDV relations between array write and read iteration domains

when the two iteration domains are not overlapping or the MDV is negative in at least one dimension. Figure 4.2 shows these kind of cases assuming the execution ordering is fixed with the *i*-dimension outermost and *k*-dimension innermost. In Figure 4.2(a) the array write iteration domain and the array read iteration domain are not overlapping at all. In Figure 4.2(b) the two iteration domains are partially overlapping while the dependency distance of the MDV at any loop dimension is negative but it is still legal.

In Figure 4.2(a), the dependency distance at the i-dimension is 3, which is larger than the bound distance of the array write iteration domain at the same dimension. Using the upper and low bound distance difference directly would result in over estimation during the counting of nodes at that loop dimension. This can easily be handled by adding a comparison of the dependency distance and the bound difference (calculated as $\mid UB_g - LB_g \mid +1$) at the *g*th loop dimension. If the dependency distance has a value larger than the bound difference, the number of nodes should be calculated using $(\mid UB_g - LB_g \mid +1)$ instead of the $MDV[g]$ value. Since there are no new nodes constrained at inner dimensions of the array write iteration domain, the counting procedure also stops at this dimension. When the MDV has negative value for any dimension, e.g., -2 at the *j*-dimension in Figure 4.2(b), direct using the bounds difference would also result in incorrect counting. The counting procedure should stop without counting even at this dimension.

Figure 4.3 shows the pseudo-code of our algorithm of memory footprint estimation for single pair dependencies. All the different cases are taken into account.

#### 4.1.1.3 Comparison with the EDV Approach

In contrast to the MDV based approach, a similar estimation technique is presented by Kjeldsberg et al. in [79, 81, 121] based on the EDV. The EDV is required by Kjeldsberg et al. since the execution ordering in their case can be partially fixed. When they calculate the lifetime window constrained by the dependency vectors, they have to take into account the worst case w.r.t. all the possible execution ordering. The EDV is hence used since it reflects the worst case lifetime window for any execution ordering. For the code in Figure 3.4, the EDV is equal to $[2,4,0]$ as shown in Figure 3.7. This vector is not among the actual DVs. It is a result of the projection of all DVs at each dimension and is caused by the two DVs $[0,4,0]$ and $[2,2,0]$ in this case.

When the execution ordering is unfixed or partially fixed, their technique results in upper and lower bounds instead of the exact size requirement. When the execution order is fixed as it is in this work, the upper and lower bounds converge to a single value. Their estimation result is equal to what is achieved with this algorithm but based on the EDV instead of the MDV. When using the EDV instead of the MDV, the memory footprint requirement for the above example turns out to be 61 instead of 55. This demonstrates that the EDV based approach used by Kjeldsberg et al. can result in over-estimate compared to the MDV since they do not take all available ordering information into account even when the execution ordering is fully fixed.

However, they have to use the EDV when the execution ordering can be partially fixed. When the execution ordering is partially unfixed, it would be possible for them to calculate the largest dependency distance vector by taking into account the partially fixed execution ordering. It could, however, be computationally expensive to do so as it needs to be performed iteratively. For most of our targeted applications, the estimation difference between using MDV and EDV is small. If computation time is the highest priority objective, it is hence sufficient for them to just use the EDV. But if estimation accuracy has the highest priority, the MDV approach is the best suited. Both approaches are therefore 2 Pareto points of estimation accuracy versus computation time in the estimation “design space”.

```
0:   size = 0
     b_continue = 1
     g = 1
     nodes = 0
10:  while (g <= m-1) and b_continue
11:      nodes = 0 //count nodes at one dimension
12:      if (MDV[g] >= 0)
13:          if (MDV[g] <= abs($U_g$ - $L_g$) +1 )
14:              nodes = MDV[g]
15:          else:
16:              nodes = abs($U_g$ - $L_g$) +1
17:          for (l=g+1; l <= m, l++)
18:              nodes *= abs($U_l$ - $L_l$) +1
19:          if (MDV[g] > abs($U_l$ - $L_l$) +1 )
20:              b_continue = 0
22:      else
23:          b_continue = 0
24:      size += nodes
25:      g++;
26:  if (b_continue) //count the nodes at the innermost dimension and constant 1
27:      if (0 <= MDV[m] <= abs($U_m$ - $L_m$) +1 )
28:          size += MDV[m] + 1
29:      elsif (MDV[m] > abs($U_m$ - $L_m$) +1 )
30:          size += abs($U_m$ - $L_m$) +1
31:  return size
```

***Figure 4.3:*** *Pseudo-code of memory footprint estimation algorithm for single pair dependencies*

### 4.1.2 Estimation for Multiple Pair Dependencies

So far has presented the memory footprint estimation algorithm for single pair dependencies. That is, for dependencies between one array write reference and one array read reference.

In general, multiple write and read references of the same array may exist in the code. Flow dependencies may exist between every write and read pair, if they access the same elements. The maximal lifetime window now can not be simply derived among the multiple pair dependencies due to their various relation. This makes it complex to achieve accurate estimation. In order to achieve a fast estimation, this work hence chooses a simple approach. If two write-read pairs access non-overlapping data, their sizes can be computed individually and summed. If on the other hand they overlap, the overlapping data needs to be stored only once. In this case, only the maximum of the two sizes needs to be stored. Basic set analysis [9] splits dependencies in this way. It needs to be performed only once, since behavior-preserving transformations cannot change the dependency relations.

In the current version of tool implementation supporting the technique described above, we do not go to basic set analysis but simply take the maximum of the two sizes if the dependencies are overlapping. This however could result in over-estimations or under-estimations, as various dependency relatons can happen between the multiple references. Our estimation does not take them into account. Further analysis is necessary for improving the estimate accuracy. A straight forward way is to perform the basic set analysis before our estimation is performed. This is, however, left for future work.

In the non-single assignment application code, one array element can be written and read multiple times and there also exist other dependencies beside the data flow dependency. In such a case, the EDV/MDV is calculated by taking the worst case. That is, it is calculated by starting from the node at where one array element is written for the first time till the node at where it is read for the last time (consumed). As it can be already written multiple times in between, the last read only consumes what has been written most recently. The actual MDV/EDV can hence be smaller than the calculated one. Preprocessing the index expressions into dynamic single assignment (DSA) form [142] avoids this problem. It has to be performed only once, giving a quite acceptable overhead even in a system design space exploration context. Indeed, once the initial code is in DSA form, also the code transformations become simpler and they will maintain the DSA form.

## 4.2 MDV/EDV Calculation

The previous section presents the technique on how to estimate the array size requirement for individual arrays based on the MDV or EDV. As shown, Equation 4.1 and its extension algorithm in Figure 4.3 are very simple and extremely fast, independent to the complexity of the loop dimensions and the array index expressions. This is critical for achieving fast estimation. To reach the goal, it is also critical to perform a fast MDV calculation.

Current techniques on how to calculate the MDV and also the EDV is very computationally expensive. They are based on a dependency analysis, i.e., a calculation of dependency vectors as introduced in Chapter 3. Different techniques has been presented on how to calculate dependency vectors by using integer linear programming [123, 125, 146, 93] and Fourier-Motzkin variable elimination [116, 96]. The EDV is calculated by further performing a projection of all dependency vectors at each loop dimensions and taking the maximal value. It is well-known that both the integer linear programming algorithm and Fourier-Motzkin variable elimination algorithm are computationally expensive. To perform a projection of all dependency vectors at each loop dimensions is also non-trivial.

In this section two techniques on how to calculate the MDV is introduced: the general ILP formulation and the vertexes approach on the bounding box geometrical model. Both of them are again 2 Pareto points in the design space of estimation accuracy versus computation time. These techniques can also be used for a fast calculation of the EDV which could be useful for others. Since the MDV calculation is equivalent to calculating the EDV with some extra constraints added during computation, how to calculate the EDV is hence presented firstly. This is followed by the MDV calculation.

### 4.2.1 The ILP Approach for MDV/EDV Calculation

This section presents the ILP formulation [123, 125] of calculating the EDV/MDV between single pair dependencies on the polyhedron geometrical model. Their iteration domains ($I$ and $I'$) consist of a set of inequality constraints, as expressed in Equation 3.1. When data flow dependencies exist, the index value of the array write and read references ($E$ and $E'$) must be equal at each of the array dimensions. This is because, when dependencies exist between these two array references, both references access the same data element(s). Equation 3.4 hence results in a set of equalities:

$$E'_h = E_h \mid 1 \leq h \leq n \tag{4.2}$$

in addition to the set of inequality constraints for the iteration domains. In the above expression, $n$ is the number of array dimensions. For the code in Figure 3.4, the set of equalities are $i = i' - k'$, $j = j' - i' + k'$ and $k = k'$ for the three array dimensions. For this example, the numbers of array dimensions and loop dimensions are the same. This does not necessarily have to be the case.

One way to find the EDV is to calculate all the DVs, or the corresponding dependency distance polytope, with existing dependency analysis techniques. This is then followed by the projection operation at each loop dimension, keeping the maximal projected distance value at each dimension. In this dissertation, the EDV is directly calculated by solving ILP problems:

$$MAX(i'_g - i_g) \tag{4.3}$$

based on the set of inequality constraints for $I$ and $I'$ expressed in Equation 3.1 and the set of equality constraints expressed in Equation 4.2 of each loop dimensions. In Equation 4.3, $g$ is the analyzing dimension of the iteration domain in which array elements are written. For the above example, the ILP maximal problem needs to solved at each loop dimension, i.e., $MAX(i' - i), MAX(j' - j), MAX(k' - k)$ . Using an ILP solver, the EDV is found to be $[2, 4, 0]$. Note that, for this simple example, there are not many constraints and variables. The ILP problem is easy to solve when using an ILP solver. For the more general real-life applications, there are usually more loop dimensions, and arrays can also have more dimensions and/or more complex index expressions. The complexity of the ILP problem will then grow exponentially.

The MDV is similarly calculated by solving the ILP MAX problems of Equation 4.3 at each loop dimension sequentially, starting from the outermost dimension. In addition, the calculated maximum distances of the outer dimensions are propagated as equality constraints when calculating the dependency distance at the inner loop dimensions. The reason for this is that ordering information must be taken into account when calculating the actual dependency distance of the MDV at the analyzing dimension. This actual distance is not necessarily the maximal distance at every loop dimension, as shown in Section 3.1.6. Note that Equation 4.2 gives one equality constraint for each array dimension while Equation 4.3 gives one ILP problem for each loop dimension. The solution to each ILP problem has to satisfy all constraints of Equation 4.2 and Equation 4.3.

The procedure is now demonstrated using the example code of Figure 4.1. The ILP problem $MAX(i'-i)$ is first solved giving a dependency distance at the outmost dimension equal to 2. By propagating the equality constraint $i'-i=2$, the ILP problem $MAX(j'-j)$ is then solved and the dependency distance at the second outermost dimension is equal to 2. The outer two equality constraints are then propagated at the innermost loop dimension. The ILP problem $MAX(k'-k)$ is solved, giving the dependency distance at the innermost dimension equal to 0. The MDV is hence equal to $[2,2,0]$. From Figure 4.1 we see that the MDV does not have the maximal dependency distance for dimensions $j$ (which is 4). This is because the DV with length 4 in dimension $j$ has lenght 0 in the outer $i$ dimension.

Above has described how to calculating the MDV/EDV using the polyhedron GM by solving ILP problems. The ILP approach naturally works when the bounding box GM is used while less constraints exist. As the constraints are exact in the polyhedron GM, the calculated MDV/EDV are accurate and there are not pseudo DVs introduced. Pseudo DVs can only occur for the bounding box GM case. In Figure 3.11 pseudo DVs exist, but they do not contribute to the calculation of the MDV/EDV. The bounding box based MDV/EDV is then also accurate. This is usually the case for the targeted applications at where data accesses are regular.

### 4.2.2 The Vertexes Approach for MDV/EDV Calculation

Above has shown how to calculate the EDV/MDV by solving one ILP problem at each loop dimension. Since solving ILP problems in general is computationally expensive, a much simpler approach, named vertexes approach, is further introduced. The vertexes approach is a simplification of the ILP approach, based on the bounding box geometrical model. It is performed with basic algebra computations without using any ILP solver. As will be shown in Chapter 9, it is orders of magnitude faster. Following we first transform the ILP problems into an alternative form and then discuss how to solve these ILP problems with the novel vertexes approach.

In the ILP approach, the dependency distance of the MDV/EDV at one loop dimension is calculated by solving the ILP problem of Equation 4.3. This ILP problem provides a solution which satisfies the equality constraints in Equation 4.2 for all array dimensions. Instead of solving the ILP problem for one loop dimension in one step with all array dimension constraints, the problem can be decomposed and solved for each array dimension individually. The global decision among the solutions for all array dimensions is taken afterwards. At one array dimension ($h$), the MAX ILP problem Equation 4.3 is solved based on its own equality constraints only:

$$E'_h = E_h \tag{4.4}$$

together with the inequality constraints for the iteration domains. One MAX value output corresponds to one array dimension. Since all the MAX values are valid for the analyzing loop dimension, the minimal one among all the MAX values is still valid for the analyzing loop dimension, which is the global distance of the EDV/MDV at this loop dimension.

**Proposition 4-2**: the minimal value of all the MAX values calculated at each array dimension by solving the MAX ILP problem with its equality constraint in Equation 4.4 defines the dependency distance of the MDV/EDV.

With such a decomposition, the MDV/EDV can be calculated at each loop dimension by solving one ILP problems for each array dimension.

Above has shown how to decompose the general ILP approach of calculating the dependency distance of the MDV/EDV at one loop dimension to the problem of solving a set of ILP problems for each array dimension. Let us now present how to solve these ILP problems without using any ILP solver.

The equality of Equation 4.2 can be rewritten as:

$$e_g^{h'} \cdot i'_g + O' = e_g^h \cdot i_g + O \tag{4.5}$$

in which $e_g^h$ and $e_g^{h'}$ are the index coefficients of the analyzing loop dimension $g$ for the array write and read references, at the analyzing array dimension $h$. $O$ and $O'$ are the parts of the index expressions that do not contain the analyzing variables $i_g$ or $i'_g$. This is equivalent to

$$e_g^{h'} \cdot i'_g - e_g^h \cdot i_g = O - O' \tag{4.6}$$

which can be rewritten to isolate the difference between $i'_g$ and $i_g$ which needs to be maximized:

$$i'_g - i_g = \frac{(e_g^h - e_g^{h'}) \cdot i_g + O - O'}{e_g^{h'}} \tag{4.7}$$

If $e_g^{h'}$ is equal to zero, the expression at the right-hand side of Equation 4.7 is divided by zero which is not legal and $e_g^h$ is used as denominator instead. When both $e_g^{h'}$ and $e_g^h$ are zero, the equation is not valid which means the loop dimension is indetermined at this array dimension. The MAX problem in Equation 4.7 is hence equivalent to

$$MAX(i'_g - i_g) = \begin{cases} MAX(\frac{(e^h_g - e^{h'}_g)\cdot i_g + O - O'}{e^{h'}_g}) & \text{when} \quad e^{h'}_g \neq 0 \\ MAX(\frac{e^h_g \cdot i'_g + O - O'}{e^h_g}) & \text{when} \quad e^{h'}_g = 0 \\ undeterminant & \text{when} \quad e^h_g = 0 \wedge e^{h'}_g = 0 \end{cases} \tag{4.8}$$

The right-hand side of Equation 4.8 is a linear combination of loop variables. Because bounding box constraints are assumed on the loop variables, the maximum of this right-hand side can easily be found by replacing a loop variable with its upper bound if it has a positive coefficient, and its lower bound if it has a negative coefficient. In other words, the maximum is always reached at one of the vertexes of the bounding box iteration domain. We can immediately find which vertex by looking at the coefficients of Equation 4.8. Since only integral solutions of the MAX problem are considered, integral division result is used if the denominator $|e^{h'}_g|$ or $|e^h_g|$ in Equation 4.8 is not equal to 1, respectively.

If the array has more than one array dimension, the MAX problem formula of Equation 4.8 is performed for each array dimension with its own equality constraint found in Equation 4.2. The minimal value of all these maxima is the dependency distance at this loop dimension.

Let us illustrate the vertexes approach on the example of Figure 3.4. At the outermost loop dimension $i$, the maximal value at the first array dimension with the equality $i' - k' = i$ is calculated. The ILP problem $MAX(i' - i)$ in Equation 4.8 is then equal to $MAX(k')$. By taking the upper bound value 2 of $k'$, the maximal value for the first array dimension at the outermost loop dimension is found to be 2. Similarly, the equality $j' - i' + k' = j$ at the second array dimension leads to the ILP problem $MAX(j' - j)$, which is equal to $MAX(i' - k')$. The maximal value of $MAX(i' - k')$ is 4 when $i' = 4$ and $k' = 0$. At the third dimension of the array index expression, the coefficients of $i$ and $i'$ are zero, the problem $MAX(i' - i)$ is undeterminant and it does not affect the maximum. The minimal of the maximum among all array dimensions is the dependency distance at the outermost loop dimension, that is 2. The EDV dependency distance at the other loop dimensions can be calculated in a similar way.

For the MDV calculation, the calculated dependency distance equality constraints at the outer loop dimensions and the variables with their fixed bound values are propagated during the calculation of the dependency distance at the inner loop dimensions using their fixed values. At the outermost $i$ dimension, the dependency distance is equal to 2 as found above. During the calculation of dependency distance at the second outermost $j$ dimension, the dependency distance equality $i' - i = 2$ and the variable with its fixed bound value $k' = 2$ are propagated for the calculation of the dependency distance at the $j$ dimension. Now at the first array dimension, the coefficients of $j$ and $j'$ of the array index expression are zero and there is no fixed solution. At the second array dimension with the equality $j' - i' + k' = j$, the MAX problem $MAX(j' - j)$, equivalent to $MAX(i' - k')$, has the maximal result 2 when the upper bound value $i'$=4, is used together with the propagated constraint $k'$=2. The coefficients of $j$ and $j'$ are zero at the third dimension of the array index expression. Consequently, the dependency distance of the MDV at the second loop dimension is 2. In a similar way, the dependency distance of the MDV at the third loop dimension is calculated, equal to 0. This gives the MDV of $[2,2,0]$.

## 4.3 Estimation Errors and Exceptions

### 4.3.1 Memory Footprint Estimation Errors

The memory footprint estimation presented in this dissertation is performed using the bounding box geometrical model. Over-estimation can occur when the bounding box representation does not exactly represent the original polytope shape of the iteration domain. The counting of iteration nodes constrained by the MDV can then include nodes that are not present in the polyhedron iteration domain. For most advanced telecommunication and multi-media applications, the iteration domains are typically very close to rectangular and the bounding box approach works quite well. A major exception are skewed loops and diagonal loops with code as shown in Figure 4.4 and graphically depicted in Figure 4.5.

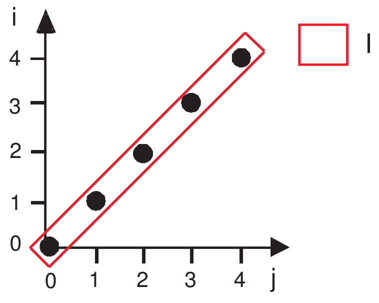
For these, the simplified bounding box iteration domain could result in overestimates with a factor of 2 (for diagonal case, power of two). Typically, however, only a few loop dimensions are not rectangular. It is then still possible to use the MDV approach combined with exact counting of the number of points in those iteration domains. In Equation 4.1, the non-rectangular factors of the product term should then be replaced by the exact counting function. An alternative would be to preprocess the code by a loop transformation which makes the skewed domain rectangular (which is feasible in the 2 examples above). The preprocessing could potentially be performed quite fast once the code has been brought in the geometrical model. But also that option has limitations because obviously, this only

```
for (i=0; i<=4; i++)
  for (j=0; j<=4; j++)
    if (i=j)
      A[i][j] = ...;
        (a)
```

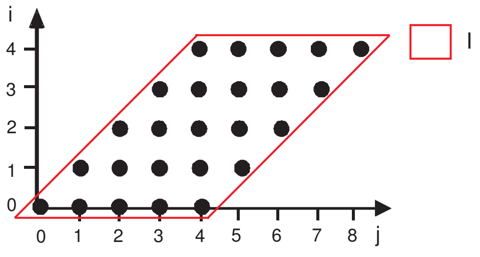
```
for (i=0; i<=4; i++)
  for (j=i; j<=i+4; j++)
    A[i][j-i] = ...;

        (b)
```

**Figure 4.4:** Code examples with (a) diagonal iteration domain, (b) loop skewed iteration domain



(a)



(b)

**Figure 4.5:** (a) diagonal iteration domain, (b)loop skewed iteration domain

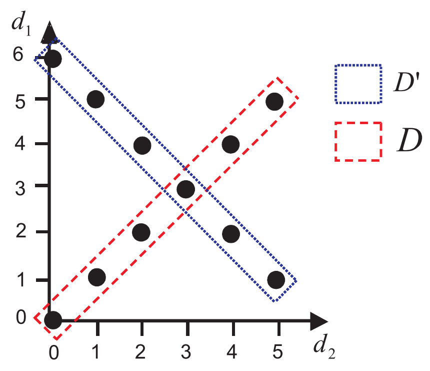


**Figure 4.6:** Data domain for Memory footprint estimation exception code

works for cases that can be considered as skewed rectangular domains. Further study of alternative solutions for accuracy improvement w.r.t. these cases is left for future work.

## 4.3.2 MDV Calculation Errors

The two approaches for MDV and EDV calculation have been presented. The general ILP approach can work on both the polyhedron geometrical model and the bounding box geometrical model. The MDVs calculated using the polyhedron geometrical model are accurate. With the bounding box geometrical model, the added pseudo dependency vectors can contribute to the calculation of the MDV. This is the case both with the ILP approach or the vertexes approach. The error appears mainly in triangular and skewed iteration domains, as discussed in Section 4.3.1. For the running example code it is for instance not a problem, even though it has several pseudo DVs. There is a trade-off between accuracy (best with ILP approach on the polyhedron geometrical model) and computation time (shortest with vertexes approach on the bounding box geometrical model).

