UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Tesi di Dottorato in Ingegneria Informatica e Automatica

XXVII Ciclo

Methodologies for automated synthesis of memory and interconnect subsystems in parallel architectures

Author:
Luca GALLO

Supervisor:
Prof. Alessandro CILARDO

Coordinator:
Prof. Franco GAROFALO

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

in the

SecLab Group

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

March 2015

"Vivi come se dovessi morire domani.
Impara come se dovessi vivere per sempre."

Gandhi

UNIVERSITY OF NAPLES *"FEDERICO II"*

Abstract

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Doctor of Philosophy

Methodologies for automated synthesis of memory and interconnect subsystems in parallel architectures

by Luca Gallo

As frequency scaling for single-core computers has today reached practical limits, the only effective source for improving computing performance is parallelism. Consequently, automated design approaches leveraging on parallel programming paradigms, such as OpenMP, appear a viable strategy for designing on-chip systems. At the same time, emerging architectures, such as reconfigurable hardware platforms, can provide unparalleled performance and scalability; in fact they allow to customize memory and communication subsystems based on the application needs. However, current high level synthesis tools often lack enough intelligence to capture those opportunities consequently leaving room for research.

As a first contribution of this thesis, a novel technique based on integer lattices is proposed for tailoring the memory architecture to the application accesses pattern. Data are automatically partitioned across the available memory banks reducing conflicts so as to improve performance. Compared to existing techniques, the rigorous spatial regularity of lattices yields more compact circuits. As a second aspect, in order to sustain the execution parallelism stemming from the presence of multiple memory banks, an efficient interconnection subsystem must be designed. To this aim, the thesis proposes a methodology capable of targeting the interconnect to the traffic profile imposed by the distributed memory. In conclusion, all the proposed techniques are merged together in a comprehensive OpenMP-based design flow, resulting in an automated framework that well fits in the electronic design automation panorama.

Ringraziamenti

Il primo ringraziamento non puo' che essere per l'Ing. Alessandro Cilardo. Alessandro e' per me un insegnante, un collega, ma soprattutto un caro amico. Da lui ho imparato moltissimo tecnicamente, molto di piu' che da chiunque altro finora. Credo che la sua trasparenza, il suo impegno genuino nella ricerca e la sua passione per i dettagli possano essere un vero esempio.

Il secondo ringraziamento va al Dr. David Thomas ed al Prof. George Constantinides, per avermi ospitato presso il gruppo di ricerca Circuits and Systems dell'Imperial College di Londra. Ringrazio inoltre Brian Durwood, C.E.O. di Impulse Accelerated Technologies, per aver fornito software ed hardware necessario per i primi esperimenti.

Nulla di cio' sarebbe stato possibile senza il supporto di tutta la mia famiglia; in particolare Mamma, Alessandro ed Hilde... mi avete dato la serenita' e la tranquillita' necessaria per spingere sempre al massimo. Questo e' un traguardo mio quanto vostro.

Marco e Raffaela, grazie per essere i miei amici piu' cari e quelli che piu' mi hanno accompagnato in questi anni.

Ringrazio infine tutti i miei amici e colleghi che mi sono stati accanto e mi hanno supportato; un ringraziamento particolare e' per Edoardo, con cui ho condiviso innumerevoli momenti divertenti e formativi.

Preface

Some of the research described in this Ph.D. thesis has undergone peer review and has been published in academic journals and conferences. In the following, I list all the papers developed during my research work as Ph.D. student.

- **Improving multibank memory access parallelism with lattice-based partitioning**
  A. Cilardo, L. Gallo
  ACM Transactions on Architecture and Code Optimization (TACO), 2015

- **Exploiting Concurrency for the Automated Synthesis of MPSoC Interconnects**
  A. Cilardo, E. Fusella, L. Gallo, A. Mazzeo
  ACM Transactions on Embedded Computing (TECS), 2015

- **Efficient and scalable OpenMP-based system-level design**
  A. Cilardo, L. Gallo, A. Mazzeo, N. Mazzocca
  Design, Automation Test in Europe Conference Exhibition (DATE), 2013

- **Design space exploration for high-level synthesis of multi-threaded applications**
  A. Cilardo, L. Gallo, N. Mazzocca
  Journal of Systems Architecture (JSA), 2013

- **Automated synthesis of FPGA-based heterogeneous interconnect topologies**
  A. Cilardo, E. Fusella, L. Gallo, A. Mazzeo
  Field Programmable Logic and Applications International Conference (FPL), 2013

- **Joint communication scheduling and interconnect synthesis for FPGA-based many-core systems**

A. Cilardo, E. Fusella, L. Gallo, A. Mazzeo

Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014

- **Automated design space exploration for FPGA-based heterogeneous interconnects**

  A. Cilardo, E. Fusella, L. Gallo, A. Mazzeo, N. Mazzocca

  Design Automation for Embedded Systems (DAES), 2014

- **Area implications of memory partitioning for high-level synthesis on FPGAs**

  L. Gallo, A. Cilardo, D. Thomas, S. Bayliss, GA. Constantinides

  Field Programmable Logic and Applications International Conference (FPL), 2014

- **Generating On-Chip Heterogeneous Systems from High-Level Parallel Code**

  A. Cilardo, L. Gallo

  Digital System Design Euromicro Conference (DSD), 2014

- **Interplay of loop unrolling and multidimensional memory partitioning in HLS**

  A. Cilardo, L. Gallo

  Design, Automation and Test in Europe Conference and Exhibition (DATE), 2015

- **Improving Multi-Bank Memory Access Parallelism with Lattice-based Partitioning**

  A. Cilardo, L. Gallo

  European Network of Excellence on High Performance and Embedded Architecture and Compilation conference (HiPEAC), 2015

Contents

List of Figures

List of Tables

Dedicato a papa'

Chapter 1

Introduction

1.1 Memory at the centre of thinking

Systems designers, whether they use FPGA or ASIC technologies, strive to increase memory bandwidth. Memory architecture has become a prevalent topic to the extent that systems architects, at any level, must tune, tweak and plan systems with the memory at the centre of their thinking. The reason behind has been clear for many years already: GigaHertz clock frequencies and hundreds of processing elements, also known as cores, flood the memory system with requests of billions of bytes per second. Unfortunately, the evolution of electronic systems has undermined the pillars of traditional memory design. Multi-threading and heterogeneity have changed the way memory is accessed, patterns are getting highly unpredictable and general rules that have driven traditional caching strategies are slowly losing their validity when coming to such complex systems. In fact, both of those trends complicate the simple locality of references upon which DRAMs and caches bandwidth depends. On the other hand, emerging computing technologies provide the unprecedented opportunity of tailoring the memory architecture to the application access pattern because they are provided with high degree of flexibility. This customization possibility appears, today, one of the most viable possibilities for tackling the thick memory wall. Field Programmable Gate Arrays (FPGAs), for example, can contain up to thousands of on-chip memory banks that can be accessed simultaneously by parallel computing elements through complex

interconnects. While an off-chip memory is often still required due to capacity requirements, on-chip memory is the highest throughput, lowest latency possible memory in a FPGA-based system. Being made of several physically independent banks, it offers a considerable access parallelism. As a direct consequence, the problem of augmenting memory bandwidth is strictly related to partitioning data wisely among the available banks reducing as much as possible the number of conflicts.

Part of this thesis analyzes the problem of automatic memory partitioning targeting platforms provided with multiple independent memory banks. Specifically, a high level specification of an application is statically analyzed in order to derive the partitioning solution that most boosts bandwidth. In that respect, the thesis, first proposes solutions and algorithms that extend the state of the art and then analyzes the impact of those choices in terms of performance and area.

1.2 Interconnection infrastructures

No matter how perfectly the memory architecture is tailored to the application, a bottleneck in the communication between two components might stall the entire chip [3], even provoking a functional failure. Given the high data rate that the communication subsystems must sustain, they have evolved considerably during the last years. First-generation on-chip interconnects consist of conventional bus and crossbar structures. Buses are mostly wires that interconnect IP cores by means of a centralized arbiter. Examples are AMBA AHB/APB [4] and CoreConnect from IBM (PLB/OPB) [5]. However, the growing demand for high-bandwidth interconnects has led to an increasing interest in multi-layered structures, namely crossbars, such as ARM AMBA AXI [6] and STBus [7]. A crossbar is a communication architecture with multiple buses operating in parallel. It can be full or partial depending on the required connectivity between masters and slaves. Crossbars provide lower latency and higher bandwidth, although this benefit usually comes at non-negligible area costs compared to shared buses. Buses and crossbars are tightly-coupled solutions, in that they require all IP cores to have exactly the same interfaces, both logically and in physical design parameters [8]. Networks on

Chip (NoCs) [9] enable higher levels of scalability by supporting loosely coupled solutions. NoCs are particularly well-suited when targeting reusability, maintainability, and testability as the main design objectives, while bridged buses/crossbar architectures ensure low-power, high-performance, and predictable interconnects.

As in the case of memory architectures, customizing the interconnect according to the applications requirements is the most rewarding solution in terms of performance, area and power/energy. Especially in presence of a partitioned memory subsystem, designing the interconnect analyzing the traffic between processing elements and memory slaves, i.e. banks, is particularly rewarding. In fact, although this topic is of high interest for industrial and academic research, coupling the memory architecture definition and the interconnect design so tightly is what makes the proposed methodologies distinguish from the current literature. When both problems are analyzed together and blended in a single design flow they enable high levels of scalability, area and power efficiency.

1.3 Electronic system level design

The same technological platforms allowing a high degree of customization for memories and interconnects pose a key challenge for today's semiconductor industry: the so-called programmability wall. As they get more and more heterogeneous, fine-tuning programs for performance becomes very complicated, or at least far more difficult than it was when the Von Neumann abstraction was mainstream. Heterogeneous computing refers to systems that use a variety of different computational units: general-purpose processors, special-purpose units, i.e. digital signal processors or the popular graphics processing units (GPUs), co-processors or custom acceleration logic, i.e. application-specific circuits, often implemented on field-programmable gate arrays (FPGAs). Unlike the general-purpose processor case, such computational units act as accelerators and need to be carefully programmed on a per-application basis to yield real performance improvements.

To overcome all those programmability issues, Electronic System Level (ESL) design is an emerging design approach that is essentially focused on raising the design

abstraction levels [10]. It encompasses methodologies and tools for design automation and it is the big picture, the scientific context in which this thesis is placed. ESL design is also defined as concurrent design of hardware and software describing the system at a behavioral level. The algorithmic nature of the system, must be captured by the designer and then formalized into an appropriate specification. The height of the abstraction level at which the specification is offered, creates a big gap between the design phase and the effective synthesis phase primarily because several design choices are possible.

To give a clearer idea of what ESL design is, an example of a C/OpenMP based flow is depicted in figure 1.1.

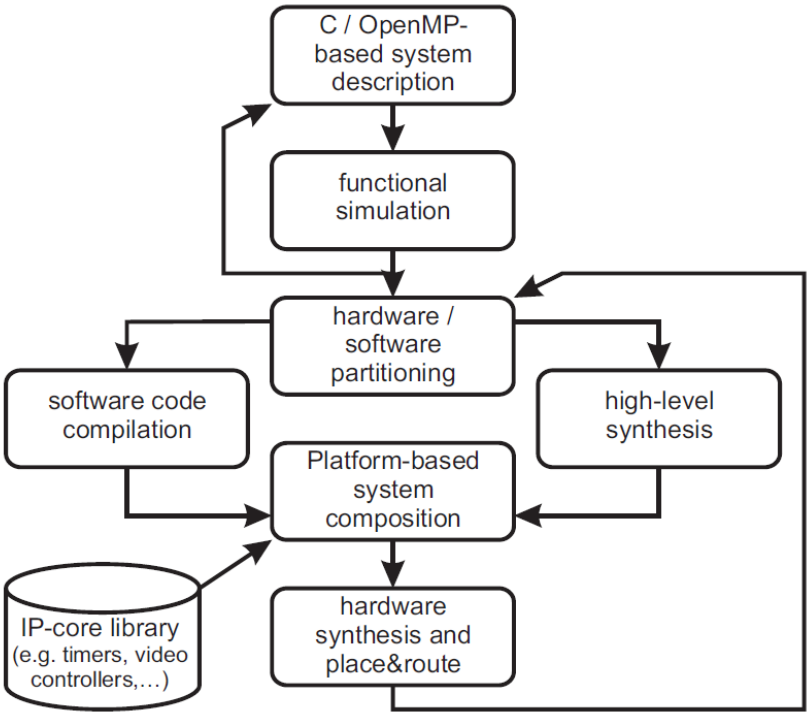


FIGURE 1.1: An example of a ESL design flow

It starts with the description of the system's behavior using an appropriate MoC (Model Of Computation) that in this case is C with OpenMP extensions. The choice of using a widespread programming language as MoC, gives the advantage of a friendly learning curve but lacks any direct formal verification and analysis method. Also, ussing a MoC where a coarse-grained parallelism is identified explicitly at task-level gives hints to the subsequent steps of the flow on how to perform

the Design Space Exploration (DSE), hence limiting the design space while achieving the specified parallelism. Furthermore, using a programming language as MoC enables the designer to immediately have an executable specification of the system ready to give a functional outcome of the specification. For example, in the case of OpenMP, a fully compliant code could be executed on the same workstation where the left steps of the flow are carried out. The hardware/software partitioning, either performed manually or supported by automated tools, corresponds to a bifurcation in the flow. It is during the hardware software partitioning that the DSE procedure takes place. The output is to determine where and how to map functional components (in this case OpenMP threads), as hardware units or software subsystems. In fact, this example flow is highly general because it combines a platform-based methodology (bottom-up) and a top-down path by means of high-level synthesis. Typically, components with strict I/O requirements, are included into the system using pre-synthesized soft-macros in the form of netlists or VHDL code whereas OpenMP threads are subject to high level synthesis. Once the whole system is assembled it can be synthesized using back-end tools and the software can be compiled using appropriate compilers for the target processors.

One of the key enabler of ESL design approaches is High Level Synthesis (HLS). Putting it simply, HLS is a technology transforming an high-level language into a functionally equivalent circuit. HLS has the potential of hiding the implementation details underlying the high-level specification of the application semantics and algorithms, including such low-level aspects as timing, data transfers, and storage [11]. This can greatly reduce design times and verification efforts [12]. In the example flow shown in figure 1.1, HLS in involved for translating the OpenMP threads into hardware circuits.

Since ESL design flows and HLS have to fill a considerable semantic gap, typically from C to gates, they leave open a number of design choices subject to discussion and improvements. While some of them are widely discussed in the existing literature such as pipelining techniques, loop scheduling and tasks mapping, others have been less subject to research even though being equally important for systems performance. One of those issues is, unsurprisingly, memory partitioning. For example, it's widely known that an FPGA has many independent banks and trivial partitioning choices are already adopted by commercial HLS engines, but

targeting the memory architecture to the application profile is a much wider thing as we will see in chapter 2 and 3. One more big issue concerns how the communication subsystem is realized and targeted to the specific application as explained in chapter 4. Later in the thesis in chapter 5, a full ESL design flow is proposed that addresses both those problems with methodologies that compare well against the existing literature.

1.4 Research contributions

This thesis presents contributions enriching the existing literature in the three areas of design automation outlined previously: memory partitioning, interconnections design and ESL design flows. The design of memory and interconnection subsystems are deeply related tasks and tailoring both of them to the application's specific needs enables very high levels of scalability. Using a bottom-up approach, the thesis first examines those two issues and then integrates the results into a comprehensive ESL design flow based on OpenMP.

In chapter 2, a novel memory partitioning strategy is presented that allows to target the memory architecture to the specific application. The application memory accesses are modeled by means of mathematical tools, so as to derive a satisfying partitioning solution formally. The proposed approach is based on the Z-polyhedral mathematical framework as it recognizes in integer lattices a powerful tool for partitioning the application's memory among the existing banks. Geometrically speaking, integer lattices are a superset of hyperplanes that are the state of the art for automatic partitioning in the context of digital synthesis. Beyond naturally enlarging the available solutions space, lattices are provided with a spatial regularity that is directly reflected into more compact datapaths avoiding unpleasant steering logic to connect memory ports and processing elements. This effect is thoroughly analyzed in chapter 3, in which we formalize, for the first time in the literature, the interplay between memory partitioning and area consumption. The adopted benchmarks have shown that the adoption of lattices instead of hyperplanes, can save up to nearly 50% of the area of the synthesized circuit while achieving the same latency. Moreover, in the same chapter, we propose a technique

to reduce the area impact of partitioning using loop unrolling, a well-known code transformation. Recognizing loop unrolling as a way of improving the synthesis in relation to memory partitioning has never been done before to the author's knowledge. Experiments have shown that unrolling, in presence of a partitioned memory, might improve the latency-area product of the synthesized circuit considerably; in fact, an average improvement of roughly 3x has been observed in the analyzed benchmarks.

As briefly explained, chapters 2 and 3 present stand-alone methodologies that can improve current HLS software, but they keep an important hypothesis: the interconnection between processing elements and memory banks is always a partial crossbar. Although this is exactly what happens in current HLS tools, the hypothesis maybe unacceptable for a comprehensive ESL design flow. Sparse crossbars are only a single point in a much wider solutions space. That is the main motivation behind chapter 4 in which we propose novel techniques for customizing the interconnect design according to the application needs. Dependencies among communications between processing elements and memory banks are considered along with the degrees of freedom the application offers in terms of communication scheduling. This enables to perform the architecture design concurrently to communication scheduling yielding efficient systems. The experimental data show a considerable impact in terms of latency-area product and energy consumption. In fact, according to the considered benchmarks, the approach can save up to 70% area if compared to a partial crossbar keeping the performance degradation around 10%, whereas the energy consumption can be up to 40% less than existing similar approaches.

Finally, in chapter 5 we put it all together. We merge the proposed techniques together into an OpenMP-based ESL design flow. The flow also embeds the support for some OpenMP features usually dropped by existing approaches like dynamic scheduling that is vital for heterogeneous systems where load balancing mechanisms are unavoidable. Experiments have demonstrated a 3.25x speedup improvement and a 4x improvement of the clock frequency against existing techniques. In addition, the well-known EPCC benchmarks show that the overhead of our OpenMP support is one order of magnitude less than common implementations for desktop workstations.

Chapter 2

Automatic memory partitioning

2.1 Introduction

This chapter presents in detail one of the main contributions of the thesis, that is a novel memory partitioning technique for high level synthesis. After having introduced the basic theory behind the approach, which relies on integer lattices, we formalize the partitioning problem and propose an algorithmic solution to solve it. The advantages over previous existing techniques are clearly stated using some concrete examples. A step-by-step case study, later in the chapter, clarifies the technique thoroughly.

2.2 The opportunity of customizing the memory architecture

During the last few years, a particularly important trend has clearly emerged in parallel computing architectures, indicating that significant improvements in performance can only be achieved by customizing the computing platform to some extent. In particular, *heterogeneous computing* refers to systems made of a variety of different general- and special-purpose computational units, such as graphics processing units (GPUs) [13], digital signal co-processors [14], and custom accelerators

typically implemented on field-programmable gate arrays (FPGAs) [15]. Many of such advanced computing platforms are provided with several independent memory banks that can be accessed simultaneously by parallel computing elements through complex interconnects. This potentially provides an opportunity for improving the memory bandwidth available to the application. However, in order to take full advantage of the memory architecture, adopting suitable memory partitioning strategies based on the actual application access patterns is of paramount importance.

One of the contributions of this thesis is to propose a methodology for automated memory partitioning in architectures provided with multiple independent memory banks, like FPGAs. The approach encompasses both the problem of bank mapping and the minimization of the total amount of memory required across the partitioned banks, referred to as storage minimization here. Most of the results presented apply to affine static control parts (SCoPs), i.e., code segments in loops where loop bounds, conditionals, and subscripts are affine functions of the surrounding loop iterators and of constant parameters possibly unknown at compile-time. The methodology is based on the Z-polyhedral mathematical framework [16] allowing a compact and comprehensive representation of the code and the related transformations. Furthermore, it adopts a partitioning scheme based on integer lattices, identifying memory banks with different full-rank lattices which constitute a partition of the whole memory. The methodology relies on a method for enumerating the solution space exhaustively and evaluating each solution based on the time overhead caused by conflicting accesses. A technique for representing the transformed program, suitable for existing tools that generate code from Z-polyhedra, is proposed as well. For the storage minimization problem, an optimal approach is adopted, yielding asymptotically zero memory waste or, as an alternative, an efficient approach ensuring arbitrarily small waste. The above techniques are demonstrated through a prototype toolchain relying on a range of software libraries for polyhedral analysis, while the experimental data are collected through high-level synthesis (HLS) of FPGA-based hardware accelerators. Adopting a lattice-based approach to memory partitioning in the context of HLS, the proposed technique improves on a few very recent results in the technical literature that formalize the same problem by means of less powerful mathematical tools,

resulting in narrower solution spaces and thus missing potential solutions. After having explained the technique, extensive comparisons with state-of-the-art proposals concerned with memory partitioning in the context of HLS are presented, showing that lattice-based partitioning can effectively identify a superset of the solutions spanned by existing works.

2.3 A motivational example

The motivational example proposed here is a downsizing algorithm used for image resizing relying on bilinear interpolation [17], where the image is shrunk by a factor of 2 in both dimensions. For the sake of simplicity, we neglect boundary conditions and the source image is supposed to be grayscale. Each pixel in the target image is determined by averaging a 2×2 block of the source image. The kernel consists of a perfect loop nest and comprises only one statement. There are neither loop-carried dependencies nor loop-independent dependencies [18]. The code is thus fully parallel. According to the bilinear interpolation method, each instance of the statement accesses simultaneously four locations of the source image. Figure 2.1.a shows the kernel code while figure 2.1.b highlights the access patterns of a few iterations, where each dotted box corresponds to the locations accessed by the same iteration (i.e., the same value of i and j). Clearly, if the number of memory ports

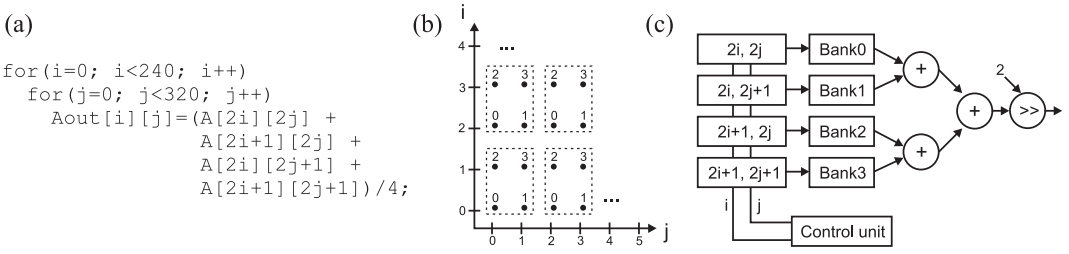


FIGURE 2.1: Motivation example. (a) Kernel code. (b) Lattice-based partitioning. Each dotted box embraces the locations accessed by a specific iteration, i.e., a specific value of i and j, while the numbers associated to each location indicate the memory bank where the location is mapped. (c) Hardware datapath inferred from the memory partitioning solution of part (b).

equaled the number of pixels in the source image, all read operations could virtually be performed in parallel[1]. In contrast, if we have limited memory banks and the array elements are not properly mapped to the available banks, memory conflicts may arise and the inherent parallelism is not fully exploited. A few examples of partitioning approaches include cyclic and a block partitioning strategies [19]. Both first linearize the array. Then, cyclic partitioning assigns a memory location m to memory bank m mod NB, where NB is the number of banks, whereas block partitioning maps m to memory bank $\lfloor m/NB \rfloor$. Assume for example we have only four memory banks available. In fact, with neither of the two approaches, four ports are sufficient to completely avoid conflicts. For instance, if we have a 480×640 image and adopt a cyclic partitioning strategy, iteration $i = 0$, $j = 0$ would simultaneously access both $A[0 \cdot 640 + 1]$ and $A[1 \cdot 640 + 1]$ in the flattened array and a conflict would arise since $1 \bmod 4 = 641 \bmod 4$. In essence, the technique proposed here seeks different approaches for partitioning the memory space such that improved parallelism in memory accesses can be achieved. Figure 2.1.b shows the mapping determined by the *lattice-based partitioning* technique, introduced later. As highlighted by the figure, the memory accesses in the statement can be completely executed in parallel. Figure 2.1.c contains the hardware datapath implementing the example kernel based on the partitioning strategy. Access parallelization can be achieved with the lowest overhead in terms of steering and control circuitry. In fact, the mapping scheme follows a regular pattern and, consequently, a direct connection between the memory banks and the functional units is sufficient to access the required data across all iterations.

2.4 Mathematical background and problem formulation

This section briefly reviews a few mathematical concepts and results that are essential for the formulation of the lattice-based partitioning technique.

[1]The same would apply to write operations, although we ignore them in this example for the sake of simplicity

Given n linearly independent vectors $\vec{b}_0 = (b_{0,0}, \ldots, b_{0,m-1}), \ldots, \vec{b}_{n-1} = (b_{n-1,0}, \ldots, b_{n-1,m-1}) \in \mathbb{R}^m$, a *lattice* $\mathcal{L}$ is a subset of $\mathbb{R}^m$ defined by $\left\{\sum_{j=0}^{n-1} z_j \cdot \vec{b}_j | z_j \in \mathbb{Z}\right\}$. We refer to $\vec{b}_0, \ldots, \vec{b}_{n-1}$ as a *basis* of the lattice [20]. The rank of the lattice is n whereas its dimensionality is m. If $n = m$ the lattice is a full-rank lattice. In particular, if the basis is made of integer points, the lattice is an *integer lattice*. This approach will only deal with integer lattices.

A basis can be also regarded as an $m \times n$ matrix $\mathbf{B} = \{b_{ij}\}$ having the vectors $\vec{b}_j$ as columns. We will say that $\mathbf{B}$ spans a lattice by using the notation $\mathcal{L}(\mathbf{B})$. Two different bases $\mathbf{B} = \{b_{ij}\}$, $\mathbf{C} = \{c_{ij}\}$ span the same lattice $\mathcal{L}$ if and only if there exists a unimodular matrix $\mathbf{U}$ (i.e., a square matrix with integer entries and determinant equal to ± 1) such that $\mathbf{B} = \mathbf{C} \cdot \mathbf{U}$.

Given a basis $\mathbf{B}$, a *fundamental parallelepiped* associated with $\mathbf{B}$ is a set of points in $\mathbb{R}^m$ defined by $FP(\mathbf{B}) = \left\{\sum_{j=0}^{n-1} r_j \vec{b}_j : 0 \leq r_j < 1\right\}$. The set $FP(\mathbf{B})$ is clearly half-open and the points in its translates $FP(\mathbf{B}) + \vec{t}$, with $\vec{t} \in \mathcal{L}(\mathbf{B})$, form a partition of $\mathbb{R}^n$. The *determinant* of a lattice, denoted $\det(\mathcal{L}(\mathbf{B}))$, is the n-dimensional volume of its fundamental parallelepiped and may be regarded as the number of integer points contained in it. For a full-rank lattice, $\det(\mathcal{L}(\mathbf{B})) = |\det(\mathbf{B})|$.

An integer *affine* lattice $\mathcal{L}_t$ is obtained by translating an integer lattice $\mathcal{L}(\mathbf{B})$ by a constant offset $\vec{t} = (t_0, \ldots, t_{m-1}) \in \mathbb{Z}^m$, i.e. $\mathcal{L}_t = \left\{\mathbf{B} \cdot \vec{z} + \vec{t} | \vec{z} \in \mathbb{Z}^n\right\}$.

An *integer polyhedron* P is a subset of vectors in $\mathbb{Z}^m$ satisfying a finite number of affine (in)equalities with integer coefficients: $P = \left\{\vec{z} \in \mathbb{Z}^m | \mathbf{Q} \cdot \vec{z} + \vec{q} \geq \vec{0}\right\}$, where matrix $\mathbf{Q}$ and vector $\vec{q}$ specify the (in)equalities. An integer *polytope* is a bounded integer polyhedron.