In Equation 4.8, if the denominator, $|e_g^{h'}|$ or $|e_g^{h}|$, is not equal to 1, an integral division is used. The result is then only an approximation. The approximation only makes a difference for border cases of an array, which contribute little to the total size. Figure 4.7 shows a typical code example where such an overestimation occurs. Here the vertexes approach with Equation 4.8 would report $MDV = [100]$ while the real MDV is $[0]$. This is because, for one MAX ILP problemn, only variable on the left side of Equation 4.8 has non-zero coefficient. The one variable has coefficient zero which means it is indeterminant and over-estimation can hence occur. This error can still be removed with further analysis.

On the other hand, when any of the two variables on the left side of Equation 4.8 have a coefficient other than one, the integral division is only an approximation. By making the real division and round the result up, a maximum is found which may be an overestimate. This overestimation due to the border cases can be removed if basic set analysis [9] is applied before the estimation is performed.

```
for (i=0; i<=200; i++) {
  if (i<=100)
      A[100][i-100] = ...;
  if (i>=100)
      ... = A[i][0];
}
```

**Figure 4.7:** Code example

## 4.4 Summary

This chapter presents a system level memory footprint estimation technique for individual arrays. It is based on calculating the maximal lifetime window of the array accesses in the iteration domain, which is constrained by the MDV (or EDV when the execution ordering is not fully fixed). By counting the number of iteration nodes constrained by the MDV/EDV, the estimation approach presented in this dissertation is very fast. In order to achieve this fast estimation, two approaches of how to calculate the MDV/EDV have also been presented. The general ILP approach can be used both with the bounding box geometrical model and the polohedron geometrical model. In the last case, it is fully accurate. As will be shown in Chapter 9, even then it is faster than the existing ILP and Fourier-Motzkin variable elimination approaches, where the DVs are calculated first. This is due to that the ILP approach presented in this dissertation works by solving the problem for one variable at a time, which is much simpler than considering all variables at the same time. The ILP approach also skips the projection operation for the EDV calculation, which is also not trivial.

The novel vertexes approach is performed based on the bounding box geometrical model and is extremely fast compared to traditional ILP based calculations. It still gives accurate output for most practical cases. This makes the memory footprint estimation presented in this dissertation especially useful for giving fast feedback during the system level exploration, where estimation speed is critical in addition to accuracy. As already indicated in chapter 1 and section 2.2, the memory footprint estimation is required in several different contexts during the system design exploration trajectory, with different requirements on accuracy and estimation speed. So all the variants that have been reported here have their own valid position in the overall Pareto trade-off space.

# Chapter 5

# Fast Inter-Array Memory Footprint Estimation

The second part of the overall memory footprint estimation methodology handles inter-array inplace effects. The fact that different arrays having non-overlapping lifetime can reuse the same memory locations is hence taken into account. The maximal combined size of overlapping arrays give the memory footprint requirement for the complete application. As mentioned in Chapter 2, all previous memory footprint estimation techniques are too slow to be used repeatedly during the system level search space exploration.

In this dissertation, the inter-array memory footprint estimation is performed based on solving the Hanoi tower puzzle problem and hence named Hanoi tower approach. The Hanoi tower puzzle was invented by the French mathematician Edouard Lucas in 1883. The basic idea is that, by pre-defining a set of towers which respond to the lifetime periods of each statements, all arrays that are simultaneously alive at any given tower is assigned to the corresponding tower. At the end, all arrays are assigned and the tower which contains the arrays having the maximal size requirement defines the memory footprint requirement for the application. The Hanoi tower approach was first used in [40] for memory hierarchy layer assignment and pre-fetching optimization to overcome the memory performance/energy bottleneck. There is however no detail information about how it is implemented. For the work in this dissertation, the problem is much simpler as only the lifetime of arrays is considered. Several approaches based on the Hanoi tower puzzle are presented in order to perform a trade-off between estimation accuracy and computation time. In this work, the memory footprint requirement for an array is assumed to be constant by ignoring the boundary cases where the memory size requirement increases from zero to the constant size or decreases from the constant size to zero. To take into account these boundary cases, as discussed in [43, 41], is quite complex and needs full knowledge of the access pattern of the arrays. Since this is very computationally expensive, it is simply ignored in this work in order to achieve a fast estimation.

This chapter is organized as follows. In Section 5.1 the so called initial one-layer Hanoi-tower approach is presented. In order to reduce the computation time, the idea of performing an estimation on multi-layer Hanoi tower is also presented in Section 5.2 and an improved one-layer Hanoi tower approach in Section 5.3. Finally a summary is given in Section 5.4.

## 5.1 Initial One Layer Hanoi Tower Approach

A straight-forward implementation of the Hanoi tower approach is to consider each statement as a tower and then evaluate all arrays's lifetime on all the towers. A tower's lifetime is defined by its iteration domain within the common iteration space. An array's lifetime is defined from the first time any of its elements are used (usually it is a write reference) until the last time any of its elements are used (usually the last read reference). An array's lifetime can be calculated by performing an union operation of the lifetimes of all the references to this array. If an array's lifetime span over a tower, the array is assigned to that tower. An array is hence assigned to all the towers within its lifetime period. Note that, different from the original Tower of Hanoi puzzle, there are no shift operations after an array is assigned to a towers. This evaluation procedure is repeated for all arrays. At the end, all the arrays are assigned to the towers within their lifetime period. The tower which contains the set of arrays having the largest memory footprint requirement defines the memory size requirement for this application.

Figure 5.1 graphically illustrates how the one-layer Hanoi tower looks like after full assignment of all arrays, assuming there are 4 statements and 10 arrays. As shown in the figure, 4 statements are represented as 4 towers named from 1 to 4 and 10 arrays are named by capital letter from A to J respectively. The horizontal dimension represents the lifetime of arrays and the vertical dimension represents the size requirement of arrays. Each array is assigned to the towers within its lifetime period. For example, array $A$ is assigned in all the 4 towers meaning it is referenced at least in the first and last statements. It is however not necessarily required to be referenced in the 2nd and 3rd towers. Array $E$ is only assigned to tower 2 which means its lifetime is limited to the corresponding statement. The height of an array represents its memory footprint requirement, which is calculated with the intra-array memory footprint estimation techniques presented in the previous chapter. For arrays assigned to the same tower, they are simultaneously alive and there is no in-place possibilities between them. When all the arrays are assigned, it is obvious that the highest tower decides the overall memory size requirement for the application. The highest tower contains the set of arrays which are simultaneously alive having the largest size requirement.

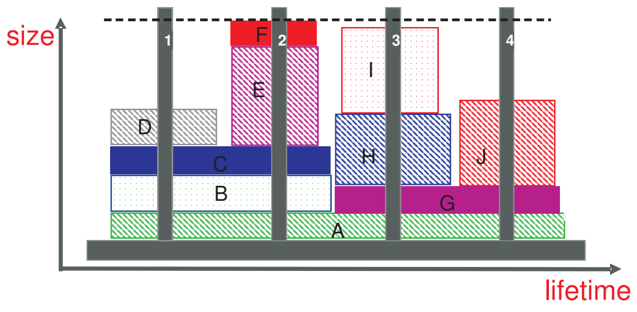


**Figure 5.1:** Illustration of the one-layer Hanoi tower approach

This approach is very straight-forward to implement. In practice, however, the number of statements are large in real life applications. For example, the QSDPCM algorithm, which will be used for experiments later on, contains 182 statements. To let each statement represents a tower will then give very long computation times. Even more so since the number of arrays can also be large. When the initial one-layer Hanoi tower approach becomes slow, it is not feasible to use it for memory footprint estimation methodology repeatedly. Alternative approaches are hence required.

## 5.2 Multiple Layer Hanoi Tower Approach

In order to reduce the number of towers before performing array-to-tower assignment to these towers, a multiple layer Hanoi tower approach is proposed. The idea is that the set of statements are unified consisting of one unified tower when their lifetimes are equivalent at a certain number of outer loop dimensions. Lifetime differences at inner loop dimensions of these statement are ignored. This set of statements within one unified tower can again consist of a number of unified towers each containing statements that are equivalent also for a number of additional inner dimensions. Obviously, an individual statement can still be a tower by itself. These inner towers are hence sub-towers of the unified tower, also called the parent tower. In this way, a multiple layer Hanoi tower structure can be built up. The sub-towers are however transparent for arrays at its parent layer. Figure 5.2 illustrates an example of a two layer Hanoi tower structure. In this example, 4 towers, instead of the total 9, are present at the first layer. For real life applications, the unification ratio will be much larger. For the QSDPCM algorithm, for example, there exist 12 towers when the lifetime of all 182 statements are unified at the outer fourth loop dimension. When unified at the sixth loop dimension instead, around 60 towers exist.

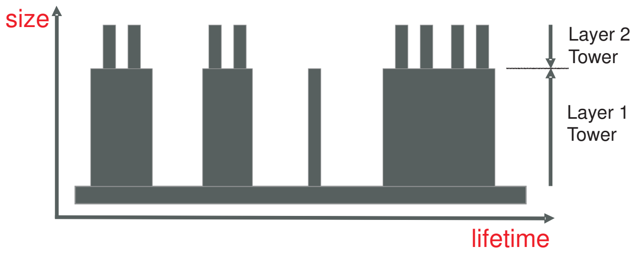


**Figure 5.2:** Illustration of the multiple layer Hanoi tower approach

Since now the number of towers is limited at a given layer, the computation time required to assign arrays to these towers can be reduced. When the array assignment has been performed at one layer, the unified tower which contains the set of arrays having the largest size requirement is selected for further evaluation at its sub-towers. This is because the arrays assigned at this unified tower are considered simultaneously alive until the loop dimension where the unified tower is created. They are however not necessarily simultaneous alive at inner loop dimensions. This means that there may exist further in-place possibilities between them, and the actual memory size requirement can be smaller. All these arrays are hence evaluated for assignment again at the sub-towers. This procedure is repeated until the inner most tower layer is reached and the largest size requirement at one of these sub-towers is found. However, the largest size requirement at one sub-tower can be smaller than the size requirement at its parent tower, since additional in-place mapping may exist at the inner loop dimensions. This may lead to a situation where the largest size requirement at the sub-tower is smaller than the size requirement at another tower at its parent tower layer. In this case, the evaluation should also be performed at sub-towers of the other tower. This procedure is continued until the actual largest size requirement is found, which should be equal to what is reported by the initial one layer approach.

Above the technique to build up a multiple layer Hanoi tower is discussed. The critical issue for this approach is on deciding how many layers it should be and at which loop dimension the statements should be unified. This varies at different loop dimension for different applications. An algorithm which makes decision at which dimension the union is performed is hence required. In this work, a proposed approach is to make the decision based on the evaluation of estimation accuracy ratio which can be achieved at each loop dimension. The estimation accuracy ratio indicates to what degree an accurate estimate will be found at one loop dimension compared to the result achieved without performing inter-array estimation. At one loop dimension, all the arrays whose lifetime can be exactly identified are summed up. The summed size over the estimated size achieved without performing any inter-array estimation gives the estimation accuracy ratio. The MDV is used to decide at which loop dimension an array's lifetime is known. The innermost dimension of the MDV with a nonzero value is the dimension at where an array's lifetime can be exactly determined while its lifetime spans over the remaining inner loop dimensions. This is because all these identified arrays are alive at this dimension and hence simultaneously alive. There is no inplace possibilities between them. However, the non-identified arrays's actual lifetimes are transparent at this dimension. There is a chance for inplace between them when their lifetimes are actually not overlapping. This can, however, only be detected at the inner dimensions. In the multiple layer Hanoi tower structure, the non-identified arrays are assumed to be simultaneously alive at the unified tower. The inplace possibilities between them will be identified at the sub-towers of the union one when required. The designer hence needs to pre-define to what accuracy the multiple layer Hanoi tower should be built at each layer or it can be interactively defined during the estimation based on the accuracy ratio output.

The multiple layer approach can reduce the number of towers to be considered at each layer. This is useful for large applications at where the number of statements can be very large. However, to build the multiple layer Hanoi tower represents an overhead for this approach. Based on the analysis of experiments, it was found that the overhead is not trivial even for not very big applications. Both the accuracy and efficiency of this approach depend on at what loop dimensions the multi layer tower structure is built, and the computation time required also varies. Due to these factors, a third inter-array estimation alternative has been developed, as described in the following section.

## 5.3 Improved One Layer Hanoi Tower Approach

The improved one-layer Hanoi tower approach is inspired by the multiple layer approach in order to reduce the computation time compared to the initial one layer approach. The idea is still to limit the number of towers in order to reduce the computation overhead, but now there is only one layer of towers. The number of towers is reduced based on evaluation of the estimation accuracy ratio The loop dimension where a satisfactory estimation accuracy ratio can be achieved is used to build towers by unifying statements based on their lifetime till the selected loop dimension. This will reduce the number of towers while still giving a satisfactory estimation result. The estimation accuracy can either be pre-defined before the estimation is performed or interactively defined by the designer during the estimation. As mentioned, this can be achieved based on the evaluation of the MDV for each array.

The estimation accuracy can further be improved. Let us consider the case that the estimation accuracy could be much higher at one inner dimension than the dimension at where the pre-defined accuracy is reached. It may worth to perform the estimation at the inner dimension with achieving much higher accuracy while the increased computation time is still acceptable. The computation time increases due to that the number of towers unified increase when it is performed at one dimension inner. This solution would bring trade-off between estimation accuracy and computation time. It is useful as the optimization effect can change during the large search space exploration at the early design stage. The result however varies depending on the pre-defined parameters for different applications.

## 5.4 Summary

This chapter presents a system-level inter-array memory footprint estimation technique. It is based on the principle that different arrays with non- overlapping lifetime can reuse the same memory locations. In the current work, a constant size requirement is used for individual array during the in-place estimation. Three different inter-array in-place estimation techniques have been introduced: initial one-layer Hanoi tower approach, multiple layer Hanoi tower approach, and improved one-layer Hanoi tower approach. The efficiency and estimation accuracy of the first and the last approaches are demonstrated in the experiments later in Chapter 9. The first approach is fast when the application is small while the last two approach can potentially speed up the estimate when the applications become larger. The multiple layer Hanoi tower approach has not been implemented in this work and is left for future work.

# Chapter 6

# Initial Hierarchical Memory Size Estimation

As mentioned earlier, loop transformations play a crucial role in optimizing the memory accesses at the earlier design stage. Traditionally improvement in data locality and regularity is used to steer the code transformations. This is, however, a very abstract cost function, which does not represent how the data will be mapped onto a memory platform. Further more, different loop transformations may result in optimal utilization of different memory platform instances. Ad-hoc decisions without estimating their impact on the actual memory utilization often lead to final sub-optimal solutions. An estimation of data mapping onto the memory hierarchy is hence necessary to evaluate the loop transformations' effect during the its design space exploration. This allows to find the right version(s) of loop transformations which will result in an optimal memory usage later on. A fast estimation to rapidly evaluate the impact of different loop transformations on multiple candidate memory platform instances are necessary. This can be of great use to a system designer or a steering tool, by providing accurate and very fast feedback during the huge loop transformation search space exploration.

This chapter presents a hierarchical memory size estimation methodology which estimates both the memory footprint requirement (of both individual arrays and between multiple arrays) and the data mapping onto a hierarchical memory architecture. The methodology presented in this chapter is named the initial Hierarchical Memory Size Estimation (HMSE) methodology, in order to distinguish it from the incremental approach presented in the next chapter. Since there exist typically a huge number of code transformation possibilities, and the estimation needs to be performed for each transformation, a fast estimation is crucial. Furthermore, as loop transformations are usually performed at an early design stage where the data memory platform, e.g., the number of layers and the exact size of the memories in each layer, is often not known yet, the estimation should be able to handle these unknown factors.

The initial HMSE methodology contains five main steps: intra-array size requirement estimation, data reuse analysis (DRA), memory hierarchy layer assignment (MHLA) estimation with Pareto curve output, inter-array memory footprint estimation, and Pareto curves comparison to find the interesting one(s). Figure 6.1 illustrates the flow of the methodology. The source code is first parsed into the geometrical model on which the estimation is performed. Each step is presented at the following sections.

When a sequence of loop transformations are performed incrementally, the estimation with the above five steps can be performed repeatedly directly on the geometrical model. This saves the time for parsing and dumping of C-code to/from the geometrical model. These processes usually consume quite some time for real life applications. During the tool implementation of the HMSE methodology, the first two phases are actually interleaved with each other, as are the third and fourth phases.

Now let us explain how these steps are performed based on the small code examples in Figure 6.2.

## 6.1 Intra-array Size Estimation

The first step of the initial HMSE methodology is to perform memory footprint estimation for each individual array. This is essential as loop transformations improves data access locality and regularity and also changes the data's lifetime. The size requirement for arrays will hence be affected. It is performed first because this step explores memory location reuse, which has a direct effect on other steps but not vice versa. For example, when the memory footprint estimation for one array reports that the memory footprint requirement for this array is small enough, it may be worth to assign this array on-chip directly. This makes it unnecessary to perform data reuse exploration for this array, which identifies parts of the array that is frequently accessed and therefore a candidate to be copied on-chip. This

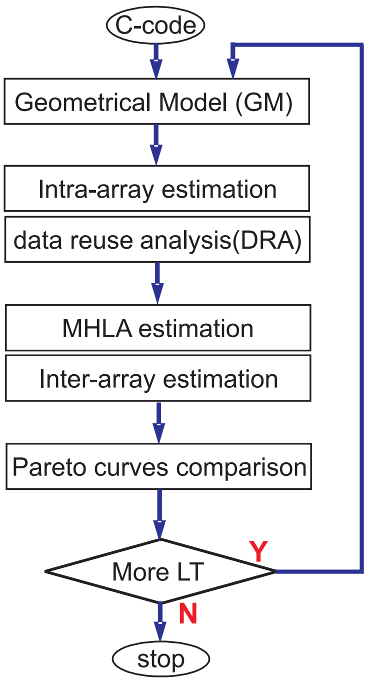


**Figure 6.1:** initial HMSE flow graph

```
for (i=0; i<=109; i++)
  for (j=0; j<=69; j++)
    for (k=0; k<=59; k++)
      for (l=0; l<=29; l++)
S1:
... = f(A[40i+k][20j+l]);
for (i=1; i<=109; i++)
  for (j=1; j<=69; j++)
    for (k=0; k<=59; k++)
      for (l=0; l<=29; l++)
S2:
... = g(A[40i+k-40][20j+l-20]);
```

(a)

```
for (i=0; i<=109; i++)
  for (j=0; j<=69; j++)
    for (k=0; k<=59; k++)
      for (l=0; l<=29; l++) {
S1:
... = f(A[40i+k][20j+l]);
        if (i>0 and j>0)
S2:
... = g(A[40i+k-40][20j+l-20]);
      }
```

(b)

```
for (j=0; j<=69; j++)
  for (i=0; i<=109; i++)
    for (k=0; k<=59; k++)
      for (l=0; l<=29; l++) {
S1:
... = f(A[40i+k][20j+l]);
        if (i>0 and j>0)
S2:
... = g(A[40i+k-40][20j+l-20]);
      }
```

(c)

**Figure 6.2:** Code examples: (a) original code, (b) after loop fusion, (c) after loop interchange

step also affects the mapping decision of the MHLA estimation as the array requires smaller memory size, which saves space and allow more data to be kept on-chip.

In Chapter 4, a fast intra-array memory footprint estimation methodology has been introduced which is performed based on counting the iteration nodes constrained by a maximal lifetime window. The lifetime window is constrained by the MDV. Two techniques for calculating the MDVs have further been presented. In the prototype CAD tool of the HMSE methodology, the intra-array size estimation step is implemented using the vertexes approach based estimation technique presented in Chapter 4.

## 6.2 Data Reuse Analysis

Data reuse analysis is used to detect the most frequently accessed data in an application. The frequently accessed data can be a whole array or parts of an array and are interesting candidates for mapping (copying) onto the memory hierarchy. They are called copy candidates (CCs). It is normally beneficial to copy these CCs from the main memory to smaller (on-chip) memories from where they are accessed multiple times. This can both save energy and improve performance since accessing on-chip memory is faster and more energy efficient. A full reuse exploration thus makes it possible to perform an optimal data mapping on the memory hierarchy, which can be either a customized memory hierarchy or a predefined one. Previous work on data reuse analysis [140, 24, 69] are all based on the polyhedral geometrical model resulting in a more accurate analysis. This model is however too slow to be used for the estimation purpose when analysis needs to be performed repeatedly during design space exploration of a huge number of loop transformations.

In this dissertation, a fast data reuse analysis technique is proposed which is performed on the bounding box geometrical model. It can result in an overestimate, but in practical cases it turns out to be as good as other more exact analysis techniques. Following firstly illustrates how this fast approach works and then highlights the differences of it compared to other techniques.

Initially, data reuse analysis is performed for all array references (both write and read) of one array, assuming all loops iterate over their complete set of iterator values. It results in the root for that array: i.e., the union of the data domains and the sum of accesses of all references. Reuse is detected between different array references as well as between the different iterations of the outer loop dimensions for the same array reference. The analysis is performed at every loop dimension, starting from the outermost dimension. The root, together with the CCs at every loop dimension, form the hierarchical data reuse tree (DRTree) for a given array. The root and CCs consist of a parent-children relation. In DRTree, a CC created at one loop dimension is a parent of CCs created at its inner loop dimensions. The root is hence the parent of all the CCs of this array.