By combining multiple affine constraints on integer variables, either equalities or inequalities, by means of logical operators ($\neg$, $\wedge$, and $\vee$) and quantifiers ($\forall$ and $\exists$), we can specify special sets known as *Presburger sets*, which are particularly relevant to this context. Consider an integer polyhedron, $P = \left\{\vec{z} \in \mathbb{Z}^m | \mathbf{Q} \cdot \vec{z} + \vec{q} \geq \vec{0}\right\}$, and an affine function $F(\vec{z}) = \mathbf{F} \cdot \vec{z} + \vec{f}$, where $\mathbf{F}$ is an $n \times m$ matrix. The image of P under F is in the form $F(P) = \left\{\mathbf{F} \cdot \vec{z} + \vec{f} | \mathbf{Q} \cdot \vec{z} + \vec{q} \geq \vec{0}, \vec{z} \in \mathbb{Z}^m\right\}$. These structures are *linearly bound lattices* (LBLs) [21] and can in fact be expressed as Presburger sets.

A *Z-polyhedron* Z is the intersection of a polyhedron $P^c = \left\{\vec{z} \in \mathbb{Z}^m | \mathbf{Q} \cdot \vec{z} + \vec{q} \geq \vec{0}\right\}$ and an integer full dimensional lattice $\mathcal{L} = \{\mathbf{B} \cdot \vec{z}, \vec{z} \in \mathbb{Z}^m\}$: $Z = \mathcal{L} \cap P^c$. A Z-polyhedron can in fact be regarded as an affine image of an integer polyhedron. The hypotheses on the affine function ensure the compliance with LeVerge's conditions [22] for an LBL to be a Z-polyhedron. This representation has been showed to be complete by [23]. We will rely on a number of previous results concerning Z-polyhedra [16]:

- The intersection of two Z-polyhedra is still a Z-polyhedron;
- The union of Z-polyhedra, called a *Z-Domain*, is not a Z-polyhedron in general;
- The difference between two Z-polyhedra is a Z-Domain;
- The preimage of a Z-polyhedron by an affine invertible function is a Z-polyhedron;
- The image of a Z-polyhedron by an arbitrary affine function is an LBL;
- An LBL, and hence a Presburger set, can be expressed as a union of Z-polyhedra [16]. A solution for deriving such union from a generic LBL is given in [24].

(a)

```
for(i=1; i<N; i++){
   parfor(j=1; j<4; j++){
      a[i][j]=a[i-1][j-1]+a[i][j];
   }
}
```

(b) $\Theta = [\,1, 0, 0, 1\,] \cdot [\,i, j, \mathrm{N}, 1\,]^{\mathrm{T}}$

(c)

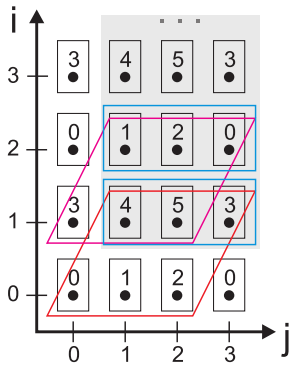


FIGURE 2.2: (a) A simple example of a parallel loop. (b) The schedule function of the statement in the loop body. (c) A representation of the memory space. The (i, j) pairs corresponding to the iteration domain of the loop are emphasized by the gray backgroud. The blue rectangles represent parallel sets of iterations. (d) The red parallelograms represent the memory locations (data sets) accessed by the parallel iterations in part (c).

Z-polyhedra can be used to represent execution information of program loop nests, particularly the so-called affine *Static Control Parts* (SCoPs), having compile-time predictable control flow as well as loop bounds, subscripts, and conditionals expressed as affine functions of the loop iterators, which is common in a wide range of HPC and scientific program kernels, such as image processing and linear algebra operations. Representing SCoP code by means of parametric polyhedra enables a precise and instance-wise representation of the execution information, unlike abstract syntax trees (ASTs) normally used by compilers, as well as the composition of well-known loop transformations in a single step [25]. In fact, some compilers currently embody tools for code manipulation based on the polyhedral abstraction, i.e., GCC Graphite, LLVM Polly, RStream [26]. Following are a few mathematical concepts linking the Z-polyhedral model with the representation of SCoP code.

The *iteration vector* $\vec{v}$ of a given loop nest is a vector having as elements the indices of the surrounding loops along with the parameters and the constant term. As an example, the iteration vector for the loop nest in figure 2.2.a is $(i, j, N, 1)$[2]. While the iteration vector represents an execution instance of the loop nest, the *iteration domain* D_S of a statement S in a loop nest is the set of values $\vec{v}$ satisfying the loop-bound constraints. Because an iteration domain is delimited by affine loop-bound constraints, it can be expressed as $D_S = \left\{\vec{v} : \mathbf{D}_S \cdot \vec{v} \geq \vec{0}\right\}$ for a suitable matrix $\mathbf{D}_S$. Hence, it turns out to be a (parametric) integer polytope, or a Z-polytope in case of non-unit strides.

Each reference to an array A in the loop nest is characterized by a *memory access function* $F(\vec{v}) : \mathbb{Z}^n \longrightarrow \mathbb{Z}^d$, where d is the dimensionality of array A in memory. The access function F associates each value of the iteration vector $\vec{v}$ with a unique cell of array A. Since the subscripts in SCoP code are affine functions, F can always be expressed as $F = \mathbf{F} \cdot \vec{v}$, where $\mathbf{F}$ is a $d \times |\vec{v}|$ matrix. As an example, the first reference to array A in figure 2.2.a has the access function $F(\vec{v}) = \mathbf{F} \cdot \vec{v} = (i - 1, j - 1)$. The set of integer points (memory locations) accessed by a certain reference in a statement S can be regarded as the image of the Z-Polytope D_S, i.e., the iteration domain, by the affine access function F of that memory reference.

[2]The notation $(i, j, \ldots)$ will always denote a column vector.

The result is in general an LBL, hence a union of Z-polyhedra, since we cannot guarantee the invertibility of F.

Although not relevant to this context, polyhedra are also essential for representing data dependencies between statements. In particular, given two statements S_i and S_j and their iteration vectors $\vec{v}$ and $\vec{w}$, respectively, we can build a *dependence polyhedron*, having a dimensionality equal to $|\vec{v}| + |\vec{w}|$, that includes all pairs of instances $\langle \vec{v}, \vec{w} \rangle$ such that $\vec{v} \in D_{S_i}$, $\vec{w} \in D_{S_j}$, and $\vec{w}$ has a dependency on $\vec{v}$.

Each instance $\vec{v}$ of a statement S in a loop nest can be associated to a point in time based on a *schedule function* $\Theta_S(\vec{v})$, which in fact establishes an ordering of the instances of S, possibly based on a parallelizing transformation. The schedule function has the form $\Theta_S(\vec{v}) = \boldsymbol{\Theta} \cdot \vec{v}$, where $\boldsymbol{\Theta}$ is an $n \times |\vec{v}|$ matrix. The parallelism in the code can be exposed by transforming the iteration domain and making the schedule matrix $\boldsymbol{\Theta}$ have a zero-column corresponding to each parallel `for` loop. In figure 2.2.b the schedule has four columns: two for the loop iterators i and j, one for parameter N, and one for the constant term. In this particular example, the outer loop was kept serial, while the inner loop was parallelized. Consequently, the schedule is one-dimensional, i.e., the matrix has one row. The parallelism can be easily recognized by the fact that the schedule has a zero term in the position corresponding to the parallel loop iterator: all instances with the same value of i can be executed in parallel independent of the value taken by j.

To model data-level parallelism, we rely on the concept of parametric *polyhedral slice*, defined for each statement S as follows:

$$P_S\left(\vec{k}\right) = \left\{ \vec{v} : \left(\begin{array}{cc} \mathbf{D}_S & \mathbf{0} \\ -- & -- \\ \boldsymbol{\Theta}_S & -\mathbf{I} \end{array} \right) \left(\begin{array}{c} \vec{v} \\ \vec{k} \end{array} \right) \begin{array}{c} \geq \\ -- \\ = \end{array} \vec{0} \right\}$$

The horizontal line in the above notation separates the inequality constraints, here $\mathbf{D}_S \cdot \vec{v} \geq \vec{0}$, from the equality constraints, here $\boldsymbol{\Theta} \cdot \vec{v} = \vec{k}$. The slice $P_S\left(\vec{k}\right)$ identifies all iteration instances in D_S that correspond to the same schedule value $\vec{k}$. Notice that, being an intersection of Z-polyhedra, $P_S\left(\vec{k}\right)$ is still a Z-polyhedron. Each component k_h of the parameter vector $\vec{k}$ varies within the projection of the domain D_S on the corresponding serial dimension. In the example of figure 2.2, the set

of parallel instances are the points along the planes having i as a constant value. Two of these sets are represented by the blue rectangles in figure 2.2.c. Vector $\vec{k}$ has only one component because we have only one outer sequential dimension corresponding to the i iterator.

Given an array A and a statement S containing a number of memory references, each with affine access function F_h, call $M(\vec{v})$ the set of memory cells of array A accessed by the statement instance $\vec{v}$. Clearly, $M(\vec{v}) = \bigcup_h F_h(\vec{v})$. The definition can be obviously extended to sets of instances. In particular, $M\left(P_S\left(\vec{k}\right)\right) = \bigcup_h \text{Image}_{F_h}\left(P_S\left(\vec{k}\right)\right)$ is the set of the memory cells referenced by the parallel iterations in the parametric slice $P_S\left(\vec{k}\right)$. We call $M\left(P_S\left(\vec{k}\right)\right)$ a *data set*. For a certain statement S, the data set is a function of the parameter $\vec{k}$ and includes all memory locations that can be potentially accessed simultaneously according to the given schedule. Notice that, although $P_S\left(\vec{k}\right)$ is a Z-polyhedron, $M\left(P_S\left(\vec{k}\right)\right)$ is not necessarily a Z-polyhedron for two different reasons. First, the union of a finite number of Z-polyhedra is not generally a Z-polyhedron. Second, the image of a Z-polyhedron is not necessarily a Z-polyhedron, since the access functions may be non-invertible. In general, $M\left(P_S\left(\vec{k}\right)\right)$ is an LBL, which is formally equivalent to a union of Z-polyhedra, as remarked above. Figure 2.2.d depicts two data sets, represented by two red parallelograms, corresponding to the parametric slices $i = 1$ and $i = 2$, shown in figure 2.2.c.

The presence of multiple statements affects the calculation of the data sets. In particular, some of the statements may have the same schedule function and, hence, be run in parallel. The above formulation could be extended by defining multiple data sets, each being the union of the data sets of single statements, having the same schedule. For the sake of simplicity, however, in the following we will refer to the case of a single statement.

The parallel data sets defined above can be expressed as finite unions of Z-polyhedra, allowing a closed formulation of the memory partitioning problem. Furthermore, they can be easily manipulated by existing polyhedral tools, such as [27], [24] and [28].

2.4.1 Problem formulation

Although all the statement instances in a parametric slice $P_S\left(\vec{k}\right)$ are scheduled in parallel, they normally access the same data structure in memory. Accesses that conflict on the same memory port may cause parallel instances to be serialized, introducing a considerable performance bottleneck. Ideally, if the memory locations accessed by the iterations within the same parametric slice (described by the $M\left(P_S\left(\vec{k}\right)\right)$ set) are mapped to independent physical banks, then full parallelization can be achieved.

Consider again figure 2.2. Each point, i.e., each memory location of array A, is labelled with the identifier of the bank where the location is mapped. For instance, $A[3][1]$ is mapped to memory bank 4. As shown in the figure, six different banks are used and the mapping is such that there never are two equal labels in each red parallelogram of figure 2.2.d, i.e., full access parallelization is achieved.

To generalize the above reasoning, we introduce the concept of *conflict count*, denoted $MC\left(\vec{k}\right)$, identifying the maximum number of distinct memory locations in $M\left(P_S\left(\vec{k}\right)\right)$ mapped to the same bank, as a function of $\vec{k}$. In essence, $MC\left(\vec{k}\right)$ represents the number of *serialized* distinct memory accesses in slice $P_S\left(\vec{k}\right)$, which is indicative of the time spent for handling memory references. Notice also that the definition of $MC\left(\vec{k}\right)$ only refers to distinct memory accesses. In fact, concurrently scheduled accesses might also include multiple references to the same location, possibly due to more than one parallel statement being executed. Notice that, since we assume that the code is correctly parallelized, concurrent conflicting accesses cannot include more than one write operation, while the semantic of concurrent write and read operations assumes that the read access gets the old value held by the accessed location, which is consistent with the physical meaning of the concurrent accesses (e.g., in an FPGA memory bank). On the other hand, multiple read operations are simply handled by broadcasting the value to the different operators requesting it (e.g., FPGA-implemented processing blocks).

Based on the definition of conflict count, we can introduce a cost function, simply defined as the summation of values $MC\left(\vec{k}\right)$ across all the values of $\vec{k}$:

$$C_{\text{time}} = \sum_{\vec{k}} MC\left(\vec{k}\right) \tag{2.1}$$

The above function is representative of the overall time cost incurred for memory operations across the execution of the entire kernel. As we will show in the following section, it can be expressed in a closed mathematical form with lattice-based memory partitioning. In particular, the Z-polyhedral theory provides useful results for counting the points in unions of parametric Z-polyhedra. Based on such results, the conflict count turns out to be a piecewise quasi-polynomial function of the parameter $\vec{k}$ [24]. In case of multiple statements, as explained above, the data set is indeed a list of sets, each parameterized in $\vec{k}$. In this case, the cost functions C_{time} can still be calculated for every data set separately and then summed over all the sets of the list in order to evaluate the overall cost, since any two accesses of two different data sets are executed sequentially.

For a given number of physical memory banks NB, each of size $SIZE_i$, the memory partitioning problem can be formulated as follows. A *memory partition* of an array A is a pair of scalar integer functions $g\left(\vec{m}\right), f\left(\vec{m}\right)$. Vector $\vec{m}$ has as many components as the dimensionality of A. It represents a memory location and varies in the integer parallelepiped defined by A. Function $g\left(\vec{m}\right)$ identifies the bank to which $\vec{m}$ is mapped, while $f\left(\vec{m}\right)$ is the (linear) address in that bank. Clearly, we must have $0 \leq f\left(\vec{m}\right) < SIZE_{g(\vec{m})}$, while the total amount of memory taken by the arrays across the physical banks is

$$C_{\text{size}} = \sum_{0 \leq i < NB} \max_{g(\vec{m})=i} \left\{f\left(\vec{m}\right)\right\} \tag{2.2}$$

The memory partitioning problem can be decomposed into

- a *bank mapping problem* which consists in finding a suitable function $g\left(\vec{m}\right)$ that assigns all used locations to existing banks (i.e., $0 \leq g\left(\vec{m}\right) < NB$) and minimizes C_{time};

- a *storage minimization problem* which consists in finding a suitable function $f(\vec{m})$ that avoids colliding assignments to the same bank (i.e., $\forall \vec{m} \neq \vec{n}, g(\vec{m}) = g(\vec{n}) \implies f(\vec{m}) \neq f(\vec{n})$) and minimizes the total amount of memory C_{size}.

In the ideal case, C_{size} coincides with the number of locations in array A, essentially meaning that the partitioned arrays are perfectly exploited with no holes in memory allocation.

The two problems above are tackled separately by two sequential steps. Since we aim to maximize parallelism, we determine the bank mapping as a first step by using the proposed lattice-based partitioning technique. Then, the other problem is solved by an optimal storage minimization approach described later.

2.5 Lattice-based memory partitioning

The approach presented here aims to automatically define efficient partitioning choices minimizing structural conflicts on the memory ports. The proposed methodology assumes that the code is already parallelized by properly rearranging the loops [29–31], which corresponds to having zero columns in the schedule matrix, through a preliminary code transformation step based on existing polyhedral tools [24, 27, 28]

In essence, the *lattice-based memory partitioning* proposed technique

- regards a d-dimensional memory array as a polyhedron, precisely a hyper-rectangle;
- partitions the hyper-rectangle into separate sets. The sets are Z-polyhedra delimited by the same polyhedron, i.e. the memory array, but having different underlying affine lattices, obtained as translates [32] of a particular lattice chosen by the methodology so as to minimize the conflict count;
- generates the new polyhedral representation of the code featuring optimized memory accesses.

As an example, in figure 2.2 each set of locations assigned to the same bank forms an integer lattice. Each lattice can be thought of as a translate of the lattice marked with 0, that we denote $\mathcal{L}_0$. Mathematically, $\mathcal{L}_0$ is a full-rank bidimensional lattice spanned by the basis $\mathbf{B}_0 = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$. The remaining lattices are affine lattices. For instance, the set of integer points $\mathcal{L}_1$ can be obtained by translating $\mathcal{L}_0$ by one position along the j dimension: $\mathcal{L}_1 = \mathcal{L}_0 + (1, 0)$. Similarly, $\mathcal{L}_3 = \mathcal{L}_0 + (0, 1)$.

In general, we define the lattice containing the origin the *fundamental lattice*. As implied by the results summarized in section 2.4, it forms a partition of $\mathbb{Z}^d$ along with its translates. The number of distinct translates, including the fundamental lattice itself, is equal to the determinant of the lattice [32]. Since lattice-based partitioning maps each memory bank to a different translate, the determinant of the fundamental lattice must be equal to the number of actually used banks to fully cover the d-dimensional memory. This establishes a fundamental link between a physical characteristic, the number of memory banks, and the main mathematical object handled by this technique, the lattices. The problem of bank mapping, i.e., determining the $g(\vec{m})$ function defined above, directly corresponds to finding the best fundamental lattice.

2.5.1 Generation of the solution space

As implied by the previous remark, the total number of available memory banks NB is an upper bound to the determinant of the fundamental lattice to choose. We carry out an exhaustive search of all candidate lattices based on the ideas presented in [33], considering all lattices having a determinant less than or equal to NB. Among the solutions reaching the minimum conflict count, we then choose the one requiring the least number of banks, i.e. the least determinant. Two solutions ensuring the minimum conflict count and the same number of banks are deemed equivalent, although they may have different second-order implications on the implementation costs. In chapter 3, we will look at those implications.

Since d-dimensional lattices are spanned by full-rank $d \times d$ matrices, we can equivalently generate all matrices corresponding to distinct lattices. In that respect, given a matrix $\mathbf{B}$ of rank d in $\mathbb{Z}^{d \times d}$, there exists a unique lower triangular matrix

$\mathbf{H} = \{h_{ij}\}$ and a unimodular matrix $\mathbf{U}$ such that $\mathbf{H} = \mathbf{B} \cdot \mathbf{U}$ and $0 \leq h_{ij} < h_{ii}$ for all $j < i$. The matrix $\mathbf{H}$ is called the *Hermite Normal Form* (HNF) of $\mathbf{B}$ [34]. Consequently, we can generate all integer lattices of a given determinant δ by enumerating all distinct lower triangular matrices $\mathbf{H} = \{h_{ij}\}$ such that $\det(\mathbf{H}) = \delta$ and $0 \leq h_{ij} < h_{ii}$. For a given rank d and determinant δ, the number of such matrices, denoted $H_d(\delta)$, is equal to $\prod_{j=1}^{d-1} (p^{k+j} - 1)/(p^j - 1)$, when $\delta = p^k$ for a prime p, while $H_d(\delta) = H_d(p)H_d(q)$, if $\delta = p \cdot q$, where p and q are relatively prime numbers [35]. For a maximum number of banks NB, thus, the search space contains $\sum_{i=2}^{NB} H_d(i)$ solutions. As an example, the fundamental lattice of figure 2.2 is one of 32 potential solutions, since we have a 2-dimensional array and 6 banks. In case we had 8, 12, 16, 24, or 32 banks, the number of candidates would become 55, 116, 209, 480, 846, respectively.

2.5.2 Evaluation of the solutions

Lattice-based partitioning enables the conflict count $MC\left(\vec{k}\right)$ to be expressed in a closed mathematical form. Call $\mathcal{L}_t$, the tth translate of a given fundamental lattice $\mathcal{L}$, with $0 \leq t < \det(\mathcal{L})$. Then, we simply have

$$MC\left(\vec{k}\right) = \max_t \left| \mathcal{L}_t \cap M\left(P_S\left(\vec{k}\right)\right) \right|$$

The intersection picks the points of the data set M that belongs to translate $\mathcal{L}_t$. For each value of $\vec{k}$, the lattice incurring the maximum conflict count determines the worst-case serialization in the memory accesses. The set $\mathcal{L}_t \cap M\left(P_S\left(\vec{k}\right)\right)$ is a union of parametric Z-polyhedra. We first use the technique proposed in [24] to count the integer points in the set as a function of $\vec{k}$, obtaining different expressions for determined intervals of $\vec{k}$. Then, by summing such expressions together, we obtain the overall value of the cost function $C_{\text{time}} = \sum_{\vec{k}} MC\left(\vec{k}\right)$. As an example, in figure 2.2 we have $\left| \mathcal{L}_t \cap M\left(P_S\left(\vec{k}\right)\right) \right| = 1$ for each of the six translates t, and hence $C_{\text{time}} = N - 1$.

2.5.3 Enumeration of the translates

The evaluation of the solutions requires the exhaustive enumeration of the translates of a given fundamental lattice. To this aim, we rely on the following property, which can be easily proven.

Property 1. Let $\mathbf{H} = \{h_{ij}\}$ be the HNF basis of a lattice $\mathcal{L}$ and $\vec{m}$ be one of its points. Then, there does not exist any point $\vec{n} \neq \vec{m}$ such that $\forall i \ 0 \leq n_i - m_i < h_{ii} \wedge \vec{n} \in \mathcal{L}$.

The property implies that if we take a point in $\mathcal{L}$, we cannot find any different point of $\mathcal{L}$ in the parallelepiped having $\vec{m}$ as the lower corner and the diagonal elements h_{ii} as heights. In fact, notice that the parallelepiped has a volume equal to the determinant of $\mathcal{L}$ because $\mathbf{H}$ is in HNF. As a direct consequence, considering $\vec{m} = \vec{0}$, if we take all the elements in the parallelepiped having the diagonal elements h_{ii} as heights, we pick exactly $|\det(\mathbf{H})|$ points that belong to different translates. Since there are exactly $|\det(\mathbf{H})|$ translates, those points cover all the possible translation vectors. As an example, the HNF of the fundamental lattice $\mathcal{L}_0$ in figure 2.2 is $\begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$, so the remaining five translates are given by $\mathcal{L}_0 + \vec{t}$, with $\vec{t} = (1,0), (2,0), (0,1), (1,1), (2,1)$.

2.5.4 Generation of the new polyhedral representation

After picking the best partitioning solution, a new version of the code must be generated so as to capture the allocation to memory banks. A possibility is to use an additional dimension in the partitioned array and then apply block or cyclic mapping. Notice however that we are mainly interested in the synthesis of hardware accelerators from high-level C code. While existing high-level synthesis tools [36] allow block or cyclic partitioning along a given dimension, they look at statements, not *instances* of statements. In other words, they parallelize two memory accesses, say M_1 and M_2, only if it can be proved that all the instances of M_1 and M_2 in the iteration domain of the loop nest access different banks. To address this problem, we decided to explicitly partition the array in the code by declaring multiple distinct arrays. Consider the example in figure 2.2. The

statement in the loop body contains two memory references, or accesses, i.e., $A[i][j]$ and $A[i-1][j-1]$. Different values of the iterators (j, i) lead to a different bank accessed for each of the two references. For example, instance $(1, 1)$ accesses banks 4 and 0, while instance $(2, 1)$ accesses 5 and 1. Indeed, since the dataset has a constant shape and the banks in the example are periodic with period 3 along the j dimension and 2 along the i dimension, there are exactly 6 different cases, i.e., the number of banks NB, each corresponding to a different statement body:

`Y[i][j] = A0[i-1][j-1] + A4[i][j];` when $(j \bmod 3, i \bmod 2) = (1, 1)$

`Y[i][j] = A1[i-1][j-1] + A5[i][j];` when $(j \bmod 3, i \bmod 2) = (2, 1)$

`Y[i][j] = A2[i-1][j-1] + A3[i][j];` when $(j \bmod 3, i \bmod 2) = (0, 1)$

`Y[i][j] = A3[i-1][j-1] + A1[i][j];` when $(j \bmod 3, i \bmod 2) = (1, 0)$

`Y[i][j] = A4[i-1][j-1] + A2[i][j];` when $(j \bmod 3, i \bmod 2) = (2, 0)$

`Y[i][j] = A5[i-1][j-1] + A0[i][j];` when $(j \bmod 3, i \bmod 2) = (0, 0)$

In general, depending on the structure of the dataset, there may be up to NB^h different combinations of bank accesses, where h is the number of memory references in the statement. The essential idea in the approach is to generate a different statement for each combination of bank accesses, e.g. six statements in the above example. In order to describe the new statements in terms of polyhedral representation, we start from the original iteration domain D_S and generate NB smaller integer sets, each corresponding to a different statement body, covering a specific combination of bank accesses. To express this mathematically, further constraints must be added to the algebraic form describing the original polyhedron. Call r the number of distinct memory references in the statement and let d be the dimensionality of the array, e.g., $d = 2$ for bidimensional arrays. For each combination p of bank accesses, the iteration domain of the corresponding statement can be expressed as follows:

$$D_{Sp} = \vec{v} : \exists \ (\vec{y}_0 \dots \vec{y}_{r-1}) \in \mathbb{Z}^{r \cdot d} :$$

$$\begin{pmatrix} \mathbf{D}_S & \mathbf{0} & \mathbf{0}\ldots & \mathbf{0} \\ -- & -- & -- & -- \\ \mathbf{F}_0 & -\mathbf{B} & \mathbf{0}\ldots & \mathbf{0} \\ \mathbf{F}_1 & \mathbf{0} & -\mathbf{B}\ldots & \mathbf{0} \\ \ldots & \ldots & \ldots & \ldots \\ \mathbf{F}_{r-1} & \mathbf{0} & \mathbf{0}\ldots & -\mathbf{B} \end{pmatrix} \begin{pmatrix} \vec{v} \\ -- \\ \vec{y}_0 \\ \ldots \\ \vec{y}_{r-1} \end{pmatrix} \begin{matrix} \geq \\ -- \\ = \end{matrix} \begin{pmatrix} \vec{0} \\ -- \\ \vec{T}_0^p \\ \ldots \\ \vec{T}_{r-1}^p \end{pmatrix}$$

The matrix has as many columns as the original polyhedron matrix $\mathbf{D}_S$ plus the number of accesses r times the lattice rank d. $\mathbf{B}$ is the $d \times d$ basis of the chosen fundamental lattice. $\mathbf{F}_i$ are the $d \times |\vec{v}|$ access matrices in the original code. The r vectors $\vec{y}_i$ contains each d integer parameters. $\vec{T}_i^p$ is a d-element vector denoting the translate, i.e., the bank, where the ith access is mapped in combination p. The set of vectors $\vec{T}_i^p$ identifies one of the possible combinations of bank accesses. Each line in the bottom part of the above matrix multiplication results in a constraint looking like $\mathbf{F}_i \cdot \vec{v} - \mathbf{B} \cdot \vec{y}_i = \vec{T}_i^p$. For instance, concerning the example in figure 2.2, the constraint $\exists\ (y_1, y_2) \in \mathbb{Z}^2 : (j, i) - \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix} (y_1, y_2) = (1, 0)$ for access $A[i][j]$ picks the iterations (j, i) that access locations belonging to translate $(1, 0)$ of the fundamental lattice $\begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$. Notice that $\vec{T}_i^p$ cannot be made parameters since we need to pass separate algebraic structures to the code generator, one for each version of the statement. The above structures can be handled by existing tools, e.g., ISL, CLoog, for code generation from the polyhedral model which can identify and efficiently prune out empty polyhedra. Notice also that, due to the potentially non-invertible nature of the affine access functions, the sets D_{Sp} can be unions of Z-polyhedra rather than single Z-polyhedra. However, in case the access functions are invertible, D_{Sp} can be proved to be a single Z-polyhedron, delimited by the same polyhedron as D_S, yielding a representation that can be manipulated even easier.

2.5.5 Storage minimization

The problem of storage minimization, i.e., the definition of the $f(\vec{m})$ function, consists in assigning a new address within the new bank to each original memory location $\vec{m}$. This corresponds to determining new access functions for each memory reference such that the proper location is accessed in each iteration. Call $F(\vec{v})$ the access function corresponding to the iteration vector $\vec{v}$ for a certain reference and let $\vec{m} = (m_0, \ldots, m_{d-1})$ be the accessed location, i.e., $\vec{m} = F(\vec{v})$. Let $\vec{m}' = \left(m'_0, \ldots, m'_{d-1}\right) = F'(\vec{v})$ be the new access function. Lattice-based partitioning allows a straightforward solution to the storage minimization problem. In fact, because of the cyclic repetitions along the axes of the memory space, which occurs for any lattice chosen, the address in each bank can simply be obtained by scaling the original addresses. Precisely, given $\mathbf{H} = \{h_{ij}\}$, the HNF basis of the fundamental lattice, the new address $\vec{m}'$ can simply be obtained as $\vec{m}' = \left(\left\lfloor \frac{m_0}{h_{00}} \right\rfloor, \ldots, \left\lfloor \frac{m_{d-1}}{h_{d-1\,d-1}} \right\rfloor\right)$. In other words, the new access can be written as $\vec{m}' = \lfloor \mathbf{S} \cdot \mathbf{F} \cdot \vec{v} \rfloor$, where

$$\mathbf{S} = \begin{pmatrix} \frac{1}{h_{00}} & 0 & 0 & \ldots & 0 \\ 0 & \frac{1}{h_{11}} & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & \frac{1}{h_{d-1\,d-1}} \end{pmatrix}$$

The scalar function $f(\vec{m})$ is then given by the linearized address of $\vec{m}'$ in bank $g(\vec{m})$. It can be easily shown that the above solution is consistent and asymptotically optimum. In fact, by Property 1, if $\vec{m}$ and $\vec{n}$ are two distinct locations belonging to the same lattice, then there must be at least one i such that $|m_i - n_i| \geq h_{ii}$. Consequently, they are certainly mapped to different locations in the same bank, since $\left|\left\lfloor \frac{m_i}{h_{ii}} \right\rfloor - \left\lfloor \frac{n_i}{h_{ii}} \right\rfloor\right| \geq 1$. Furthermore, calling D_i the size of the original array along dimension i, the size of each scaled array along the same dimension is no larger than $\left\lceil \frac{D_i}{h_{ii}} \right\rceil$. In other words, each array will require no more than $\prod_i \left\lceil \frac{D_i}{h_{ii}} \right\rceil$ locations. Since the number of banks is $NB = \prod_i h_{ii}$, then the total number of locations across all banks, i.e., the C_{size} cost function defined earlier, is upper bounded by $\prod_i \left\lceil \frac{D_i}{h_{ii}} \right\rceil \cdot h_{ii} = \prod_i \left(\frac{D_i}{h_{ii}} + \delta_i\right) \cdot h_{ii} = \prod_i (D_i + \delta_i h_{ii})$, with $\delta_i < 1$. Under the assumption that $h_{ii} = o(D_j)\ \forall i, j$, the above upper bound to C_{size} can be written as $\prod_i D_i + o\left(\prod_i D_i\right) = D + o(D)$, where $D = \prod_i D_i$ is the total amount of

memory taken by the original array. In other words, C_{size} is asymptotically equal to the original amount of memory, i.e., the banks are densely populated with a number of holes becoming negligible as the size of the array is increased.

The above optimal solution involves a zero amount of asymptotic memory waste. Unfortunately, however, it requires an integer division by h_{ii} for each component in $\vec{m}$ before starting a memory access. In case that h_{ii} is a power of 2, which indeed is likely to happen in practice, division simply coincides with a right-shift. For the cases where h_{ii} is not a power of 2, on the other hand, we propose an efficient solution which involves an arbitrarily small amount of asymptotic waste along each dimension of the memory. In essence, the solution is based on the idea of replacing the integer division by h_{ii} with a more efficient multiplication by an integer constant a, followed by a right-shift by b bits. In the following, we will refer to a single component m_i for the sake of simplicity. In essence, we replace $m'_i = \left\lfloor \frac{m_i}{h_{ii}} \right\rfloor$ with $m'_i = \left\lfloor \frac{m_i \cdot a}{2^b} \right\rfloor$, with $\frac{2^b}{a} \leq h_{ii}$. The maximum value taken by the m'_i will thus increase from $\left\lfloor \frac{D_i - 1}{h_{ii}} \right\rfloor$ to $\left\lfloor \frac{(D_i - 1) \cdot a}{2^b} \right\rfloor$, stretching the memory used by around $\frac{h_{ii} \cdot a}{2^b}$ along dimension i. We show that, in principle, it is possible to choose a and b such that any arbitrarily small amount of waste can be obtained. Call ω the fraction of memory waste to obtain (e.g., $\omega = 0.1$ indicates a memory waste of 10% along dimension i). Take $b > \log_2 \frac{h_{ii}}{\omega}$ and $a = \left\lceil \frac{2^b}{h_{ii}} \right\rceil$. First of all, based on this choice we have that $\frac{a}{2^b} \geq \frac{1}{h_{ii}}$, still ensuring that $\left| \left\lfloor \frac{m_i \cdot a}{2^b} \right\rfloor - \left\lfloor \frac{n_i \cdot a}{2^b} \right\rfloor \right| \geq 1$ if $|m_i - n_i| \geq h_{ii}$. Furthermore, the stretching factor is $\frac{h_{ii} \cdot a}{2^b} = \frac{h_{ii} \cdot \left\lceil \frac{2^b}{h_{ii}} \right\rceil}{2^b} = \frac{h_{ii} \cdot \left(\frac{2^b}{h_{ii}} + \delta \right)}{2^b} = 1 + \frac{h_{ii} \cdot \delta}{2^b}$, with $\delta < 1$. The percentage waste, represented by the term $\frac{h_{ii} \cdot \delta}{2^b}$, is by construction less than ω, as required, because $b > \log_2 \frac{h_{ii}}{\omega}$.

Depending on the values of h_{ii} and the actual cost of an integer division, one of the two solutions above can be adopted to effectively address the problem of storage minimization, As an example, assume that we have $h_{ii} = 3$ for dimension i, as in the example of figure 2.2, and that we require a percentage waste upperbounded by 10%. We can choose $b > \log_2 \frac{h_{ii}}{\omega} = \log_2 30$, e.g., $b = 5$, and $a = \left\lceil \frac{2^b}{h_{ii}} \right\rceil = \left\lceil \frac{32}{3} \right\rceil = 11$. Then, in the transformed code, each array subscript corresponding to dimension i, expressed as $\mathbf{F}_i \cdot \vec{v}$ (where $\mathbf{F}_i$ is the ith row of the access matrix $\mathbf{F}$) will be replaced with $(11 \cdot \mathbf{F}_i \cdot \vec{v}) >> 5$.

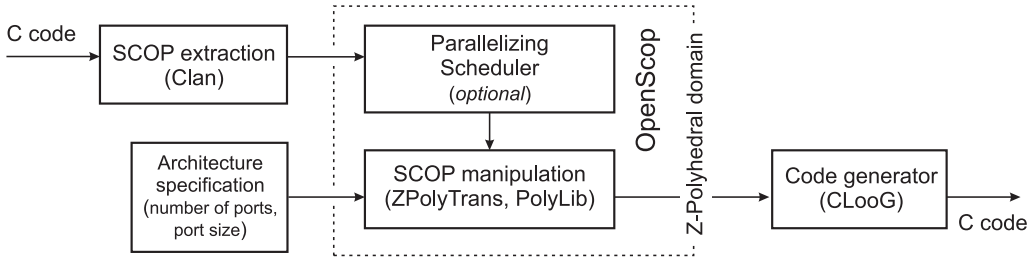


FIGURE 2.3: Prototype toolchain implementing lattice-based partitioning.

2.6 A detailed case study

To demonstrate the lattice-based partitioning technique, we built a prototype toolchain, depicted in figure 2.3. The implementation is based on a variety of open-source tools for polyhedral analysis. The input is first analyzed by Clan[3], used to extract a polyhedral representation. Then, it is scheduled in order to extract as much internal parallelism as possible. This step is optional since the mathematical framework does not deal with parallelism extraction but assumes an already scheduled loop nest. Subsequently, the representation is manipulated by ad-hoc tools applying the proposed methodology. The process relies on the OpenScop[4] format to represent the information on the code being transformed, PolyLib [27] to manipulate polyhedra, and ZPolyTrans [24] to handle images of Z-polyhedra by non-invertible functions and parametric counting of points in unions of Z-polyhedra. There are other tools that could serve the purpose, particularly [28], while ISL supports relations and hence Z-polyhedra, but it does not provide specialized algorithms [37]. Last, after manipulation, C code is regenerated using CLooG [38].

Based on the above toolchain, this section provides a step-by-step illustration of the lattice-based partitioning technique to a real-world application, a bidimensional window filtering kernel, where a waveform/sequence is multiplied by a window function [39]. Applications of window functions include spectral analysis, filter design, and beamforming. More specifically, in this section we target bidimensional signals expressed as arrays (e.g., images) and 3×3 window functions expressed as 9 separate coefficients. Figure 2.4.a shows the bidimensional filter kernel consisting

[3] http://icps.u-strasbg.fr/~bastoul/development/clan/index.html
[4] http://icps.u-strasbg.fr/~bastoul/development/openscop/

(a)

```
for(t=0; t<T; t++)
  for(i=1; i<N-1; i++)
    for(j=1; j<N-1; j++)
      S(i,j): Y[i][j] =
         W1*A[i-1][j-1] + W2*A[i-1][j] +
         W3*A[i-1][j+1] + W4*A[i][j-1] +
         W5*A[i][j]     + W6*A[i][j+1] +
         W7*A[i+1][j-1] + W8*A[i+1][j] +
         W9*A[i+1][j+1];
```

(b)

```
for(t=0; t<=T-1; t++)
  for(i=0; i<=N-3; i++)
    for(j=0; j<=(N-4)/2; j++)
      parfor(p=0; p<=1; p++)
        S(i,j+p): Y[i+1][2*j+1+p] =
           W1*A[i][2*j+p]    + W2*A[i][2*j+1+p] +
           W3*A[i][2*j+2+p]  + W4*A[i+1][2*j+p] +
           W5*A[i+1][2*j+1+p]+ W6*A[i+1][2*j+2+p]+
           W7*A[i+2][2*j+p]  + W8*A[i+2][2*j+1+p]+
           W9*A[i+2][2*j+2+p];
```

FIGURE 2.4: 2D Windowing kernel. (a) Original version. (b) 2x Stripmined and parallelized version.

of a perfect nest made of three loops. The inner loops iterate over space whereas the outer loop iterates over time since the filter may be applied several times to the same signal in order to amplify the effect or use different coefficients.

Some parallelism has been extracted by stripmining the innermost loop by a factor SF. Figure 2.4.b shows the stripmined and parallelized code. While we can set SF to any value less than $N-1$, for the sake of simplicity we use here $SF = 2$. The constant N is assumed to be even. All loops in the code are normalized, i.e., their iterators start from 0 and have unit stride. The innermost `for` construct is marked as parallel. In fact, it is evident that it can be parallelized since there are no true dependencies carried by the three innermost loops.

In the example, the memory architecture is assumed to provide up to 6 independent memory banks. We apply the lattice-based partitioning technique to the input array A; a similar procedure may be applied to the output array Y.

2.6.1 Polyhedral model of the kernel

The kernel contains four loops, hence the iteration vector $\vec{v}$ is $\vec{v} = (t, i, j, p, T, N, 1)$, N and T being two constant parameters. The normalized iteration domain of

statement S is the following integer polyhedron:

$$D_S = \left\{ \vec{v} : \mathbf{D}_S \cdot \vec{v} \geq \vec{0} \right\}, \qquad \mathbf{D}_S = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & -3 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & 1 & -4 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 \end{pmatrix}$$

A three-dimensional schedule can be derived for statement S directly from the code in figure 2.4.b:

$$\Theta_s(\vec{v}) = \mathbf{\Theta}_S \cdot \vec{v} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \vec{v}$$

The fourth column, corresponding to p, contains all zero elements, meaning that the loop iterations are scheduled concurrently.

2.6.2 Modeling data parallelism

The first step of the methodology consists in modelling the parallelism of the memory accesses. First, we need to build the parallel parametric slice $P_S\left(\vec{k}\right)$. In this example, $\vec{k}$ is $\vec{k} = (k_1, k_2, k_3)$ because we have three serial loops. Parameters k_h directly corresponds to the iterators of the three outermost loops, i.e., t, i, and j, and vary within the same intervals. The parametric slice $P_S\left(\vec{k}\right)$ is the

polyhedron of iterations $\vec{v}$ satisfying:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & -3 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & 1 & -4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ - & - & - & - & - & - & - & - & - & - \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} t \\ i \\ j \\ p \\ T \\ N \\ 1 \\ k_1 \\ k_2 \\ k_3 \end{pmatrix} \begin{matrix} \geq \\ - \\ = \end{matrix} \vec{0}$$

For each value of the parameter $\vec{k} = (k_1, k_2, k_3)$, the slice turns out to contain two iterations, those corresponding to $p = 0$ and $p = 1$.

The statement in the kernel of figure 2.4.b contains 9 bidimensional affine accesses to array A, each having its own access function. As an example, $A[i+1][2j+p]$ has $F(\vec{v}) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 1 & 0 & 0 & 0 \end{pmatrix} \cdot \vec{v}$ as its access function. The next step thus concerns the identification of the data sets determined by these accesses, i.e., the set $M\left(P_S\left(\vec{k}\right)\right)$ containing the memory locations accessed in parallel by concurrent iterations. As explained earlier, the data sets are the union of the images of $P_S\left(\vec{k}\right)$ by the affine access functions. Each image represents a set of locations accessed in parallel by a specific memory reference in the statement. In this case, although the memory access functions are non-invertible, the images are still Z-polyhedra (here indeed they are simply polyhedra). Furthermore, their unions also turn out to be polyhedra. Figure 2.5 depicts a few examples of images and data sets for a couple of values of the parameter $\vec{k} = (k_1, k_2, k_3)$. Although the data sets are a function of $\vec{k}$ and their cardinality may in general vary, here all data sets always contain 12 memory points. In this example they can be formally expressed as follows:

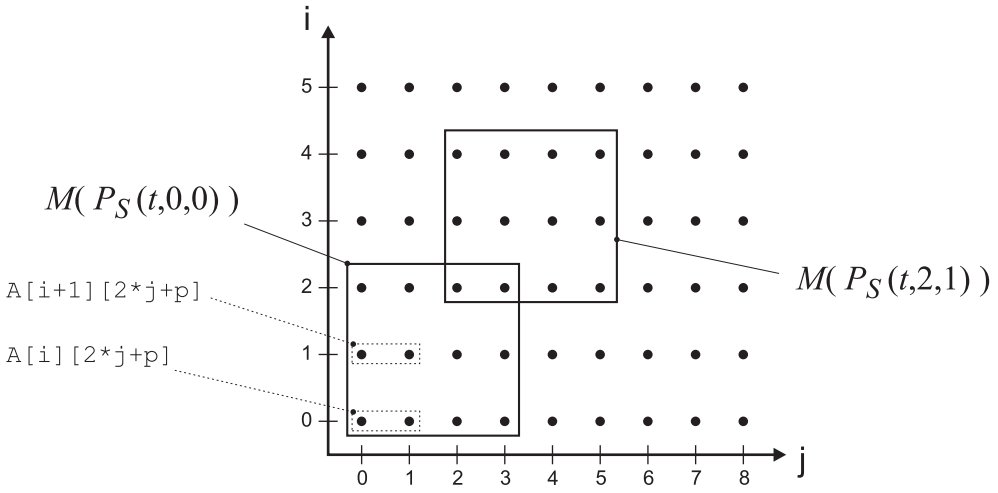


FIGURE 2.5: Example of data sets for the example kernel. For data set $M\left(P_S\left(t,0,0\right)\right)$ the figure also shows the images of the slice P_S by two single access functions. Overall, each data set is formed by nine such images.

$$M\left(P_S\left(\vec{k}\right)\right) = \left\{ (j,i) \in \mathbb{Z}^2 : \begin{pmatrix} 1 & 0 & 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 2 \\ 0 & 1 & 0 & 0 & -2 & 0 \\ 0 & -1 & 0 & 0 & 2 & 3 \end{pmatrix} \begin{pmatrix} i \\ j \\ k_1 \\ k_2 \\ k_3 \\ 1 \end{pmatrix} \geq \vec{0} \right\}$$

2.6.3 Generation of the solution space

Since there are 6 memory banks available, the methodology looks for the best bidimensional lattice-based partitioning solution having determinant less than or equal to 6. Indeed, since each data set contains 12 distinct accesses, the best we can do is to have two accesses per memory bank in each data set, which is only possible with 6 banks. As a consequence, in this example we will only discuss the case $\det\left(\mathcal{L}\right) = 6$. As explained in section 2.5.1, enumerating all distinct bidimensional lattices is equivalent to generating all possible 2×2 matrices in HNF with a determinant equal to 6. Relying on the quantitative formula given in section 2.5.1, we know that there are $H_2(6) = H_2(3)H_2(2) = 3 \times 4 = 12$ such matrices. Below we list all of them:

$$\mathbf{B}_1 = \begin{pmatrix} 6 & 0 \\ 0 & 1 \end{pmatrix}; \quad \mathbf{B}_2 = \begin{pmatrix} 1 & 0 \\ 0 & 6 \end{pmatrix}; \quad \mathbf{B}_3 = \begin{pmatrix} 1 & 0 \\ 1 & 6 \end{pmatrix}; \quad \mathbf{B}_4 = \begin{pmatrix} 1 & 0 \\ 2 & 6 \end{pmatrix};$$

$$\mathbf{B}_5 = \begin{pmatrix} 1 & 0 \\ 3 & 6 \end{pmatrix}; \quad \mathbf{B}_6 = \begin{pmatrix} 1 & 0 \\ 4 & 6 \end{pmatrix}; \quad \mathbf{B}_7 = \begin{pmatrix} 1 & 0 \\ 5 & 6 \end{pmatrix}; \quad \mathbf{B}_8 = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix};$$

$$\mathbf{B}_9 = \begin{pmatrix} 2 & 0 \\ 1 & 3 \end{pmatrix}; \quad \mathbf{B}_{10} = \begin{pmatrix} 2 & 0 \\ 2 & 3 \end{pmatrix}; \quad \mathbf{B}_{11} = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}; \quad \mathbf{B}_{12} = \begin{pmatrix} 3 & 0 \\ 1 & 2 \end{pmatrix}.$$

Each enumerated solution is representative of an infinite number of bases all equal up to a unimodular transformation. Figure 2.6 shows the lattices corresponding to $\mathbf{B}_8$ and $\mathbf{B}_3$. Once the solution space is generated, we need to evaluate each potential solution to identify the best fundamental lattice.

2.6.4 Enumeration of the translates

To proceed, we first need to enumerate the translates of each lattice previously identified. Take $\mathbf{B}_8 = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$ as an example. As pointed out in section 2.5.3, we need to enumerate all the points in the parallelepiped identified by the diagonal of $\mathbf{B}_8$, here a 2×3 rectangle, with the lower corner in the origin. The points are:

$$t_0 = (0,0), \quad t_1 = (1,0), \quad t_2 = (0,1), \quad t_3 = (1,1), \quad t_4 = (0,2), \quad t_5 = (1,2).$$

These vectors correspond to the six translates of the fundamental lattice needed to cover $\mathbb{Z}^2$. By construction, their number is equal to the number of memory banks.

2.6.5 Evaluation of the solution space

To evaluate a solution, we compute the maximum number of conflicting accesses (i.e., accesses to the same memory bank) within all parallel data sets (see section 2.5.2). For each parallel data set, the conflicts for a given translate $\mathcal{L}_t$ are

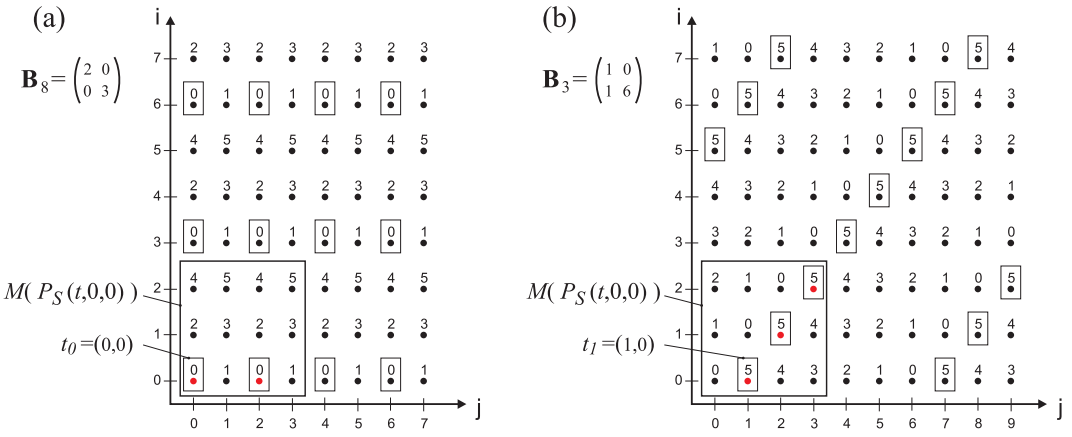


FIGURE 2.6: Two different mapping solutions corresponding to the fundamental lattices $\mathcal{L}(\mathbf{B}_8)$ and $\mathcal{L}(\mathbf{B}_3)$. For each of the two solutions, the figure highlights one of the translates causing the highest conflict count.

given by the intersection of the affine lattice $\mathcal{L}_t$ itself and the polyhedron describing the data set. The number of integer points, what we called $MC\left(\vec{k}\right)$, contained in such Z-polyhedron represents, parametrically, the number of conflicts corresponding to a particular translate $\mathcal{L}_t$ for a certain solution $\mathcal{L}$. Figure 2.6 also shows graphically $MC\left(\vec{k}\right)$ for $\vec{k} = (t, 0, 0)$ and the translates corresponding to the vectors $(0, 0)$ and $(1, 0)$ for $\mathcal{L}(\mathbf{B}_8)$ and $\mathcal{L}(\mathbf{B}_3)$, respectively. The number of points in the first Z-polyhedron is 2 whereas we count 3 points in the second Z-polyhedron, resulting in worse performance. In both cases, they are independent of $\vec{k}$. Following are the overall memory access times, computed by taking into account the worst-case conflicts within the data sets (which are necessarily serialized in time).

$$C_{\text{time}}(\mathbf{B}_1) = (N-2)\cdot(N-2)\cdot T\cdot 3 \qquad C_{\text{time}}(\mathbf{B}_2) = (N-2)\cdot(N-2)\cdot T\cdot 4$$

$$C_{\text{time}}(\mathbf{B}_3) = (N-2)\cdot(N-2)\cdot T\cdot 3 \qquad C_{\text{time}}(\mathbf{B}_4) = (N-2)\cdot(N-2)\cdot T\cdot 3$$

$$C_{\text{time}}(\mathbf{B}_5) = (N-2)\cdot(N-2)\cdot T\cdot 2 \qquad C_{\text{time}}(\mathbf{B}_6) = (N-2)\cdot(N-2)\cdot T\cdot 3$$

$$C_{\text{time}}(\mathbf{B}_7) = (N-2)\cdot(N-2)\cdot T\cdot 3 \qquad C_{\text{time}}(\mathbf{B}_8) = (N-2)\cdot(N-2)\cdot T\cdot 2$$

$$C_{\text{time}}(\mathbf{B}_9) = (N-2)\cdot(N-2)\cdot T\cdot 2 \qquad C_{\text{time}}(\mathbf{B}_{10}) = (N-2)\cdot(N-2)\cdot T\cdot 2$$

$$C_{\text{time}}(\mathbf{B}_{11}) = (N-2)\cdot(N-2)\cdot T\cdot 4 \qquad C_{\text{time}}(\mathbf{B}_{12}) = (N-2)\cdot(N-2)\cdot T\cdot 3$$

In section 2.5.2, the expression of C_{time} contained a sum on $\vec{k}$. Here, because of the fact that there is no dependence on $\vec{k}$, the sum is reduced to a multiplication by the number of values spanned by $\vec{k} = (k_1, k_2, k_3)$, i.e., $(N-2)\cdot(N-2)\cdot T$. Notice that $\mathbf{B}_5$, $\mathbf{B}_8$, $\mathbf{B}_9$, and $\mathbf{B}_{10}$ all achieve the minimum number of conflicts. Here, we choose $L = \mathbf{B}_8$.

2.6.6 Storage minimization

The chosen fundamental lattice is $\mathcal{L}(\mathbf{B}_8)$, with $\mathbf{B}_8 = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$. Based on the results in section 2.5.5, we can simply replace each original access in the new arrays by scaling the column subscript by 2 and the row subscript by 3, obtaining an asymptotically zero memory waste. As an alternative, to avoid the integer division by 3, we can adopt the approximate approach introduced in section 2.5.5, which consists in replacing the division with a multiplication by an integer a followed by a b-bit right-shift. As already exemplified in section 2.5.5 for the case $h_{ii} = 3$ and a maximum waste of 10%, we can use $b > \log_2 \frac{3}{0.1}$, e.g., $b = 5$, and consequently $a = \left\lceil \frac{2^5}{3} \right\rceil = 11$. Formally, for each access function in the statement of figure 2.4, the new access function can be obtained as $F'(\vec{v}) = \left\lfloor \begin{pmatrix} 1/2 & 0 \\ 0 & 11/32 \end{pmatrix} \cdot F(\vec{v}) \right\rfloor$.

As a practical example of memory waste, consider the case $N = 100$, requiring $N^2 = 10000$ memory locations. The highest modified subscripts are $[11\cdot(i+2) >> 5][(2\cdot j+2+p) >> 1]$, and they take as their maximum value [34][48], corresponding to the bounds $i = N-3 = 97$, $j = (N-4)/2 = 48$, $p = 1$. In other words, we need $35 \cdot 49 = 1715$ locations for each of the 6 arrays, i.e., 10290 locations overall, with an actual waste below 3%.

2.6.7 Code generation

Based on the technique presented in section 2.5.4, we can generate the new polyhedral representation maximizing parallel memory bank accesses. The representation leads to the following optimized code:

```
for( t=0; t<=T-1; t++ ) {
  for( i=0; i<=N-3; i++ ) {
    if( i%3==0 ) {
      for( j=0; j<=(N-4)/2; j++ ) {
        parfor( p=0; p<=1; p++ ) {
          if( (j+p)%2==0 )
           S1:  Y[i+1][2*j+1+p] = W1*A0[(11*i)>>5][(2*j+p)>>1] +
           W2*A1[(11*i)>>5][(2*j+1+p)>>1] + W3*A0[(11*i)>>5][(2*j+2+p)>>1] +
           W4*A2[(11*i+11)>>5][(2*j+p)>>1] + W5*A3[(11*i+11)>>5][(2*j+1+p)>>1] +
           W6*A2[(11*i+11)>>5][(2*j+2+p)>>1] + W7*A4[(11*i+22)>>5][(2*j+p)>>1] +
           W8*A5[(11*i+22)>>5][(2*j+1+p)>>1] + W9*A4[(11*i+22)>>5][(2*j+2+p)>>1];
          else
           S2:  Y[i+1][2*j+1+p] = W1*A1[(11*i)>>5][(2*j+p)>>1] +
           W2*A0[(11*i)>>5][(2*j+1+p)>>1] + W3*A1[(11*i)>>5][(2*j+2+p)>>1] +
           W4*A3[(11*i+11)>>5][(2*j+p)>>1] + W5*A2[(11*i+11)>>5][(2*j+1+p)>>1] +
           W6*A3[(11*i+11)>>5][(2*j+2+p)>>1] + W7*A5[(11*i+22)>>5][(2*j+p)>>1] +
           W8*A4[(11*i+22)>>5][(2*j+1+p)>>1] + W9*A5[(11*i+22)>>5][(2*j+2+p)>>1];
        }
      }
    }else if( i%3==1 ){
      for( j=1; j<=N - 2; j+=2 ) {
        parfor( p=0; p<=1; p++ ) {
          if( (j+p)%2==0 )
           S3:  Y[i+1][2*j+1+p] = W1*A2[(11*i)>>5][(2*j+p)>>1] +
           W2*A3[(11*i)>>5][(2*j+1+p)>>1] + W3*A2[(11*i)>>5][(2*j+2+p)>>1] +
           W4*A4[(11*i+11)>>5][(2*j+p)>>1] + W5*A5[(11*i+11)>>5][(2*j+1+p)>>1] +
           W6*A4[(11*i+11)>>5][(2*j+2+p)>>1] + W7*A0[(11*i+22)>>5][(2*j+p)>>1] +
           W8*A1[(11*i+22)>>5][(2*j+1+p)>>1] + W9*A0[(11*i+22)>>5][(2*j+2+p)>>1];
          else
           S4:  Y[i+1][2*j+1+p] = W1*A3[(11*i)>>5][(2*j+p)>>1] +
           W2*A2[(11*i)>>5][(2*j+1+p)>>1] + W3*A3[(11*i)>>5][(2*j+2+p)>>1] +
           W4*A5[(11*i+11)>>5][(2*j+p)>>1] + W5*A4[(11*i+11)>>5][(2*j+1+p)>>1] +
           W6*A5[(11*i+11)>>5][(2*j+2+p)>>1] + W7*A1[(11*i+22)>>5][(2*j+p)>>1] +
           W8*A0[(11*i+22)>>5][(2*j+1+p)>>1] + W9*A1[(11*i+22)>>5][(2*j+2+p)>>1];
          }
        }
    }else if( i%3==2 ){
      for( j=1; j<=N - 2; j+=2 ) {
        parfor( p=0; p<=1; p++ ) {
          if( (j+p)%2==0 )
           S5:  Y[i+1][2*j+1+p] = W1*A4[(11*i)>>5][(2*j+p)>>1] +
           W2*A5[(11*i)>>5][(2*j+1+p)>>1] + W3*A4[(11*i)>>5][(2*j+2+p)>>1] +
           W4*A0[(11*i+11)>>5][(2*j+p)>>1] + W5*A1[(11*i+11)>>5][(2*j+1+p)>>1] +
```

```
          W6*A0[(11*i+11)>>5][(2*j+2+p)>>1] + W7*A2[(11*i+22)>>5][(2*j+p)>>1] +
          W8*A3[(11*i+22)>>5][(2*j+1+p)>>1] + W9*A2[(11*i+22)>>5][(2*j+2+p)>>1];
         else
          S6:  Y[i+1][2*j+1+p] = W1*A5[(11*i)>>5][(2*j+p)>>1] +
          W2*A4[(11*i)>>5][(2*j+1+p)>>1] + W3*A5[(11*i)>>5][(2*j+2+p)>>1] +
          W4*A1[(11*i+11)>>5][(2*j+p)>>1] + W5*A0[(11*i+11)>>5][(2*j+1+p)>>1] +
          W6*A1[(11*i+11)>>5][(2*j+2+p)>>1] + W7*A3[(11*i+22)>>5][(2*j+p)>>1] +
          W8*A2[(11*i+22)>>5][(2*j+1+p)>>1] + W9*A3[(11*i+22)>>5][(2*j+2+p)>>1];
        }
      }
    }
  }
}
```

Complying with the mapping choice previously identified, no more than two conflicts per parallel set of iterations are incurred. Consider the first two parallel statements S_1 and S_2 as an example. $A0$ appears 3 times, thus the iteration seemingly contains three simultaneous accesses. The three accesses can however only involve two different values because of the mapping we enforced. In fact, $A0[(11 \cdot i) >> 5][(2 \cdot j + 1 + p) >> 1]$ in the `else` branch coincides with $A0[(11 \cdot i) >> 5][(2 \cdot j + 2 + p) >> 1]$ in the `then` branch if j is even (because if j is even, p must be equal to 1 in the `else` branch and 0 in the `then` branch) or with $A0[(11 \cdot i) >> 5][(2 \cdot j + p) >> 1]$ in the `then` branch if j is odd (because if j is odd p must be equal to 0 in the `else` branch and 1 in the `then` branch). The same considerations apply to the other accesses.