At one loop dimension, we analyze the data domain accessed within one iteration of that loop by keeping all outer iterators constant and expanding the inner iterators. Since the index expressions are affine functions of the surrounding loop iterators and the iteration domains are bounding boxes, the data domain in any given iteration of a loop dimension is the same as in its first iteration, but shifted. We can therefore simply set the analyzing loop dimension to a constant and all outer dimensions to 0. An interesting CC exists if the data domains accessed at two consecutive iterations are overlapping. If overlap is found for two consecutive iteration values it will exist for any two consecutive iteration values because of the affine index expression and iteration domains. We therefore only need to calculate the data domains at two consecutive iteration values, 0 and 1, regardless of the actual bound values for that dimension. We calculate the data domain of one array reference in statement $S_x$ at these two consecutive iterations of the analyzing dimension $i_m$ as follows:

$$\overleftrightarrow{D}_{S_x,i_m} = E_{S_x}(\overleftrightarrow{I}_{S_x}(\bigwedge_{g=1}^{m-1} = 0, i_m = 0)) \tag{6.1}$$

$$\overleftrightarrow{D}^{+}_{S_x,i_m} = E_{S_x}(\overleftrightarrow{I}_{S_x}(\bigwedge_{g=1}^{m-1} = 0, i_m = 1)) \tag{6.2}$$

When analyzing the j-dimension of the example in Figure 6.2(a), i is set 0 while j is set to two constants 0 and 1 respectively. k and l iterates from their lower to their upper bound values, that is 0 to 59 and 0 to 29, respectively. The data domain of the array references in statement S1 in Figure 6.2(a) at two consecutive j-iterations (e.g., j=0 and j=1) are therefore

$$\begin{aligned}\overleftrightarrow{D}_{A,S_1,j} &= E_{A,S_1}(\overleftrightarrow{I}_{S_1}(i=0,j=0))\\ &= \{[d_1,d_2]^T \mid \exists i,j,k,l : d_1 = 40i+k \wedge d_2 = 20j+l\\ &\wedge [i,j,k,l]^T \in I_{S_1} \wedge i=j=0\}\\ &= \{[d_1,d_2]^T \mid 0 \le d_1 \le 59 \wedge 0 \le d_2 \le 29\}\end{aligned}$$

$$\begin{aligned}\overleftrightarrow{D}^{+}_{A,S_1,j} &= E_{A,S_1}(\overleftrightarrow{I}_{S_1}(i=0,j=1))\\ &= \{[d_1,d_2]^T \mid \exists i,j,k,l : d_1 = 40i+k \wedge d = 20j+l\\ &\wedge [i,j,k,l]^T \in I_{S_1} \wedge i=0 \wedge j=1\}\\ &= \{[d_1,d_2]^T \mid 0 \le d_1 \le 59 \wedge 20 \le d_2 \le 49\}\end{aligned}$$

Suppose this CC is assigned to the on-chip SPM layer, the overlapping part would be reused during the second iteration without fetching it from main memory again. The overlapping part is therefore called the reused part (denoted *reuse_part*), which can be calculated by performing an intersection operation on the two data domains. The non overlapping part accessed at the second iteration value needs to be fetched from off-chip main memory before it is accessed at that iteration. The non overlapping part is called the update part (denoted *update_part*). It is calculated by taking the difference between the data domain accessed at one iteration and its reuse part. The reuse part and update part can be calculated by

$$reuse\_part = \overleftrightarrow{D}_{S_x,i_m} \cap \overleftrightarrow{D}^{+}_{S_x,i_m} \tag{6.3}$$

$$update\_part = \overleftrightarrow{D}^{+}_{S_x,i_m} \setminus reuse\_part \tag{6.4}$$

For this example, the reuse part and the update part are

$$\begin{aligned}reuse\_part &= \overleftrightarrow{D}_{j,A,S_1} \cap \overleftrightarrow{D}^{+}_{j,A,S_1}\\ &= \{[d_1,d_2]^T \mid 0 \le d_1 \le 59 \wedge 20 \le d_2 \le 29\}\\ update\_part &= \overleftrightarrow{D}^{+}_{A,S_1,j} \setminus reuse\_part\\ &= \{[d_1,d_2]^T \mid 0 \le d_1 \le 59 \wedge 30 \le d_2 \le 49\}\end{aligned}$$

For each CC, three numbers need to be calculated: the size required to keep it in SPM (denoted *size*), the total number of accesses to the CC (denoted *#accesses*), and the number of misses from SPM to the main memory if the CC is kept on SPM (denoted *#misses*). They are calculated by

$$size = \prod_{h=1}^{n}(|U_{\overleftrightarrow{D}_h} - L_{\overleftrightarrow{D}_h}| + 1) \tag{6.5}$$

$$\#accesses = \prod_{g=1}^{m}(|U_{\overleftrightarrow{I}_g} - L_{\overleftrightarrow{I}_g}| + 1) \tag{6.6}$$

$$\#misses = \#iter_c \left(\#reuse\_part + (|U_{\overleftrightarrow{I}_c} - L_{\overleftrightarrow{I}_c}| + 1)\#update\_part\right) \tag{6.7}$$

where

$$\#reuse\_part = \prod_{h=1}^{n}(|U_{reuse\_part_h} - L_{reuse\_part_h}| + 1),$$

$$\#update\_part = \#D_c - \#reuse\_part$$

and

$$\#iter_c = \prod_{g=1}^{c-1}(|U_{\overleftrightarrow{I}_g} - L_{\overleftrightarrow{I}_g}| + 1)$$

$\#iter_c$ is the number of iterations of the outer loop dimensions, $c$ is the current loop dimension. $\#D_c$ means the number of elements within the data domain at one iteration of the current loop dimension. *#reuse_part* and *#update_part* are the number of elements within their part respectively. $(U_{\overleftrightarrow{I}_c} - L_{\overleftrightarrow{I}_c} + 1)$ is the number of times an *#update_part* must be fetched from the main memory for all iterations at the current dimension. In addition to that we need to fetch *#reuse_part* once. Multiplied with the number of iterations outside $c$, $\#iter_c$, this gives *#misses*.

This analysis is performed at each loop dimension resulting in a set of CCs. The CCs, together with the root, constitute the data reuse tree (DRTree) for that array. For the example code in Figure 6.2(a), the data reuse tree is shown in Figure 6.3(a). The CCs at the $i$ and $j$ dimensions are potentially interesting, but at the inner $k$ and $l$ dimensions the *reuse_part* is empty so no interesting CCs are found here. The same analysis is also performed for the array reference in statement $S_2$.

A reuse gain (*gain*) is also calculated for each CC. It will be used by the following MHLA estimation as discussed in Section 6.3. It is defined as the number of accesses to main memory that are avoided, per size unit, by assigning the CC to the SPM.

$$gain = \frac{\#accesses - \#misses}{size} \tag{6.8}$$

In general, the higher reuse gain a CC has, the more beneficial it is to copy it on-chip.

In the discussion above, we have shown how to perform the data reuse analysis for one array access. This analysis can also be performed between multiple array references as there might also exist data reuse between them. We do this when, for the current and all outer loop dimensions, the iteration domains of the multiple references are overlapping and their index expression coefficients are identical. When this is the case, the chance for potential reuse to be present is high. The minimal lower bounds and the maximal upper bounds are then used instead of individual bounds for these iteration dimensions. If the outer index expression coefficients are not identical, reuse is usually small and it is very complexity to exploit it in the final implementation. This makes it not worthy to be exploited. In the current tool implementation, the union CC, if it exists, will replace the individual CCs in the DRTree, provided it has a larger *gain*. In fact, both the union CC and all the individual CCs are valid candidates to be assigned to on-chip later on. But they can only be contradictively assigned to the SPM: the assignment of one would invalidate the possible assignment of the other. To keep both alternatives would require a large memory space during the running of the tool. More important, it will increase the complexity of the methodology especially for the following MHLA estimation step.

For the example code in Figure 6.2(b), the union CC between the two array accesses at the *i*-dimension has larger *gain* than the individual CCs and is kept in the DRTree instead of the individual ones. This is shown in Figure 6.3(b). At the *j*-dimension, a union CC between the two accesses also exists ($size = 5000$, $\#reuse\_part = 2000$, $\#update\_part = 3000$, $\#misses = 23320000$, $gain = 815.56$). It is not kept because it has lower *gain* than the individual CCs (average *gain* equals to 2500).

## 6.2.1 Comparison with Other DRA Techniques

The main difference between the DRA technique presented here and previous techniques [140, 139, 138] is that the previous ones are performed on the polyhedral GM while the one presented here is performed on the bounding box GM.

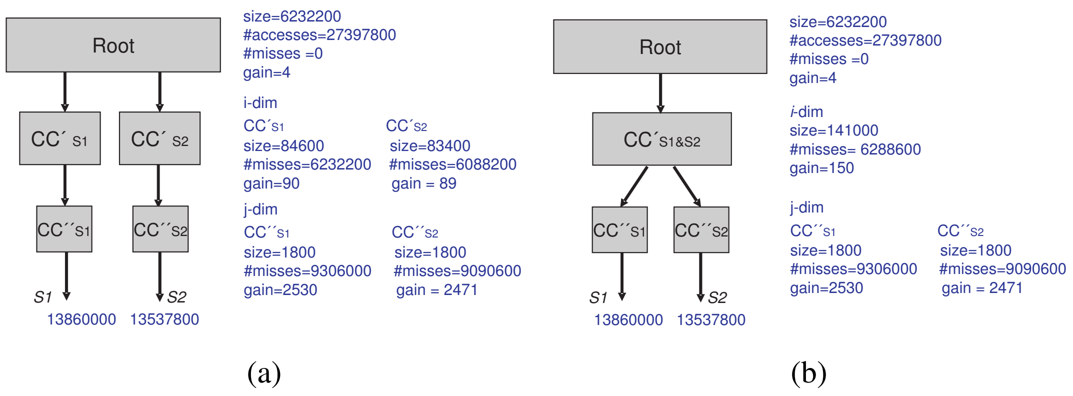


**Figure 6.3:** (a) DRTree for the original code and (b) DRTree for the fused code

This results in significant difference of the execution time for them. This is because operations, such as intersection or union of two domains or calculating the size of one domain, on the bounding box GM are much simpler and faster than on the polyhedral domain.

Worth to note that there are no data reuse for the update parts at the first and last iteration of the dimension being analyzed within that loop dimension. This non reuse phenomena only happens at the boundary of each loop dimension. Within the DRA technique presented here, the boundary cases are simply ignored as [69] does. Suppose CC is assigned to the on-chip SPM, we assume the data at the boundaries would also be fetched from the main memory to the SPM. [24] proposes a method to evaluate the lifetime of stencil elements. This means an exact reuse analysis but has high complexity for analysis and for generated code. [140, 139, 138] attempts to explore tradeoffs between SPM size and power, assuming an optimal run-time placement of data in SPM. They propose to perform the reuse analysis between both the same loop dimension level and also crossing loop dimensions. Both of these two techniques take into account the boundary cases. In contrast, [69] and our techniques do not perform analysis between crossing loop dimensions and ignores the boundary cases. Ignoring the boundary cases means lower analysis complexity and would also result in simpler code generation. [69] does the analysis by evaluating the reuse distance of the same set of data between the current and next iteration of a certain loop level. All these methods work on the general polyhedral GM and are quite computation expensive for large applications. In general, [140] has lower complexity than [24] but higher than [69]. In contrast, the complexity of our technique is significantly reduced due to the use of bounding box geometrical model. On the other hand, overestimation can occur due to the use of bounding box geometrical model.

As we will demonstrate in Chapter 9, the results using the simplified bounding box GM turns out to be as good as exact analysis for most practical cases in the multi-media target domain where the polyhedra are usually rectangular shaped. This simplification is critical to achieve a fast HMSE estimation. The technique presented here also differs from the previous version of DRA [139] by the introduction of the size dependent gain. Previous work has focused solely on accesses and misses.

## 6.3 Memory Hierarchy Layer Assignment Estimation

The main difference between the MHLA estimation techniques presented here and a previous technique developed earlier at IMEC [27], is that the techniques presented here perform a platform independent data assignment onto memory hierarchy while the previous does not. Further more, the techniques presented here aim at a fast estimation with a reasonably accurate (but not fully optimized) result. The previous technique aims at an optimal data assignment solution for a given memory hierarchy configuration.

The MHLA estimation aims to estimate which arrays and copies should be stored in the SPM, so that the number of off-chip memory accesses is minimized. The existing technique for MHLA [27] finds the optimal selection for one given memory hierarchy (SPM size). Their approach is not feasible to be used for the estimation purpose as the memory platform can usually not be given at the early loop transformation exploration stage: it is not realistic to perform an estimate for each possible memory hierarchy instance. Their backtracking algorithm also has high complexity and is hence too slow for large applications, making it unfeasible to be used frequently during the loop transformation design space exploration.

The MHLA estimation is performed for each version of the application code, based on the DRTrees output of the previous step. As there is usually no detailed memory platform defined at the loop transformations stage, we propose a platform-independent MHLA estimation approach based on a two-layer memory hierarchy template: the on-chip SPM layer and the main memory. The size of the main memory is assumed to be large enough to hold all arrays while the on-chip

SPM layer has an unfixed size varying from zero and up to the size required to store all arrays.

At the start, the SPM is empty and all accesses from the processor go to the main memory. Then at each pass of the estimation, the most promising unassigned CC/array is selected for assignment to the SPM layer. At the following subsections, we present two MHLA estimation heuristics on how to select the most promising unassigned CC at each pass of the estimation: a greedy MHLA estimation algorithm and an improved MHLA estimation algorithm. This procedure is repeated until all CCs and arrays in the DRTrees are assigned. At the end, all arrays are assigned. Each assignment results in a Pareto point between the number of main memory accesses and that SPM size requirement. The set of Pareto points consists of the Pareto curve for one version of code.

### 6.3.1 Greedy MHLA Estimation Algorithm

With the greedy MHLA estimation algorithm, the unassigned CC/array which has the biggest gain calculated based on Eq. 6.8 is selected and assigned to the SPM layer. The evaluation of all the CCs and arrays is done only once. This evaluation procedure can be performed directly based on the DRTrees output by sorting out all the CCs and arrays based on their gain value. The rationale behind the assignment based on its gain is that the one with the highest gain replaces, per size unit increase of SPM, the largest number of main memory accesses with accesses to the SPM. Since accessing main memory is more costly than SPM, this represents the most power saving per size unit increase of SPM. For each assignment, the information regarding its size requirement and which CCs that have been assigned are kept. This is used for the evaluation of the parent-children relation. When one CC or an array root is assigned, all its children, if already assigned, are removed since their data are not presented in SPM through their parents.

For the fused code with DRTree shown in Figure 6.3(b), for example, $CC''_{S_1}$ has the largest gain and is assigned to SPM first. It results in less main memory accesses and more SPM accesses than assigning any of the other CCs or the root. $CC''_{S_2}$ is assigned next. Then $CC'_{S_1,S_2}$ is assigned and its children $CC''_{S_1}$ and $CC''_{S_2}$ are removed from the SPM. The number of SPM accesses hence needs to be updated accordingly by adding in $CC''_{S_2}$'s effect and removing that of $CC''_{S_1}$ and $CC''_{S_2}$. Finally the array root is assigned replacing its child $CC'_{S_1,S_2}$ resulting in no misses to the main memory. The Pareto curve output is shown in Figure 6.4.

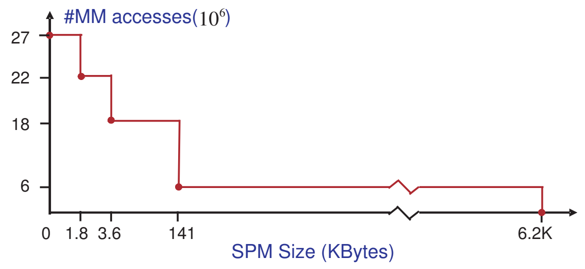


**Figure 6.4:** the Pareto curve output for the fused code

### 6.3.2 Improved MHLA Estimation Algorithm

The greedy heuristic is very simple and each CC/array is considered for assignment just once. However, due to the direct replacement of the already assigned CCs with its parent, the greedy algorithm can lead to local sub-optimal solutions. This is because we calculate the gain only once for each CC and array. When the children CCs are already assigned, the actual number of accesses to the parent should be equal to the number of misses from the assigned children, instead of the total number of accesses of all its children. The actual gain of the parent should hence be smaller when considering for assignment. It can in turn be more beneficial if other CC having larger gain is assigned. This over-estimate only has local effect but can still affect the data mapping result at certain critical SPM size(s). An improved MHLA estimation algorithm is hence proposed.

With the improved algorithm, the gain for the CCs and arrays are recalculated when any of its children is already assigned. As mentioned, all the CCs and array roots from the DRTrees are sorted out in an order based on their gains before the MHLA estimation starts. When one CC or an array root in the ordered list is considered for assignment, it is firstly checked whether any of its children is already assigned. If false, it is simply assigned and a new Pareto point is created as the greedy algorithm does. If true, the actual accesses and the gain to the parent, when its children already assigned, is recalculated as

$$\#accesses = \#misses_{assigned\_children} + \#accesses_{un\_assigned\_children} - \#misses \quad (6.9)$$

$$gain = \frac{\#accesses}{size} \quad (6.10)$$

$\#misses_{assigned\_children}$ means the number of misses for all the assigned children. $\#accesses_{un\_assigned\_children}$ means the number of accesses for all the un-assigned children. *#misses* means the number of misses for this parent. This Parent is then resorted within the ordered list for the unassigned ones. If it still has the highest gain, it is assigned replacing its children. Otherwise, the one having the highest gain, instead of it, is assigned. However, resorting procedures need to be performed each time when an parent's gain is calculated. This is not efficient to recursively perform the gain recalculation procedure during MHLA estimation. This is especially the case in practice as a large number of parents usually exist in the sorted list which takes time when performing resorting.

Alternatively, the gain recalculation procedure can be performed based on the DRTrees output before the MHLA estimation starts. In order to recalculate the gain for the root, the gain of its children needs to be recalculated and so on until the end leaves (the CCs created at the innermost loop dimension). This can be realized with a recursively procedure. If a CC has larger gain than its parent, then the CC will be assigned to the SPM before its parent is considered. The actual accesses to the parent can hence be calculated based on Equation 6.9 by taking into account all its children's effect. All the children which have larger gain are used as $assigned\_children$ in the equation. During the recursive procedure, an evaluation is also performed for each CC. If its parent has larger gain than the CC after recalculation assuming the CC would have been removed from the DRTree, this CC will be removed from the DRTree as its parent is more beneficial to be assigned. This removing ensures that only the CCs which will be assigned to the SPM are kept. At the end, all CCs and the roots for the new DRTrees are sorted descendingly based on their gain. This sorted list is then used from MHLA estimation.

Figure 6.5 shows the DRTree for the original code when the gain has been recalculated with the improved algorithm. The DRTree should be compared to Figure 6.2(a) for the simple algorithm. For this simple example, the Pareto curve is the same as shown in Figure 6.4 as the assignment order is not changed. This is, however, not the case in general as there are usually multiple DRTrees for different arrays and the recalculated gain for one CC/array usually changes the assignment order.

### 6.3.3 Algorithm Complexity

Because of the stepwise assignment, where each array and CC need to be evaluated only once for assignment, the greedy algorithm is very fast. As the gain for each CC/array is already calculated in the previous data reuse step, all the CCs

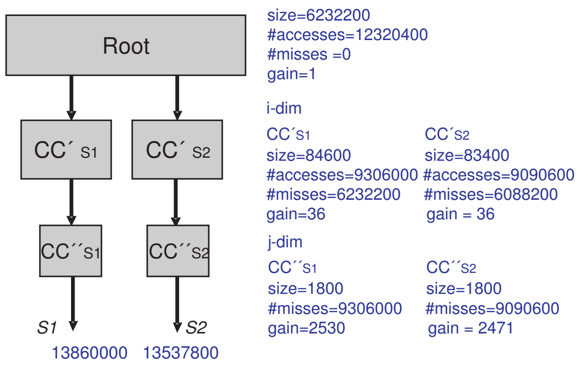


**Figure 6.5:** DRTree output by the improved MHLA algorithm for the original code

and arrays just need to be sorted based on their gains. At the MHLA estimation stage, the assignment is performed based on the sorted list of all CCs and arrays. The complexity of this algorithm is equal to the complexity of the sorting algorithm used. For example, when quick sorting algorithm is used, the complexity is $O(n \log n)$ (where $n$ is the number of CCs and array roots in the DRTrees). $n$ is usually smaller than the total number of CCs and array roots in the original DRTrees as all the uninteresting CCs are removed before the MHLA estimation starts.

In contrast, the improved algorithm further contains gain re-computation by traversing through the DRTrees. The complexity of that part depends on the depth of the DRTrees (denoted by $m$) and the maximal number of children (denoted by $l$) one parent can have. In theory, the complexity of the re-computation part is $O(m^l)$. The complexity of the improved algorithm hence is $O(m^l) + O(n \log n)$. In practice, the maximal number of children that one parent has usually is small. It has not been larger than 13 for the applications studied in Chapter 9 and it is usually smaller for most parents. Further more, not all CCs within the DRTrees are beneficial to be assigned on-chip which reduces the number of interesting CCs within the DRTrees. This makes the computation time of the gain re-computation part insignificant compared to the whole HMSE methodology.

For comparison, [27] uses a backtracking algorithm which has a complexity of $O(2^n n^2 \log n)$ for a given two-layer memory platform instance. To estimate the mapping of a K-layer memory platform, the complexity of their approach is $O(k^n n^k \log n)$. The approaches presented here give a quick MHLA estimate and the accuracy is quite reasonable as demonstrated on real life applications in Chapter 9. Their work is also targeting (and requires) one specific memory platform instance at a time, while our MHLA estimation is platform instance independent. This means that our MHLA estimation only needs to be performed once for a given version of application code.

## 6.4 Inter Array In-place Estimation

Loop transformations change the data access lifetime, affecting the memory footprint requirement of individual arrays and hence the in-place memory footprint requirement between multiple arrays for the whole application. The inter array memory footprint requirement should hence also be estimated at each round when loop transformations are performed. Since a platform independent MHLA estimation is proposed here and the data are incrementally assigned, the inter array in-place estimation should be integrated with the MHLA estimation step and performed each time a CC or a root is assigned. The inter array in-place estimation speed hence becomes critical since there are many CCs and array roots that are going to be assigned. The techniques introduced in Chapter 5 can hence be used to achieve such a fast inter array in-place estimation.