2.6.8 Validation of the cost function

To conclude the case study, we show a few results collected from a practical implementation of the kernel targeted at an FPGA technology by means of a high-level synthesis process. In particular, the above C code was synthesized to VHDL by the Vivado HLS tool [36] and analyzed by cycle-accurate simulation. The actual execution times, denoted ET and expressed in terms of clock count, refer to the case $N = 20$, $T = 8$. They were measured for each of the 12 lattice-based partitioning solutions explored above:

$$ET(\mathbf{B}_1) = 3889, \quad ET(\mathbf{B}_2) = 5187, \quad ET(\mathbf{B}_3) = 3889, \quad ET(\mathbf{B}_4) = 3889,$$

$$ET(\mathbf{B}_5) = 2595, \quad ET(\mathbf{B}_6) = 3889, \quad ET(\mathbf{B}_7) = 3889, \quad ET(\mathbf{B}_8) = 2595,$$
$$ET(\mathbf{B}_9) = 2595, \quad ET(\mathbf{B}_{10}) = 2595, \quad ET(\mathbf{B}_{11}) = 5187, \quad ET(\mathbf{B}_{12}) = 3889$$

The results provide a clear confirmation of the significance of the cost function C_{time} we used to drive the optimization process and confirm the impact that memory conflicts have on the actual execution time of the synthesized kernel.

2.7 Related work

The application of integer lattices to memory allocation problems is addressed by [33], which proposes a methodology for reducing the memory needed for the execution of an algorithm by analyzing variable liveness. The authors establish a correspondence between an integer lattice and a modular mapping of the array indices. As an example, the lattice spanned by the basis $\begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$ may equivalently be expressed by the two-dimensional mapping $(b_0, b_1) = (j \bmod 2, i \bmod 3)$, where the pair (b_0, b_1), with $0 \leq b_0 < 2$ and $0 \leq b_1 < 3$, is used as a bidimensional bank index. Although the approach here described is based on a similar mathematical framework and use some of their results, e.g., the enumeration of the solutions, they solve a different problem, i.e., memory reuse. In particular, in this context the determinant of the lattice corresponds to the number of memory banks and the objective function is chosen to minimize the number of conflicts on memory ports. On the other hand, in their work the number of memory locations required by an optimum modular mapping is equal to the determinant of the underlying lattice. Furthermore, they consider indices as conflicting when they are simultaneously live under a given schedule, whereas, in this context, there is a conflict when there is an access to the same memory bank by simultaneous executions of loop iterations.

The development of mathematical methodologies for the problem of partitioning was also studied extensively for optimizing Locally Sequential Globally Parallel (LSGP) computations. The authors of [40] aim to find valid boxes, i.e., parallelepipeds of points, for systolic arrays. Their solution space consists of all the left Hermite forms of the so called activity basis that identifies the computations simultaneously active. In [40] integer points represent computations that may be

mapped or not onto the same processor and not memory locations to map onto specific banks. A similar problem is analyzed in [41], providing a closed form for enumerating all the schedules on a cluster of VLIW processors and constructing a linear function that schedules precisely one iteration per cycle of a loop nest. Memory partitioning for LSGP systems has been also studied by [42–44] in order to reduce the communication among processors.

Recently, mathematical optimization techniques have targeted reconfigurable devices provided with multiple independent fine-grained memory blocks. In this scenario, minimizing the number of conflicts on the distributed memory banks is of paramount importance for performance. A few recent contributions [19, 45–47] are concerned with partitioning of on-chip memory. In [19] an automated memory partitioning algorithm is proposed to support multiple simultaneous affine memory references to the same array. In [45] memory accesses in different loop iterations are partitioned across different memory banks and scheduled in the same cycle minimizing the number of required banks. In the above works, a multidimensional array is first flattened into a single-dimensional array before partitioning. Since memory addresses after flattening are dependent on the array size, different partitioning schemes are generated for different array sizes, many of which are suboptimal. An improved approach is proposed in [46]. They model memory ports as n-dimensional hyperplanes. However, by limiting their solutions to a single family of hyperplanes, they might ignore some potential solutions. While [46] uses a single family of hyperplanes to express bank mapping, the presented technique essentially exploits a number of hyperplanes equal to the dimensionality of the array to be partitioned. Furthermore, [46] does not express parallel data sets formally. Hence, the methodology is valid only for the specific problem of avoiding memory conflicts when loop pipelining techniques are applied. The presented technique, on the other hand, explicitly models parallel iterations by means of Z-polyhedral techniques and generalizes the solution space by adopting lattices instead of hyperplanes, i.e., a strict superset of the solution space in [46]. The same authors in [47] extend their previous work by introducing block-cyclic partitioning, but still relying only on a single family of hyperplanes. The work in [48] introduces a geometric programming framework combining data reuse with data level parallelism. Although the problem is still related to data-level parallelization, they

assume that each processing element is connected to a single memory and they replicate data when needed. This assumption largely simplifies the problem but neglects the possibility of avoiding the replication of data, possibly incurring memory waste and a potential loss of available memory ports. Similar to the approach presented here, the authors of [49] use integer lattices in the context of memory partitioning, in order to improve memory bandwidth by using different physical storage blocks. The work explores different periodic partitioning strategies by varying both the number of banks and the system clock frequency. Their main contribution is related to address generation given a specific partitioning strategy. Their solution space exploration, however, does not resort to any mathematical technique to model the code and its memory access patterns. As a consequence, the work is based on evaluating the schedule length and the system cost for each possible solution by adopting scheduling algorithms, e.g., list scheduling.

There are other approaches concerned with optimizing on-chip memory usage, although focused on orthogonal problems. The work in [50] develops a compile-time framework for data locality optimization via data layout transformation targeting NUCA chip multiprocessors. The authors of [51] use polyhedral abstractions to reduce the occupied on-chip scratchpad memory. The approach is useful when dealing with algorithms working with a great deal of data. Unlike the proposed technique, it aims to improve data reuse instead of maximizing the available on-chip memory bandwidth. Differently, [52] introduces a framework in the context of HLS for transforming loop nests in order to maximize on-chip memory reuse and minimize accesses to off-chip memory blocks. A different problem is tackled in [53], which proposes a technique to embed some physical features related to external SDRAMs in the iteration domains of code statements in order to minimize row activations and maximize reuse. It addresses external memory modules and, in that respect, it is complementary to this technique, which is focused on on-chip memory. The work in [54] optimizes remote accesses for offloaded kernels on reconfigurable platforms.

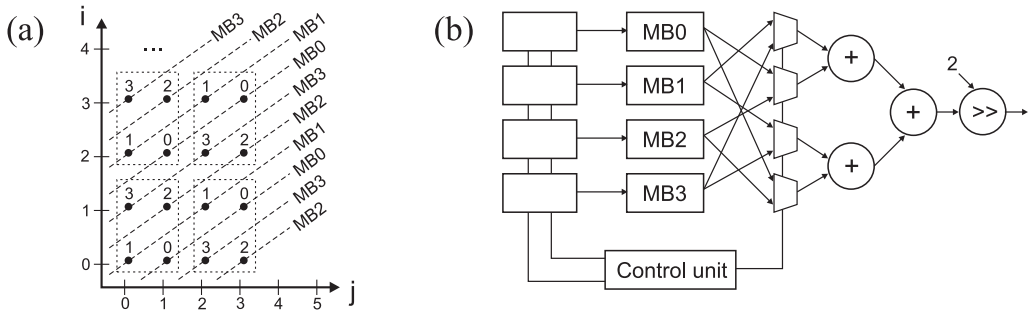


FIGURE 2.7: Hyperplane-based memory partitioning. (a) Memory bank mapping. (b) Hardware datapath inferred from memory bank mapping. The symbol MBi in the figures denotes the ith bank.

2.7.1 Comparisons with the hyperplane-based approach

As summarized above, [46, 47] are the current state-of-the-art solutions for memory partitioning in the context of high-level synthesis. They are limited to solutions based on a single family of hyperplanes. In that respect, lattices represent an extension of previous approaches. In fact, partitioning solutions based on one single family of hyperplane, e.g., $(j + i) \mod 2$, can always be represented as lattices, e.g., $\begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}$. In some cases, hyperplane-based partitioning may achieve the same level of access parallelism as lattice-based partitioning, but yielding a more complex steering circuitry. For instance, the memory in the example of figure 2.1.a can be partitioned as shown in figure 2.7.a by using hyperplanes, fully exploiting the available four memory banks. However, since the access patterns to the different banks are not always the same in each data set, each input of each adder needs to receive the operands from two different memory banks depending on the specific value of the iterators, complicating the steering logic of the datapath, as well as causing potential increases in area, clock period, and power consumption. Notice that, because of the commutativity of addition, this logic overhead may be optimized out in this specific case. However, this is not true in general. On the other hand, covering a strict superset of hyperplane-based partitioning, lattice-based partitioning allows us to identify the more efficient solution shown in figure 2.1.c, where, in addition to the fully parallelizable read operations, the access patterns to the banks are the same for each data set, inherently simplifying the steering logic. To confirm the above considerations, Table 2.1 shows the area occupation

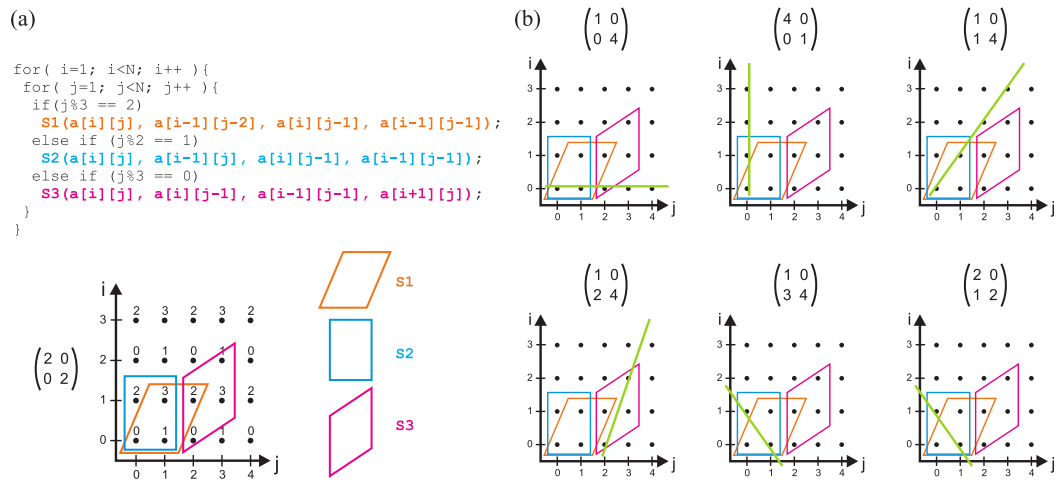


FIGURE 2.8: (a) Code snippet and related data sets on a lattice-based partitioning solution. (b) Possible hyperplane-based solutions.

measured in Look-up Tables (LUTs) on a Xilinx Virtex-7 FPGA for three different benchmarks. The considered benchmarks are typical HLS applications with loop nests having a high degree of parallelism. In particular, the first benchmark is an ordinary image resize kernel using bilinear interpolation. The remaining two benchmarks perform different stencil computations, i.e., a 2D Jacobi kernel and a 2D Gauss-Seidel kernel. See for example [55] for the details of these algorithms. The kernels were coded in plain C and synthesized by means of the Vivado HLS tool [36]. For each benchmark we varied both the stripmining factors of the two innermost loops and the number of memory banks, here coinciding with BlockRAM components (BRAMs), normally available on Xilinx FPGAs. The table reports the number of LUTs used by the two techniques for the respective optimal solutions. Because of the spatial regularity of lattices, the proposed technique achieves more compact datapaths having a simplified steering logic. These improvements are also reflected in the code complexity. In fact, Table 2.1 also contains the overall number of statements present in the final code. As shown by the table, for the chosen benchmarks the method also helps keep the code complexity reasonably low.

In addition to a more complex steering logic, there may be situations where [46] requires more memory banks than strictly necessary to achieve full parallelization, particularly when the kernel has a complex control flow. As an example, consider the code in figure 2.8. The kernel contains a normalized perfect loop nest with

TABLE 2.1: Synthesis results. (OSF: outer stripmining factor, ISF: inner stripmining factor)

Benchmark	**NB / BRAMs**	**Lattices**		**Hyperplanes**	
		LUTs	**# Statements**	**LUTs**	**# Statements**
Image Resize kernel (OSF=2, ISF=2)	2	536	4	536	4
	4	536	4	536	4
	8	592	4	1024	8
	16	526	4	1136	8
Image Resize kernel (OSF=4, ISF=4)	2	2130	16	2130	16
	4	2130	16	2130	16
	8	2212	16	2212	16
	16	2212	16	2580	32
2D-Seidel kernel (OSF=2, ISF=2)	2	2745	4	2745	4
	4	2526	4	2526	8
	8	5584	8	7387	16
	16	8178	16	11254	32
2D-Seidel kernel (OSF=4, ISF=4)	2	10384	16	10348	16
	4	10339	16	10339	16
	8	10381	16	17516	32
	16	10202	16	11290	32
2D-Jacobi kernel (OSF=2, ISF=2)	2	1862	4	1862	4
	4	2118	4	3409	8
	8	4007	8	5753	16
	16	6419	16	9129	16
2D-Jacobi kernel (OSF=4, ISF=4)	2	4893	16	4893	16
	4	5387	16	5387	16
	8	5049	16	11369	32
	16	4878	16	16978	32

three statements. There is a statically predictable branch, so the nest is still an affine SCoP. Loop iterations access a variable shape 2×2 window. In particular, figure 2.8.a shows the parallel data sets for $(i = 1,\ j = 1)$, $(i = 1,\ j = 2)$, and $(i = 1,\ j = 3)$. Each of them contains four points. If four different memory banks are considered, the only possible solution in order not to have conflicts is showed in part (a) of the figure, corresponding to the fundamental lattice $\mathcal{L}\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$. This solution cannot be expressed as a set of translate hyperplanes. Figure 2.8.b shows all the possible hyperplane-based solutions. The green line represents the hyperplane causing the conflict. It can be easily recognized that four memory banks are not sufficient here to avoid conflicts completely, and the optimal solution identified by the lattice-based partitioning technique is missed.

2.8 Remarks and future developments

This chapter addressed the problem of automated memory partitioning for emerging architectures, such as reconfigurable hardware platforms, providing the opportunity of customizing the memory architecture based on the application accesses pattern. Targeted at affine static control parts (SCoPs), the technique exploits the Z-polyhedral model for program analysis, yielding a powerful and elegant formalism capturing both the problem of bank mapping and storage minimization. In particular, the approach is based on integer lattices, enabling us to generate a solution space for the bank mapping problem which includes previous results as particular cases. The problem of storage minimization, on the other hand, is tackled by an optimal approach ensuring asymptotically zero memory waste or, as an alternative, an efficient approach ensuring arbitrarily small waste. The theoretical results were also demonstrated through a prototype toolchain and a detailed step-by-step case study, along with some comparisons with different approaches found in the technical literature.

The approach also opens up a range of further investigation paths. First of all, as pointed out by the detailed case-study, there may be different lattices all achieving the minimum number of conflicts for a given number of banks. They might however be not equivalent in terms of the generated code, which may cause different delays and possibly area results in those computing platforms where the code is directly translated to hardware, such as FPGAs. Analyzing these effects systematically is, in fact, the main subject of chapter 3. Furthermore, although the search space is likely to be limited for practical problems, we will also explore the adoption of ad-hoc heuristics to explore it more efficiently. In particular, from a purely mathematical point of view, there are situations where, given a certain determinant, the solution space of the lattice-based partitioning technique collapses to one single family of hyperplanes. Although this happens for low dimensionalities of the array and very low numbers of memory banks, a precise formalization would help reduce the solution space in such cases. A further possibility of improvement concerns the storage minimization scheme. Although it is asymptotically optimal in terms of memory waste, it does not include liveness analysis of memory locations unlike [33]. The plan is thus to extend the methodology with per-bank

liveness analysis. Also, like most similar works, the presented approach is focused on partitioning a single array in memory. When different arrays are concurrently accessed in the same kernel, they may be processed separately or, alternatively, they may be seen as parts of a single memory space. These choices might variously impact performance or result in additional opportunities for optimization, leaving room for further developments in this direction. Lastly, instead of making the partitioning explicit in the code, e.g., by using different array names, which leads to a static assignment of banks, a different possibility would be to insert ad-hoc hardware components which route the memory requests to the corresponding banks by computing the mapping dynamically. This possibility was not explored here, essentially because a static code-level solution can be easily automated and does not interfere with the HLS process in itself. Its implementation, however, would transfer the complexity of the approach from the software to the underlying hardware architecture. Thus the automated generation of such hardware memory access managers is a potential future development of this approach.

Chapter 3

Reducing the area impact of the memory subsystem

3.1 Introduction

Chapter 2 has introduced the lattice-based memory partitioning technique and has shown it is more general than other existing methods. Besides beneficially impacting latency, it leads to very compact datapaths in terms of connections between memory banks and processing elements. Intuitively, that can be explained by the spatial regularity of integer lattices; throughout this chapter, we analyze and formalize the reasons why of such area efficiency. First, we capture by examples the existing relation between memory partitioning and area consumption. Then, we formalize the relation mathematically and propose a technique to improve the area efficiency by using a well-known code transformation, namely loop unrolling.

The related literature is quite scarce. To the author's knowledge, this has been the first work to formalize the interplay between memory partitioning and area efficiency in the context of high level synthesis. The only other attempt has been made in [47], in which the authors recognize the presence of an area overhead due to address generation and data assignment. However, the two effects are not modeled mathematically and, importantly, the overhead estimation does not take into account the bank switching phenomenon which, as we will notice, may have

a considerable impact on area efficiency, even more than the number of memory banks itself. Also, this is the first attempt ever to identify loop unrolling as key transformation in presence of memory partitioning,.

3.2 Impact of partitioning on area

Normally, in a HLS tool flow, the statements in a loop body are synthesized to a parallel physical datapath, which processes the memory locations embraced by the statements as concurrently as possible. The input/output ports of the datapath correspond to the read/write memory references in the loop body. When the memory is partitioned across multiple banks, the memory locations can be accessed in parallel, ideally in a single access cycle when a one-to-one correspondence between accessed memory locations and memory banks is achieved [56] as shown in the previous chapter. Furthermore, in case each specific memory reference always corresponds to the same bank, the connection between the respective datapath port and the physical bank can be direct, otherwise accesses must be multiplexed. Bank switching essentially refers to different instances of the same memory reference accessing different banks. We say that a reference $A[F(\vec{v})]$ has an amount of bank switching equal to bs, if it accesses bs different memory banks over the whole execution.

To clarify the above definition, refer to the example in figure 3.1. The code contains a two-level nested loop representing a rectangular window sliding over a bidimensional array. Figure 3.1.a shows the code while figure 3.1.b illustrates the array and the data sets accessed by a few iterations of the loop (coinciding with the rectangular-shaped sliding window). At each iteration, the window picks nine different memory locations and multiplies the values by nine different weights. Figure 3.1.b also shows a possible partitioning solution using four memory banks: the number associated with each memory location represents the memory bank to which it is allocated. As explained in chapter 2, that is a lattice based solution having as basis $\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$. Notice that the algorithm's weight associated with each memory location depends on the position within the sliding window. For example, the point located in the lower left corner of the window is always multiplied by w_0,

whereas the central by w_4. As the window slides through the array, each w_i is multiplied by a value from a different memory bank, as figure 3.1.b also highlights, causing the bank switching effect. Take weight w_0 as an example. In iteration $(i = 1, j = 1)$, the bank containing the data to be multiplied by w_0 is Bank 0, whereas in the consecutive iteration $(i = 1, j = 2)$ it is located in Bank 1.

Bank switching has a direct effect on the datapath synthesized from the high-level code. Figure 3.1.c shows the datapath corresponding to the example. Notice the additional steering logic required at the output data ports of the memory banks and at the address input ports. In fact, over the different iterations of the loop, each multiplier takes input values from all of the memory banks. As a second effect, bank switching results in a more complicated logic for the generation of the addresses to the memory banks, since each bank must be addressed by one of nine different combinations of the (i, j) indices as the loop proceeds through the iterations. In other words, we also need additional multiplexers driving different addresses to each of the four memory banks, depending on the iteration. The steering logic caused by bank switching may cause a significant area overhead, which is not inherently required by the algorithm itself, but rather by the way it is coded and the particular use of the available memory banks.

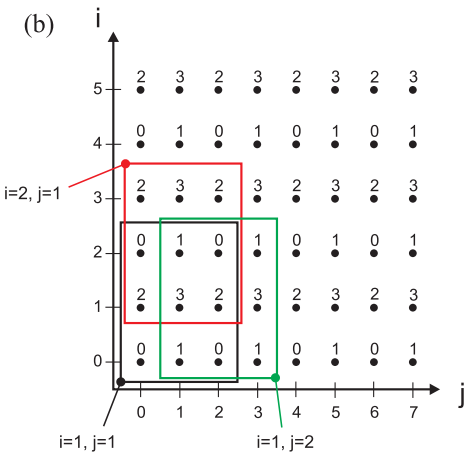
Clearly, the latter effect is not necessarily present. For example, consider the case when there is only one memory reference and memory is partitioned across two banks. Suppose that half of the statement instances access the first memory bank, while the remaining half access the second bank. Since there is only one memory reference, each memory bank stays idle half of the time and there is no need for an additional multiplexer on the address ports. As we will see during the rest of the discussion, a larger amount of bank switching corresponds to reduced area efficiency.

One interesting aspect to notice is that loop unrolling can potentially reduce bank switching, reducing the steering logic as immediate consequence. Loop unrolling is a common technique in HLS and, although it usually results in replicated datapaths, we show here that its interplay with bank switching may enable improved efficiency in the use of the hardware resources. Consider figure 3.2.a, in which the inner loop of the previous code has been unrolled by a factor of two. Like the

(a)

```
void imageWindowing(int imageOutput[N][N], imageInput[N][N]) {
   int i, j;
   for (i = 1; i <= N - 1; i++)
      for (j = 1; j <= N - 1; j ++)
         imageOutput[i][j] = w0*imageInput[i-1][j-1]
                           + w1*imageInput[i-1][j]
                           + w2*imageInput[i-1][j+1]
                           + w3*imageInput[i][j-1]
                           + w4*imageInput[i][j]
                           + w5*imageInput[i][j+1]
                           + w6*imageInput[i+1][j-1]
                           + w7*imageInput[i+1][j]
                           + w8*imageInput[i+1][j+1];
}
```

(b)



(c)

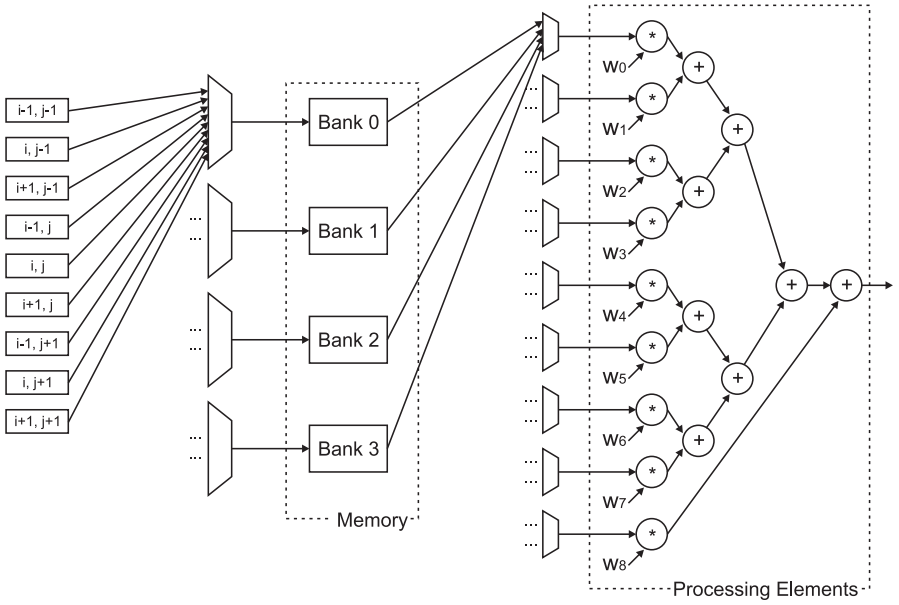


FIGURE 3.1: The original version of the generic bidimensional sliding window filter. a) Code, b) Memory accesses to the input image for some iterations, c) Synthesized datapath

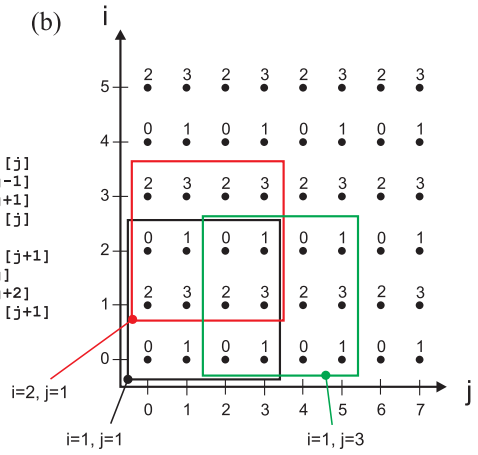
previous example, figure 3.2.b and figure 3.2.c depict, respectively, the partitioned memory and the synthesized datapath. Now, there is no bank switching affecting the inner loop because of the unrolling transformation. In fact, when sliding the window along the j axis, every location of the window always happen to touch the same memory bank (on the other hand, when sliding the window along the i axis, bank switching still occurs, with every location of the window touching two different banks). This results in significantly smaller multiplexers at both the address input port and the data output port of the memory banks. Of course,

more resources are going to be used for the replicated processing elements in the datapath because of the unrolling, but they are purely used for computation rather than being steering overhead, making the resulting design more area and power efficient.