## 6.5 Pareto Curve Comparison

As explained in Section 6.3, one Pareto curve is created after each round of code transformation. Among all these curves, a global Pareto curve is generated. A global Pareto point is the one that has a smaller number of main memory accesses than any other point with an SPM size not larger than the one this point has. A global Pareto point will hence result in the most energy efficient use of a certain two layer memory hierarchy instance with an SPM size equal to that of this point. Hence, the corresponding loop transformation is interesting and should be kept among the potential solutions. As a result, all curves which contribute to global Pareto point and hence the global Pareto curve are kept.

Figure 6.6(b) shows the three Pareto curves generated by the greedy MHLA estimation algorithm for the three versions of the code shown in Figure 6.2. We can see that, for small SPM sizes, the interchanged code will result in less main memory accesses than the fused code. This means that the interchanged code can result in the most energy efficient use of certain two layer memory platform

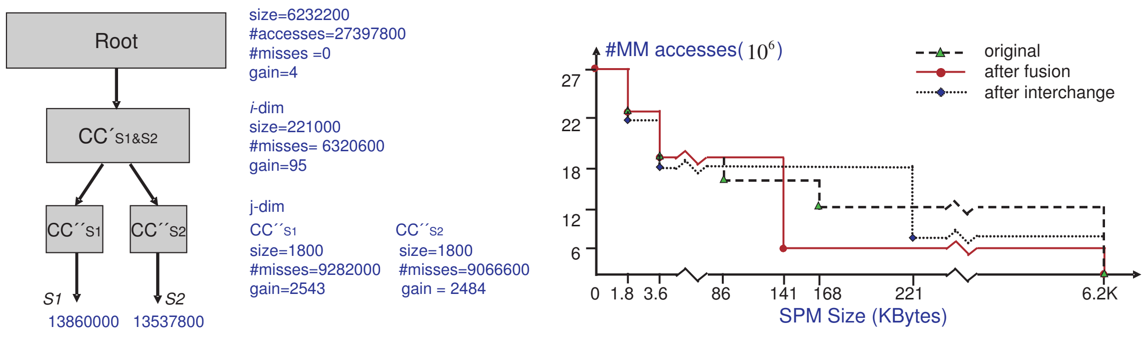


**Figure 6.6:** (a) DRTree output for the code after loop interchange and (b) Pareto curves comparison

instances. On the other hand, the fused code has the smallest number of main memory accesses for larger SPM sizes. The original code is the best for the SPM having the size in between these two parts. Hence these two loop transformations and the original code are all interesting and should be kept for selection until the actual memory platform instance is defined. Although the difference in memory accesses is not that significant, this small example still demonstrates the fact that different loop transformations might be optimal for different memory platform instances. This just demonstrates the usefulness of the HMSE methodology presented in this dissertation.

The Pareto curve comparison allows us to find all interesting loop transformations during the exploration. Later when the memory hierarchy is given, the Pareto curves allow to find the most beneficial loop transformation with the corresponding version of code. The initial HMSE methodology presented here is fast. It only takes between a few milliseconds till a few seconds for real life applications as demonstrated later in Chapter 9. The generated Pareto curves allow not only to find the optimal loop transformations, but also to customize the optimal memory hierarchy while trading off power consumption and SPM layer size requirement. This is discussed in Chapter 8.

## 6.6 Summary

This chapter has represented the initial HMSE methodology. It estimates the low power data mapping on a hierarchical memory platform and the memory size requirement for one version of code. The methodology can be coupled together with any loop transformation algorithms when they are performed on the geometrical model. The estimation can be performed repeatedly for any sequence of loop transformations. This can save the dumping and parsing procedure between the C-code and the geometrical model.

In order to achieve a fast estimation, advanced techniques have been used at each step during the estimation: fast vertexes based MDV approach for intra array in-place mapping estimation, data reuse analysis based on the bounding box geometrical model, fast platform independent MHLA estimation with Pareto curve output, Hanoi tower based inter array in-place mapping estimation. The initial HMSE methodology presented here is fast and only takes between a few milliseconds till a few seconds for real life applications as demonstrated later in Chapter 9.

# Chapter 7

# Incremental Hierarchical Memory Size Estimation

The previous chapter has presented the initial HMSE methodology, which performs estimation starting from the source code. When a sequence of loop transformations are performed incrementally, the estimation is repeated based on the GM on which loop transformations are performed and represented as matrix operations as shown in Chapter 3. The GM contains the data and control flow information for the previous version of code. Performing the estimate directly on the GM saves the interactive dumping and parsing procedures between the source code and the GM. Advanced techniques have also be introduced at each step of the initial HMSE in order to achieve a fast estimate. It requires between a few millisecond and a few seconds cpu time for real life applications as will be demonstrated in Chapter 9. This is fast enough to be used as estimation basis in the later stages of the system design trajectory (see Chapter 1 and Section 2.2).

However, in earlier stages of the system design trajectory, the freedom is much larger still. Then, usually a huge number of (combined) loop transformation possibilities exist, often in the order of tens of thousands or more. For real life applications, even existing greedy automatic loop transformations algorithms, i.e. [144, 58], need to perform a large number of loop transformations to reach their optimal solution. Though the initial HMSE is pretty fast, it is still time consuming when the number of loop transformation possibilities are large and the estimation needs to be performed repeatedly. On the other hand, when loop transformations are performed incrementally, the transformations usually just have local effect. This means that only a few loop nests and arrays are transformed. Furthermore, the transformation can potentially be performed just on some array references and even at a limited number of loop dimensions. Because of this kind of characteristics, it is not necessary to perform the initial HMSE each time loop transformations are performed. The execution time can be reduced when performing HMSE by locally updating the transformations is faster than performing the initial HMSE again. In this Chapter, the incremental HMSE methodology is introduced which exploits the local effect of loop transformations during estimation. This further speeds up the hierarchical memory size estimation making the estimation feasible to be used repeatedly during the loop transformation search space exploration.

This chapter is organized as follows. Section 7.1 gives an overview of the whole HMSE methodology including both incremental HMSE and initial HMSE. How to perform incremental intra-array memory footprint requirement estimation is discussed in Section 7.2. An incremental data reuse analysis is presented in in Section 7.3. A summary is given in Section 6.6.

## 7.1 Incremental Hierarchical Memory Size Estimation

Figure 7.1 shows the flow for the whole HMSE methodology including the initial and incremental HMSEs. Based on the initial HMSE output, we first check if we are going to perform incremental loop transformations. If not, the estimation stops. If yes, incremental HMSE is performed. When loop transformations are performed, both the previous GM information, the transformation matrix information, and the updated GM information are kept.

When a sequence of loop transformations are performed incrementally based on the previous version, the loop transformations performed at each step usually have very local effect compared to the previous version. The incremental HMSE methodology intends to exploit the local effect to save the execution time when repeating the estimation. For the incremental HMSE methodology, as shown Figure 7.1, the intra-array memory footprint estimation and data reuse analysis are performed incrementally. They are performed based on the incremental loop transformation information, the previous GM information and the previous output at that step. The MHLA estimation and inter-array memory footprint estimation are

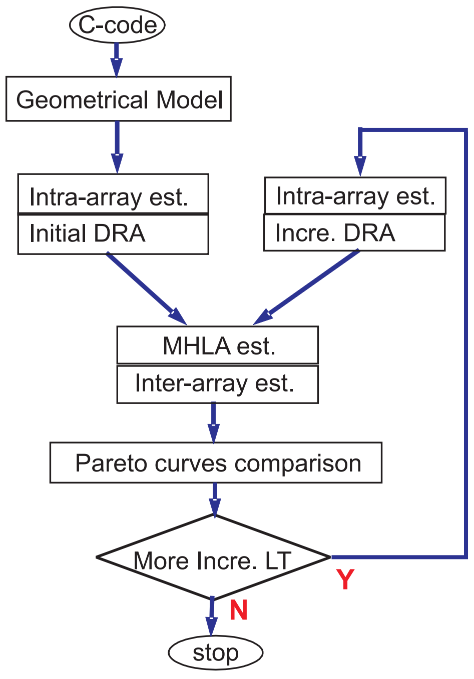


**Figure 7.1:** HMSE flow graph

redone as they are in the initial HMSE, with a Pareto curve output. This curve is then compared to those generated previously, and only those that contribute to any global Pareto point are kept. This procedure is repeated until there are no more incremental loop transformations to be performed.

The reason why only the intra-array memory footprint estimation and data reuse analysis are performed incrementally is that local transformation effects can easily be detected at these two steps, which might reduce the execution time. Experiments show that these two steps dominate the overall execution time for the HMSE methodology while the later MHLA estimation and inter-array memory footprint estimation steps only takes a small part of the time. Furthermore, it is more complex to exploit the local effect at the MHLA estimation and inter-array memory footprint estimation steps.

We will now take a closer look at the incremental techniques.

## 7.2 Incremental Intra-array Memory Footprint Estimation

When loop transformations are performed incrementally, usually not all arrays and loop nests are transformed. It is hence only necessary to perform the memory footprint estimation for the arrays that have been transformed. A straight forward approach is to perform the estimate for all the arrays which have been effected by any of the loop transformations performed. They can be found easily in the GM by first identifying all the statements which have been transformed and then all the arrays which have been referenced within these statements. These arrays are named as transformed arrays from now on although loop transformations performed may not always have affect on their memory footprint requirement. The same intra-array memory footprint estimation technique as used in the initial HMSE is then performed for these arrays based on the updated GM information. The GM information is updated based on the previous GM information together with the loop transformations information which is represented as matrices.

## 7.3 Incremental Data Reuse Analysis

As motivated in Chapter 3, when affine loop transformations are performed incrementally, they are in fact represented as matrix operations. Based on the analysis of the transformation matrix, it is possible to detect which arrays and loop dimensions that are transformed. Based on this, we can directly update the data reuse trees built previously only for the transformed arrays, instead of always recomputing the data reuse trees from scratch for all arrays. Furthermore, we can simply update the parts of the data reuse trees where the loop transformations take effect. The key is hence to identify the loop dimensions at where loop transformations are performed.

In [67], a simple incremental DRA algorithm has been introduced which is limited to only translation (loop fusion and loop shifting). As discussed previously in Chapter 3, translation can only make changes on the constant offset part of the unimodular transformation matrix. The following matrix corresponds to the loop fusion which transformed the code of Figure 3.16(a) to the code in Figure 3.16(b)

$$\tilde{A}_{S_2} = \begin{bmatrix} 1 & 0 & 0 & 0 & -\mathbf{1} \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The last column in the matrix corresponds to the translation offset part. It is very easy to identify at which loop dimensions of one statement translations are performed. If any offset parts have a non zero value, this means the corresponding loop dimension has been transformed. In contrast, linear transformation changes the linear part of the unimodular transformation matrix. An extended algorithm will be presented in this section which works for all affine loop transformations. It also contains a simple algorithm that can handle translations. As a result, nearly all relevant loop transformations (see chapter 2) are handled. For the others, that occur much more rarely, the non-incremental methodology should still be used. The algorithm is demonstrated on a small example code.

The effect on arrays and loop dimensions varies depending on the incremental loop transformation performed. Let us demonstrate the incremental DRA using the example code in Figure 7.2 as used previously in Chapter 3. The outer $i$ and $j$ dimensions of the code in Figure 7.2(a) are interchanged in 7.2(b). The two statements $S_1$ and $S_2$ are hence affected as their loop nests are transformed. The

```
for (i=0; i<=4; i++)
  for (j=0; j<=5; j++)
    for (t=0; t<=1; t++) {
      if (t=0)
S1:     A[i][j] = ...;
      if (i>=1 and j>=2 and t=1)
S2:     ... = f(A[i-1][j-2]);
    }
            (a)
```

```
for (j=0; j<=5; j++)
  for (i=0; i<=4; i++)
    for (t=0; t<=1; t++) {
      if (t=0)
S1:     A[j][i] = ...;
      if (j>=1 and i>=2 and t=1)
S2:     ... = f(A[j-1][i-2]);
    }
            (b)
```

**Figure 7.2:** Loop interchange example: (a) before (b) after

transformation matrices of the two statements are identical for this transformation, represented as

$$\tilde{A}_{S_1} = \tilde{A}_{S_2} = \begin{bmatrix} 0 & \mathbf{1} & 0 & 0 \\ \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In this case, loop transformations are performed at the outermost two dimensions. This is seen at the matrix since the diagonal elements of the first two rows are different from 1 and any of the other elements on those rows have a non-zero value. As mentioned in Chapter 3, all affine loop transformations can be represented as matrix operations. Based on the transformation matrix, it can easily be detected at which loop dimensions loop transformations have effect. For this example, it is necessary to perform DRA update at these two transformed dimensions. Sometimes a transformation at outer dimensions has ripple effects at inner loop dimensions. Analysis must then also be performed at these inner dimensions. Otherwise, no analysis is required to be redone at the inner dimensions. The results from the previous run of DRA (initial or incremental) can be reused instead. For real life applications, loop transformations usually also have effect on a limited number of arrays, the untransformed arrays are not changed and DRA hence does not need to be redone for them. Because of this, incremental DRA can significantly save computation time when the transformation effect is local.

An evaluation is performed to decide whether to perform a complete DRA or to perform the normally much faster local updating. Identification of the dimensions that are transformed is based on evaluation of the transformation matrix. As mentioned in the theory part in Chapter 3, a dimension is transformed if the diagonal element of a row is different from 1 and any of the other elements on that row are non-zero. Thus it is necessary to identify the outermost dimension that is transformed (denoted as $tra\_OMD$) and the innermost dimension that is transformed (denoted as $tra\_IMD$) for all the transformed array accesses. We then assume that the array accesses is transformed for all dimensions within this range. The strip mining transform is not simply a matrix multiplication since it creates a new dimension and divides the old dimension with the new dimension's bound difference. The $tra\_OMD$ and $tra\_IMD$ for strip mining are hence the old dimension and the new dimension. The updating of the iteration domain bounds during strip mining is trivial and we will not go into further details. If an array is transformed for all its dimensions, we choose to rebuild its DRTree from start. Otherwise, local updating is performed by recomputing DRA only at the transformed loop dimension range. Figure 7.3 shows the pseudo code of our incremental DRA algorithm.

In the procedure *incremental_DRA*, the $GM\_update$ is executed for the first time at line 11. At this line the iteration domain of each transformed statement and the index function of all array accesses in the transformed statements are updated based on the Equations 3.15 and 3.16. This is required even though the data domain of the whole array is unchanged as proved by Equation 3.17. The reason is that the data domain calculated for one iteration value of a certain loop dimension, see Section 6.1, is usually changed by the loop transformation. Consequently, it must be calculated based on the updated iteration domain and updated index function.

Let us now illustrate our incremental DRA algorithm based on the loop interchange example code. After the execution of $GM\_update$, one transformed array is investigated at a time. $tra\_OMD$ and $tra\_IMD$ for the current array are identified at line 13. They are actually the outermost and innermost transformed dimensions found for any of the transformed array references. Since only the outermost two loop dimensions have been interchanged for the two statements, the if-condition at line 14 is false and the if-condition at line 17 is true. The procedure $local\_update$ is called at line 18 for the parent that are one dimension above $tra\_OMD$. In this case the parent is the root. If both the outermost and innermost dimensions had been transformed, the if-condition at line 14 would have been true, and initial DRA would have been performed. If the outermost dimensions

```
0:   DRTrees : The data reuse trees for all arrays
     GM : the GM info before incremental loop transformation
     OMD : the outermost dimension
     IMD : the innermost dimension
     tra_OMD : the outermost dimension that has been transformed
     tra_IMD : the innermost dimension that has been transformed
     LTs_info: the performed LT info

10:  procedure incremental_DRA()
11:      GM_updated = GM_update(LTs_info, GM)
12:      for (each transformed array)
13:          identify tra_OMD and tra_IMD based on LTs_info
14:          if (tra_OMD = OMD and tra_IMD = IMD )
15:              recompute DRTrees[array] based on GM_updated
16:          else
17:              if (tra_OMD = OMD )
18:                  local_update(DRTrees[array], GM_updated, LTs_info)
19:              else
20:                  locate each parent whose children's dimension is OMD_tra
22:                      local_update(parent, GM_updated, LTs_info)

30:  procedure local_update(parent, GM_updated, LTs_info)
31:          if (parent.children != {} and (parent contains transformed array accesses))
32:              CCs_new = CCs_calc(parent, GM_updated, LTs_info)
33:              if (parent.children != CCs_new) /* children have been changed * /
34:                  for (i=0; i<#parent.children; i++)
35:                      if (parent.children[i] exists in CCs_new)
36:                          local_update(parent.children[i], GM_updated, LTs_info)
37:                      else
38:                          update parent.children[i] from the CC within CCs_new
39:                          if (parent.children[i] and CC contains the same array accesses)
40:                              local_update(parent.children[i], GM _updated, LTs_info)
41:                          else /* DRTree for parent.children[i] is changed after transformation*/
42:                              compute DRTree below parent.children[i]
43:              else
44:                  if (parent's dim < (tra_IMD of array accesses parent contains))
45:                      local_update(parent.children[i], GM_updated, LTs_info)
46:          else
47:              compute DRTree below parent

50:  function GM_update(LTs_info, GM)
         /* update the iteration domain of each transformed statement */
         /* and the index function of the corresponding array accesses */
         return GM_updated

60:  function CCs_calc(parent, GM_updated, LTs_info)
         return CCs_new /* recalculate the children CCs of parent */
```

**Figure 7.3:** *Pseudo code of incremental DRA algorithm*

had not been transformed, the parents at one dimension above *tra_OMD* would have been found at line 20, followed by a call to local_update at line 21 for each parent.

Within the procedure call, the parent is the root for this example and the if-condition at line 31 is true. The CCs at the dimension below the parent is recomputed (denoted as *CCs_new*) based on *GM_updated*. Since *CCs_new* is not equal to the old CCs at the same position within the *DRTrees*[*array*], the if-condition at line 33 is true. In this case there is only one CC and the *CCs_new* is updated at line 38. Since both the old and new CC contain the two array accesses, the if-condition at line 39 is true and *local_update* is called again at line 40. This procedure call will continue the evaluation at the next inner dimension, in this case at the second outermost dimension. The updated CC at the outer dimension now becomes parent. The if-condition at line 31 is true, and the new CCs below the parent are calculated at the second outer dimension. Since the loop interchange takes effect at the two outermost dimensions, the new CC is not equal to the corresponding old CC in the DRTrees. Hence it is updated with the new CC at line 38. As both the new and old CC contain the same array accesses at the second dimension, the function *local_update* is again called at line 40. The parent now becomes the updated CC at the second outer dimension. This time the analysis is performed at the third dimension even if it is beyond the *tra_IMD*. If the analysis detects any CC changes, *DRTrees*[*array*] will be updated with the new CCs at this dimension and the analysis continues at the next inner dimension. If no changes are detected, we go to line 44 and check if the parent contains any array accesses which is transformed at that/inner dimension(s). In this case, there are no changes at the third dimension, so the analysis stops for this branch. Since the if-conditions at line 33 and 44 are both false, the analysis stops without analyzing the third and fourth dimensions. The analysis also stops for this array since only this CC exists. The DRTree output is the same as achieved by performing initial DRA as shown in Figure 6.6(a).

For this small example, the local DRA needs to be performed at the three outermost dimensions. Consequently, not much computation time can be saved with incremental DRA compared to using the initial DRA. For real life applications, e.g., the QSDPCM driver as demonstrated in the experiment chapter in Chapter 9, there can be more than ten loop dimensions. Typical loop transformations are performed at the outer loop dimensions with no effect on the remaining inner dimensions, or at inner dimensions affecting only a limited number of statements. In that case, the incremental DRA with local updating can significantly reduce the computation time. As explained in Chapter 3, it is often necessary to insert extra time dimension before each loop dimension. The computation time saving of incremental DRA is then even more evident. Furthermore, the number of CCs in a DRTree can potentially increase exponentially at each level in the loop nest. The number of CCs to recompute will therefore become very large for deep loop nests. As a consequence, it is very beneficial to only recompute the CCs that have actually been changed.

## 7.4 Summary

In this chapter the incremental HMSE methodology has been presented. The main difference from the initial HMSE is that incremental intra-array memory footprint estimation and incremental data reuse analysis are only performed for the arrays that have been transformed. An incremental data reuse analysis algorithm is presented which can further limit data reuse analysis to be performed on the transformed array references and also at the loop dimensions at where they are potentially transformed. This can significantly reduce the estimation time since incremental loop transformations usually have very local effect. Locally updating DRTrees only where the data reuse are affected saves the unnecessary computation for building the DRTrees. This improvement can be globally significant for the whole HMSE, as data reuse analysis is the most time consuming step. The incremental HMSE can speed up the estimation in the order of two magnitudes, as demonstrated on experiments in Chapter 9.

# Chapter 8

# Low Power Data Memory Platform Exploration

As mentioned, at the early system level design exploration stage, the memory platform structure can usually not be given, i.e., either the number of memory layers or the size requirement for one layer is not defined. This is, for example, the case at the high level loop transformations stage within the DTSE methodology. The designers should be able to customize the memory hierarchy configuration in order to trade off cost functions of power, performance and size requirement. This customization is however usually done at later design stages when more constraints are determined and there are more information about the implementation. Designers traditionally perform such a customization by evaluating different architectural configurations based on their intuition and experience or on simulation. Usually a limited number of configuration instances are evaluated. This would in most cases result in a sub-optimal solution. For the memory platform, there usually exist many possible configuration instances. Due to the large scope of the memory platform configurations, it is infeasible to exhaustively simulate the performance and energy characteristics of an application for each configuration. Meanwhile, there equivalently exist the large scope of the code transformation exploration issues. Thus, exploration tools are necessary to rapidly evaluate the impact of different candidate memory platform configurations. Such tools can be of great utility to a system designer by giving fast initial feedback on a wide range of memory platform instances.