(a)

```
void imageWindowingUnrolled(int imageOutput[N][N], imageInput[N][N]){
   int i, j;
   for (i = 1; i <= N-1; i++)
      for (j = 1; j <= N-2; j +=2)
         imageOutput[i][j]   = w0*imageInput[i-1][j-1] + w1*imageInput[i-1][j]
                             + w2*imageInput[i-1][j+1] + w3*imageInput[i][j-1]
                             + w4*imageInput[i][j]     + w5*imageInput[i][j+1]
                             + w6*imageInput[i+1][j-1] + w7*imageInput[i+1][j]
                             + w8*imageInput[i+1][j+1];
         imageOutput[i][j+1] = w0*imageInput[i-1][j]   + w1*imageInput[i-1][j+1]
                             + w2*imageInput[i-1][j+2] + w3*imageInput[i][j]
                             + w4*imageInput[i][j+1]   + w5*imageInput[i][j+2]
                             + w6*imageInput[i+1][j]   + w7*imageInput[i+1][j+1]
                             + w8*imageInput[i+1][j+2];
}
```

(b)



(c)

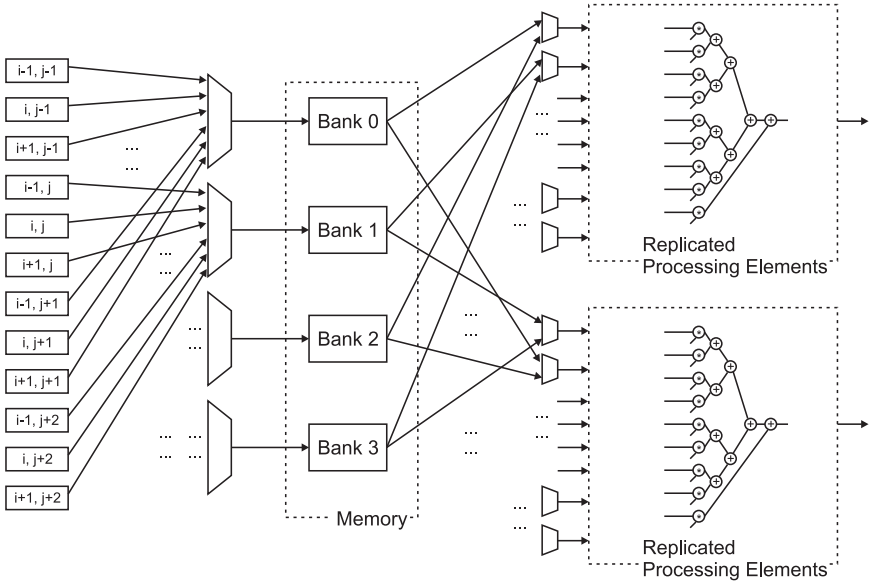


FIGURE 3.2: The unrolled version of the generic bidimensional sliding window filter. a) Code, b) Memory accesses to the input image for some iterations, c) Synthesized datapath

In order to assess the impact of bank switching quantitatively, we define the *area efficiency* as the product of latency, i.e., the overall execution time of a synthesized algorithm, and its area cost. Usually, the area cost of FPGA designs is quantified in terms of elementary components such as Look-Up Tables (LUTs), Flip-Flops

(FFs), and, in some cases, Digital Signal Processing (DSP) blocks; we consider here LUTs primarily because FPGA designs are often LUT-bounded.

TABLE 3.1: Example of area implications of bank switching

Partitioning solution	**Banks**	**Bank switching**	**Latency·Area**
Lattice	4	0	1
Lattice	8	2*	2.02
Lattice	16	4*	3.43
Hyperplane	4	2**	2.44
Hyperplane	8	4**	4.33
Hyperplane	16	8**	8.37

As a preliminary experimental confirmation of bank switching hardware implications, consider table 3.1, concerning an image resizing kernel with bilinear interpolation synthesized to an FPGA technology through a commercial HLS toolchain, namely Vivado HLS by Xilinx [36]. For two different partitioning solutions, a lattice based and an hyperplane based, the table lists the number of memory banks, the amount of bank switching, and the latency-area product (normalized to the case of minimum bank switching). Due to the regularity of the considered benchmark, the amount of bank switching is the same for all memory references, so they are not considered separately. For the bank switching amount, two asterisks indicate that we need multiplexers both on the address and data output ports, while a single asterisk indicates that only data output multiplexers are present. The benchmarks were not manipulated before synthesis, e.g. by unrolling the loop.

The results in the table clearly point out the impact of bank switching on area efficiency. Solutions associated with lattices, are more regular than hyperplanes[1]. This regularity is reflected directly by more efficient datapaths generally reducing the steering logic needed. This is also one more confirmation of the enhanced efficiency of the lattice-based partitioning technique presented in chapter 2 compared to hyperplane-based techniques.

In the following sections, we model bank switching mathematically and also present a formal model to determine the unrolling factors for a certain algorithm in order to avoid bank switching completely.

[1]This happens in the sliding window filter example because of a rectangular window.

3.3 Mathematical formalization

Let the $\mathbb{Z}^n$ set represent the whole n-dimensional memory space containing the n-dimensional array A, and $\mathcal{L} = \{\mathbf{B} \cdot \vec{z}, \vec{z} \in \mathbb{Z}^n\}$ be the integer lattice used for partitioning. $\mathcal{L}$ is a subset (and a subgroup) of $\mathbb{Z}^n$. As seen in the previous chapter, there are overall $NB = \det(\mathbf{B})$ different translate lattices, constituting a partition of $\mathbb{Z}^n$, each corresponding to a physical memory bank. To associate a different location to the corresponding bank, we rely on the following property, proven in [57]. Let $\mathbf{S} = \mathbf{U}_1\mathbf{B}\mathbf{U}_2$ be the *Smith Normal Form* (SNF) of the lattice basis $\mathbf{B}$. $\mathbf{S} = \{s_{ij}\}$ is a diagonal matrix and its diagonal elements, denoted $\vec{s} = (s_{00}, s_{11} \ldots s_{n-1,n-1})$, are such that s_{ii} divides $s_{i+1,i+1}$, while $\mathbf{U}_1$ and $\mathbf{U}_2$ are unimodular matrices, and hence $NB = \det(\mathbf{B}) = \det(\mathbf{S}) = \prod_{i=0}^{n-1} s_{ii}$. Then, the *modular mapping* defined by $\sigma(\vec{z}) = \mathbf{U}_1 \cdot \vec{z} \bmod \vec{s}$, $\vec{z} \in \mathbb{Z}^n$ has $\mathcal{L}$ as the *kernel*, i.e., $\vec{z} \in \mathcal{L} \Rightarrow \mathbf{U}_1 \cdot \vec{z} \bmod \vec{s} = \vec{0}$.

As a consequence, the quotient group $\mathbb{Z}^n/\mathcal{L}$, containing the NB translates of $\mathcal{L}$, each corresponding to a memory bank, is in one-to-one correspondence with the different values taken on by the modular mapping $\sigma(\vec{z}) = \mathbf{U}_1 \cdot \vec{z} \bmod \vec{s}$. In other words, each bank can be denoted by an n-dimensional index $\vec{b_F} = (b_0, \ldots, b_{n-1})$ with $b_i < s_{ii}\ \forall i$ and, given a particular memory reference of indices $\vec{z} = (z_0, \ldots, z_{n-1})$ (e.g., $A[3][5]$ in a bidimensional space), the corresponding bank can be simply obtained as $(b_0, \ldots, b_{n-1}) = \sigma(\vec{z}) = \mathbf{U}_1 \cdot \vec{z} \bmod \vec{s}$. Consider now a memory access in the code, and the corresponding access function $\vec{z} = F(\vec{v}) = \mathbf{F} \cdot \vec{v} + \vec{t_0}$. The bank accessed by the memory reference in a given iteration $\vec{v}$ is $\vec{b_F}(\vec{v}) = \mathbf{U}_1(\mathbf{F} \cdot \vec{v} + \vec{t_0}) \bmod \vec{s}$. Quantifying bank switching is equivalent to computing how many different values are taken by the modular mapping $\vec{b_F}(\vec{v})$ as $\vec{v}$ spans $\mathbb{Z}^n$ (we assume that the memory array is large enough to take all the different values of $\vec{b_F}$, which is normally the case in practice). In the following, we show how the previous formulation can drive the choice of the unrolling factors in order to reduce, or even eliminate, the bank switching effect. Notice that, as discussed later, we assume that the choice of the unrolling factors is not constrained here by possible data dependencies across iterations.

To formalize the impact of loop unrolling note that, in terms of memory accesses, the essential effect of unrolling is to change the expressions of the memory access

functions. In general, let e_i and e_j be the unrolling factors of the two loops in the nest. Each memory access function of the form $F(j,i) = \begin{pmatrix} f_{00} & f_{01} \\ f_{10} & f_{11} \end{pmatrix} \begin{pmatrix} j \\ i \end{pmatrix} + \vec{k}$ is transformed after unrolling into:

$$F'(j,i) = \mathbf{F'} \cdot \vec{v} + \vec{k'} = \begin{pmatrix} e_j \cdot f_{00} & e_i \cdot f_{01} \\ e_j \cdot f_{10} & e_i \cdot f_{11} \end{pmatrix} \cdot \begin{pmatrix} j \\ i \end{pmatrix} + \vec{k'}$$

with $\vec{k'}$ being a suitable constant vector (not affecting bank switching). The following result connects the amount of bank switching with the transformation induced by the unrolling factors. Let $\mathbf{U}_1 = \begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix}$ and $\vec{s}$ refer to the SNF of the lattice basis $\mathbf{B}$ used for partitioning. Then, the bank mapping function is

$$\vec{b_F}(\vec{v}) = \mathbf{U}_1 \cdot \left[\mathbf{F'} \cdot \vec{v} + \vec{k'}\right] \bmod \vec{s} = \left[\mathbf{T} \cdot \vec{v} + \vec{k''}\right] \bmod \vec{s}$$

where $\mathbf{T} = \mathbf{U}_1 \cdot \mathbf{F'} = \begin{pmatrix} t_{00} & t_{01} \\ t_{10} & t_{11} \end{pmatrix} = \begin{pmatrix} q_{00} \cdot e_j & q_{01} \cdot e_i \\ q_{10} \cdot e_j & q_{11} \cdot e_i \end{pmatrix} =$

$$\begin{pmatrix} (f_{00}u_{00} + f_{10}u_{01}) \cdot e_j & (f_{01}u_{00} + f_{11}u_{01}) \cdot e_i \\ (f_{00}u_{10} + f_{10}u_{11}) \cdot e_j & (f_{01}u_{10} + f_{11}u_{11}) \cdot e_i \end{pmatrix}$$

and $\vec{k''} = \mathbf{U}_1 \cdot \vec{k'}$ is a constant term.

Bank switching is thus given by the number of different values taken on by $[\mathbf{T} \cdot \vec{v}] \bmod \vec{s}$. In case the two rows of $\mathbf{T} \cdot \vec{v}$ are a multiple of s_{00} and s_{11}, respectively, the above modular expression takes on always one value. A sufficient condition is ensured by the following choice for the unrolling factors:

$$e_j^* = \operatorname{lcm}\left(\frac{s_{00}}{\gcd(q_{00}, s_{00})}, \frac{s_{11}}{\gcd(q_{10}, s_{11})}\right)$$

$$e_i^* = \operatorname{lcm}\left(\frac{s_{00}}{\gcd(q_{01}, s_{00})}, \frac{s_{11}}{\gcd(q_{11}, s_{11})}\right)$$

In fact, taking the first row as an example, we have that $e_j^* = \alpha \cdot \frac{s_{00}}{\gcd(q_{00}, s_{00})}$ and $e_i^* = \beta \cdot \frac{s_{00}}{\gcd(q_{01}, s_{00})}$ by the above choices. Write $q_{00} = q'_{00} \gcd(q_{00}, s_{00})$ and

$q_{01} = q'_{01}\gcd(q_{01}, s_{00})$. Then, the first row of $\mathbf{T} \cdot \vec{v}$ is

$$q_{00}e^*_j \cdot j + q_{01}e^*_i \cdot i = q'_{00}\gcd(q_{00}, s_{00}) \cdot \alpha\frac{s_{00}}{\gcd(q_{00}, s_{00})} \cdot j +$$

$$q'_{01}\gcd(q_{01}, s_{00}) \cdot \beta\frac{s_{00}}{\gcd(q_{01}, s_{00})} \cdot i = s_{00}\left[q'_{00}\alpha \cdot j + q'_{01}\beta \cdot i\right]$$

Notice that, although we considered two nested loops for simplicity, the treatment can be easily generalized to the case of an n-level loop nest. Furthermore, the technique was discussed for a single memory reference. When more memory accesses to the same array are found in the loop body, the technique can still be applied to each reference separately, then picking for each unrolling factor the least common multiple of the different solutions determined.

3.4 Experimental evaluation

In order to validate the theoretical results, Xilinx VivadoHLS [36] has been used as synthesis tool targeting a Xilinx Virtex-7 device [58]. The C code featuring partitioned memory accesses for each case study has been written and given as input to VivadoHLS. We have chosen three benchmarks, that are:

- An Image Resizing algorithm using bilinear interpolation
- A 2D Gauss-Seidel kernel
- A 2D-Jacobi algorithm

The benchmarks are representative of typical HLS applications and widely used in many scientific applications. The Gauss-Seidel is often used for the resolution of linear systems whereas Jacobi is a popular algorithm for solving Laplace differential equations on a regularly discretized square domain. The partitioning solutions were derived following the approach explained in chapter 2.

In table 3.2 there are the solutions, i.e. the unrolling factors, returned by the application of the methodology for an increasing number of banks. The Gauss-Seidel and Jacobi kernel exhibit the same behavior in terms of unrolling choices

for avoiding bank switching since they both process the input array sliding on it with a unitary stride. In contrast, the Image Resizing kernel processes the input data block-wise; in fact, it can be noticed that the unrolling factors tend to be lower because the accesses pattern helps avoid bank switching by itself. According to the methodology, the identified unrolling factors eliminate bank switching and, hence, avoid degrading the area efficiency due to complex steering logic.

Image Resize	**NB**	e_i^*, e_j^*
	1	1,1
	2	1,1
	4	1,1
	8	1,2
	16	2,2
2D Gauss-Seidel	**NB**	e_i^*, e_j^*
	1	1,1
	2	1,2
	4	2,2
	8	2,4
	16	4,4
2D Jacobi	**NB**	e_i^*, e_j^*
	1	1,1
	2	1,2
	4	2,2
	8	2,4
	16	4,4

TABLE 3.2: Unrolling factors returned by the application of the methodology

Figure 3.3.a depicts the area efficiency (latency-area product), for the Image Resize algorithm in function of the number of banks; each curve corresponds to a different unrolling configuration. Figure 3.3.b and figure 3.4.a do the same for the Gauss-Seidel and Jacobi kernels. The latency-area product, in each plot has been normalized to the case of 1 memory bank, hence it shows the area efficiency improvement/loss with respect to a non-partitioned solution. When the curve decreases it means we are gaining efficiency.

For each of the unrolling configurations, first the area latency product is on the decrease, i.e., the efficiency improves; then after a certain number of banks, it starts increasing. The point in which the trend changes is exactly where bank switching starts. For each curve, we can distinguish clearly two regions; one where

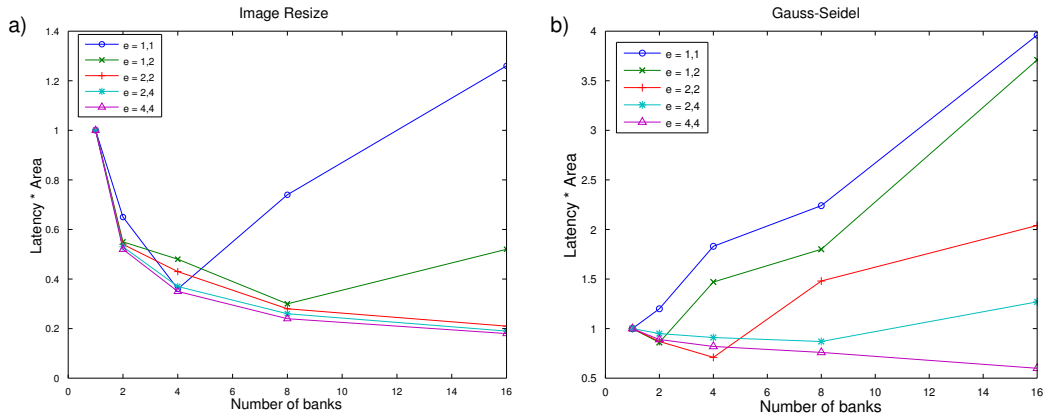


FIGURE 3.3: Area efficiency (normalized to the case of 1 bank) versus number of banks. a) Image Resize algorithm, b) Gauss-Seidel kernel

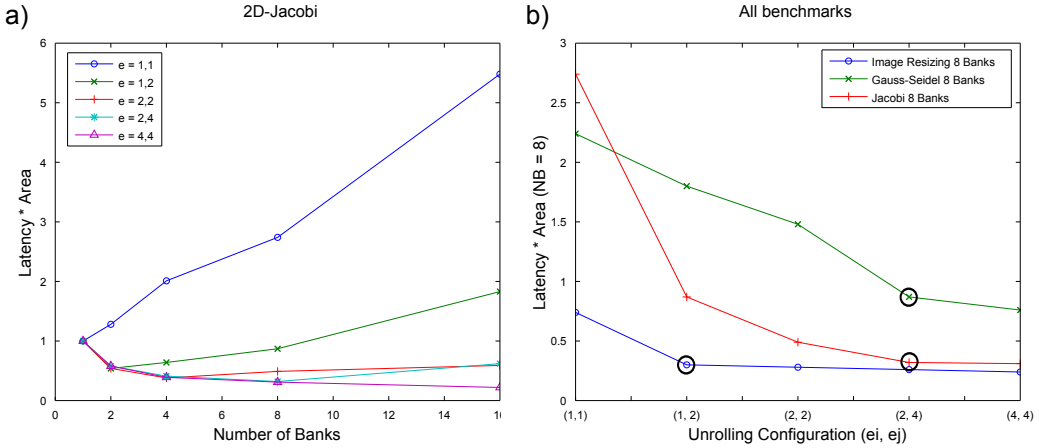


FIGURE 3.4: a) Area efficiency (normalized to the case of 1 bank) versus number of banks for the 2D-Jacobi. b) Area efficiency versus unrolling configurations for all benchmarks and 8 memory banks. Each sample is intended to be normalized to the case of a rolled version using 1 memory bank. The black circles are the solutions predicted by our methodology for avoiding bank switching, unrolling beyond them does not yield substantial advantages

the curve decreases that is the region without switching and the other where it increases because of increasing switching. For example, in figure 3.3.b (Gauss Seidel), the unrolling configuration e=(2,2) is free of switching until 4 banks. The unrolling configuration e=(2,4) can resist until 8 banks, whereas a completely rolled version e=(1,1), is subject to switching immediately using only two memory banks. Therefore, the more aggressive is the unrolling, the bigger is the number of banks we can use without incurring switching.

Moreover, independently of the number of banks, loop unrolling can effectively improve the efficiency of the synthesized circuit. This can be noticed in figure 3.4.b that depicts the latency-area product versus unrolling configurations in case of 8 memory banks. The figure shows the area efficiency against unrolling configurations for a fixed number of memory banks (NB = 8) for the three benchmarks. We can notice that up to a certain point, unrolling benefits efficiency considerably due to reduced amounts of bank switching. Beyond those points (marked in the plot by black circles), there are diminishing or null returns. Those points are actually the minimum unrolling factors needed to avoid bank switching completely and are all correctly predicted by the mathematical model. The improvement of an unrolled circuit with unrolling factors e=(4,4) over a rolled version is respectively of 5.6x, 2.3x and 2x for the three benchmarks.

Experiments also give more confirmations of the validity of the mathematical model described in section 3.3. For example, our mathematical model has predicted (see table 3.2) that an unrolling configuration of e=(2,2), is the minimum amount of unrolling needed in order not to have switching with 4 memory banks in case of Gauss-Seidel. As can be noticed in figure 3.3.b, that is exactly what happens in the experiment; the red curve, corresponding to e=(2,2), is free of switching up to 4 banks, after which the efficiency sharply decrease. In fact, a more aggressive unrolling (light blue and purple curves) can guarantee no switching also with more banks, but lower unrolling factors (e = (1,1) and e = (1,2)) cannot avoid it; in fact, the curves with (e = (1,1) and e = (1,2)) are in their switching regions with 4 banks.

The above conclusions refer to the lattice-based technique, but similar considerations apply to cyclic [19] and hyperplane-based [46] partitioning. However, the three techniques may perform very differently in terms of area and performance. In particular, while there can be a significant advantage in terms of latency when comparing the lattice-based approach with the cyclic approach, the hyperplane-based solutions are general enough to achieve full parallelization in the memory accesses for the very regular benchmarks considered. The most important advantage of the lattice-based method over hyperplanes is related to the area overhead, as the bank switching effect occurs more frequently when only hyperplanes are used.

Figure 3.5 depicts latency and area obtained for the 2D-Image Resizing Kernel using the three techniques. As can be noticed, lattices and hyperplanes are comparable in latency but the area advantage increases with the number of banks up to more than 2 times.

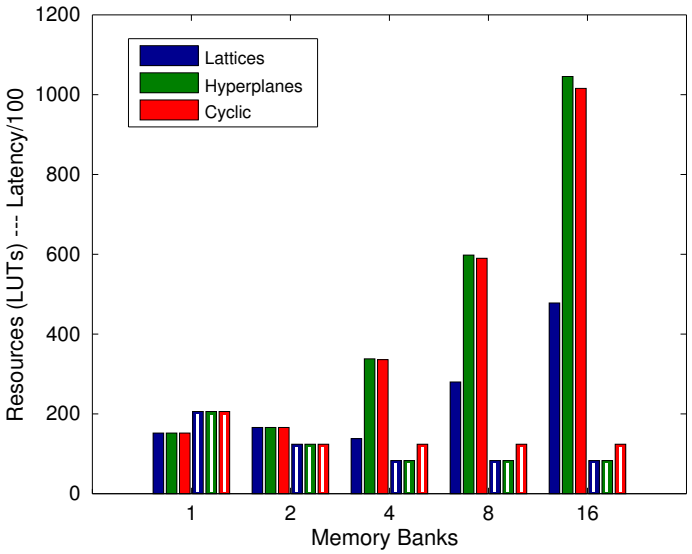


FIGURE 3.5: Comparison of the three techniques. The area is reported by solid bars, the latency by white filled bars.

3.5 Remarks and conclusions

In this chapter we have recognized the main relationship existing between memory partitioning and area overhead. Bank switching has been identified as the main cause of efficiency decrease, and for this reason, we have proposed a mathematical formalization and an approach for reducing it based on loop unrolling. In the light of the experimental results two general and non-intuitive conclusions can be drawn.

- In the presence of partitioning, a more aggressive unrolling does not directly turn into larger area, but instead depends on how partitioning is done. The reason is that the steering logic may outpace the area increase caused by unrolling.

- A lattice-based partitioning leads to a more efficient synthesis compared to the hyperplane-based one. This is due to fact that the former is more general and includes the solutions proposed by the latter; an hyperplane based approach may exclude the solutions having the highest spatial regularity affecting area badly.

As a last point, regarding the usage of loop unrolling to improve the synthesis it is important to notice that we have not taken into account loop carried and dataflow dependencies, which may hurt the unrolling effectiveness. Those dependencies are inherent in the algorithm implementation rather than its synthesis and, although loop unrolling always decreases bank switching, it can hurt efficiency in presence of data dependencies since some allocated resources might not be used in parallel. However, notice that:

- If there are dependencies in an unrolled loop such that computation cannot take place in parallel, current advanced HLS tools are generally able to detect such situation and not allocate more resources than strictly necessary;
- Restricting the interest to loops with uniform dependencies, we can always extract $n-1$ level of parallelism in a loop hierarchy made of n loops [59]. In other words, transforming the loop nest (possibly automatically [29, 60]) we can unroll all the loops, apart from one (e.g. the outermost, in the case we have parallelized the inner loops of the hierarchy) avoiding building an underutilized datapath.
- Even if there are data flow dependencies among operators and HLS allocates more resources than necessary, the loss in efficiency, if any, with respect to the non-unrolled case is limited by the fact that HLS tools can still take advantage of an unrolled code by exploiting the parallelism of accessory operations (e.g. arithmetic operations for the generation of addresses). Therefore, overall, unrolling is likely to increase efficiency because of the benefit in terms of avoided bank switching.

Although the above arguments, introducing loop carried dependencies in the formalization is an evident opportunity of improvement that is not going to be missed in future developments.

Chapter 4

Interconnecting memory banks and processing elements

4.1 Introduction

As seen previously, the memory architecture is of paramount importance for performance and area of synthesized circuits. In chapter 3 the implications of the partitioning choices on area have been analyzed; different solutions lead to different steering circuitry, in that they affect how computing elements are interconnected to memories. However, there was an implicit assumption behind the approach: if a computing resource eventually needs data from a memory bank it is directly connected to it by means of multiplexers. As a consequence, this generates an actual partial crossbar as communication infrastructure. In this chapter that assumption is removed and a novel approach for targeting the interconnect architecture to the application requirements is proposed. Partial crossbars are only a single point in a much bigger design space that is constrained, as usual, by available area, required performance and energy consumption. Differently from previous chapters, the discussion is more system-level oriented; processing elements are not limited to simple components anymore, such as adders or multipliers inferred by an high-level synthesizer but can be of any type; for example, they can be general purpose processors or specialized accelerators. However, the orientation of the approach remains the same, it targets performance optimization and analyzes

area impact as side effect trying to limit it. In the dedicated experimental section, the advantages of the proposed methodology over trivial choices, such as crossbars and shared buses, as well as other existing approaches, are shown using many benchmarks having different traffic profiles.

4.2 The importance of interconnects and concurrency

The choice of the underlying topology, depending on specific application requirements, is critical because it affects the entire inter-component data traffic and impacts the overall system performance and cost [61]. Not surprisingly, the industry and the academia are continuously introducing new architectures and components dictating the evolution of on-chip communication. First-generation on-chip interconnects consist of conventional bus and crossbar structures. Buses are mostly wires that interconnect IP cores by means of a centralized arbiter. Examples are AMBA AHB/APB [4] and CoreConnect from IBM (PLB/OPB) [5]. However, the growing demand for high-bandwidth interconnects has led to an increasing interest in multi-layered structures, namely crossbars, such as ARM AMBA AXI [6] and STBus [7]. A crossbar is a communication architecture with multiple buses operating in parallel. A crossbar can be full or partial depending on the required connectivity between masters and slaves. Slaves are quite often memories, or memory banks as seen in previous chapters. Crossbars can provide lower latency and higher bandwidth, although this benefit usually comes at non-negligible area costs compared to shared buses. Buses and crossbars are tightly-coupled solutions, in that they require all IP cores to have exactly the same interfaces, both logically and in physical design parameters [8]. Networks on Chip (NoCs) [9] enable higher levels of scalability by supporting loosely coupled solutions. NoCs are particularly well-suited when targeting reusability, maintainability, and testability as the main design objectives, while bridged buses/crossbar architectures ensure low-power, high-performance, and predictable interconnects.

Due to the large spectrum of choices for the definition of the on-chip interconnect, including the selection of the appropriate components and topology, determining the solutions that best suit given applications requirements is a non-trivial task [61], especially when targeting embedded many-core systems. The interconnect topology should be designed so as to provide a high aggregate bandwidth by allowing many separate communication tasks to operate concurrently. In order to achieve this goal, taking advantage of spatial locality, i.e. placing the blocks that communicate more frequently closer to each other, is critical [62]. In particular, the communicating elements can be grouped in *local domains* that, depending on their internal architecture, may allow different communication interactions to take place concurrently.

In this scenario, we can recognize three different levels of parallelism:

- Global parallelism: communication in different local domains can take place concurrently.
- Local parallelism or intra-domain parallelism: local domains may be implemented by inherently concurrent architectures (i.e. crossbars).
- Inter-domain parallelism: multiple parallel paths among different local domains are allowed.

Figure 4.1 exemplifies these three forms of communication parallelism. The three levels of parallelism can potentially enable lower-cost, lower-latency concurrent interconnects that scale well with the system size.

On the other hand, taking into account dependency constraints between communication tasks is essential to understanding the actual communication timing and properly dimensioning the interconnect [63]. Based on this observation, this chapter proposes a complete methodology supporting joint interconnect synthesis and communication scheduling based on communication task dependencies. The resulting architecture improves the level of communication parallelism that can be exploited, while keeping area requirements low, as proven by some case-studies presented later.

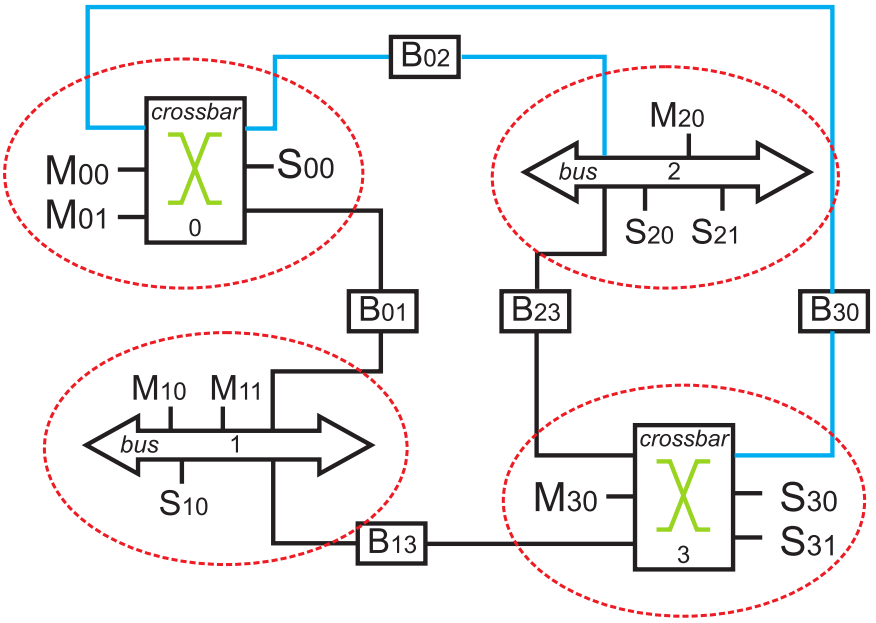


FIGURE 4.1: An example of topology exhibiting three levels of parallelism ("M" : master, "S" : slave, "B" : bridge.): Global parallelism (red), Intra-domain parallelism (green), and Inter-domain parallelism (blue).

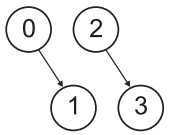
4.3 Problem definition

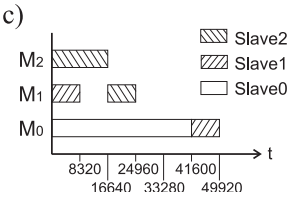
4.3.1 Definitions

An application-specific on-chip network topology synthesis and communication scheduling method is proposed. It takes as input the information on the communication tasks and their dependency relationships, and generates the specification of an on-chip interconnect along with the communication task schedule. In the following we give some useful definitions as well as a simple example.

a)

ID	Master	Slave	Traffic
0	0	0	50
1	0	1	10
2	1	1	10
3	1	2	10
4	2	2	20





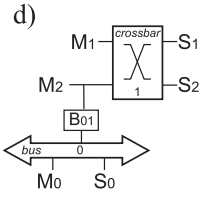


FIGURE 4.2: A few examples ("M" : master, "S" : slave, "B" : bridge.) (a) A *Task List* (TL). (b) A *Dependency Graph* (DG). (c) A *Communication Schedule* (CS). (d) A *Synthesizable Topology* (ST).

Definition 4.1. A *Task List* (TL) is a list, made up of n_{task} *communication tasks*, indexed by a unique *taskID* where each entry t_i contains the *master* and the *slave* involved in the communication and a *cost* c_i (i.e. the data traffic) in terms of number of bytes to be transferred.

The communication traffic between a master and a slave is made up of communication tasks which are non-preemptive atomic entities with an arbitrary load, i.e. the amount of transmitted bytes, transferred in a burst mode. Notice that two different tasks can have the same master/slave pair. This allows modeling any traffic pattern. Figure 4.2.a contains a TL with five tasks where, for each task, the master/slave pair and the amount of traffic they exchange are specified.

Definition 4.2. A *Dependency Graph* (DG) is a *Directed Acyclic Graph* (DAG) in which the vertex set $V = \{v_i : i = 0, 1, ..., n_{task} - 1\}$ is in one-to-one correspondence with the set of communication tasks and the edge set $E = \{(v_i, v_j) : i, j = 0, 1, ..., n_{task} - 1\}$ represents *dependency relationships* between the above tasks. An edge connects one vertex to another, such that there is no way to start at some vertex v_i and follow a sequence of edges that eventually loops back to v_i again. As a result, it gives rise to a partial order relationship $\leq$ on its vertices, where the relationship $v_i \leq v_j$ occurs exactly when there exists a directed path from v_i to v_j. It describes a set of parallel communication tasks (the vertices) having some inter-task precedence relationships, i.e. dependency constraints. Unlike the classical scheduling problems, the inputs and outputs of a node do not convey data, but they only express communication dependency constraints. The communication represented by a node can start as soon as all its parent nodes have finished. A node with no parents is called *source*, while a node with no children is called *sink*. The weight of a node is called the *communication cost* (i.e. the data traffic of a node v_i in terms of amount of bytes to be transferred) and is denoted by $c(v_i)$, while the weight on an edge is called the *computation cost* of the edge and is denoted by $w(v_i, v_j)$. This cost represents the delay, due to computation, that might take place between two consecutive communication tasks. Notice that $c(v_i) = c_i \ \forall i : v_i \in V$.

As an example, consider the DG in figure 4.2.b. It expresses the existing dependency relationships between the tasks present in the TL in figure 4.2.a. In this case, there are dependency constraints between t_0 and t_1 and between t_2 and t_3.

Definition 4.3. A *Communication Schedule* (CS) is defined as a vector indexed by a task identifier containing in each entry s_i the *start time* of communication task t_i in clock cycles. The *latency* of the CS is the number of cycles to execute the entire schedule, or equivalently, the difference in start time of the sink and source vertices. If two or more sources/sinks are available, the source/sink with the smallest/biggest start time is used. Since a feasible solution must satisfy data dependency constraints, the start time of a communication task is at least as large as the start time of each of its predecessors plus their *execution delay* d_i. An execution delay is an integer representing the amount of clock cycles required to execute a communication task in a given architecture. Therefore a schedule must satisfy the following relations:

$$s_i \geq s_j + d_j \quad \forall i, j : (v_j, v_i) \in E \tag{4.1}$$

It is of course desirable to find the *minimum latency* schedule that can be run on a physical topology under given area constraints.

In figure 4.2.c, a possible scheduling solution is depicted. It is the as-soon-as-possible (ASAP) schedule for the given DG and TL. Notice that although t_3 and t_4 do not have any data dependency, their execution is serialized due to a slave incompatibility: two different tasks cannot simultaneously access the same slave. Similarly, accessing simultaneously the same bridge in a multi-hop communication would lead to a task incompatibility.

Definition 4.4. A *Synthesizable Topology* (ST) is made up of *local domains* interconnected by means of bridges. A local domain can be either a bus or a crossbar where some masters and slaves are involved in communication tasks. Communication on a global scale between different local domains can be done via a proper configuration of bridge address ranges. The number of bridges crossed by a communication task t_i is called *hop count* and is denoted by h_i. Formally, a Synthesizable Topology is defined as a triplet (C, O, I). C is the set of *clusters* that is in one-to-one correspondence with the set of local domains. O is the set of directed

edges between two clusters which physically correspond to a unidirectional bridge decoupling the local domains, which makes the inter-cluster traffic possible. I is the implementation function specifying the local domain architecture. Given an element $c \in C$ which contains n masters and m slaves, $I(c)$ can be equal to either an $n \times m$ bus or a $l \times k$ crossbar, with $2 \leq l \leq n$ and $2 \leq k \leq m$ where l and k are the number of master and slave ports. In the crossbar case, the output will include a *connection matrix* (call it Z) that fully specifies the connections between every master port and every slave port, possibly defining sparse crossbars. Obviously, the values of l and k directly impact the resulting area requirements. An ST must meet the area constraints and must allow the concurrent executions identified in the scheduling solution.

Figure 4.2.d depicts the synthesizable topology made of two clusters (C_0 and C_1) and one bridge allowing task t_1 to be performed. C_0 is implemented as a shared bus and C_1 as a crossbar. This topology exhibits the required parallelism to run the above schedule. This is obtained by taking advantage of both global parallelism, because there are two clusters running concurrently, and local parallelism, because cluster C_1 is implemented as a 2×2 crossbar.

It is important to emphasize that the two sets C, O and the implementation function I of a ST are each responsible for a different level of parallelism. The C set defines the number of clusters and the distribution of the masters and slaves between them. Since the communication in each local domain is concurrent, the global-parallelism depends on the definition of the C set. The O set contains the number and the position of the bridges and consequently its configuration affects the inter-domain parallelism. Last, function I returns the architectural implementation, i.e. a bus or a crossbar, of each domain and hence the number of local concurrent channels, i.e. the degree of local or intra-domain parallelism.

4.3.2 Objectives

The interconnect synthesis problem is stated as follows. Given

- a communication Task List (TL) containing, for each task, the master and the slave involved and the amount of bytes to be transferred;

- a Dependency Graph (DG) describing the possible inter-task precedence relationships, i.e. data dependency constraints;
- area constraints;

find:

- a Synthesizable Topology (ST) specification based on a heterogeneous bus/crossbar architecture minimizing the target cost function;
- a minimum-latency communication task schedule (CS) compatible with the identified architecture.

The problem of communication scheduling and synthesizable topology definition are deeply interrelated. The schedule identifies the precise start time of each communication task. The start times must satisfy the DG dependencies, which limits the amount of parallelism of the communication, because any pair of communication tasks involved in a direct dependency, or a chain of dependencies, may not execute concurrently. Determining the concurrency of the topology implementation, scheduling affects the resulting performance and cost. Similarly, the maximum number of concurrent communication tasks at any step of the schedule is bounded by the interconnection topology. The final architecture must be able of handling the parallelism offered by a certain schedule.

The cost of the interconnect in terms of area can be upper bounded to satisfy some design requirements. When resource constraints are imposed, the number of concurrent communication tasks whose execution can overlap in time is limited by the parallelism of the topology. Tight bounds on the interconnect area cause serialized communication. As a limiting case, a scheduled sequencing graph may be such that all communication tasks are executed in a linear sequence. This is indeed the case when only a single bus is available to execute all communication tasks. On the contrary, the unconstrained solution is a crossbar with a suitable number of connections enabling the maximum number of concurrent communication tasks according to DG dependencies. Area/latency trade-off points can be derived as the solutions to different constrained scheduling problems. The methodology jointly

considers them to derive a heterogeneous interconnection topology satisfying both area and dependency constraints.

4.3.3 Assumptions

Following are the main assumptions we made for the interconnection design problem:

- First, we target a memory mapped communication scenario where the memory blocks and I/O devices are mapped to slave nodes while the initiators, such as CPUs and DMAs, are mapped to master nodes sharing the same address space in the system. IP cores that have both roles, i.e. initiator and target, are mapped to both a master node and a slave node, handled like the other nodes.
- All the interconnect channels use the same protocol and channel width.
- The routing is deterministic and defined by statically setting bridge addresses.
- We neglect the overhead due to bus contention when computing latencies.
- A pre-characterized library of crossbars and buses is ready for various configurations (e.g., different numbers of masters and slaves). We built an accurate area cost model, obtained through interpolation of extensive area characterizations using RTL synthesis. We will explain the proposed method and show the experimental results for AXI-based interconnects. However, the proposed method can also be applied to crossbar-based interconnects implementing other protocols.

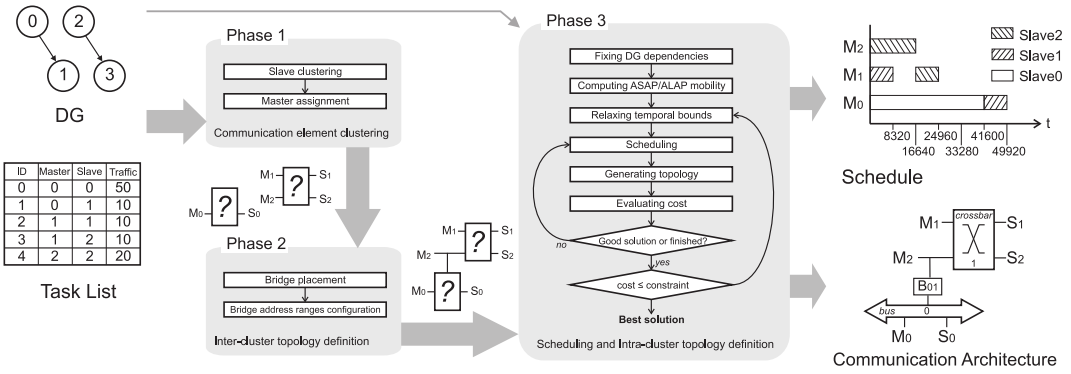


FIGURE 4.3: Proposed interconnect synthesis flow

4.4 Proposed methodology

4.4.1 Overview of the proposed method

This section presents a novel approach to automatically build highly parallel interconnection structures, minimizing the communication overhead and maximizing the degree of parallelism that can be achieved by concurrent communication interactions. The approach also deals with dependency constraints expressed using the DG. The proposed topology synthesis flow, shown in figure 4.3, consists of three phases:

1. Communication elements clustering
2. Inter-cluster topology definition
3. Scheduling and intra-cluster topology definition

The global parallelism is made possible by defining *local domains*, i.e. subsets of nodes in the interconnect that can directly communicate with each other, independent of other domains where different communication interactions can take place concurrently. Local domains are implemented as either crossbars or buses. Crossbars also enable local parallelism, i.e. concurrent communication among nodes within the same domain.

The first step performed (Phase 1) is the clustering of communicating elements in local domains. The capability of exploiting spatial locality in the communication

patterns is here key, as we attempt to place the nodes that communicate more frequently closer to each other, minimizing the traffic between communicating elements and matching the localized traffic patterns induced by a given application.

In Phase 2 the clusters are connected in order to make all inter-cluster communications feasible by means of bridges. By properly setting the mapping of the address spaces in each bridge, furthermore, multiple physical paths between different domains can be realized. Multiple paths introduce flexibility in the network topology, as they create a further opportunity for balancing the load across the interconnect, as well as concurrency in inter-cluster communication (inter-cluster parallelism).

Finally, we need to figure out how single clusters will be implemented (Phase 3). This step is performed jointly with the communication scheduling: an iterative procedure finds an optimal communication tasks schedule (in terms of latency) and a global topology containing enough resources to execute the found schedule. The identification of the final implementation for the local domains is driven by the area constraints. The chosen implementation must ensure a degree of intra-cluster parallelism compatible with the identified schedule. The definition of an effective method generating the intra-cluster architecture given a global scheduling solution is essential.

Notice that these three steps are each responsible for the two sets C and O and for the implementation function I, and, hence, for a different level of parallelism. Notice that the above approach targets a single application. In case different applications are anticipated to share the same interconnect architecture, the methodology can still be adopted by identifying a clustering that averages the characteristics of all applications, similar to cluster ensemble techniques [64]. The three main steps covered by the methodology are explained in the following sections.

4.4.2 Communication elements clustering

As shown in figure 4.3, Phase 1, clustering is one of the main steps involved in the interconnect architecture definition. In fact, the quality of clustering heavily affects the degree of communication parallelism the final interconnect can exploit.

Each cluster corresponds to a local domain made up of some masters and some slaves connected by a local communication architecture. Clustering takes place in two separate substeps. First, slaves are clustered in order to form local domains. Then, masters are assigned to local domains according to the traffic profile. In the following the two substeps are described.

4.4.2.1 Hierarchical slave clustering

This sub-step takes as input the TL and provides as output a part of the C set of the Synthesizable Topology that will be taken as input by the following sub-step in order to determine the remaining elements. Therefore, the clustering determines how many local domains will be present and which slaves belong to which domains. Notice that the clustering step fixes the number of clusters in C and determines the partitioning of the slave nodes, while the allocation of master nodes is not constrained at this step. Since an agglomerative hierarchical clustering is performed, the number of merging steps is essential to the architecture definition for both the area efficiency and the global communication overhead. The more clustering steps are performed, the fewer clusters are created at the expense of area occupation. This phase moves the degree of parallelism from global to local parallelism. To cope with this problem, slaves are represented in an Euclidean n-dimensional space. For each slave h, we build an array containing n_{master} elements, where each element m represents the fraction of the total traffic from/to the slave h involving master m. Then, in order to decide which clusters should be combined, the Euclidean distance is used as a measure of dissimilarity between slaves. It can be simply proven that $d(h,k) \leq \sqrt{n_{master}} \quad \forall h,k$, where $d(h,k)$ is the Euclidean distance between slave h and k.

Hence, the clustering algorithm can proceed by merging clusters until we meet a stop condition depending on the worst-case area occupation (computed assuming all possible bridges between clusters and full matrices for intra-cluster communications). Without area constraints, the clustering iterations will lead to a single crossbar, which is consistent with the goals because a single large crossbar guarantees the lowest possible communication overhead. With increasingly stringent

area constraints, on the other hand, we will have a growing number of smaller clusters.

Figure 4.4 shows an example of the clustering algorithm applied to benchmark APPIV (see section 4.5 for the details) where, due to the imposed area constraints, the maximum possible value of inter-cluster Euclidean distance is 0.75, giving an outcome of four clusters.

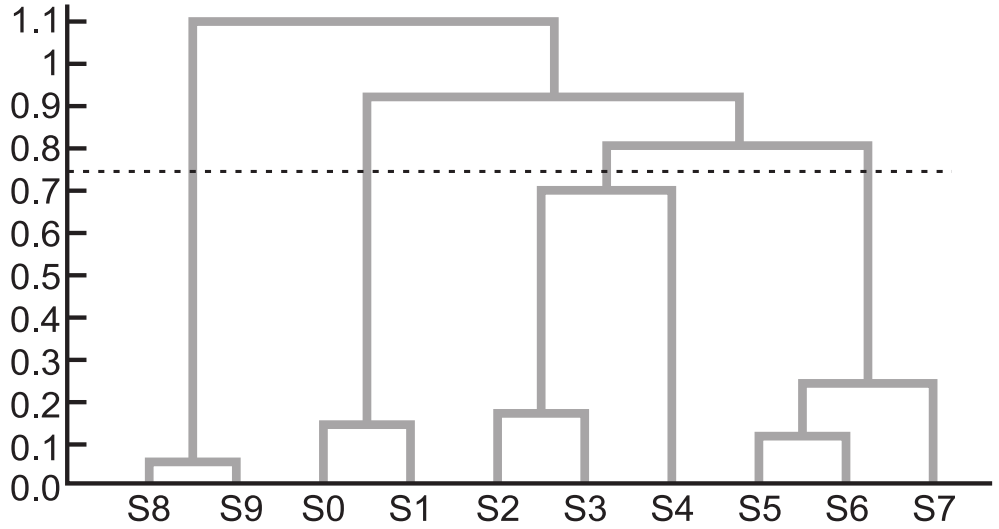


FIGURE 4.4: An example of slave clustering. Slave nodes are on the x-axis, while the Euclidean Distance is on the y-axis.

4.4.2.2 Master assignment to slave clusters

The aim of the second step is to fill the clusters in the C set of the Synthesizable Topology by assigning each master to a unique cluster. We relied on a heuristic improving the communication parallelism and reducing the area overhead enabled by the interconnect topology. The heuristic simply assigns the masters to the clusters with which they exchange most data in order to keep as much communication as possible within single clusters (intra-cluster communication) and to minimize the communication through bridges. In order to perform this assignment, we need to consider the aggregate traffic that each master exchanges with the slaves inside all clusters. At the end of this phase, all masters and slaves are divided in local domains whose internal and external topologies are still to be defined.

4.4.3 Inter-cluster topology definition

As already mentioned, both inter-cluster (i.e. the O set of the ST) and intra-cluster topologies (i.e. the I function of the ST) need to be defined. The aim of this phase is to determine the global inter-cluster architecture: the O set defines the actual links between the clusters (from an implementation viewpoint, it determines the number and the positions of the bridges between crossbars and buses). The intuition behind the proposed algorithm is that, in order to have low-latency communication, we have to maximize intra-cluster communication while keeping inter-cluster communication as low as possible satisfying given area constraints. Hence, we resort to an approach that is biased towards optimizing the I function more than the O set. As shown in figure 4.3, this is achieved in two steps.

1) Populate the O set, making all inter-cluster communications feasible.

2) Define bridge addresses by solving a path balancing problem.

For step 1), we solve an optimum branching problem taking as nodes the C set. We rely on a well-known algorithm (i.e. Edmonds' algorithm [65]). The important clue is that we set the weights on the arcs to the inverse of the communication requirements between clusters. In other words, we prioritize arcs having higher communication requirements.

Concerning step 2), we resort to the approach in [66] that is capable of solving a path balancing problem with a complexity of $O(n \log n)$. This approach allows us to configure bridge addresses in a balanced way without overloading a reduced number of links. Furthermore, this possibly enables the use of parallel communication paths between masters and slaves in different clusters, enabling the exploitation of inter-cluster parallelism. Figure 4.5 exemplifies the benefits of inter-cluster parallelism. Figure 4.5.a shows some communication requirements of an example application, while figure 4.5.c depicts the derived topology. Depending on the bridge address ranges, two different paths can be used between clusters 0 and 2. For instance, the communication between M_{00} and S_{20} can go through B_{02} and the communication between M_{01} and S_{21} can follow a multi-hop path through B_{01} and B_{12}. Figure 4.5.b and figure 4.5.d contain two possible schedules

obtained, respectively, without and with exploiting multipaths. Bridge configurations enabling parallel communication lead here to an improvement in terms of communication overhead roughly equal to 33%.

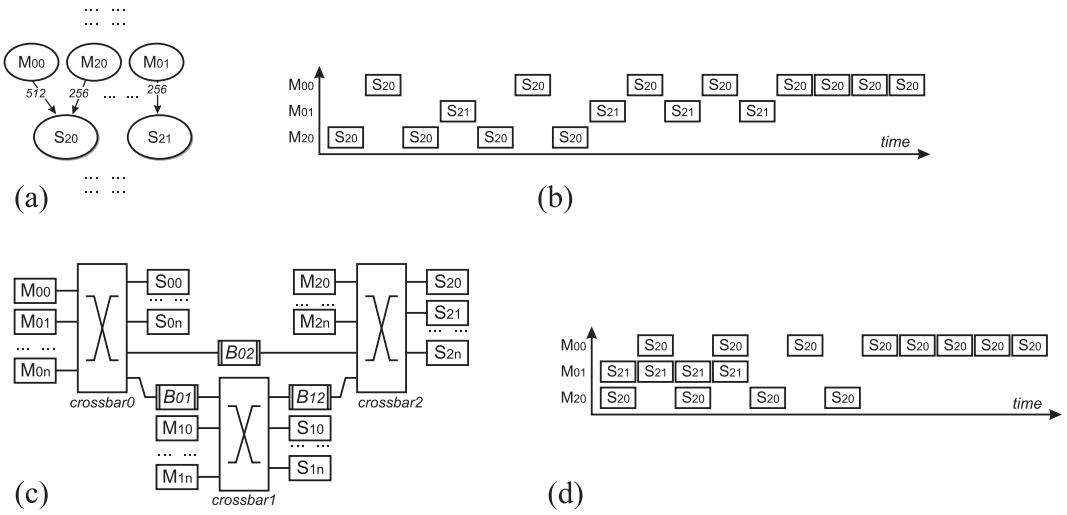


FIGURE 4.5: The effect of considering multiple paths. ("M" : master, "S" : slave, "B" : bridge.) (a) Some communication requirements of an example application. (b) Schedule with no multiple paths. (c) The communication architecture implementation. (d) Schedule with multiple paths.

4.4.4 Scheduling and intra-cluster topology definition

This section presents the approach to the concurrent definition of the intra-cluster architecture and the scheduling solution. This is the most complex phase of the methodology since it concurrently involves the design of the intra-cluster interconnections and the communication scheduling. The whole procedure, consisting of several steps, is depicted in figure 4.3, Phase 3.

At the beginning, the DG is fixed in order to take into account and preemptively solve any potential structural conflict due to slave and bridge accesses. This is because the original DG, taken as input by the methodology, only expresses dependency relationships. The resulting DG will comply with the following rule: *all tasks using the same slave or bridge must be connected by a single path.* This means that there will be a partial order relation for slave and bridge accesses. This is achieved by prioritizing tasks that, in an ASAP schedule, start first. This must

be computed only once by the methodology and is represented by the first step. In addition, in this step the execution delay d_i of each communication task t_i is derived in order to solve equations (4.1). Notice that d_i does not only depend on the cost of the task but also on the interconnect topology: inter-cluster communications require additional clock cycles due to bridge crossing. These values are strictly dependent on the technology and can be computed only after the definition of the global topology.

After fixing the DG, the second step computes the ASAP and ALAP schedules and the mobility values. Then, the iterative phase takes place. First, a temporal bound, subsequently relaxed at each iteration of the outer loop, is fixed (third step). Since we are optimizing the global execution time, we start with the minimum temporal bound, i.e. the latency of the ASAP schedule. In fact, the ASAP schedule finds the optimum execution time in an unconstrained problem [67]. At each iteration of the inner loop we calculate the optimal schedule, in terms of area occupied by an interconnection architecture able to run that communication schedule, under that temporal bound (fourth, fifth, and sixth steps). The temporal bound is relaxed until a solution, satisfying the area constraints, is found. A key decision here is how much the temporal bound should be relaxed. These steps rely on an algorithm to find the synthesizable architecture with the minimum degree of parallelism allowing a certain schedule to be run (fifth step). Then, the cost of the whole architecture is quantified by using a suitable area model. The last step checks if the area of the architecture found meets the given constraints. If this is not the case, we go back to the third step where, in addition to relaxing the temporal bound, the mobility values are recomputed accordingly. In the following, several essential aspects related to the above flow are described in more detail.

4.4.4.1 Relaxing the temporal bound

The granularity of the temporal bound, relaxed at each repetition of the third step above, is critical to guaranteeing a comprehensive exploration of the available design choices. Of course, relaxing the bound by a single clock cycle at each step would be infeasible. We chose to relax the bound by the minimum time allowing a task to eliminate at least one overlap in the schedule. The intuition behind this

choice is that less overlapping leads to less concurrency, and hence the architecture will take less area. As an example, in figure 4.6 the temporal bound is relaxed by two time units. Had the temporal bound been shifted by only one time unit, task t_1 would have still overlapped with task t_2.

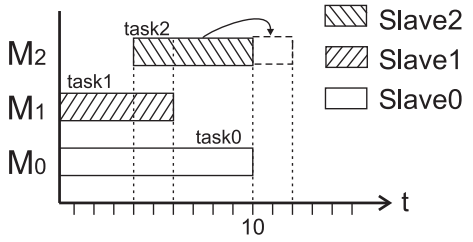


FIGURE 4.6: Example of temporal bound relaxing

4.4.4.2 Scheduling algorithms

The scheduling algorithm is responsible for determining a schedule compatible with the lowest-area synthesizable topology satisfying a given temporal bound and all the constraints expressed by the modified DG (including slaves and bridges conflicts). Table 4.1 summarizes the characteristics of the different scheduling algorithms evaluated.