This chapter presents an approach for performing a fast low power data memory platform exploration. It is performed based on the Pareto curve output from the HMSE methodology for each version of code. Based on the number of accesses to each layer of the memory platform, a power estimation can be performed for all possible memory platform configurations. Using the quick power estimation of all possibly memory platform configuration, an early low power memory platform exploration can be performed.

This chapter is organized as follows. Section 8.1 presents how the power estimation is performed for any given memory platform configuration. In Section 8.2, the low power data memory exploration is discussed.

## 8.1 Power Estimation For A Given Memory Platform Configuration

This section introduces the principle of how to perform a power estimation for any given memory platform configuration. First the power model used is discussed in Section 8.1.1. Then Section 8.1 presents the principles on how to perform the power estimation for any given memory platform configuration. This is followed with discussion about how to perform an estimation for multiple layer memory platform configurations in Section 8.1.3 .

### 8.1.1 Power Estimation Model

As mentioned, we propose to perform power estimation based on our HMSE methodology Pareto curve outputs. As Pareto curve only contains the number of accesses information, the power estimation is performed based on a simple high level power model:

$$Power_{total} = \sum_{i=1}^{eachlayers} \#accesses_i * power_{size}/access_i \tag{8.1}$$

In this equation, $power_{size}/accesses_i$ means the power per access for a memory with the considered size as the power per access is different for different sized memory. The static power consumption of the memory can be taken into account within the power model or it can be ignored. The total power consumption for the memory platform is hence the sum of the power consumed at each layer. The estimation accuracy hence also depends on the power model which will be used. In this work, a model which is confidential to IMEC is used, that is based on a realistic memory macro. This makes it feasible to compare the estimate achieved by this work with theirs. However, any other memory model can also be used. For example, [128] has used CACTI [150, 151] model in their work.

In the current version of this work, the power model only considers the case that all data are accessed directly by the CPU. When a block of data is accessed, it is however more power efficient for the data to be accessed by a DMA controller and the power per access can hence be lower. This have not been considered yet in the current version of work and is left for future work. This power model is simple and still useful at the early design stage. To obtain a precise power estimation from a high-level system description requires a full design effort. It would be very slow which is not acceptable at the early stage. On the other hand, precise estimation is not always needed. It is often sufficient to be able to make a relative comparison among a large number of memory platform configurations so further evaluation can be proceeded on a limited number of alternatives.

### 8.1.2 Power Estimation for a Given Memory Platform Configuration

The power estimation for a given memory platform configuration is performed based on the Pareto curve output from the HMSE methodology. As explained in Chapter 6, the Pareto curve is created based on a pre-assumed two-layer memory platform structure where the on-chip SPM has unspecified size. A Pareto point is created by assigning the most promising CC/array replaces, per size unit increase of SPM, the largest number of main memory accesses with accesses to the SPM. Since accessing main memory is more costly than SPM, this represents the most power saving per size unit increase of SPM. The rationale behind is that each Pareto point corresponds to a low power two layer data memory with that SPM size requirement. We have information of the number of accesses to different layers of memory. Then using the above mentioned power model, we can per-

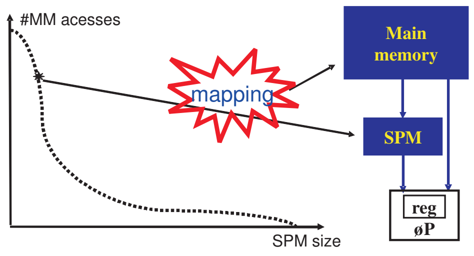


**Figure 8.1:** Power estimation for any two layer memory platform

form power estimation based on the Pareto curve for any given two layer memory platform configuration.

Figure 8.1 illustrates how the data mapping for a given two layer memory platform instance is performed. For a given two layer memory platform instance, the Pareto point selected for simulating the on-chip SPM layer should be the one having a size as close as possible to, but not larger than, the SPM size of the selected platform. The chosen Pareto point defines which data (both CCs and the root assigned at that point) that should be mapped on the on-chip SPM layer. The CCs assigned at the point define which part of the arrays to copy from the off-chip memory to the SPM. The off-chip memory of the selected platform should be large enough to store the remaining roots unassigned at that Pareto point. The off-chip memory should always be large enough to keep all the data and its size is less relevant. The Pareto point contains all the access information: the number of accesses from processor to the on-chip SPM layer, the number of misses from the SPM to the main memory. The number of accesses to the main memory can easily be calculated by adding the misses from the SPM and the bypass accesses, which is equal to the total number of accesses minus the number of accesses from processor to the SPM. The energy can hence be estimated based on the number of accesses to each layer together with the abstract energy-per-access model, which depends on the SPM size.

### 8.1.3 Power Estimation for a Multiple-layer Memory Platform Configuration

So far has described the principles of how to perform power estimation for a two layer memory platform configuration. This technique can also work for a multiple layer data memory platform, assuming multiple SPM layers exist as shown in Figure 8.2. For a multiple layer data memory platform, each layer is assigned a Pareto point with a size as close as possible to, but not larger than, the layer size. Based on the selected Pareto points, the number of accesses to each layer can be found. Power estimation can hence be performed based on the number of accesses to each layer.

However, for multiple layer memory platform, over-estimate occurs. For example, for a three layer memory platform, two Pareto points are selected and the number of accesses to each of the three layers are calculated. Each Pareto point defines the optimal data mapping for the two layer memory platform with the SPM layer having a size equal to the size at the Pareto point. When the two points are used for the three layer platform, the Pareto point having larger size in principle also contains the data which are mapped for the Pareto point having smaller size. In other words, some data are considered for mapping at both the two Pareto points. In practice, as soon as data are mapped to the smaller SPM layer, it should not be used again at the larger SPM layer. This potentially enable other data to be assigned to the larger SPM layer. This overestimation cannot be avoided. The key issue is how much effect such an overestimation would be and whether this will still result in reasonable estimate. It will vary depending on the sizes of the different SPM layers and also depending on the applications considered. Power estimation for multiple layer memory platform has not been evaluated yet in the experiments. Further evaluation is hence required and is left for future work.

## 8.2 Data Memory Platform Exploration

Above has presented the techniques used to perform power estimation for a given memory platform. Since the estimation is performed based on the Pareto curve, the estimation can be performed for any memory platform configurations. This is very useful since estimation of one platform at a time is not feasible when a large number of memory platform configurations can exist. Based on a power

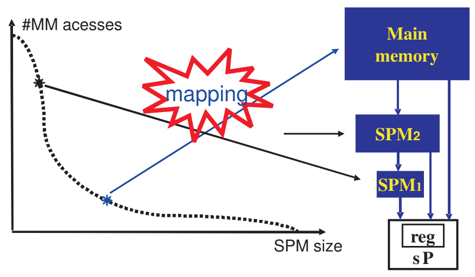


**Figure 8.2:** Power estimation for any multiple layer memory platform

estimation for all possible memory platform configurations, we can easily select the most low power memory platform(s) for further more exact evaluation.

As stated previously, the goal of our HMSE methodology is to perform a fast data mapping estimation to steer the loop transformation exploration. It is also beneficial to perform this early memory platform exploration. The designer can select the limited number of most lower power memory platform(s) at the early design stage for further optimization, together with selecting the most interesting transformed codes which may result in optimal data mapping for that memory platform instances. This significantly limits the number of memory platform instances together with the limited versions of the code that should be retained for more detailed (and time-consuming) optimization at later design stages.

## 8.3 Summary

In this chapter we have represented the principle on how to perform power estimation for any given memory platform and how the low power data memory platform exploration is performed based on that. The memory platform exploration enables the designer to select the most lower power memory platform(s) at the early design stage for further evaluation, together with selecting the most interesting transformed codes which may result in optimal data mapping for that platform instances. This significantly limits the number of memory platform instances together with the limited versions of code for optimization at later design stages.

# Chapter 9

# Experiments

This chapter presents experiments performed for the estimation methodologies presented in this dissertation on several real life applications. It is organized as follows. Section 9.1 first gives a brief description of prototype tools which implement the estimation methodologies. In Section 9.2, the Cavity Detection algorithm is used for demonstration. Section 9.3 studies the QSDPCM algorithm as another test vehicle. Section 9.4 presents experiment results for the 2-dimensional Wavelet transform algorithm. A summary is given afterwards.

## 9.1 Prototype Tools

Two prototype tools, MFE, for fast memory footprint estimation methodology, and HMSE, for hierarchal memory size estimation, have been developed to prove the feasibility and usefulness of the methodologies presented in this dissertation. The current versions have been implemented in the object-oriented language Python [3].

The MFE tool covers both intra-array memory footprint estimation and inter-array memory footprint estimation. In the current version of tool implementation, we do not go to basic set analysis but simply take the maximum of the two sizes if the dependencies are overlapping. Note that, for the ILP approach of the MDV calculation, the ILP problem is first formulated in Python and then solved by calling an ILP solver, *lp_solve* [2] in this case which is implemented in C/C++.

For inter-array memory footprint estimation, the initial one layer Hanoi tower approach and the improved one layer Hanoi tower approach have been implemented and compared. For the improved approach, the tower union operation is performed at a loop dimension at where the pre-defined accuracy ratio is reached. In this case, the accuracy ratio is defined as 95%. The multiple layer Hanoi tower approach has not been implemented. The estimation result achieved by the MFE tool is also compared with what is achieved by the Atomium/MC tool. Atomium [68] is the abbreviation of "A Toolbox for Optimizing Memory I/o Using geometrical Models" and operates at the behavioral level of an application, expressed in C. It is a tool suite supporting the DTSE methodology as reviewed in Section 2.2. The Atomium/MC tool refers to the memory compaction tool and is implemented based on the techniques presented in [41].

For the HMSE methodology, both the initial HMSE and incremental HMSE approaches are implemented. Note that the inter-array memory footprint estimation step is not implemented yet within the prototype HMSE tool and is left for future work. To evaluate the estimation accuracy of the HMSE, the HMSE estimation results are compared with the results obtained with the Atomium/MH tool. The Atomium/MH tool refers to the memory hierarchy tool and is implemented partially based on the techniques presented in [27]. Note that in order to make the fare comparison, the inter-array inplace optimization option in their tool is not selected since this option is not implemented yet within the current HMSE implementation.

## 9.2 Demonstration on Cavity Detection Algorithm

The MFE and HMSE methodologies have been applied to the Cavity Detection algorithm, which is a medical image-processing application that extracts contours from images to help physicians detect brain tumors. The initial algorithm consists of several functions, each with one image frame as input and one as output. The new value of a pixel in one function depends on its neighbors in the previous function, which are too big to be on-chip and saved in background memory. There are approximately 100 lines of C code (ignoring file I/O etc.) Experiments are performed with images of 640*400 pixels.

The estimations are performed for a selected sequence of loop transformations. This selected sequence of loop transformations is chosen for illustration purposes. The HMSE methodology does, however, not require any specific sequence of loop transformations. It can work for any kind of sequence, depending on the loop transformation algorithm used. The sequence of loop transformations performed varies depending on the algorithm chosen for performing loop transformations. Note also that for one application, at each step, it can be a specific loop transformation technique, as presented in Chapter 3, performed on a statement, or it can also be a set of loop transformation techniques performed on a statement or multiple statements.

Figure 9.1 gives the code used as start point for our estimations. This is the resulting code after applying preprocessing and pruning on the straightforward source code. Preprocessing and pruning is the first step in the DTSE methodology as presented previously in Section 2.2. Linear transformations have also been applied to this code. In this case, loop interchange has been performed at the outmost two dimensions (*x*- and *y*-dimension) for all the loop nests. This is because that *y*- dimension has a smaller range and would result in larger buffers after transformations, if kept second outermost.

As the first in the sequence of transformations, the two loop nests, input loop and horizontal Gaussian blur are fused together at the outermost two dimensions. This transformation is named LT-1. The part of transformed code is presented in Figure 9.2 while the rest part of code is unchanged. The horizontal Gaussian blur loop uses input pixels with coordinates [x-1][y] to [x+1][y]. The optimal locality (for these two loops) is reached when input pixel [x+1][y] is used immediately after it has been produced in the input loop. The input buffer size is then limited to 3 pixels, instead of N*M as it is in Figure 9.1. To obtain this, the input and horizontal Gaussian blur loops are fused. Before this is possible, the horizontal Gaussian blur x-loop has to be shifted by 1, so that it starts only when all pixels it needs are available. The y loop has no problem. Note that the conditions due to the shifting are fused with the existing condition, if possible. Note also that the input statement could have been further fused with the k-loop of the horizontal Gaussian blur (guarded by the condition k==1), but this would create more complexity while reducing the size of in_image only by 1. The locality improvement above is a local effect: the global effect, however, is also positive. Indeed, the in_image signal is not used anywhere else, so any local improvement is also good globally.

The above fusion is followed by a set of loop transformations fusing all the loop nests at the outermost y-dimension. This transformation is named LT-2. The part of transformed code is presented in Figure 9.3. The same ideas as for the input and Gaussian blur loops can be applied to the subsequent loops. Now, also

```
void cav_det_l2 () {
    ... // Declaration
    // Input
    for (y=0; y<M; ++y)
      for (x=0; x<N; ++x)
        in_image[x][y] = input();
    // GaussBlur
    tot = init_gausss(1);
    // Horizontal Gaussian blur x
    for (y=0; y<M; ++y)
      for (x=1; x<N; ++x) {
        g_acc_x[x-1][y] = 0;
        for (k=-1; k<=1; ++k)
          g_acc_x[x-1][y] += gauss_filt(in_image[x+k][y], k);
        g_tmp_image[x][y] = g_acc_x[x-1][y]/tot; }
    // Vertical Gaussian blur y
    for (y=1; y<M-1; ++y)
      for (x=1; x<N-1; ++x) {
        g_acc_y[x-1][y-1] = 0;
        for (k=-1; k<=1; ++k)
          g_acc_y[x-1][y-1] += gauss_filt(g_tmp_image[x][y+k], k);
        gauss_image[x][y] = g_acc_y[x-1][y-1]/tot;  }
    // Compute Edges
    for (y=2; y<M-2; ++y)
      for (x=2; x<N-2; ++x) {
        maxdiff[x-2][y-2] = 0;
        for (k=-1; k<=1; ++k)
          for (l=-1; l<=1; ++l)
            if (k!=0 || l!=0)
              maxdiff[x-2][y-2] = maxdiff_compute (gauss_image[x+k][y+l],
                                      gauss_image[x][y], maxdiff[x-2][y-2]);
        ce_image[x][y] = maxdiff[x-2][y-2];  }
    // Reverse kernel & DetectRoots kernel & Output
    for (y=3; y<M-3; ++y)
      for (x=3; x<N-3; ++x) {
        out_tmp[x-3][y-3] = 0;
        for (k=-1; k<=1; ++k)
          for (l=-1; l<=1; ++l)
            if (k!=0 || l!=0)
              if (out_tmp[x-3][y-3] == 0)
                if (ce_image[x+k][y+l] < ce_image[x][y])
                  out_tmp[x-3][y-3] = 1;
        output(out_tmp[x-3][y-3], x, y);    }
}
```

**Figure 9.1:** Cavity Detection Algorithm (orig)

```
    for (y=0; y<M; ++y)
      for (x=0; x<N; ++x) {
        // Input
        in_image[x][y] = input();

        // Horizontal Gaussian blur
        if (x>=2) {
          g_acc_x[x-1][y] = 0;
          for (k=-1; k<=1; ++k)
            g_acc_x[x-2][y] += gauss_filt(in_image[x+k-1][y], k);
          g_tmp_image[x-1][y] = g_acc_x[x-2][y]/tot;
        }
      }
}
```

**Figure 9.2:** The part of code after LT_1

shifting in the y direction is required. Fusing the y loops reduces all intermediary signals. The memory footprint requirement for g_tmp_image and gauss_image are reduced from N*M to 3*(N-2) pixels, and ce_image from N*M to 3*(N-4) pixels.

Next, a set of loop transformations are performed fusing all the loop nests also at the x-dimension. This transformation is named LT-3 and the transformed code is shown in Figure 9.4. Fusing of the x-loops in addition to the y-loops improves locality even more, as the data is consumed immediately after it has been produced. Fusing the x loops further reduces all intermediary signals with N pixels. g_tmp_image is now reduced to 2*(N-2)+1 pixels, gauss_image to 2*(N-2)+3 pixels and ce_image to 2*(N-4)+3 pixels.

Locality could be improved even further by reversing all the k- and l-loops. Indeed, then the production of a pixel is immediately followed by its first read. However, this effect is extremely small: the distance between the write and last read references is virtually unchanged, and the distance between different consumptions does not change. Therefore, this transformation is not applied.

Following let us look at the experiment results achieved on the different version of the loop transformation codes. All the experiments are performed on a server with four 2.8GHz Intel Xeon processors with 2G memories.

```
for (y=0; y<M; ++y) {
  for (x=0; x<N; ++x) {
    // Input
    in_image[x][y] = input();
    // Horizontal Gaussian blur
    if (x-1>=1) {
      g_acc_x[x-1][y] = 0;
      for (k=-1; k<=1; ++k)
        g_acc_x[x-2][y] += gauss_filt(in_image[x+k-1][y], k);
      g_tmp_image[x-1][y] = g_acc_x[x-2][y]/tot; }
  }
  // Vertical Gaussian blur
  for (x=1; x<N-1; ++x)
    if (y-1>=1 && y<M) {
      g_acc_y[x-1][y-1-1] = 0;
      for (k=-1; k<=1; ++k)
        g_acc_y[x-1][y-1-1] += gauss_filt(g_tmp_image[x][y+k-1], k);
      gauss_image[x][y-1] = g_acc_y[x-1][y-1-1]/tot;
    }
  // Compute Edges
  for (x=2; x<N-2; ++x)
    if (y-2>=2 && y-2<M-2) {
      maxdiff[x-2][y-2-2] = 0;
      for (k=-1; k<=1; ++k)
        for (l=-1; l<=1; ++l)
          if (k!=0 || l!=0)
            maxdiff[x-2][y-2-2] = maxdiff_compute (gauss_image[x+k][y+l-2],
                                       gauss_image[x][y-2], maxdiff[x-2][y-2-2]);
      ce_image[x][y-2] = maxdiff[x-2][y-2-2];
    }
  // Reverse kernel & DetectRoots kernel & Output
  for (x=3; x<N-3; ++x)
    if (y-3>=3 && y-3<M-3) {
      out_tmp[x-3][y-3-3] = 0;
      for (k=-1; k<=1; ++k)
        for (l=-1; l<=1; ++l)
          if (k!=0 || l!=0)
            if (out_tmp[x-3][y-3-3] == 0)
              if (ce_image[x+k][y+l-3] < ce_image[x][y-3])
                out_tmp[x-3][y-3-3] = 1;
      output(out_tmp[x-3][y-3-3], x, y);
    }
}
```

**Figure 9.3:** The part of code after LT_2

```
void cav_det_l2 () {
    ... // Declaration
    // GaussBlur
    tot = init_gausss(1);
    for (y=0; y<M; ++y)
      for (x=0; x<N; ++x) {
        // Input
        in_image[x][y] = input();
        // Horizontal Gaussian blur
        if (x-1>=1) {
          g_acc_x[x-1][y] = 0;
          for (k=-1; k<=1; ++k)
            g_acc_x[x-2][y] += gauss_filt(in_image[x+k-1][y], k);
          g_tmp_image[x-1][y] = g_acc_x[x-2][y]/tot; }
        // Vertical Gaussian blur
        if (x-1>=1 && x-1<N-1 && y>=2) {
          g_acc_y[x-1-1][y-2] = 0;
          for (k=-1; k<=1; ++k)
            g_acc_y[x-1-1][y-2] += gauss_filt(g_tmp_image[x-1][y+k-1], k);
          gauss_image[x-1][y-1] = g_acc_y[x-1-1][y-2]/tot; }
        // Compute Edges
        if (x-2>=2 && x-2<N-2 && y>=4) {
          maxdiff[x-2-2][y-2-2] = 0;
          for (k=-1; k<=1; ++k)
            for (l=-1; l<=1; ++l)
              if (k!=0 || l!=0)
                maxdiff[x-2-2][y-2-2] = maxdiff_compute(gauss_image[x+k-2][y+l-2],
                                           gauss_image[x-2][y-2], maxdiff[x-2-2][y-2-2]);
          ce_image[x-2][y-2] = maxdiff[x-2-2][y-2-2]; }
        // Reverse kernel & DetectRoots kernel & Output
        if (x-3>=3 && x-3<N-3 && y>=6) {
          out_tmp[x-3-3][y-3-3] = 0;
          for (k=-1; k<=1; ++k)
            for (l=-1; l<=1; ++l)
              if (k!=0 || l!=0)
                if (out_tmp[x-3-3][y-3-3] == 0)
                  if (ce_image[x+k-3][y+l-3] < ce_image[x-3][y-3])
                    out_tmp[x-3-3][y-3-3] = 1;
          output(out_tmp[x-3-3][y-3-3], x, y); }
      }
}
```

**Figure 9.4:** Cavity detection algorithm after loop transformation (LT-3)

### 9.2.1 Intra-array Memory Footprint Estimation

Table 9.1 compares the intra-array memory footprint estimation between our two MDV calculation approaches (ILP approach and vertexes approach) and also with the Atomium/MC tool. In the table, $t$ is the CPU execution time. As shown, the vertexes approach and ILP approach based memory footprint estimations are identical with the optimized results reached by Atomium/MC. When comparing the computation time, the ILP approach based estimation takes less than 200ms for all versions of Cavity Detection codes. The ILP approach based estimation is in the same magnitude of computation time as the Atomium/MC tool does. The vertexes approach based estimation just takes less than 5 milliseconds, which is 3 orders of magnitude faster.

| Application | Declared | MDV approach | | | Atomium/MC | |
|---|---|---|---|---|---|---|
| code | size | est. size | $t_{vertexes}$ | $t_{ILP}$ | est. size | $t_{MC}$ |
| orig | 2536880 | 1016984 | 124.0ms | 5.8s | 1016984 | 24.9s |
| LT-1 | 2536880 | 760987 | 114.3ms | 6.0s | 760987 | 27.1s |
| LT-2 | 2536880 | 5743 | 166.4ms | 6.0s | 5743 | 26.5s |
| LT-3 | 2536880 | 3838 | 168.5ms | 6.0s | 3838 | 82.4s |

**Table 9.1:** Intra-array memory footprint estimation for Cavity Detection Algorithm

### 9.2.2 Inter-array Memory Footprint Estimation

Within the inter-array estimation experiments, the effect of intra-array memory footprint estimation is also included. For the inter-array estimate, the vertexes approach based intra array estimation is used.

As shown in Table 9.2 , the improved one layer Hanoi tower approach gives estimation result very close to the original one layer Hanoi tower approach while the computation speed is between one and two orders of magnitude faster. The two Hanoi tower based estimation techniques still give very reasonable estimates when compared to what Atomium/MC gives. The improved Hanoi tower approach is however 3 orders of magnitude faster.

| Application code | Declared size | Inter Hanoi tower approach | | | | Atomium/MC | |
|---|---|---|---|---|---|---|---|
| | | Initial approach | | Improved approach | | | |
| | | size | time | size | time | size | time |
| orig | 2536880 | 511202 | 306.0ms | 511202 | 31.2ms | 511213 | 25s |
| LT-1 | 2536880 | 509126 | 361.4ms | 509126 | 7.8ms | 509137 | 25s |
| LT-2 | 2536880 | 5741 | 594.0ms | 5741 | 1.2ms | 5741 | 25s |
| LT-3 | 2536880 | 3836 | 846.4ms | 3840 | 28.5ms | 3840 | 85s |

**Table 9.2:** Inter-array memory footprint estimation comparison for Cavity Detection Algorithm

### 9.2.3 HMSE Estimation

Figure 9.5 shows the HMSE estimation result for the Cavity Detection algorithm. First the initial HMSE is performed for the *orig* code. Then the incremental HMSE is performed for the sequence of loop transformations based on the geometrical model information for the previous version of code and the loop transformation matrices information. The figure demonstrates that the fully transformed version LT-3 always results in the global Pareto curve. Compared to the first two code versions orig and LT-1 which require over 1M memory to store all arrays, the last two versions only require 5K and 3K, respectively. It is therefore possible to keep all the data of the last two versions on a small SPM layer. With the same SPM size (3K), the original code would need more than 3 million accesses to main memory. The figure also shows why it is important for the memory size estimation to take into account the memory hierarchy. The total memory size requirement is 3838Bytes for LT-3 and over 1M for the LT-1. Without taking into account the memory hierarchy exploration, the conclusion would therefore be that the code

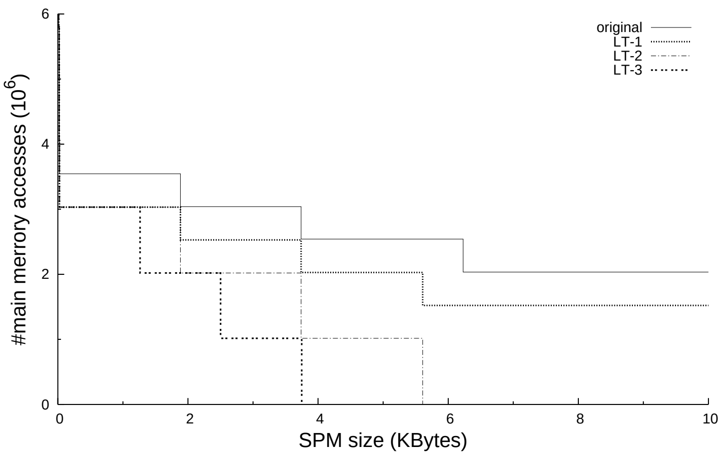


**Figure 9.5:** HMSE estimate with Pareto curves output for Cavity Detection Algorithm

LT-1 is not interesting at all. However, when the hierarchical memory size estimation is performed, it turns out that for SPM sizes up to 1286Bytes, the code LT-1 is a viable alternative. Since the LT-1 code has lower complexity (the loop shifting of LT-3 adds `if`-clauses), it is actually preferred for small SPM sizes. Analysis of the code complexity as a third trade-off axis can be performed at a later stage when the memory platform is given.

Table 9.3 compares the execution time of the initial HMSE and incremental HMSE. The *GMparsingupdate* corresponds to the time required to parse the source code into the geometrical model or update the geometrical model when loop transformations are performed. This is required to perform the HMSE estimation, but is usually anyway performed during loop transformation exploration and is not considered as a step within the estimation. The $Intra - +DRA$ means the time required to perform both intra-array estimation and DRA. As shown, the incremental HMSE takes only a few tens of milliseconds. This is at most one-quarter of the execution time of the original code for this example. Even for this small application, incremental HMSE reduces the execution time significantly.

To evaluate the estimation speed and accuracy of the HMSE methodology, the HMSE estimate results are compared with what is obtained with the Atomium/MH tool. Note that in order to make a fair comparison, the inter-array inplace optimization option in their tool is not used since this option has not been implemented yet for HMSE. For the Cavity Detection algorithm, the HMSE estimates

| | GM parsing/update | Intra- + DRA | MHLA est. | total/ init. HMSE |
|---|---|---|---|---|
| initial HMSE (orig) | 208.5ms | 120.0ms | 1.2ms | 100ms |
| incre. HMSE (LT-1) | 142.0ms | 19.0ms | 1.2ms | 16.7ms |
| incre. HMSE (LT-2) | 140.1ms | 30.1ms | 1.3ms | 25.9ms |
| incre. HMSE (LT-3) | 140.4ms | 19.2ms | 1.2ms | 16.8ms |

**Table 9.3:** Execution time comparison of HMSEs for Cavity Detection Algorithm

are identical to the results that Atomium/MH achieves. This is, because for this small testbench, the estimation result achieved by the HMSE is the same as what Atomium/MH is got. The computation time for the Atomium/MH tool varies between 2s to 10s of the CPU time while the initial HMSE takes about 120ms and the incremental HMSE just takes maximal 30ms. This shows that the initial HMSE is over one order of magnitude faster than Atomium/MH while the incremental HMSE is 4 times faster than the initial HMSE. The speedup is not very significant yet, as the Cavity Detection algorithm is rather small and not much gain can be achieved then.

Based on the Pareto curve outputs, a power estimation has been performed on a number of two layer memory platform configurations for the sequence of loop transformations as shown in Figure 9.6. The power estimation is performed based on the model presented in Chapter 8. The two layer memory platform consists of an on-chip SPM layer and the off-chip main memory. The horizontal dimension refers to the size requirement of the SPM layer and the vertical dimension refers to the total power consumed on the memory platform. As shown, the power estimated on several different memory platform configurations just matches what is represented in the Pareto curves. Further study shows that the results achieved are equivalent to what are achieved with the Atomium/MH tool.

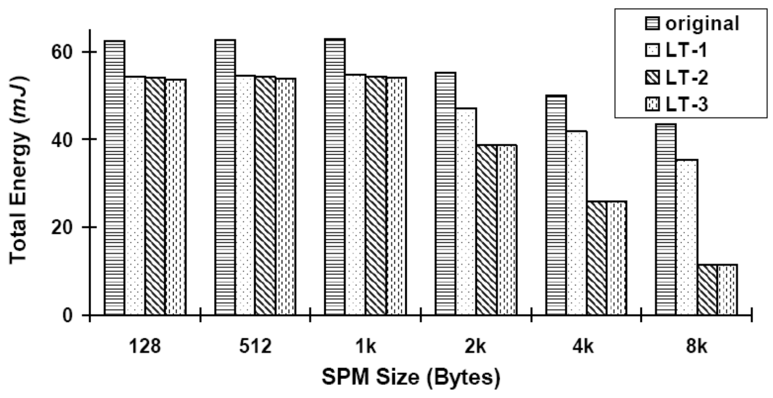


**Figure 9.6:** Power estimation for two layer memory platforms for Cavity Detection Algorithm

## 9.3 Demonstration on QSDPCM Algorithm

The QSDPCM algorithm is an inter-frame compression technique for video images, involving hierarchical motion estimation and a quadtree-based encoding of the motion compensated frame-to-frame differences [130]. Our estimation is performed on a number of code versions resulting from different loop transformations. The QSDPCM algorithm is a medium size application, its core having more than 800 lines of c-code.

### 9.3.1 Intra-array Memory Footprint Estimation

Table 9.4 compares the results of two code version using the memory footprint estimate techniques presented in this dissertation and Atomium/MC. As shown, the vertexes approach takes a few seconds, which is one order of magnitude faster than the ILP approach. It is again significantly faster than the Atomium/MC approach. For this algorithm, the estimated size by the MDV approach is not identical with the size achieved by the Atomium/MC tool. This over-estimate is caused by the fact that the current techniques does not perform an exact analysis when multiple array write references exists for the same array. This was discussed in Chapter 4.

| Application | Declared | MDV approach | | | Atomium/MC | |
|---|---|---|---|---|---|---|
| code | size | est. size | $t_{vertexes}$ | $t_{ILP}$ | est. size | $t_{MC}$ |
| orig | 154019 | 159250 | 6.0s | 58.9s | 151203 | 93.1s |
| LT-7 | 154019 | 96044 | 1.2s | 12.4s | 85128 | 48.0s |

**Table 9.4:** Intra-array memory footprint estimation for QSDPCM Algorithm

## 9.3.2 Inter-array Memory Footprint Estimation

Table 9.5 compares the the inter-array memory footprint estimation between the two Hanoi tower approaches and the Atomium/MC tool. For this algorithm, the improved one layer Hanoi tower approach is at least three orders of magnitude faster than the original one layer Hanoi tower approach. This is achieved with the same loss in estimate accuracy. The estimate accuracy for this algorithm is still within 97% of the initial one layer Hanoi tower approach. The estimate accuracy can be pre-configured before the experiments or interactively defined by the designer during the estimation. It can hence be used to trade off estimation speed. For this algorithm, the estimated size by the Hanoi tower based approaches are not identical with the size achieved by the Atomium/MC tool due to the overestimate for the multiple pair dependencies that occur at the intra-array estimation step. This can however be removed by performing basic set analysis as mentioned.

| Application | Declared | Inter Hanoi tower approach | | | | Atomium/MC | |
|---|---|---|---|---|---|---|---|
| code | size | Initial approach | | Improved approach | | | |
| | | size | time | size | time | size | time |
| orig | 118906 | 118442 | 78.1s | 118670 | 11.9ms | 117338 | 97s |
| LT-7 | 118906 | 94868 | 24.1s | 96044 | 2.0ms | 85128 | 55s |

**Table 9.5:** Inter-array memory footprint estimation comparison for QSDPCM Algorithm

### 9.3.3 HMSE Estimation

Figure 9.7 and Table 9.6 show the Pareto curves output and the execution time comparison for QSDPCM algorithm. For QSDPCM, several versions of loop transformations have been implemented incrementally. The fully transformed code LT-7 is always the best. This means that this version of code will result in optimal memory usage for all possible hierarchy instances. For QSDPCM, the incremental HMSEs take between 0.7% and 15% of the execution time needed for the initial HMSE. This is because the incremental loop transformations are all performed at the outermost two of over 12 loop dimensions and only a limited number of arrays are transformed. Most of the loop transformations performed have no effect at the inner dimensions and the incremental DRA only needs to locally update the transformed arrays at the two outermost dimensions. This significantly reduces the computation time. LT-5 and LT-6 are big transformation steps which affect about 50% of the array accesses. Still, since not all dimensions are transformed, 85% of the time can be saved by performing HMSE incrementally.

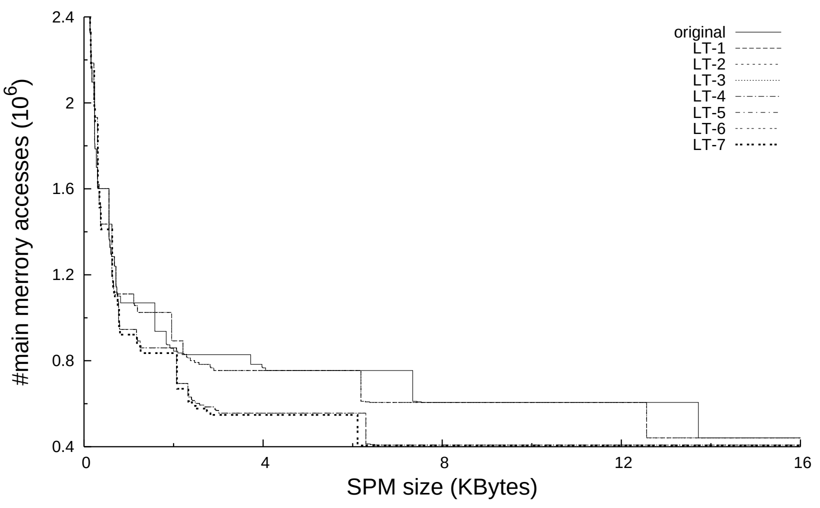


**Figure 9.7:** HMSE output of Pareto curves for QSDPCM

Figure 9.8 shows the comparison of the *#mainmemoryaccesses* of HMSE and Atomium/MH for the QSDPCM algorithm for several realistic two layer memory

| | GM parsing/update | Intra- + DRA | MHLA est. | total/init. HMSE (%) |
|---|---:|---:|---:|---:|
| initial HMSE | 542.7ms | 1627.6ms | 10.2ms | 100ms |
| incre. HMSE (LT-1) | 6.1ms | 9.1ms | 9.3ms | 1.2ms |
| incre. HMSE (LT-2) | 1.6 ms | 2.0ms | 9.5ms | 0.7ms |
| incre. HMSE (LT-3) | 1.5 ms | 2.1ms | 9.3ms | 0.7ms |
| incre. HMSE (LT-4) | 4.7 ms | 8.2ms | 9.0ms | 1.1ms |
| incre. HMSE (LT-5) | 64.3ms | 222.1ms | 11.5ms | 14.3ms |
| incre. HMSE (LT-6) | 61.1ms | 177.9ms | 8.9ms | 11.4ms |
| incre. HMSE (LT-7) | 3.5ms | 9.3ms | 8.9ms | 1.1ms |

**Table 9.6:** Execution time comparison for QSDPCM Algorithm

hierarchy instances. The horizontal dimension means the on-chip SPM layer size of a two layer main memory hierarchy and the vertical dimension presents the number of accesses to the off-chip main memory.

Compared to the Atomium/MH result, the HMSE estimation in general gives close estimates. When the two layer memory hierarchy instances have an SPM layer size between 2K and 8K, HMSE gives some overestimates compared to the Atomium/MH tool. This is because the Atomium/MH tool finds data dependent copies for some arrays that the HMSE does not find. The reason is that the simplified bounding box geometrical model, which is the input of the HMSE, does not model data dependent terms in the index expressions fully accurate.

For the QSDPCM algorithm, Atomium/MH takes a few minutes of the CPU time. In contrast, the initial HMSE takes less than 2s which is two order of magnitude faster than the Atomium/MH tool. The incremental HMSE takes between 12ms and 234ms of the CPU time, which is again between one and two order of magnitude faster than the initial HMSE. It varies depending on what loop transformations are performed and their effect. The computation time is trivial when the loop transforms performed at an instance have very local effect. For the transformation instance LT-5, the transformation effect is fairly global as over one-quarter of statements and two-third of all arrays are transformed. The computation time of it is just 15% of the initial HMSE approach.

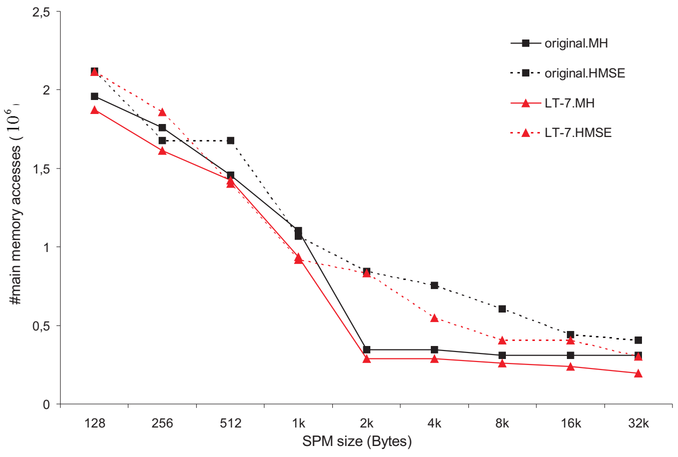


**Figure 9.8:** HMSE and Atomium/MH comparison for QSDPCM

## 9.4 Demonstration on 2D Wavelet Transformation Algorithm

Wavelet-based compression schemes are an important part of modern multimedia codecs and can be exploited to build novel and inherently scalable video codecs(SVC). For these applications, data transfer and storage are the main cost factor in an efficient implementation and loop transformations have significant impact on the global data accesses. In this section, the 2-dimensional (2D) Wavelet transformation algorithm is chosen, with a sequence of loop transformations, to evaluate the estimation methodologies presented in this dissertation. The 2D Wavelet algorithm is another intermediate size application , its core having nearly 1000 lines of c code.

### 9.4.1 Intra-array Memory Footprint Estimation

Table 9.7 shows results of the intra-array memory footprint estimation for the 2D Wavelet Algorithm. As shown, the vertexes approach based estimation takes about two seconds for the two versions of code while the ILP approach based estimation takes over a hundred seconds. The ILP approach is still faster than the Atomium/MC tool. For this application, the size difference between different approaches and the declared approaches is small as there are not much intra-inplace possibilities.

| Application code | Declared size | MDV approach | | | Atomium/MC | |
|---|---|---|---|---|---|---|
| | | est. size | $t_{vertexes}$ | $t_{ILP}$ | est. size | $t_{MC}$ |
| orig | 2739220 | 2723648 | 2.0s | 27.2s | 2721364 | 158s |
| LT-4 | 2739220 | 2723448 | 2.1s | 34.7s | 2721364 | 168s |

**Table 9.7:** Intra-array memory footprint estimation for 2D Wavelet Algorithm

#### 9.4.1.1 Inter-array Memory Footprint Estimation

Table 9.8 compares the the inter-array memory footprint estimation for 2D Wavelet algorithm between the two Hanoi tower approaches and the Atomium/MC tool. For this algorithm, the improved one layer Hanoi tower approach is at the same order of magnitude as the original one layer Hanoi tower approach. Both are between one and two order of magnitude faster than the Atomium/MC tool. For LT-4, the improved approach is even slower than the initial approach. In which, it takes 0.5s to union the towers which dominates the overall time of this approach. Further analysis is required to find out why it comes like this. Possible improvement may be necessary to make this approach valuable if this is not a bug. For this application, there is not much inter-inplace possibilities.

| Application code | Declared size | Inter Hanoi tower approach | | | | Atomium/MC | |
|---|---|---|---|---|---|---|---|
| | | Initial approach | | Improved approach | | | |
| | | size | time | size | time | size | time |
| orig | 2739220 | 2723648 | 0.04s | 2723648 | 0.02s | 2721364 | 6.2s |
| LT-4 | 2739220 | 2723448 | 0.4s | 2723448 | 0.6s | 2721364 | 4.1s |

**Table 9.8:** Inter-array memory footprint estimation comparison for 2D Wavelet Algorithm

## 9.4.2 HMSE Estimation

Figure 9.9 shows the Pareto curves output for the 2D Wavelet transform after having performed initial HMSE and 4 incremental HMSEs, corresponding to four sequences of loop transformations. As shown, the Pareto curves are interleaved with each other. The original code results in the least main memory accesses when the SPM size is not larger than 270K. For sizes smaller than 4K it has 2,000,000 fewer main memory accesses than any of the other versions. This huge amount of main memory accesses are consequently then replaced by SPM accesses with a corresponding boost in performance and reduction in power consumption. Similarly, LT-2, LT-3 and LT-4 have the least number of main memory accesses for certain SPM sizes. LT-1 is not optimal for any SPM sizes, and can be discarded. This demonstrates the importance of performing HMSE in order to find the right versions of code during loop transformation exploration. Without doing so, it can easily end up with a sub-optimal solution.

Table 9.9 compares the execution time required for the initial HMSE and each of the incremental HMSEs. All the 4 incremental HMSEs take less than 30% of the time required by the initial HMSE. The execution time for performing HMSE is also compared to the time needed to read in the GM information for the initial HMSE case and to update GM for incremental HMSE. This GM read-in and updating must in any case be done once during loop transformations and is in fact not part of our iteratively applied HMSE.