TABLE 4.1: Scheduling Algorithms

Name	Abbreviation	Solution	Type
Genetic Algorithm	GA	Approx	Evolutionary
Priority-based List Scheduling PBLS	PBLS	Approx	Approx
Randomized Priority-based List Scheduling	R-PBLS	Approx	Approx
Random search	Rand	Approx	Random
Exhaustive Exploration	EX	Exact	Exhaustive
Smart-Exhaustive Exploration	Smart-EX	Exact	Exhaustive

4.4.4.3 Genetic algorithm (GA)

The scheduling problem can be solved by general iterative methods, like genetic algorithms [68] that starts from a random initial population and then mutate and

combine individuals in an iterative fashion. For implementing a scheduling algorithm relying on a genetic approach, we used the Opt4J Java-based modular framework [69]. In order to obtain only acceptable solutions at each step, we created a new *genotype* to encode the solutions. The scheduling genotype is a class containing a vector corresponding to the scheduling vector itself, indexed by the task identifier and containing in each entry the starting clock cycle of the corresponding task. The *phenotype* is simply the scheduling vector contained in the corresponding genotype. The evaluator of the phenotype is described in the subsequent paragraph and is the same for all other algorithms. The *fitness function* returns the minimum area required by a communication architecture exhibiting enough parallelism to run the identified scheduling. Due to the custom implementation of a personalized genotype, phenotype, and evaluator, a custom optimizator operator was also implemented to drive the generation of the new population. The initial population is generated starting with the ASAP solution by moving a random number of tasks by a random quantity not larger than the mobility. This guarantees a good diversity at the chromosome level with a relatively small population size. During the movements, all data and structural dependency relationships must be preserved. When generating the new population, we do not perform crossover but we only rely on mutation. Specifically, we substitute the worst L elements with L new individuals mutating the remaining ones with probability p. Mutation takes place in the same way as random generation but starting from the parent's schedule and not from the ASAP schedule. After the last iteration, the best individual is chosen. L, p, the initial population size, and the number of iterations are configurable.

4.4.4.4 Priority-based list scheduling

Our implementation of the priority-based list approach (PBLS) tries, at each step, to make the best move as possible within a list of admitted moves, i.e. the moves satisfying the data-dependency constraints. The list is ordered according to the area cost incurred by the topologies associated with any potential move. The procedure to derive a topology, given a schedule, is explained in section 4.4.4.7. Its behavior is radically different than the genetic approach. It does not admit uphill moves (causing a temporary cost increase), hence the probability of sticking

at a local minimum solution tends to be high. The starting point must be a valid solution. At each step, the algorithm evaluates all neighboring solutions by analyzing what happens by moving each task by the minimum quantity to eliminate an overlap. The list is comprised of all neighboring solutions, then sorted according to the benefits achieved by taking that move. Only the best solution is chosen. When, at a certain step, there are no moves improving the solution, the algorithm takes randomly one of the possible moves leading to an equivalent cost solution and goes on. The number of consecutive random moves is bounded by a value configurable by the user. This value must be accurately tuned, keeping in mind that the solution space may be very smooth and have several equivalent neighboring solutions. When this condition persists, the identified solution is likely to be a local minimum. The search process exits when the maximum number of iterations is reached or when there are only uphill moves available. The algorithm does not perform well, in general, when the design space is irregular. Since it does not admit uphill moves, this approach is likely to yield the optimal solution only if we can take an initial scheduling that is already quite close to the optimum, otherwise it gets stuck at a local minimum (a point not having better neighboring solutions).

4.4.4.5 Randomized priority-based list scheduling

This is, essentially an optimized version of the previous algorithm that tries to avoid getting stuck at a local minimum. This algorithm tries to explore a certain number of regions of attraction according to how many iterations are performed. The algorithm is very similar to its conventional counterpart, but when it arrives to a local minimum (a point not having better neighboring solutions) it records the solution and generates another random starting point in the hope of falling in a different region of attraction. The procedure exits as soon as a maximum number of iterations (or a maximum number of local minima) is found.

4.4.4.6 Random search, exhaustive exploration, and smart-exhaustive exploration

In order to extend the comparisons, some general problem-solving techniques were implemented: a random search and two different exhaustive search approaches.

The Smart-Exhaustive exploration is an optimized version of the standard exhaustive approach. The optimization consists in relying on an adaptive granularity when moving tasks. Specifically, the quantity by which a task must be moved is automatically computed in order to eliminate at least an overlapping in the current schedule configuration. Obviously, whenever a task is moved, all tasks dependent on it must be moved as well to preserve dependency and structural constraints. Albeit this optimization largely reduces complexity, exhaustive approaches are still too complex and can only be used for small-sized problems.

4.4.4.7 Evaluation of scheduling solutions

All the evaluated algorithms rely on a procedure to evaluate a communication task schedule. In that respect, we need to find the lowest cost architecture that exhibits enough parallelism to accommodate the identified scheduling. This process is responsible for the quality of the local parallelism. Concerning the derivation of valid intra-cluster topologies, we rely on compatibility graphs [67]. Compatibility graphs are usually adopted in binding problems, to figure out whether two scheduled operations can share a hardware resource. In the methodology, compatibility graphs are used to determine the number of master and slave ports needed by local interconnects.

Definition 4.5. Given a Communication Scheduling, two tasks t_i and t_j are *compatible* if and only if they do not overlap in time.

Definition 4.6. Two masters (slaves), including bridge master (slave) ports, are *compatible* if and only if all the tasks in which they are involved are compatible and they belong to the same cluster.

From the definitions given above, the construction of compatibility graphs for communication tasks and then for masters and slaves is straightforward. First,

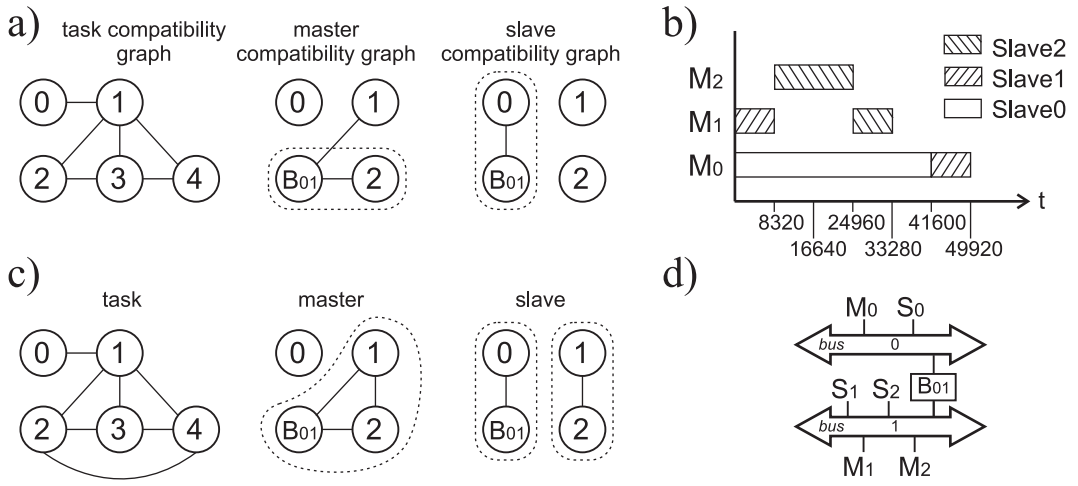


FIGURE 4.7: Deriving a topology from a given schedule. (a) Compatibility graphs for the schedule in figure 4.2. (b) An enhanced schedule, with less concurrency, for the same application. (c) Its compatibility graphs. (d) The derived topology.

a communication task compatibility graph $CG_t(V, E_t)$ is built. The vertex set $V = \{v_i : i = 0, 1, ..., n_{task} - 1\}$ is in one-to-one correspondence with the set of communication tasks, while the edge set $E_t = \{(v_i, v_j) : i, j = 0, 1, ..., n_{task} - 1\}$ denotes compatibility between the above tasks: if two tasks do not overlap, then an arc between their two vertices is placed. Then, a master $CG_m(V_m, E_m)$ and a slave $CG_s(V_s, E_s)$ compatibility graph are built. The vertex sets $V_m = \{v_i : i = 0, 1, ..., n_{master} - 1\}$ and $V_s = \{v_i : i = 0, 1, ..., n_{slave} - 1\}$ are in one-to-one correspondence with the set of masters and slaves, while the edge sets $E_{master} = \{(v_i, v_j) : i, j = 0, 1, ..., n_{master} - 1\}$ and $E_{slave} = \{(v_i, v_j) : i, j = 0, 1, ..., n_{slave} - 1\}$ denote compatibility between the masters and slaves. In order to build the master (slave) compatibility graph, we start from a completely connected graph and we delete arcs progressively as follows. For each pair of non-compatible tasks, we delete the arc between the corresponding master (slave) vertices. Starting from those compatibility graphs, for each connected component we solve a clique-partitioning problem [70] that identifies the minimum number of cliques[1]. Then, we assign a master (slave) port to each clique in its cluster. If for a cluster a single clique is identified, both in the master and the slave compatibility graphs, then a single shared bus is enough because all tasks are compatible, hence sequential. Otherwise

[1]In graph theory, a clique in an undirected graph is a subset of its vertices such that every two vertices in the subset are connected by an edge [71].

a crossbar is necessary. If there are no clique in a cluster, then a full crossbar will be used, otherwise the number of concurrent channels will be dependent on the number of cliques. Figure 4.7 shows the differences between compatibility graphs for two distinct schedules. Figure 4.7.a depicts the compatibility graphs for the schedule in figure 4.2.c. Masters M_1 and M_2 and slaves S_1 and S_2 are incompatible with each other. The derived topology, consisting of a crossbar for the cluster C_1 and a shared bus for C_0, is shown in figure 4.2.d. Notice that masters M_2 and B_{01} are compatible and, hence, they share the same crossbar port. Figure 4.7.b shows a different schedule that removes the above incompatibility (figure 4.7.c). The solution of the clique-partitioning problem is highlighted by the circles. This leads to a less parallel and less expensive architecture, shown in figure 4.7.d.

4.5 Experiments and case studies

4.5.1 Experimental setup

For the experiments, we used a prototyping FPGA board, namely a ZedBoard by Avnet Design Services for the Xilinx Zynq™-7000 [72]. The communication architecture synthesis flow uses the Xilinx AXI components compliant with the AMBA® AXI version 4 specification from ARM. The components we used for generating the system architecture include:

- Xilinx LogiCORE IP AXI Interconnect (v1.06.a)
- Custom AXI to AXI bridge
- Xilinx LogiCORE IP AXI to AXI Connector (v1.00.a)

The Xilinx AXI Interconnect may be configured as a bus or a crossbar. All data channels have the same bus width of 32 bits. Albeit the number of total crossbars, buses, and bridges provides an indication of the cost of the overall Synthesizable Topology, in order to target physical devices, such as FPGAs, we need a measure related to the technology. Synthesized FPGA designs are normally evaluated in terms of Look-Up Tables (LUTs) and Flip-Flops (FFs). Interconnect components,

in particular, are usually dominated by the number of LUTs. Hence, to explore the design space efficiently, we need a technique to estimate how many LUTs are taken by a topology (without resorting to an actual synthesis). This measure clearly depends on the technology, because the internal architectures of FPGA chips can vary considerably from one family to the other.

We built accurate analytical models for the evaluation of the area cost, the latency, and the power consumption, obtained by interpolating extensive RTL synthesis results. The area model of the AXI Interconnect was obtained by synthesizing the AXI Interconnect IP core for a subset of all the possible configurations with 1 to 16 master ports and 1 to 16 slave ports with a step of 3, a data bus width of 32 bits, and the two available address bus modes (shared bus or SAMD[2]) using the Xilinx PlanAhead Design Tool. From the data collected, we extrapolated the following equations, valid for the Zynq-7000 family [72], used to calculate the area information of any interconnects with 1 to 16 master ports or 1 to 16 slave ports (16 masters/slaves is the maximum value supported by AXI Interconnect IP core).

$$A_{crossbar}\ n \times m = 101n + 60nm + 42m + 874 \tag{4.2}$$

$$A_{bus}\ n \times m = 80n + 18.75m + 95.5 \tag{4.3}$$

We considered burst-based transactions with only the start address issued and the payload transferred in a single burst that can comprise multiple beats[3] Concerning the bandwidth estimation, we considered the bandwidth of a channel as the maximum rate at which a master can receive/send (r/w) data from/to a slave. Due to the burst-based communication, we considered that address arbitration does not impact that rate[4]. To compute the execution delay d_i of each communication task t_i , we used the following equation:

$$d_i = (\frac{52 + 10h_i}{64}) * c_i \tag{4.4}$$

[2]Shared Address buses and Multiple Data buses: in most systems, the address channel bandwidth requirement is significantly less than the data channel bandwidth requirement. Such systems can achieve a good balance between system performance and interconnect complexity by using a shared address bus with multiple data buses to enable parallel data transfer [6]

[3]A beat is an individual data transfer within an AXI burst [6].

[4]Arbitration latencies typically do not impact data throughput when transactions average at least three data beats [73]

where h_i and c_i are, respectively, the *hop count* and the *computation cost*, defined in section 4.3.1. Intra-cluster communication tasks require 52 clock cycles every 64 bytes of data sent: 4 clock cycles for the initial access latency due to the phases of arbitration and handshaking on the address channels, and 3 clock cycles for each transfer of a single beat on the data channel. In case of inter-cluster communication 10 additional clock cycles are required for each traversed bridge. Obviously, these numbers are dependent both on the architecture and the used IP cores. Concerning the static power consumption, we considered the power from transistor leakage on all connected voltage rails and the circuits required for the FPGA to operate normally. Obviously this power is independent of the user design and, consequently, is affected only by voltage and temperature. Unlike static power, dynamic power is the power of the user design, i.e. it depends on the input data pattern and the design internal activity. We calculated this power by means of simulation results based on the average switching rate of every signal in the interconnect. Concerning the software framework, including the implementation of all the above scheduling algorithms, the optimization tool is implemented in Java 7. For the genetic algorithm we rely on the modular framework Opt4J [69].

4.5.2 Overview of the experiments

In order to validate the clustering choices and the cost of the corresponding communication architecture, and to demonstrate the impact of scheduling on the resulting topology, we tested the presented method for six synthetic benchmarks as well as a real-world application. The benchmarks were obtained using the TGFF package [74], removing possible concurrent tasks with the same master, while the application is a Canny edge detection algorithm [75] whose TL and DG are taken from [76]. Their characteristics are summarized in Table 4.2. The number of tasks in the experiments ranges from 13 to 88, while the number of masters and slaves ranges, respectively, from 5 to 16 and from 5 to 20. We chose benchmarks with a various number of masters and slaves to evaluate the effectiveness and the scalability of the proposed method for larger systems. Table 4.2 also gives the number of clusters $n_{cluster}$ found after the Communication Element Clustering phase. In addition, the table contains an index representing the completeness of the DG.

It summarizes how much the design space exploration is constrained by data dependencies. The index has been derived as the ratio between the number of arcs in the DG and $(n_{task}^2 + n_{task})/2$, the maximum possible number of arcs with no cycles. The last column of the table contains the localization factor[5]. Notice that the localization factor depends only on the mapping of communication elements within local domains and hence is schedule-independent.

TABLE 4.2: Benchmarks Characteristics

	n_{task}	n_{master}	n_{slave}	$n_{cluster}$	DG completeness	Localization factor
App-I	36	9	6	3	0.0570	0.9460
Bench-I	13	5	5	2	0.2200	0.9697
Bench-II	25	7	8	3	0.1267	0.9178
Bench-III	31	8	10	4	0.1275	0.9267
Bench-IV	43	11	14	5	0.0731	0.8261
Bench-V	62	12	16	6	0.1100	0.7087
Bench-VI	88	16	20	8	0.1350	0.7935

We first carried out a set of experiments to evaluate the benefits of using the exploration techniques presented here and to analyze the area/latency trade-off. Area/latency trade-off points can be derived as the solutions to different constrained scheduling problems. The presented methodology was applied to each benchmark and application with different area constraints. The results obtained on Bench-III are discussed in subsection 4.5.3. Here we will also give some general remarks.

In a second set of experiments, a stringent area constraint, ranging between 30% and 70% of the area of a full crossbar implementation, was fixed. Then, we compared the proposed approach to the most popular design choices and to [1], used under the same area constraints, as well as to a Full Crossbar, a Hierarchical Bus, a single Shared Bus and a Network-on-Chip. Furthermore, in order to appreciate the overall impact of the scheduling algorithm on the whole methodology, we also analyzed and compared the different scheduling algorithms and the resulting solutions. In addition, we give some general remarks about power consumption.

[5]The localization factor is a metric introduced in [77], expressing the ratio between the local traffic and the total traffic

Finally, we focused on the area saving by applying the proposed methodology under different area constraints able to obtain the same latency. Notice that [1] does not perform the scheduling step but only relies on aggregated parameters (i.e. the total amount of traffic exchanged between master/slave pairs) to automate the interconnection design. These results are presented in subsection 4.5.4.

4.5.3 Exploring area/latency trade-offs

As shown by Table 4.2, the number of clusters generated depends on the complexity of the application as well as on the area constraint. For the most complex benchmark (Bench-VI), 8 clusters are derived by the procedure, while only two are required for the most simple benchmark (Bench-I). Figure 4.4 shows an example of clustering algorithm applied to Bench-III (of medium complexity) where, due to the imposed area constraints, the maximum possible value of inter-cluster Euclidean distance is 0.75, giving an outcome of 4 clusters. This result is obtained with an area constraint of 4000 LUTs. Above this value, the number of clusters starts decreasing. Furthermore, figure 4.8 shows the schedules and the corresponding topologies obtained from the same benchmark with area constraints of 2700 LUTs and 4000 LUTs using the randomized priority-based list scheduling.

Communication tasks involving the same bridges or slaves, or exhibiting dependency relationships, never overlap. Concerning unrelated tasks, if they do not overlap, their communication interactions are serialized, so that they can share the same resources and a single communication link can be synthesized. On the other hand, when two tasks do overlap, multiple resources must be synthesized. Four clusters are connected by means of bridges that allow the traffic to be exchanged according to the requirements expressed by the Task List. The location of the bridges depends on the solution of the optimum branching problem that tries to place the clusters that exchange more data as near as possible to each other (determining less hops to go through). Notice that there are two parallel communication paths between masters inside cluster C_1 and slaves inside cluster C_0: the first goes through bridge B_{10}, while the second goes through the multi-hop path consisting of bridges B_{13} and B_{30}. The difference between the two architectures

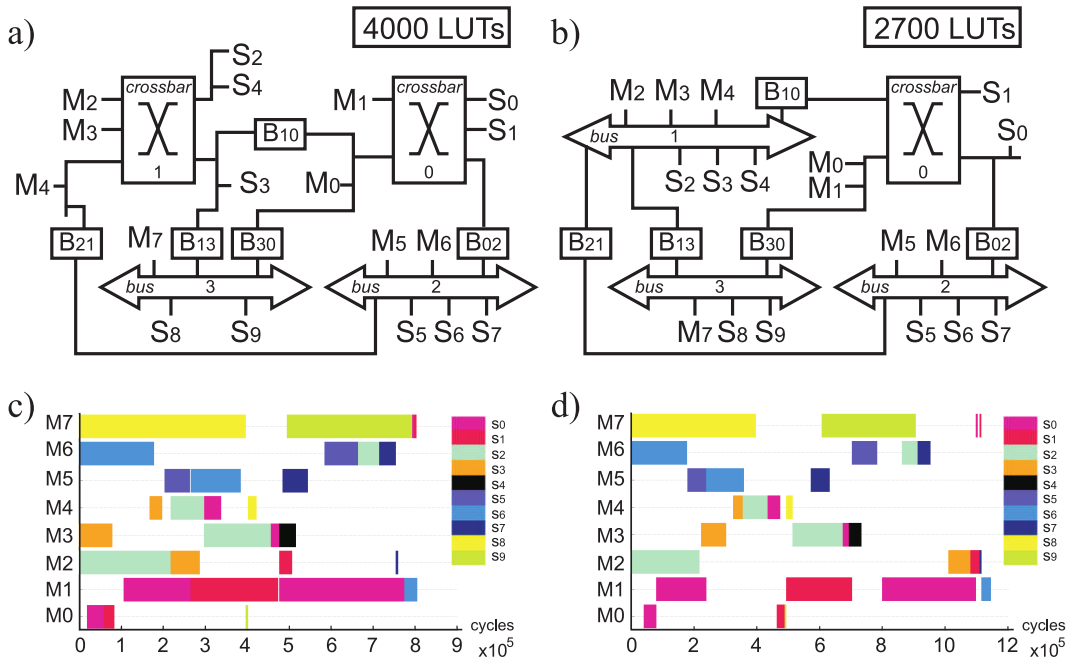


FIGURE 4.8: Synthesized topologies and their schedule found for Bench-III with the Randomized Priority-based List Scheduling and two different Area constraints. (a) The *Synthesizable Topology* (ST) obtained with an area constraint of 4000 LUTs. (b) The ST obtained with an area constraint of 2700 LUTs. (c) The *Communication Scheduling* (CS) obtained with an area constraint of 4000 LUTs. (d) The CS obtained with an area constraint of 2700 LUTs.

lies in the intra-cluster architectures. With 4000 LUTs we have a more parallel intra-cluster architecture achieving a gain of roughly 32% in execution time (around 11.75 Mega Clock-cycles against 8.10) due to the presence of the crossbar in cluster C_1. Instead, with an area constraint of 2700 LUT, all tasks involving M_2, M_3 and M_4 as well as S_2, S_3, S_4 are executed in a linear sequence, leading to a single bus executing all communication tasks in cluster C_1. It is important to highlight that, in this case, the two area constraints lead to a difference only on the intra-cluster parallelism, while the global and the inter-cluster parallelism does not change. For Bench-III we achieved a localization factor of approximately 0.92. It indicates a highly localized traffic and hence improved opportunities for global parallelism, resulting in a lower overall execution latency. Localization factors for each benchmark and application are shown in the last column of Table 4.2.

4.5.4 Comparisons with existing methods for various scheduling algorithms

To illustrate the benefits of the approach, a comparison with all the common design choices and the approach presented in [1] is presented. We evaluate the improvement of area, latency and energy profiles of seven benchmarks against a crossbar implementation, a single shared bus, a hierarchical bus composition and a basic NoC implementation. Figure 4.9 summarizes the results obtained by applying the two methodologies to all the case-studies under a fixed area constraint. Concerning the hierarchical bus, we implemented each local domain found after the clustering step with a single shared bus and we keep the inter-cluster topology unchanged. On the other hand, a NoC implementation is more customizable than a bus-based interconnect. We considered a basic NoC implementation with a 2-D mesh topology, an oblivious minimum-path routing, and a wormhole flow control. The network channel width and flit size was set to 8 bits. Each packet in the network contains 64 body flits and 1 header flit which carries the address information. The FIFO buffers at the output ports of the router have a depth of 32 flits. The router takes 4 cycles to process the header flit [78]. After the virtual channel is acquired by the header flit, the remaining flits follow the header flit in a pipelined fashion. Furthermore, the network interface overhead due to packetization/depacketization was set to 2 cycles [79]. Then, the mapping of cores to their cross-points was done at a high level of abstraction (based on the traffic between cores) exploiting the traffic locality such as to reduce the number of router in a path. Refers to [80] for more details. Concerning the presented approach, all the scheduling algorithms discussed in subsection 4.4.4.2 were used, except the too slow exhaustive searches.

For each experiment, the full crossbar implementation is the unconstrained solution and, hence, it exhibits the minimum possible latency at the price of a highly oversized area. For Bench-VII we were not able to synthesize a single Full Crossbar and a single Shared Bus due to the size of the benchmark[6]. Obviously, the approach shows different behaviors according to the scheduling algorithm used. The R-PBLS algorithm is able to obtain a latency that on average is only 11% larger

[6]The AXI IP Core can be configured to comprise maximum 16 Slave Interfaces (SI) and 16 Master Interfaces (MI) [73].

compared to the latency obtained with a full crossbar implementation, while, due to the imposed constraints, the area ranges between 30% and 70% of the area of a full crossbar implementation. The communication architectures found with the PBLS and GA show a similar trend with an average latency overhead of respectively 16% and 15% compared with the full-crossbar implementation. This happens mainly because of the adopted clustering technique. For the small fraction of inter-cluster communication, the small degradations due to the extra clock cycles taken by bridge crossing (10 cycles every 64 bytes for each bridge) are not appreciable in the figure (scheduling involves millions of clock cycles). This means that the proposed iterative method is capable of achieving roughly the same latency as a full crossbar despite stricter area constraints. Using the approach in [1] we have a performance degradation instead: our approach with R-PBLS scheduling found solutions that on the average lead to a latency reduction of about 35%. In fact, ignoring dependency relationships may result in a crossbar even when it is not strictly necessary as well as buses in cases where some communication tasks can be parallelized. In other words, it may cause the serialization of tasks on the critical path, increasing the execution time. The solution points corresponding to Shared Buses, as expected, are associated with the minimum area and the largest latency (on the average 6.5× less area than a Full Crossbar with a latency degradation of about 2.6×) followed by the Hierarchical buses (on the average 4.4× less area than a Full Crossbar with a latency overhead of about 36%). The NoC implementation exhibits a different behavior. In order to keep the router cost down, the channel width was set to only 8 bits. This leads to a performance degradation. In addition we considered a 4 cycle overhead to process the header flit and a 2 cycle overhead to handle the Packetization/Depacketization. As a consequence, the NoC solutions show on the average a latency that is about 14% bigger than this approach solutions. Furthermore, there is a considerable gap between the areas of the two approaches: the NoC implementations occupy about twice the area of the proposed approach implementations for a quarter of the channel width (32 bits against 8 bits). This is a well-known drawback of soft overlay NoCs compared to the hard implementations [81]. As an example, a 4×4 Hermes [82] NoC implementation with 1, 2 and 4 virtual channels occupies respectively 11511, 22036 and 50962 LUTs on a Xilinx XC2V6000 FPGA device [83].

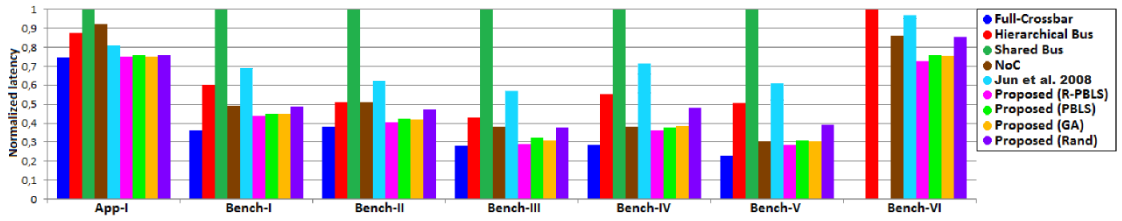


FIGURE 4.9: Latency comparison. The proposed approach and [1] are used under the same area constraints

To better explain these results, we provide Table 4.3. Element (i, j) in the table represents the average percentage of time in which the communication links are used by application i using the approach j. A low value means wasted area due to low channel usage. A value equal to 1 means a communication architecture always working as in a single shared bus. Notice that the approach in [1] (as well as the use of a single crossbar architecture) may lead to an underutilized network because it does not consider dependency relationships, which of course are very likely to be found in any real application.

TABLE 4.3: Utilization of communication channels

	Full Crossbar	Hier. Bus	Shar. Bus	[Jun 2008]	Prop. (R-PBLS)	Prop. (PBLS)	Prop. (GA)	Prop. (Rand)
App-I	0.207	0.389	1	0.273	0.341	0.341	0.336	0.336
Bench-I	0.550	0.865	1	0.577	0.787	0.774	0.771	0.715
Bench-II	0.438	0.728	1	0.448	0.554	0.524	0.532	0.475
Bench-III	0.505	0.645	1	0.476	0.549	0.492	0.511	0.421
Bench-IV	0.391	0.450	1	0.321	0.487	0.469	0.458	0.368
Bench-V	0.401	0.492	1	0.307	0.478	0.439	0.488	0.347
Bench-VI	0.450	0.595	1	0.412	0.521	0.506	0.506	0.444

In addition, Table 4.4 shows the runtime of the scheduling algorithms to reach the above solutions. Obviously, the results show a dependence on the application size in terms of number of tasks. The Smart-Ex is computationally infeasible while the PBLS and hence the R-PBLS scale poorly due to their inherent complexity. On the other hand, the GA scales more smoothly as the number of tasks increases.