Figure 9.10 shows a comparison of the *#main memory accesses* of HMSE and Atomium/MH for 2D Wavelet algorithm on several realistic two layer memory

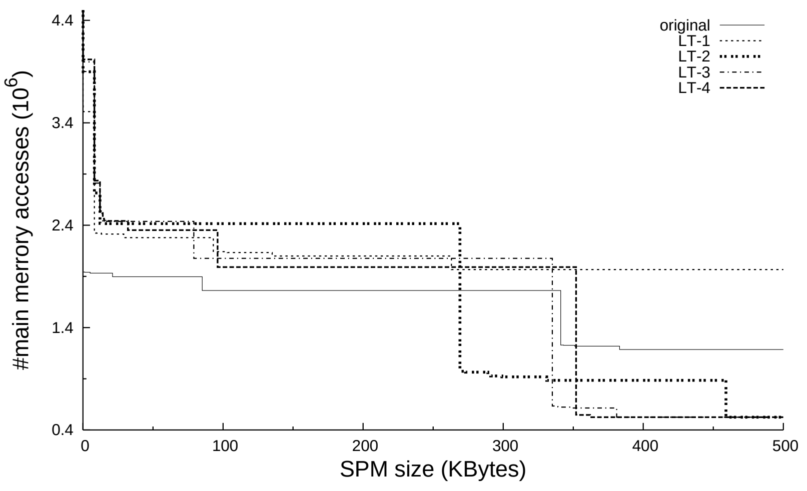


**Figure 9.9:** HMSE output of Pareto curves for 2D Wavelet Algorithm

| | GM parsing/update | Intra- + DRA | MHLA est. | incr./init. HMSE (%) |
|---|---|---|---|---|
| initial HMSE | 208.5ms | 911.1ms | 6.6ms | 100ms |
| incre. HMSE (LT-1) | 142.0ms | 249.4ms | 6.4ms | 27.9ms |
| incre. HMSE (LT-2) | 140.1ms | 253.3ms | 6.4ms | 28.4ms |
| incre. HMSE (LT-3) | 140.4ms | 252.2ms | 6.4ms | 28.2ms |
| incre. HMSE (LT-4) | 140.4ms | 254.7ms | 6.4ms | 28.5ms |

**Table 9.9:** Execution time comparison for 2D Wavelet Transform Algorithm

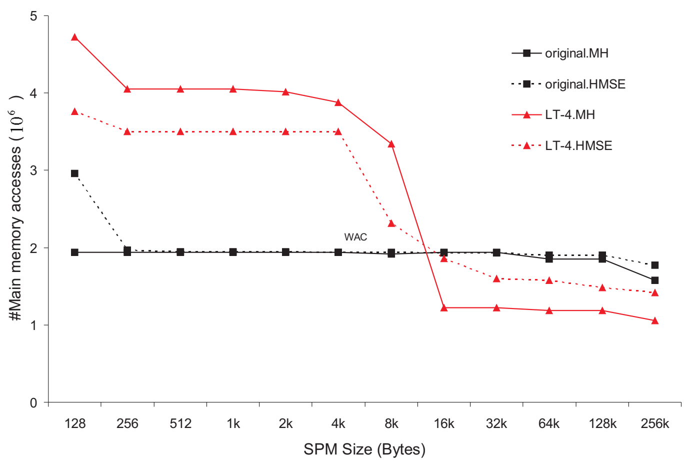


**Figure 9.10:** HMSE and Atomium/MH comparison for 2D Wavelet Algorithm

hierarchy instances. For the original version of the 2D Wavelet code, HMSE produces estimates that are very close to the Atomium/MH tool. For the transformation $LT-4$, there exists estimation difference between HMSE and Atomium/MH. This is partially due to that the different intra-array memory size estimation technquies give different results.

For this application, Atomium/MH takes a few minutes of the CPU time. In contrast, the initial HMSE takes less than 1s of the CPU time which is two order of magnitude faster than the Atomium/MH tool. The incremental HMSE takes around 260ms for all the selected transformation instances presented in the table, which is around one order of magnitude faster than the initial HMSE. For other unlisted transformations,

## 9.5 Experiments on Other Applications

The memory footprint estimation is also performed for several other realistic applications. As shown in Table 9.10, the technique presented in this dissertation present very accurate estimation results compared to what are achieved with Atomium/MC. For all applications, the technique presented in this dissertation is at least three orders of magnitude faster than Atomium/MC. In [11], Balasa et

| Application | Declared | MDV approach | | Atomium/MC | |
|---|---|---|---|---|---|
| code | size | size | time | size | time |
| durbin.c (N=500) | 502000 | 251498 | 3.0ms | 251003 | 70s |
| dynprog.c | 1020001 | 19701 | 1.2ms | 19604 | 7.0s |
| gauss.c | 5280008 | 1436016 | 2.1ms | 1436008 | 6.1s |
| reg_detect.c | 8392 | 4337 | 1.7ms | 4297 | 5.8s |

**Table 9.10:** Estimation comparison for other applications

al. has presented a memory footprint estimation tool, named K2, for exactly the minimum memory footprint requirement, developed at the University of Illinois at Chicago. Their approach is performed based on the polyhedron GM while their previous work has been reviewed in Chapter 2. For their approach, it takes 15s for the Durbin algorithm, 137s for the 2-D gauss blue filter (gauss.c), and 0.8s for the regularity detection algorithm (reg_detect.c). Their approach is also much slower than the technique presented in this dissertation. Their estimation results are not listed here as different parameters are used in their experiments, resulting in different results. In general, their approach can result in smaller memory footprint requirement as their estimate are performed on smaller units, basic set as reviewed in Chapter 2. The minimum memory footprint requirements are difficult to use in practical allocation problems (typically requiring significantly more complex hardware for address generation).

## 9.6 Summary

Experiments on several test vehicles demonstrate that the MFE methodology presented in this dissertation is very fast and gives reasonable estimation result at the early design stage. It is orders of magnitude faster than the compared Atomium/MC. The MFE methodology contains the intra-array estimate using the novel vertexes approach based MDV calculation and the Hanoi tower based inter-array estimation.

For HMSE, the initial HMSE just takes a few seconds for all the experiments. It is also fairly fast, which is two orders of magnitude faster than Atomium/MH. However, it is still too expensive considering the estimation has to be performed interactively over the large number of loop transformation search space exploration. The incremental HMSE can further speed up the estimation, just taking between a few milliseconds to hundred milliseconds for all the experiments. Its computation time varies depending on the effect of the loop transformations performed. But in principle, when the application is larger, usually one instance of the sequence of incremental loop transformations performed will have more local effect. The estimation time will be less ratio of the initial HMSE. This, however, also varies depending on what sequence of loop transformations performed, which again varies depending on the chosen loop transformation algorithm. This makes the incremental HMSE scalable for even larger size applications.

The prototype tools presented here are implemented in Python while the Atomium tool suite is implemented in C++. The prototype tools would be even faster when implemented in C/C++. A fast estimation is critical since it has to be performed frequently during the early system level exploration.

The number for the *#main memory accesses* of HMSE and Atomium/MH presented above can be used to perform a power consumption for a set of memory hierarchy instances as discussed in Chapter 8. This is also illustrated for the Cavity Detection algorithm and the estimated power is comparable to what Atomium/MH gives as the number of accesses to each layer are quite close for them. Further experiments on other applications are left for future work.

# Chapter 10

# Conclusions and Future Work

## 10.1 Conclusions

The work presented in this dissertation has been devoted to the development of fast estimation methodologies. It consists of two major parts, one about fast memory footprint estimation and one about fast hierarchical memory size estimation. At the early design stage, usually a large degree of freedom is present and the system optimization search space is huge. This is the case, for example, at the global loop transformation and control flow transformation stage. It is not realistic to evaluate each alternative through full implementation, since this is too time consuming. On the other hand, due to high-level system descriptions, the early stage transformations are usually steered using abstract cost functions. This can lead to sub-optimal end- products. As an alternative to this, the estimation methodologies presented in this thesis evaluates the later design stage's effect at the early design stage. The estimation is required to be performed interactively during the transformation exploration. They help the designer or automatic tool to find the right transformation instance(s) for a global optimization during the system level search space exploration.

Fast estimation is critical as usually a huge number of transformation possibilities need to be evaluated. To achieve this goal together with achieving reasonable estimation accuracy, a number of advanced techniques have been introduced in this dissertation.

In order to perform a fast memory footprint estimation for an application, we propose to split it into two steps: first to perform intra-array memory footprint estimation as discussed in Chapter 4 and secondly to perform inter-array estimation as discussed in Chapter 5. The intra-array memory footprint estimation is performed at the iteration domain and the memory footprint requirement for an array is defined by the maximal lifetime window which is constrained by the maximal dependency vector. The memory footprint requirement for an array is easily calculated by counting the number of iteration nodes constrained by the maximal dependency vector. This technique is extremely fast as the estimation complexity is independent on the complexity of the data access pattern. Instead it is a linear function of the number of surrounding loop iterators. For fast maximal dependency vector calculation, two approaches have been presented: an ILP formulation and the vertexes based approach. The intra-array estimation techniques have been published in [65].

Inter-array memory footprint estimation is performed based on a lifetime analysis between multiple arrays. Several Hanoi tower approaches have been introduced: initial one layer Hanoi tower approach, multiple layer Hanoi tower approach and improved one layer Hanoi tower approach. The initial approach is fast but the improved approach is faster and scalable for large applications.

The next part of the dissertation presents the hierarchical memory size estimation methodology used to evaluate the loop transformations' effect on both data mapping onto the memory hierarchy and the resulting storage requirement. The methodology is classified as an initial approach and an incremental approach as described in Chapter 6 and 7 respectively. Several advanced techniques have been introduced to achieve a fast estimation, such as bounding box data reuse analysis, platform independent memory hierarchy layer assign estimation with Pareto curve output, and fast memory footprint estimation. In order to further speed up the estimation when it is performed repeatedly, the incremental approach is introduced that performs incremental intra-array memory footprint estimation and incremental data reuse analysis [66]. It exploits the fact that incremental loop transformations usually have very local effect. Locally updating the estimation can then remove redundant recomputation and hence speed up the estimation. The estimation makes it possible to find the most interesting loop transformations which might result in optimal usage of any memory platform selected later. The Pareto curve further allows the designer to perform a fast low power memory platform exploration as represented in Chapter 8. The initial HMSE techniques have been published in [63, 67].

Prototype tools have been implemented that proves the usefulness and efficiencys of the estimation methodologies. As demonstrated on several real life applications in Chapter 9, the estimation is sufficiently accurate for the purpose and at the same time very fast. This shows that the methodologies are very suitable to be used repeatedly at the early system level design space exploration stage.

Several additional journal and conference papers are in progress, based on the material presented in this dissertation. Besides the material presented in this dissertation, the author has also been contributed to other work, partially published in [64, 132, 133, 11].

## 10.2 Future Work

Even though it has been shown that the estimation methodologies are useful in its current version, a number of research topics are open for future work. The implementation of integrating the fast inter-array memory footprint estimation within the hierarchical memory size estimation has not been completed. Further experiments after this is included should also be performed to check its effect. Future work is also required to verify the accuracy of the power estimation and the data memory platform exploration.

The current intra-array memory footprint estimation methodologies are accurate for most cases of practical applications. Overestimation however occurs for some cases which may occur in real life applications. Techniques are introduced to handle this in order to reduce the overestimate, but they have not been implemented. A search for more general solutions is hence still interesting, e.g., for non-rectangular arrays. For inter-array memory footprint estimation, the multiple layer Hanoi tower approach has not been implemented and further investigation is required to evaluate its efficiency.

The current hierarchical memory size estimation methodology targets the uniprocessor case. For future work, the challenge is to apply it also for the multiprocessor case.

# Bibliography

[1] ... in *International Roadmap for Semiconductors, 2000 Update*.

[2] "lp_solve web site." http://lpsolve.sourceforge.net/5.5/.

[3] "Python programming language – official website." http://www.python.org/.

[4] J. Absar and F. Catthoor, "Compiler-based approach for exploiting scratch-pad in presence of irregular array access," in *Proc. 4th ACM/IEEE Design and Test in Europe Conf.*, (Munich, Germany), pp. 1162–1167, Mar. 2005.

[5] A. V. Aho, R. Sethi, and J. D. Ullman, eds., *Compiulers - Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[6] J. R. Allen and K. Kennedy, "Automatic loop interchange," *Proc. of the SIGPLAN'84 Symposium on Compiler Construction, SIGPLAN Notices*, vol. 19, pp. 233–246, June 1984.

[7] S. Amarasinghe, J. Anderson, M. Lam, and C. Tseng, "The SUIF compiler for scalable parallel machines," in *Proc. 7th SIAM Conf. on Parallel Proc. for Scientific Computing*, 1995.

[8] D. F. Bacon, S. L. Grahan, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM computing surveys*, vol. 26, no. 4, pp. 245–420, Dec. 1994.

[9] F. Balasa, F. Catthoor, and H. De Man, "Background memory area estimation for multi-dimensional signal processing systems," *IEEE Trans. on VLSI Systems*, vol. 3, pp. 157–172, June 1995.

[10] F. Balasa, P. G. Kjeldsberg, M. Palkovic, A. Vandecappelle, and F. Catthoor, “Loop transformation methodologies for array-oriented memory management,” in *Proc. IEEE International Conference on Application-Specific Systems, Architectures, and Processors(ASAP’06)*, (Colorado, USA), Sept. 2006.

[11] F. Balasa, P. G. Kjeldsberg, A. Vandecappelle, M. Palkovic, Q. Hu, H. Zhu, and F. Catthoor, “Loop transformation methodologies for the memory management of signal processing applications,” *submitted to Journal of VLSI Signal Processing Systems*, 2007.

[12] F. Balasa, F. Catthoor, and H. De Man, “Practical solutions for counting scalars and dependencies in ATOMIUM – a memory management system for multi-dimensional signal processing,” *IEEE Trans. on Comp. Aided Design*, vol. CAD-16, pp. 133–145, Feb. 1997.

[13] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, “Scratchpad memory: design alternative for cache on-chip memory in embedded systems.,” in *CODES*, pp. 73–78, 2002.

[14] U. Banerjee, *Dependence Analysis for Supercomputing*. Boston, USA: Kluwer Acad. Publ., 1988.

[15] U. Banerjee, “A theroy of loop permutations,” in *Languages and Compilers for Parallel Computing* (D. Gelernter, A. Nicolau, and D. Padua, eds.), (London), Pitman, 1990.

[16] U. Banerjee, *Loop transformation for restructuring compilers: the foundations*. Boston, USA: Kluwer Acad. Publ., 1993.

[17] U. Banerjee, *Loop parallelization*. Boston, USA: Kluwer Acad. Publ., 1994.

[18] A. Barvinok, “A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed,” *Mathematics of Operations Research*, vol. 19, pp. 769–779, Nov. 1994.

[19] C. Bastoul, “Code generation in the polyhedral model is easier than you think,” in *PACT’13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, (Juan-les-Pins), pp. 7–16, september 2004.

[20] L. Benini, A. Macii, and M. Poncino, “Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation,” *IEEE Design and Test of Computers*, vol. 17, pp. 74–85, Apr. 2000.

[21] L. Benini and G. Micheli, “System-level power optimization: techniques and tools,” *ACM Trans. on Design Automation of Electronic Systems*, vol. 5, pp. 115–192, Apr. 2000.

[22] L. Benini, D. Bruni, A. Macii, and E. Macii, “Memory energy minimization by data compression: algorithms, architectures and implementation.,” *IEEE Trans. VLSI Syst.*, vol. 12, no. 3, pp. 255–268, 2004.

[23] L. Benini, A. Ivaldi, A. Macii, and E. Macii, “Block-enabled memory macros: Design space exploration and application-specific tuning.,” in *DATE*, pp. 698–699, 2004.

[24] K. Beyls and E. D‘Hollander, “Reuse distance as a metric for cache behavior,” in *Proc. of the IASTED International Conference on Parallel and Distributed Computing and Systems* (T. Gonzalez, ed.), (Anaheim, California, USA), pp. 617–622, IASTED, Aug. 2001.

[25] P. Briggs, *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, Texas, 24, 1998.

[26] E. Brockmeyer, J. D’Eer, F. Catthoor, N. Busa, P. Lippens, and J. Huisken, “Code transformations for reduced data transfer and storage in low-power realization of DAB synchro core,” in *Proc. IEEE Wsh. on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, (Kos, Greece), pp. 51–60, Oct. 1999.

[27] E. Brockmeyer, M. Miranda, H. Corporaal, and F. Catthoor, “Layer assignment techniques for low energy in multi-layered memory organisations,” in *Proc. 6th ACM/IEEE Design and Test in Europe Conf.*, (Munich, Germany), pp. 1070–1075, Mar. 2003.

[28] E. Brockmeyer, A. Vandecappelle, and F. Catthoor, “Systematic cycle budget versus system power trade-off: a new perspective on system exploration of real-time data-dominated applications,” in *Proc. IEEE Int. Symp. on Low Power Electronics and Design*, (Rapallo, Italy), pp. 137–142, Aug. 2000.

[29] F. Catthoor, K. D. andC. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. Van Achteren, and T. Omnes, *Data Access and Storage Management for Embedded Programmable Processors*. Boston, USA: Kluwer Acad. Publ., 2002.

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

[31] F. Catthoor, M. Janssen, L. Nachtergaele, and H. De Man, "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, pp. 39–50, 1998.

[32] P. Clauss and V. Loechner, "Parametric analysis of polyhedral iteration spaces," *J. of VLSI Signal Processing*, vol. 19, pp. 179–194, July 1998.

[33] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache, "Facilitating the search for compositions of program transformations," in *ACM Intl. Conf. on Supercomputing (ICS'05)*, (Boston, Massachusetts), pp. 151–160, June 2005.

[34] K. Danckaert, *Loop transformations for data transfer and storage reduction on multiprocessor systems.* PhD thesis, ESAT/EE Dept., K.U.Leuven, Leuven, Belgium, May 2001.

[35] K. Danckaert, F. Catthoor, and H. De Man, "A loop transformation approach for combined parallelization and data transfer and storage optimization," in *Proc. ACM Conf. on Par. and Dist. Proc. Techniques and Applications, PDPTA'00*, (Las Vegas NV, USA), pp. 2591–2597, June 2000.

[36] K. Danckaert, F. Catthoor, and H. De Man, "A preprocessing step for global loop transformations for data transfer and storage optimization," in *Proc. Intnl. Conf. on Compilers, Arch. and Synth. for Emb. Sys.*, (San Jose, CA, USA), pp. 34–40, Nov. 2000.

[37] A. Darte and G. Huard, "New complexity results on array contraction and related problems," *Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, vol. 40, pp. 35–55, May 2005.

[38] A. Darte, "On the complexity of loop fusion," *Parallel Computing*, vol. 26, no. 9, pp. 1175–1193, 2000.

[39] A. Darte and Y. Robert, "Affine-by-statement scheduling of uniform and affine loop nests over parametric domains," *Journal of Parallel and Distributed Computing*, vol. 29, pp. 43–59, Aug. 1995.

[40] M. Dasygenis, E. Brockmeyer, E. Durinck, F. Catthoor, D. Soudris, and A. Thanailakis, "A memory hierarchical layer assigning and prefetching technique to overcome the memory performance/energy bottleneck," in *Proc. 8th ACM/IEEE Design and Test in Europe Conf.*, (Munich, Germany), Mar. 2005.

[41] E. De Greef, *Storage Size Reduction for Multimedia Applications*. PhD thesis, ESAT/EE Dept., K.U.Leuven, Leuven, Belgium, Jan. 1998.

[42] E. De Greef, F. Catthoor, and H. De Man, "Array placement for storage size reduction in embedded multimedia systems," in *Proc. Intnl. Conf. on Applic.-Spec. Systems Arch. and Processors*, (Zurich, Switzerland), pp. 66–75, July 1997.

[43] E. De Greef, F. Catthoor, and H. De Man, "Memory size reduction through storage order optimization for embedded parallel multimedia applications," *Elsevier Parallel Computing Journal*, vol. 23, pp. 1811–1837, Dec. 1997.

[44] E. De Greef, F. Catthoor, and H. De Man, "Program transformation strategies for memory size and power reduction of pseudo-regular multimedia subsystems mapped on multi-processor architectures," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 8, pp. 719–723, Oct. 1998.

[45] J.-P. Diguet, S. Wuytack, F. Catthoor, and H. De Man, "Formalized methodology for data reuse exploration in hierarchical memory mappings," in *Proc. IEEE Int. Symp. on Low Power Design*, (Monterey CA), pp. 30–35, IEEE, Aug. 1997.

[46] M. L. Dowling, "Optimum code parallelization using unimodular transformations," *Parallel Computing*, vol. 16, pp. 139–144, 1990.

[47] N. M. et al., "Fusion of loops for parallelism and locality," *IEEE Transactions on Parallel and Distributed Systems*.

[48] S. V. et al., "An access regularity criterion and regularity improvement heuristics for data transfer optimization by global loop transformations," in *Proc. ODES, 2003*.

[49] J. Fabri, "Automatic storage optimization," *ACM SIGPLAN Notices*, vol. 14, pp. 83–91, Aug. 1979.

[50] P. Feautrier, "Array expansion," in *Proc.of the 2nd international conf. on supercomputing*, pp. 429–441, 1988.

[51] P. Feautrier, "Dataflow analysis of array and scalar references," *Intnl. J. of Parallel Programming*, vol. 20, no. 1, pp. 23–52, 1991.

[52] P. Feautrier, "Some efficient solutions to the affine scheduling problems," *Intnl. J. of Parallel Programming*, vol. 21, no. 5, pp. 389–420, 1992.

[53] A. Fraboulet, K. Kodary, and A. Mignotte, “Loop fusion for memory space optimization,” in *Proc. 14th ACM/IEEE Int. Symp. on System Synthesis*, (Montréal, Québec, Canada), pp. 95–100, Oct. 2001.

[54] F. Franssen, L. Nachtergaele, H.Samsom, F.Catthoor, and H. Man, “Control flow optimization for fast system simulation and storage minimization,” in *Proc. 5th ACM/IEEE Design and Test in Europe Conf.*, (Paris, France), pp. 20–24, Feb. 1994.

[55] K. Fukuda, “Frequently asked questions in polyhedral computation,” technical report, Swiss Federal Institute of Technology, Lausanne and Zurich, 2004.

[56] D. Gajski, F. Vahid, S. Narayan, and J.Gong, eds., *Specification and design of embedded systems*. Englewood Cliffs NJ: Prentice Hall, 1994.

[57] C. H. Gebotys and M. I. Elmasry, “Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis,” in *Proc. ACM/IEEE Design Automation Conf.*, (San Jose CA, USA), Nov. 1991.

[58] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, “Simi-automatic composition of loop transformations for deep parallelism and memory hierarchies,” *International Journal of Parallel Programming*, vol. 34, no. 3, pp. 261–317, 2006.

[59] M. Griebl, C. Lengauer, and S. Wetzel, “Code generation in the polytope model,” in *IEEE PACT*, pp. 106–111, 1998.

[60] P. Grun, F. Balasa, and N. Dutt, “Memory size estimation for multimedia applications,” in *Proc. ACM/IEEE Wsh. on Hardware/Software Co-Design(Codes)*, (Seattle WA, USA), pp. 145–149, Mar. 1998.

[61] P. Grun, N. D. Dutt, and A. Nicolau, *Memory architecture exploration for programmable embedded systems*. Boston: Kluwer Academic Publ., 2003.

[62] S. Gupta, M. Miranda, F. Catthoor, and R. Gupta, “Analysis of high-level address code transformations for programmable processors,” in *Proc. 3rd ACM/IEEE Design and Test in Europe Conf.*, (Paris, France), pp. 9–13, Apr. 2000.

[63] Q. Hu, E. Brockmeyer, M. Palkovic, P. G. Kjeldsberg, and F. Catthoor, “Memory hierarchy usage estimation for global loop transformations,” in *Proc. IEEE Norchip Conference*, (Oslo, Norway), pp. 301–304, Nov. 2004.

[64] Q. Hu, M. Palkovic, and P. G. Kjeldsberg, "Memory requirement optimization with loop fusion and loop shifting," in *Proc. 30th Euromicro conference*, (Rennes, France), pp. 272–278, Sept. 2004.

[65] Q. Hu, A. Vandecappelle, P. G. Kjeldsberg, F. Catthoor, and M. Palkovic, "Fast memory footprint estimation based on dependency distance vector calculation," in *Proc. 10th ACM/IEEE Design and Test in Europe Conf.*, (Nice, France), Apr. 2007.

[66] Q. Hu, A. Vandecappelle, P. G. Kjeldsberg, F. Catthoor, and M. Palkovic, "Incremental hierarchical memory size estimation for steering loop transformation exploration," *submitted to ACM Trans. on Design Automation of Electronic Systems*, 2007.

[67] Q. Hu, A. Vandecappelle, M. Palkovic, P. G. Kjeldsberg, E. Brockmeyer, and F. Catthoor, "Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications," in *Proc. 11st Proc. IEEE Asia and South Pacific Design Autom. Conf. (ASPDAC)*, (Yokohama, Japan), pp. 606–611, Jan. 2006.

[68] IMEC, "Atomium web site," 2006. http://www.imec.be/design/atomium/.

[69] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, "Data reuse analysis technique for software-controlled memory hierarchies," in *3rd ACM/IEEE Design and Test in Europe Conf.*, (Paris, France), pp. 202–207, Feb. 2004.

[70] J.M.Janssen, F.Catthoor, and H. Man, "A specification invariant technique for operation cost minimisation in flow-graphs," in *Proc. 7th ACM/IEEE Intnl. Symp. on High-Level Synthesis*, (Niagara-on-the-Lake, Canada), pp. 146–151, May 1994.

[71] J.M.Janssen, F.Catthoor, and H. Man, "A specification invariant technique for regularity improvement between flow-graph clusters," in *Proc. European Design Automation Conf.*, (Paris, France), pp. 138–143, Feb. 1996.

[72] M. Kandemir and A. Choudhary, "Compiler-directed scratch pad memory hierarchy design and management," in *Proc. 38th ACM/IEEE Design Automation Conf.*, (New Orleans, USA), pp. 628–633, June 2002.

[73] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "A compiler-based approach for dynamically managing scratch-pad memories in embedded systems," *IEEE Trans. on Comp.-aided Design*, vol. 23, pp. 243–260, Feb. 2004.

[74] M. Kandemir, O. Ozturk, and M. Karakoy, “Dynamic on-chip memory management for chip multiprocessors,” in *Proc. 2004 Intnl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, (Washington, DC), pp. 14–23, Sept. 2004.

[75] M. Kandemir, J. Ramanujam, and A. Choudhary, “Improving cache locality by a combination of loop and data transformations,” *IEEE Trans. on Computers*, vol. 48, pp. 159–167, Feb. 1999.

[76] W. Kelly and W. Pugh, “A framework for unifying reordering transformations,” Report UMIACS-TR-92-126.1, University of Maryland at College Park, Institute for Advanced Computer Studies, MD, USA, 1993.

[77] W. Kelly, W. Pugh, and E. Rosser, “Code generation for multiple mappings,” Report CS-TR-3457, Dept. of Computer Science, Univ. of Maryland, College Park, MD, 1994.

[78] H. Kim and I.-C. Park, “Array address translation for SDRAM-based video processing applications,” *Electronics Letters*, vol. 35, pp. 1929–1931, Oct. 1999.

[79] P. G. Kjeldsberg, *Storage requirement estimation and optimisation for data-intensive applications*. PhD thesis, Norwegian Univ. of Science and Technology, Trondheim, Norway, Mar. 2001.

[80] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas, “Detection of partially simultaneously alive signals in storage requirement estimation for data-intensive applications,” in *Proc. 38th ACM/IEEE Design Automation Conf.*, (Las Vegas N, USA), pp. 365–370, June 2001.

[81] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas, “Data dependency size estimation for use in memory optimization,” *IEEE Trans. on Comp.-aided Design*, vol. 22, pp. 908–921, July 2003.

[82] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas, “Storage requirement estimation for optimized design of data intensive applications,” *ACM Trans. Design Automation of Electronic Systems*, vol. 9, pp. 133–158, Apr. 2004.

[83] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor, and H. De Man, “Cache conscious data layout organization for conflict miss reduction in embedded multimedia applications,” *IEEE Trans. Computers*, vol. 54, pp. 76–81, Jan. 2005.

[84] D. Kulkarni, K. G. Kumar, A. Basu, and A. Paulraj, "Loop partitioning for distributed memory multiprocessors as unimodular transformations," in *Proc. of the 1991 international conference on supercomputing*, (Cologne, Germany), 1991.

[85] D. Kulkarni and M. Stumm, "Loop and data transformations: A tutorial," Tech. Rep. CSRI-337, Computer Systems Research Inst., Univ. of Toronto, June 1993.

[86] F. Kurdahi and A. Parker, "Real: a program for register allocation," in *Proc. ACM/IEEE Design Automation Conf.*, (Miami FL, USA), June 1987.

[87] V. Lefebvre and P. Feautrier, "Optimizing storage size for static control programs in automatic parallelizers," in *Proc. EuroPar Conf.*, vol. 1300 of *Lecture notes in computer science*, (Passau, Germany), pp. 356–363, Springer Verlag, Aug. 1997.

[88] V. Lefebvre and P. Feautrier, "Automatic storage management for parallel programs," *Parallel computing*, vol. 24, pp. 649–671, 1998.

[89] Z. Li, P. Yew, and C. Zhu, "An efficient data dependence analysis for parallelizing compilers," *IEEE Trans. on parallel distributed systems*, vol. 1, no. 1, pp. 26–34, 1990.

[90] P. Lippens, J. van Meerbergen, A. van der Werf, and W. Verhaegh, "Phideo: a silicon compiler for high speed algorithms," in *Proc. European Design Automation Conf. (EDAC)*, pp. 436–441, Feb. 1991.

[91] P. Lippens, J. van Meerbergen, W. Verhaegh, and A. van der Werf, "Allocation of multiport memories for hierarchical data streams," in *Proc. IEEE Int. Conf. Comp. Aided Design*, (Santa Clara, CA), Nov. 1993.

[92] V. Loechner, "Polylib: A library for manipulating parameterized polyhedra," internal report, Universite Louis Pasteur de Strasbourg, Mar. 1999. I can't find this document!

[93] L. Lu and M. Chen, "Subdomain dependence test for massive parallelism," in *Proc. of the international conference on supercomputing*, (New York, NY), Nov. 1990.

[94] A. Macii, E. Macii, and M. Poncino, "Increasing the locality of memory access patterns by low-overhead hardware address relocation.," in *ISCAS (5)*, pp. 385–388, 2003.

[95] P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig, “Fast, predictable and low energy memory references through architecture-aware compilation.,” in *ASP-DAC*, pp. 4–11, 2004.

[96] D. Maydan, J. Hennessy, and M. Lam, “Efficient and exact data dependency analysis,” in *Proc. of the ACM Sigplan’91 Conference on Programming Language Design and Implementation*, June 1991.

[97] K. McKinley, S. Carr, and C. W. Tsend, “Improving data locality with loop transformations,” *ACM Trans. on Programming Languages and Systems*, vol. 18, no. 4, Jul. 1996.

[98] T. Meeuwen, E.Brockmeyer, and M.Miranda, “High-level data-layout transformations for energy reduction in direct mapped caches,” in *2nd Symp. on Program acceleration by Application-driven and architecture-driven Code Transf.*, Sept. 2002.

[99] Y. Myraoka, *Parallelism exposure and exploitation in porgrams*. PhD thesis, Depart of Computer Science, University of Illinois at Urbana- Champaign, Feb. 1971.

[100] N. Nguyen, A. Dominguez, and R. Barua, “Memory allocation for embedded systems with a compiler-time- unknown scratch-pad size,” in *Proc. ACM Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, (San Francisco, California, USA), pp. 115–125, 2005.

[101] S. Y. Ohm, F. J. Kurdahi, and N. Dutt, “Comprehensive lower bound estimation from behavioral description,” in *Proc. IEEE Int. Conf. Comp. Aided Design*, (San Jose CA, USA), Nov. 1994.

[102] T. J.-F. Omnès, T. Franzetti, and F. Catthoor, “Interactive algorithms for minimizing bandwidth in high throughput telecom and multimedia,” in *Proc. 37th ACM/IEEE Design Automation Conf.*, (Los Angeles, CA), p. 00, June 2000.

[103] M. Palkovic, E. Brockmeyer, P. Vanbroekhoven, H. Corporaal, and F. Catthoor, “Systematic preprocessing of data dependent constructs for embedded systems,” in *15th International Workshop on Integrated Circuit and System Design, Power and Timing Modeling, Optimization and Simulation (PATMOS)*, (Leuven, Belgium), pp. 89–98, Sept. 2005.

[104] M. Palkovic, H. Corporaal, and F. Catthoor, “Global memory optimisation for embedded systems allowed by code duplication,” in *Proc. of the 9th International Workshop on Software and Compilers for Embedded Systems*, (Dallas, Texas, USA), pp. 72–79, Sept. 2005.

[105] P. R. Panda, N. D. Dutt, and A. Nicolau, “Local memory exploration and optimization in embedded systems,” *IEEE Trans. on Comp.-aided Design*, vol. 18, pp. 3–13, Jan. 1999.

[106] P. R. Panda, *Memory optimizations and exploration for embedded systems*. Doctoral dissertation, Dept. of Information and Computer Science, University of California, Irvine, CA, Apr. 1998.

[107] P. R. Panda, N. D. Dutt, and A. Nicolau, “Efficient utilization of scratch-pad memory in embedded processor applications,” in *Proc. 5th ACM/IEEE Europ. Design and Test Conf.*, (Paris, France), pp. 7–11, Mar. 1997.

[108] P. R. Panda, N. D. Dutt, and A. Nicolau, *Memory issues in embedded in systems-on-chip: optimization and exploration*. Boston: Kluwer Academic Publ., 1998.

[109] K. Patel, E. Macii, and M. Poncino, “Energy-performance tradeoffs for the shared memory in multi-processor systems-on-chip.,” in *ISCAS (2)*, pp. 361–364, 2004.

[110] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, “A case for intelligent ram,” *IEEE Micro*, vol. 17, no. 2, pp. 34–4, 1997.

[111] P. G. Paulin and J. P. Knight, “Force-directed scheduling for the behavioral synthesis of asics,” *IEEE Trans. on Comp. Aided Design*, vol. 8, pp. 661–679, June 1989.

[112] P. Pochmuller and M. Glesner, “Memory management for high level synthesis,” in *Proc. 5th Annual IEEE International ASIC Conference and Exhibition*, pp. 166–169, Sept. 1992.

[113] C. D. Polychronopoulos, “Compiler optimizations for enhancing parallelism and their impact on architecture design,” *IEEE Trans. on Computers*, vol. 37, pp. 991–1004, Aug. 1988.

[114] W. Pugh, “A practical algorithm for exact array dependence analysis,” *Communications of the ACM*, vol. 35, pp. 102–114, Aug. 1992.

[115] W. Pugh and D. Wonnacott, “An exact method for analysis of value-based array data dependences,” in *Proc. 6th Intnl. Wsh. on Languages and Compilers for Parallel Computing*, (Portland OR, USA), pp. 546–566, Aug. 1993.

[116] W. Pugh, "The omega test: a fast and practical integer programming algorithm for dependence analysis," in *Supercomputing*, pp. 4–13, Aug. 1991.

[117] W. Pugh and D. Wonnacott, "Nonlinear array dependence analysis," Tech. Rep. UMIACS-TR-94-123, CS-TR-3372, Dept. of Computer Science, Univ. of Maryland, College Park, MD, 1994.

[118] F. Quillere and S. Rajopadhye, "Optimizing memory usage in the polyhedral model," *ACM Trans. on Programming Languages and Systems*, vol. 22, pp. 773–815, Sept. 2000.

[119] J. Rabaey, ed., *Digital integrated circuits: a design perspective*. Englewood Cliffs NJ: Prentice Hall, 1996.

[120] J. Ramanujam, J. Hong, M. Kandemir, and A. Narayan, "Reducing memory requirements of nested loops for embedded systems," in *Proc. 38th ACM/IEEE Design Automation Conf.*, (Las Vegas NV, USA), pp. 359–364, June 2001.

[121] P. Rydland, M. Palkovic, P. G. Kjeldsberg, E. Brockmeyer, and F. Catthoor, "Inter in-place storage size requirement estimation," in *Proc. IEEE Norchip Conference*, (Riga, Latvia), pp. 240–243, Nov. 2003.

[122] H. Schmit and D. Thomas, "Synthesis of application-specific memory designs," *IEEE Trans. on VLSI Systems*, vol. 5, pp. 101–111, Mar. 1997.

[123] A. Schrijver, *Theory of Linear and Integer Programming*. New York: New York, 1986.

[124] P. Sithambaram, A. Macii, and E. Macii, "Exploring the impact of architectural parameters on energy efficiency of application-specific block-enabled srams.," in *ACM Great Lakes Symposium on VLSI*, pp. 377–380, 2005.

[125] S. S. Skiena, *The Algorithm Design Manual*. Springer: Wiley, first edition ed., 1998.

[126] P. Slock, S. Wuytack, F. Catthoor, and G. de Jong, "Fast and extensive system-level memory exploration for ATM applications," in *Proc. 10th ACM/IEEE Int. Symp. on System Synthesis*, pp. 74–81, 1997.

[127] Y. Song, R. Xu, C. Wang, and Z. Li, "Data locality enhancement by memory reduction," in *Proceedings of the 15th international conference on Supercomputing*, pp. 50–64, 2001.

[128] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proc. 5th ACM/IEEE Design and Test in Europe Conf.*, (Paris, France), pp. 409–415, Apr. 2002.

[129] L. Stok and J. Jess, "Foreground memory management in data path synthesis," *Int. Journal on Circuit Theory and Appl.*, vol. 20, pp. 235–255, 1992. interconnect on the datapath.

[130] P. Strobach, "QSDPCM – a new technique in scene adaptive coding," in *Proc. 4th Eur. Signal Processing Conf.*, (Grenoble, France), pp. 1141–1144, Elsevier Publ., Amsterdam, Sept. 1988.

[131] M. Swaaij, F.Franssen, F.Catthoor, and H. Man, *High-level modelling of data and control flow for signal processing systems*. Dordrecht, the Netherland: M.Bayoumi (ed.), Kluwer Academic Publishers, 1994.

[132] B. Thornberg, Q. Hu, M. Palkovic, M. ONils, and P. G. Kjeldsberg, "Polyhedral space generation and memory estimation from interface and memory models of real-time video systems," *Elsevier Journal of Systems and Software*, vol. 79, pp. 231–245, Feb. 2006.

[133] B. Thornberg, M. Palkovic, Q. Hu, L. Olsson, P. G. Kjeldsberg, M. O'Nils, and F. Catthoor, "Bit-width constrained memory hierarchy optimization for real-time video systems," *accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[134] R. Troncon, M. Bruynooghe, G. Janssens, and F. Catthoor, "Storage size reduction by in-place mapping of arrays," in *Proc. 3rd Intnl. Wsh. on Verification, Model Checking, and Abstract Interpretation (VMCAI'02)*, (Venice, Italy), pp. 167–181, Jan. 2002.

[135] C.-J. Tseng and D. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Trans. on Comp. Aided Design*, vol. 5, pp. 379–395, July 1986.

[136] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," in *Proc. ACM Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2003)*.

[137] S. Udayakumaran and R. Barua, "An integrated scratch-pad allocator for affine and non-affine code," in *Proc. 2006 ACM/IEEE Design and Test in Europe Conf.*, (Munich, Germany), pp. 925 – 930, Mar. 2006.

[138] T. Van Achteren, *TBD*. Doctoral dissertation, ESAT/EE Dept., Kath. Univ. Leuven, Belgium, TBD 2001.

[139] T. Van Achteren, F. Catthoor, R. Lauwereins, and H. De Man, “Search space definition and exploration for nonuniform data reuse opportunities in data-dominant applications,” *ACM Trans. on Design Automation of Electronic Systems*, vol. 8, no. 1, pp. 125–139, 2003.

[140] T. Van Achteren, G. Deconinck, F. Catthoor, and R. Lauwereins, “Data reuse exploration techniques for loop-dominated application,” in *Proc. 5rd ACM/IEEE Design and Test in Europe Conf.*, (Paris, France), pp. 428–535, Mar. 2002.

[141] M. van Swaaij, F. Franssen, F. Catthoor, and H. De Man, “Modeling data flow and control flow for high level memory management,” in *Proc. of the European Conference on Design Automation*, (Brussels, Belgium), pp. 8–13, Mar. 1992.

[142] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor, “Transformation to dynamic single assignment using a simple data flow analysis,” in *Proc. 3rd Asian Symp. on Programming Languages and Systems ( ASPLAS)*, vol. 3780 of *Lecture Notes on Comp. Sc.*, (Tsukuba, Japan), pp. 330–346, Nov. 2005.

[143] I. Verbauwhede, C. Scheers, and J. Rabaey, “Memory estimation for high-level synthesis,” in *Proc. 31st ACM/IEEE Design Automation Conf.*, (San Diego CA, USA), pp. 143–148, June 1994.

[144] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor, “Multi-dimensional incremental loop fusion for data locality,” in *Proc. IEEE International Conference on Application-Specific Systems, Architectures, and Processors(ASAP’03)*, (Leiden, The Netherlands), pp. 17–27, June 2003.

[145] S. Verdoolaege, *Incremental loop transformations and enumeration of parameter sets*. PhD thesis, Computer Dept., K.U.Leuven, Leuven, Belgium, Apr. 2005.

[146] D. Wallace, “Dependence of multi-dimensional array references,” in *Proc. of the second international conference on supercomputing*, (St. Malo, France), July 1988.

[147] D. K. Wilde and S. Rajopadhye, “Memory reuse analysis in the polyhedral model,” in *Proc. of the second international Euro-Pad conference, Vol. 1, Aug. 1996, pp.389-97, also in Parallel Processing Letters, Vol. 7, No. 2, June 1997, pp.203-215.*

[148] D. K. Wilde, "A library for doing polyhedral operations," Master's thesis, Oregon State University, Corvallis, OR, Dec. 1993. also Technical Report PI-785, IRISA, Rennes, France.

[149] M. Wilkes, "The memory gap," in *27th annual International Symposium on Computer Architecture, Keynote speech at Workshop on Solving the Memory Wall problem*, (Vancouver BC, Canada), 2000.

[150] S. J. E. Wilton and N. P. Jouppi, "An enhanced access and cycle time model for on-chip caches," tech. rep., Western Research Laboratory, July 1994.

[151] S. J. E. Wilton and N. P. Jouppi, "Cacti: An enhanced cache access and cycle time model," *IEEE J. of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, 1996.

[152] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, pp. 452–471, Oct. 1991.

[153] M. Wolfe, "Data dependence and program restructuring," *J. of Supercomputing*, no. 4, pp. 321–344, 1990.

[154] M. Wolfe, *High Performance Compilers for Parallel Computing*. Redwood City CA, USA: The Addison-Wesley Publishing Company, 1996.

[155] M. J. Wolfe, *Optimizing supercompilers for supercomputers*. PhD thesis, Depart of Computer Science, University of Illinois at Urbana- Champaign, Oct. 1982.

[156] M. Wolfe and C. Tseng, "The power test for data dependence," *IEEE Trans. on parallel distributed systems*, vol. 3, no. 5, pp. 591–601, 1992.

[157] S. Wuytack, J. P. Diguet, F. Catthoor, and H. De Man, "Formalized methodology for data reuse exploration for low-power hierarchical memory mappings," *IEEE Trans. on VLSI Systems*, vol. 6, pp. 529–537, Dec. 1998.

[158] S. Wuytack, F. Catthoor, G. de Jong, and H. De Man, "Minimizing the required memory bandwidth in VLSI system realizations," *IEEE Trans. on VLSI Systems*, vol. 7, pp. 433–441, Dec. 1999.

[159] S. Wuytack, F. Catthoor, G. De Jong, B. Lin, and H. De Man, "Flow graph balancing for minimizing the required memory bandwidth," in *Proc. 9th ACM/IEEE Int. Symp. on System Synthesis*, (La Jolla CA), pp. 127–132, Nov. 1996.

[160] S. Wuytack, F. Catthoor, L. Nachtergaele, and H. De Man, "Power exploration for data dominated video applications," in *Proc. IEEE Intnl. Symp. on Low Power Design*, (Monterey CA), pp. 359–364, Aug. 1996.

[161] S. Wuytack, J. da Silva, F. Catthoor, G. de Jong, and C. Ykman, "Memory management for embedded network applications," *IEEE Trans. on Comp. Aided Design*, vol. CAD-18, pp. 533–544, May 1999.

[162] Y. Zhao and S. Malik, "Exact memory size estimation for array computation without loop unrolling," in *Proc. 36th ACM/IEEE Design Automation Conf.*, (New Orleans, USA), pp. 811–816, June 1999.

[163] H. Zhu, I. I. Luican, and F. Balasa, "Memory size computation for multimedia processing applications," in *Proc. 11st Proc. IEEE Asia and South Pacific Design Autom. Conf. (ASPDAC)*, (Yokohama, Japan), pp. 802–807, Jan. 2006.