We also evaluated the overall energy consumption for each case-study, taking into account the power consumption (based on the Xilinx XPower power estimation tool) and the overall execution time incurred by the communication tasks. Static

TABLE 4.4: The runtime (ms) of the proposed scheduling algorithms

	Proposed (R-PBLS)	Proposed (PBLS)	Proposed (GA)	Proposed (Rand)	Proposed (Smart-Ex)
App-I	8550	250	2190	520	
Bench-I	1500	80	1610	310	523200
Bench-II	8770	240	2020	450	
Bench-III	22750	300	2110	680	
Bench-IV	87149	720	2430	1030	
Bench-V	300401	2000	3450	2060	
Bench-VI	685030	2550	3760	2300	

energy consumption, as expected, is directly proportional to the schedule latency since the results provided by XPower in terms of static power refer to the whole chip and, hence, are design-independent. The dynamic energy consumption, on the other hand, does depend on the specific structure of the implemented design. Figure 4.10 shows the main results in terms of dynamic energy, referring to a full crossbar, a hierarchical bus and a single bus implementations, as well as a previous literature solution [1], and the presented approach. The proposed method generates interconnect architectures consuming on average 28% less dynamic energy than full crossbar implementations and 40% less dynamic energy than [1]. Compared with a hierarchical bus and a single shared bus there is an higher energy consumption of respectively 15% and 77%. To understand these results, recall that dynamic power can be modeled as $P_{dynamic} = {V_{DD}}^2 \cdot \sum_{n \in nets} (C_n \times f_n)$ where V_{DD} is the supply voltage, while C_n and f_n are, respectively, the capacitance and the average toggle rate (switching activity) of a net n. The number of nets, in the case of crossbar solutions, grows quadratically with the number of ports. As a consequence, a large crossbar tends to consume more dynamic power per port than an interconnect made up of small crossbars. This explains the advantage of the proposed approach over full crossbar implementations in terms of energy consumption. The improvement over [1], on the other hand, is due to lower latencies achieved by the communication scheduling step, as shown in figure 4.9. In one specific case (Bench-V) [1] is worse than the full crossbar implementation, whereas our approach still performs better because of reduced-size components and improved latency.

Finally, we explored the design space by varying the area constraints in the two

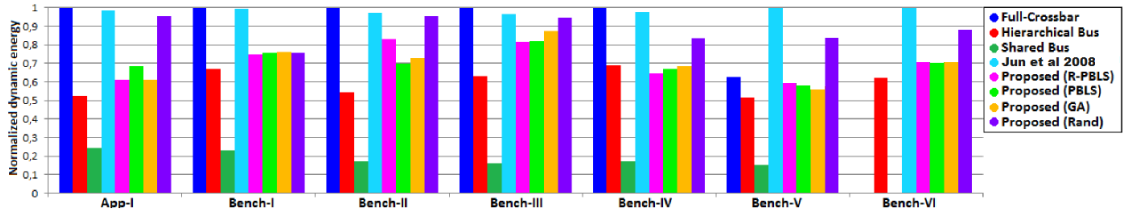


FIGURE 4.10: Dynamic energy consumption comparison.

methodologies until two architectures yielding the same latency were found. As shown in Figure 4.11, our methodology obtains an average area reduction of 48% compared to [1] under the same latency.

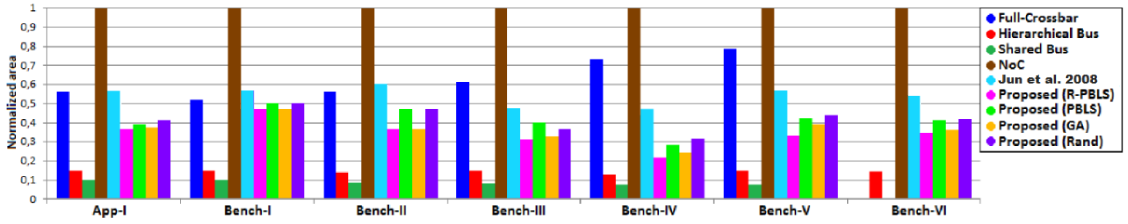


FIGURE 4.11: Area comparison of interconnects yielding the the same latency

4.6 Related work

Although dependency relationships between communication tasks are very important as they limit the communication parallelism actually available, potentially making high-bandwidth interconnects underutilized, they have only partially been considered in the context of automated interconnect synthesis. [84] describes a methodology for exploring the interconnect design space based on the overall amount of data traffic exchanged between communicating elements. They rely on a combination of analytical methods and trace-driven simulation. The authors only target an interconnect architecture based on bridged shared buses and do not cover the use of crossbars for local domains, which is a common approach today for overcoming the bandwidth limitations of shared buses. [85] proposes a framework for mapping application requirements to a given communication architecture template and optimizing design choices for the interconnect. The methodology takes an architectural template as input and performs a mapping of all computation

units on the given architecture. It does not consider inter-communication dependencies.

In order to overcome the scalability issues of single-crossbar solutions, several approaches based on cascaded crossbars have been proposed [1, 86–89], leading to topologies similar to those generated by our methodology, although they only target ASIC design flows.

An important trend emerged during the last decade is represented by Networks on Chip (NoCs) [90, 91]. While NoCs can potentially provide improved scalability for the interconnect architecture by separating transaction, transport, and physical layers, they also introduce considerable complexity and, if not designed carefully, they can even lead to performance degradation. Among the disadvantages posed by NoCs are the higher power consumption and area requirements compared to standard interconnect solutions and the lack of predictability of the performance metrics [92]. Furthermore, although research in NoCs is no more in its infancy, there are no well-established implementation solutions based on the NoC approach [93]. Since NoCs are highly customizable, there is a need of CAD tools supporting the automation of the design choices. Several such design flows have been proposed by the research community. For instance, both [94] and [95] present a method to synthesize power/performance-efficient NoC topologies. In particular, the authors in [94] present a method to concurrently statically schedule both communication transactions and computation tasks, mapping the IPs in a certain topology and allocating the routing paths. Unlike the previous work, [95] proposes a topology generation procedure using a general purpose min-cut partitioner to cluster highly communicating cores on the same router and a path allocation algorithm to connect the clusters together. Similarly, [96] addresses the unified mapping/routing problem with the objective of mapping QoS-constrained applications onto a NoC topology, statically routing the communication and allocating the TDMA time-slots on the network channels. [97] presents a method to partition the system into groups of cores using spectral clustering, a partitioning algorithm based on eigenvalue decomposition. In order to make the inter-cluster communication feasible, links between the individual routers are created using delay-constrained minimum spanning trees. The methodology presented here has some similarities with [94–97] in that it uses a custom algorithm to cluster highly

communicating cores on the same local domain and relies on a concurrent communication scheduling/topology synthesis approach. However, unlike other works, this approach targets heterogeneous interconnects made of buses and crossbars instead of NoCs. As a result, while giving as output clusters of highly interacting cores, the flow maps clusters to buses or crossbars and determines bridge allocation between cluster pairs, instead of mapping clusters to routers and allocating links between cluster pairs. For a survey on recent research contributions and problems on NoCs we refer the reader to [93].

Because of the popularity it has been gaining during the last years, crossbar-based interconnect design is addressed by several works. [98] and [99] focus on the synthesis of a single optimized crossbar. While this approach could be useful for small- or medium-sized designs, as the system size grows, a single bus matrix interconnecting a large number of components may have a prohibitive cost and incur high latencies. In order to overcome the scalability issues of single-crossbar solutions, several approaches based on cascaded crossbars have been proposed [87–89]. Constraining the interconnect architecture to using only crossbars, of course, results in higher area costs, especially for FPGA-based implementations. In many situations, in fact, there are parts of the interconnect that can be implemented as shared buses without compromising the overall performance [100].[101] and [102] address this observation by supporting the generation of heterogeneous interconnects made of crossbars and shared buses.

None of the previous works, however, considers the introduction of dependency constraints in the optimization problem in order to avoid oversized interconnection resources, i.e. wasted area on the chip. [103] and [104] address the concurrent problem of scheduling and interconnection synthesis. They both express the cost of a point in the design space as the total number of links, which however may not correctly represent the hardware cost of a heterogeneous communication infrastructure, particularly for crossbars. Furthermore, they do not consider shared buses because they are only oriented to point-to-point links. While simplifying the problem, this assumption lacks generality.

Many of the above approaches define local interconnection domains, in most cases represented by a single crossbar. The separation in local domains is only driven by

timing closure objectives (e.g., resorting to pipelining) and, unlike the methodology proposed here, it does not address directly the possibility of exploiting communication parallelism across domains. Additionally, all the cited proposals assume only one physical path between any two communicating elements, directly determined by the interconnect topology. This assumption, which limits the degree of communication parallelism that can be virtually exploited, is not inherently required when targeting an interconnect architecture based on bridged crossbars/buses. Between any two communicating domains, and even two single communicating elements, in fact, there may exist in principle different paths through different physical bridges, where each path corresponds to a different region of the addressing space. This opportunity, however, has never been considered so far in the technical literature for heterogeneous networks made of both shared buses and crossbars.

4.7 Conclusions

After chapter 2 and chapter 3 focused on memory architecture synthesis, this chapter tackles the other frequent bottleneck of many-core systems, that is the communication subsystem. An automated design methodology for the synthesis of complex on-chip interconnects has been presented. The motivation behind is that, once the application memory has been partitioned, it is necessary to identify the best interconnect topology in order to make the communication between processing elements and memory banks, id est interconnect slaves, as efficient as possible. The approach is based on a heterogeneous topology made of crossbars, buses, and bridges. The methodology can generate a synthesizable interconnection network starting from the specification of the application requirements, including dependency relationships. The approach is based on heuristic iterative optimization algorithms. The algorithms aim to get low-cost concurrent communication architectures by maximizing both intra- and inter-cluster communication parallelism as well as global parallelism, respectively by concentrating traffic within different local domains and by creating parallel, multi-hop inter-cluster communication paths. In particular, we have introduced a greedy algorithm for clustering processing elements, capable of exploiting traffic spatial locality and creating several parallel paths among clusters. In addition, we have defined a novel approach to combined

communication scheduling and interconnect generation. Several scheduling algorithms, required for the definition of intra-cluster topologies, have been analyzed and compared. We have also introduced a framework for the experimental evaluation of architectural solutions in terms of area/latency trade-offs. In conclusion, we have shown the experimental setup and some case-studies demonstrating the effectiveness of the approach as well as some comparisons against common design choices and current approaches present in the related literature.

Chapter 5

Putting it all together: OpenMP-based System level design

5.1 Introduction

Previous chapters have dealt with two of the major aspects in the automated synthesis of electronic systems, the memory subsystem and the interconnection infrastructure. They are the main responsible for systems scalability and, as such, providing techniques for improving their synthesis is of paramount importance, especially if targeting systems containing a high-degree of parallelism. In this chapter we adopt a system-level perspective and propose a design flow taking as input a functional description of a parallel system and giving as output a digital circuit. Beyond serving for putting into action the results presented in chapters 2, 3 and 4, it also introduces a design space exploration stage for solving the hardware/software partitioning problem. Moreover, several architectural optimization are introduced in order to support the OpenMP directives and clauses. In that respect, the support of dynamic loop scheduling is an important innovation because it enables run-time load balancing that is key to heterogeneous systems.

5.2 General overview

The mainstream of the electronic design automation research is to enable the adoption of high-level formalisms and programming languages borrowed from the software domain to describe complex embedded applications. In this chapter we present an approach to electronic system-level (ESL) design allowing the application semantics to be described by a familiar multi-threaded application model, namely using the popular OpenMP C extensions [105]. The approach also provides a complete support for all the subsequent development phases by proposing a comprehensive design flow which includes a design space exploration at its core. OpenMP is a de-facto standard for parallel programming and is fundamentally based on a set of compiler directives and library routines for describing shared memory parallelism in C/C++ programs. Ideally suited for medium granularity, loop-level parallelism, it was chosen as a design-entry formalism in the proposed design flow since it naturally meets the characteristics of current multi-processors systems-on-chip (MPSoCs) and provides enough semantics to express parallelism explicitly, especially for data-intensive applications.

The methodology proposed here aims to create a direct path from OpenMP software applications to automatically synthesized, heterogeneous hardware/software systems implemented onto a FPGA device. It encompasses the techniques presented in the previous chapters yielding highly-scalable systems.

A first challenge we addressed in realizing this scenario, is the need for an intermediate, system-oriented Model of Computation (MoC) suitable for describing applications specified through a parallel programming language. This is key to enable an automated design space exploration process. To this aim, we introduce a new MoC, called Shared Memory Process Networks (SMPNs), that can describe parallel applications at a high-level of abstraction, capturing the essential aspects involved in the optimization and translation process. A second aspect we dealt with, relates to functional simulation. We developed an extension of a state-of-the-art simulation engine, PtolemyII [106], to enable the support of the MoC. As a third argument, we introduced an analytical optimization model to enable a comprehensive design space exploration (DSE). Determined by the underlying

SMPN application abstraction, the optimization model is based on an Integer Linear Programming (ILP) formulation and relies on an innovative approach to the early estimation of cost functions for the population of the model. The outcome of the DSE stage is a hardware/software partitioning solution determining which tasks must be executed on processors and which ones on dedicated hardware accelerators. Finally, all the architectural techniques and optimizations used to enforce the OpenMP constructs are explained.

The above contributions result in an overall design flow and environment extending the spectrum of ESL design to high-level multi-threaded applications. Unlike many existing approaches, the support and the full compliance with the OpenMP standard enables the reuse of a large body of OpenMP code and kernels developed by the parallel computing community.

5.3 Optimized OpenMP-to-MPSoC translation

5.3.1 Design flow

The essential aim is to support the translation from a multi-threaded high-level application to an automatically optimized circuit implemented onto FPGA devices. Figure 5.1 shows the fundamental steps of the overall design flow. The first step covers the definition of the application in the C language with OpenMP extensions. Additionally, by means of specialized compiler directives, the user must specify which arrays and variables must be held on-chip and which ones off-chip. This information is propagated directly to the memory partitioning module because only the on-chip structures can be partitioned due to the presence of multiple memory banks on the programmable logic. The OpenMP multithreaded application is internally translated to a newly introduced model, called Shared Memory Process Networks (SMPNs), enabling the formal description of the application structure and the interactions between components. This task is accomplished by SOPHIE (see section 5.4) that, in addition, generates all the modules needed for the following steps, particularly for simulation, software code compilation, and high-level synthesis of hardware components. SOPHIE also provides the memory

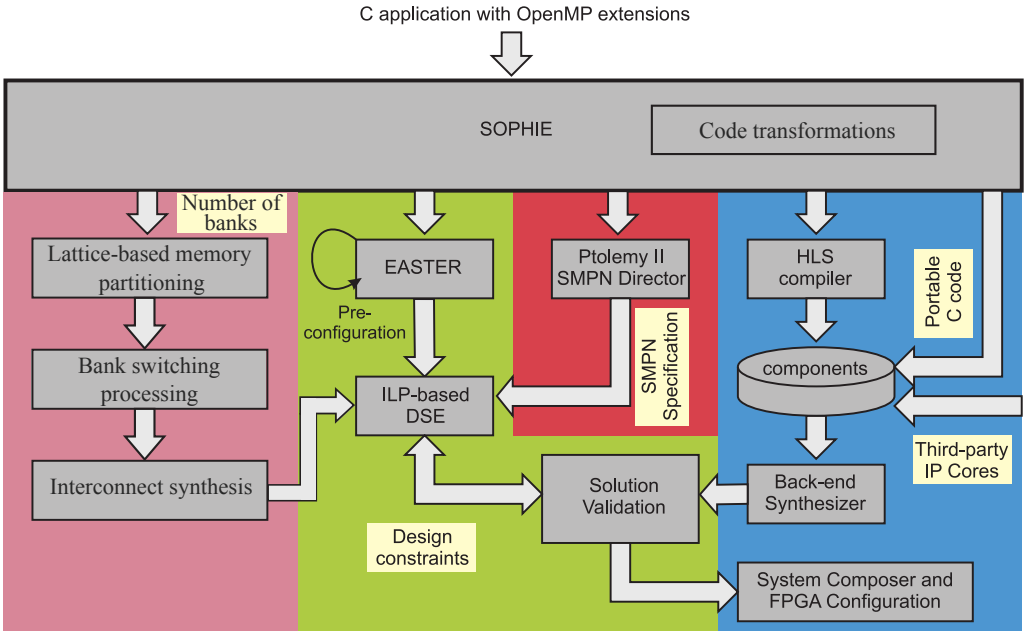


FIGURE 5.1: The overall design flow. The pink area refers to memory and communication infrastructure synthesis, the green area refers to DSE, the red area refers to simulation, and the blue area refers to hardware synthesis

partitioning module with the information about the arrays to be partitioned and the interconnect synthesis stage with synthetic traffic information needed as explained in chapter 4. The traffic information depends on how the memory has been partitioned because the slaves of the interconnection subsystem will be mainly the memory banks, therefore SOPHIE requires a feedback from the memory partitioning module in order to instruct the interconnect synthesizer.

The semantics of the translated application can be conveniently verified in a pre-synthesis stage by means of a suitable simulation infrastructure, relying on PtolemyII [107]. In particular, the methodology includes a techniques, described in section 5.5, to seamlessly plug C/OpenMP components into the Java-based simulation framework provided by Ptolemy.

Concurrently to the functional simulation but before the DSE stage, the on-chip memory partitioning and the interconnect infrastructure are automatically derived according to the procedures presented in the previous chapters. The inputs to the memory partitioning stage, apart from the original code deprived of the OpenMP directives, are the number of banks, the size of the shared variables that must be retained on chip (the information is extracted by SOPHIE using the DST, see

section 5.4.1) and the maximum bank size that is a device property. After the on-chip memory for shared variables has been partitioned lattice-wise, the partitioning solution is processed to avoid bank switching as much as possible. Clearly, the unrolling factors for bank switching avoidance cannot be bigger than the number of iterations assigned to a single task and this functionality is not supported together with dynamic scheduling (see section 5.4.3), because we cannot know beforehand which iterations will be executed by which processing unit. Notice that the interconnect synthesis step needs as input only the number of slaves, i.e. memory banks, the number of masters, i.e. processing elements, and the traffic characterization; therefore it is independent of the specific hardware/software partitioning solution, that's why a feedback from the DSE stage is not necessary.

On the other hand, the area information concerning the interconnect necessary to implement the specific data partitioning solution must be taken into account by the ILP-based DSE. Running the on-chip memory and interconnect design as separate phases means prioritizing them over the other stages in terms of area occupation. In other words, first we define the memory and interconnect architecture, and then make the remaining design choices. The motivation behind is that memory and communication are often the bottleneck for performance. However, in order to avoid that all the available area is consumed by the interconnect, the user can specify a percentage of area that must be reserved to the processing elements.

After the functional simulation, and the memory/interconnect design steps, a design space exploration process takes place relying on a mathematical ILP model directly derived from the application SMPN model, described in section 5.3.3. To allow a fast DSE, the ILP optimization model is populated with quantitative parameters derived by means of an early estimation approach based on a linear regression statistical technique, realized by an ad-hoc module called EASTER, described in section 5.6. The module allows an approximate evaluation of the latency and the hardware cost corresponding to a given hardware/software partitioning choice without resorting to the time-consuming hardware synthesis step. The ILP model takes as input the design constraints as well as the initially available resources and the ones used by interconnect that cannot be used anymore.

Following the functional simulation and the DSE process, the back-end synthesis of all hardware components and the code compilation for on-chip processors take place.

As soon as these steps are complete, the cost functions can be accurately measured by separately executing software code on processors and reading the post-synthesis results for hardware cores on the target technology. The actual values of the cost functions are used to validate the solution previously identified by the DSE step based on early estimation.

Lastly, a system composition step is executed, where the overall physical system is built, usually by assembling library hardware/software components and application-specific components generated by the previous steps along with third-party components.

5.3.2 A model of computation for multi-threaded applications

The choice of a proper Model of Computation (MoC), i.e. a formal abstract description of the application independent of the actual implementation, is vital to enable effective design space exploration and, subsequently, synthesis and verification. A large range of Models of Computation have been proposed so far to describe embedded systems. For example, the discrete-event (DE) domain supports time-oriented models of such systems as queueing systems, communication networks, and digital hardware. In this domain, actors communicate by sending events, made of a data value (a token) and a tag. The tag, in turn, contains a timestamp and microstep, used to sort simultaneous events, i.e., events having the same timestamp [107]. When simultaneous events occur, different techniques can be applied, e.g. VHDL uses a delta-delay strategy which does not prevent a zero behavior [107, 108], while Ptolemy schedules events according to model topology [107]. Kahn Process Networks [109] (KPNs) are particularly suitable to describe a fully concurrent data-flow computation. A KPN is made of a number of processes communicating through infinite FIFOs via blocking reads and non blocking writes. The control aspects are not explicitly specified by the model.

Given the hypothesis of process monotonicity, KPNs can be proven to be determinate and self-scheduled [109]. Synchronous Data Flow Graphs [110] (SDFGs) are a restriction of KPNs where each process reads and writes a fixed number of tokens on input and output channels for each iteration. The scheduling of the system is determined so that it does not block and does not cause unbounded tokens to be accumulate on channels. Several attempts have been made to propose heterogeneous MoCs in order to appropriately model both data-flow and control aspects. In fact, some MoCs, like the familiar Finite-State Machines (FSMs), are suitable to representing control-dominated algorithmic aspects unlike models such as KPNs, that only express the flow of data. As an example, Metropolis [111] allows the use of various MoCs for modeling the system at the highest level of abstraction including both data-flow and control behaviors.

In this design flow, we propose a new Model of Computation, called Shared Memory Process Networks (SMPNs). The model is particularly focused on expressing the interactions between the actors in order to formalize a system-level description deriving from a high-level parallel application written in C/OpenMP. We addressed the shared memory computing paradigm since it is essential in many different contexts where modeling communication by means of channels turns out to be too restrictive, e.g. when the parallel processing of a large quantity of data inherently requires a global vision of them. As an example of two contrasting situations having different modeling requirements, sharpening filtering only requires a local vision of data around the current pixel, while a geometric transformation of an image requires a global vision of the data, possibly made of millions of pixels. Consequently, a sharpening filter application might be well-suited for a data-flow architecture with distributed memory, a rotation transformation might not. An SMPN is made of a network of processes that evolve concurrently and communicate with shared memory synchronizing with dedicated channels. This leads to a mixed local/global approach for communication. Operations on shared memory are asynchronous, while communication with channels happens in the same way as Kahn Process Networks with blocking reads and non-blocking writes. Furthermore, as in KPNs, testing a channel is not allowed. One of the main problems of KPN synthesis is FIFO-dimensioning. A KPN is said to be strictly b-bounded if all FIFOs are strictly b-bounded. In general, determining the boundedness of a

KPN is an undecidable problem. SMPNs address this problem by simply imposing a unitary length of FIFOs and limiting the use of channels to synchronization purposes. As shown below, this provides enough expressiveness for describing OpenMP applications.

Of course, enlarging the scope of KPNs, some formal properties guaranteed by KPNs are dropped by SMPN, but this is not necessarily a limitation since such properties are not inherently required by the targeted class of applications. In particular, one major property of KPN, determinism, is lost in the shared memory model. Luckily, the semantic of an OpenMP application does not require this property. An SMPN, therefore, is deterministic if and only if the underlying OpenMP application is such that the scheduling of the threads is deterministically known a-priori.

Another important aspect is that the SMPNs model the shared memory as a monolithic abstracted block. Hence, the model does not interfere with the memory partitioning stage; it's completely independent of it.

Formally an SMPN is a quintuple SMPN $= (P, S, D, M, A)$, where

- P is a set of concurrent processes, the actors of the process networks, implementing a distributed form of control. Denote its cardinality as $|P|$.
- S and D are the set of *Start* and *Done* channels, respectively. These are unidirectional channels of a unitary size, needed for synchronization among processes.
- M is a set of shared memory locations. The shared memory size is always bounded and can be inferred by analyzing the shared clauses or by profiling the executing of the OpenMP program during the functional simulation. Denote it as $|M|$ and assume locations be of one byte.
- A is a set of shared addresses implementing a read-only-after-write access semantic. These are used to implement mutual exclusion primitives necessary for mapping OpenMP applications. Denote its cardinality as $|A|$ and assume locations be of one byte.

In order to make the SMPN a synthesizable model that can be translated into a complete on chip system, a few hypotheses are necessary:

- Each process puts a token to the *Done* channel in an unpredictable but bounded time starting from the consumption of a token on the *Start* channel.
- *D* and *S* channels are lossless. This assumption is fundamental as token loss on *Start* or *Done* channels may unpredictably affect the evolution of the whole system. Furthermore, token delivery must be ordered.

Following is an example of SMPN. Blue arrows are *S* channels, while red ones

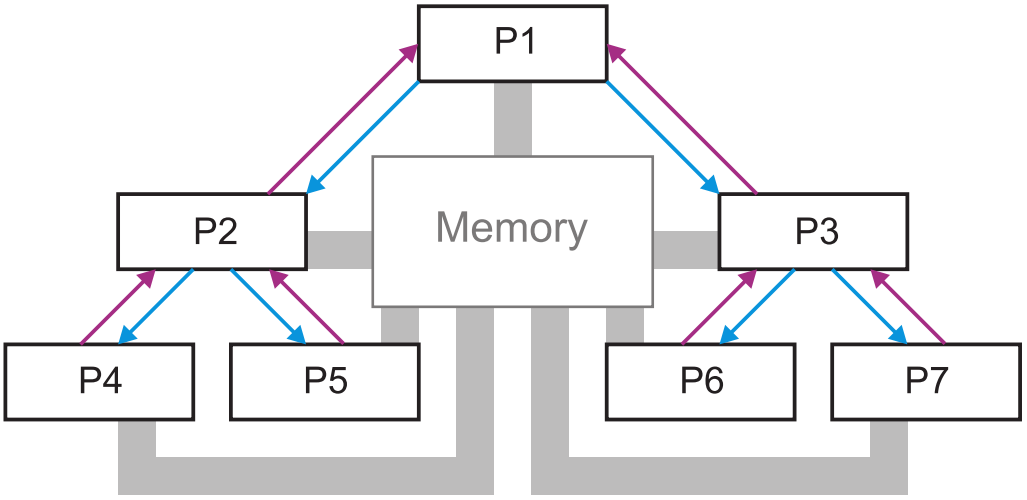


FIGURE 5.2: An example of SMPN

are *D* channels. Communication with shared memory is totally asynchronous and synchronization between processes can only take place by blocking reads on *S* channels. Writes are not blocking in the sense that a process can continue its execution after write. The SMPN model includes the following operations:

- `SendStart()`: non-blocking put of a token to a *Start* channel
- `ReceiveStart()`: blocking get of a token from a *Start* channel
- `SendDone()`: non-blocking put of a token to a *Done* channel
- `ReceiveDone()`: blocking get of a token from a *Done* channel

- `Write (Offset)`: writes to shared memory M at the specified address
- `Read (Offset)`: reads from shared memory M at the specified address.

Obviously, the above operations are not part of the OpenMP specification and the OpenMP program is completely independent from the SMPN MoC. Nevertheless, modeling an OpenMP application with an SMPN is very straightforward. As an example, consider the following simple OpenMP program.

```
int main(int argc, char* argv[]) {
   int tid;
   int a = 5;
   int c;
   int d[3];
   omp_set_num_threads(3);
   #pragma omp parallel private(tid,c)
   {
      tid = omp_get_thread_num();
      c = a+tid;
      d[tid] =c;
   }
   printf("%d, %d, %d\n",d[0],d[1],d[2]);
   return 0;
}
```

The corresponding SMPN is given in figure 5.3. P1 is the master thread. The first

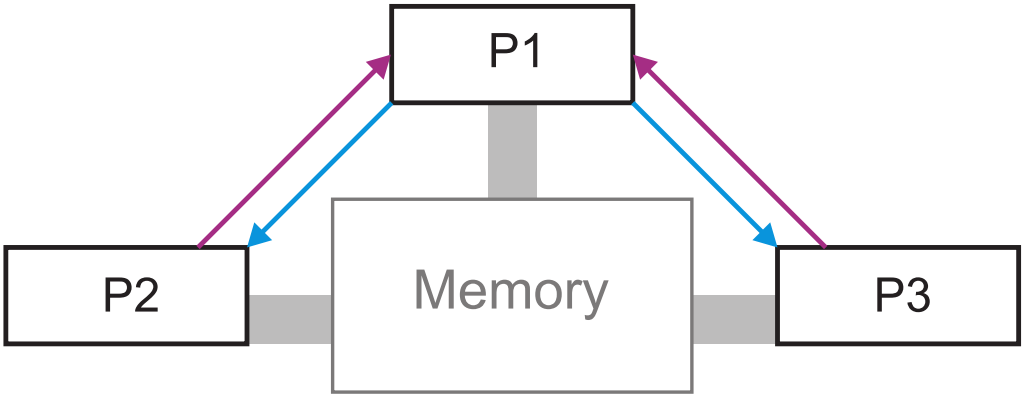


FIGURE 5.3: SMPN derived from the simple OpenMP code snippet in the text

operation of P2 and P3 is a blocking *ReceiveStart()*. After executing all operations

before `#pragma omp parallel`, P1 makes a `WriteStart()`. Then, P1, P2, and P3 execute all instructions specified in the parallel body. The `d` array is located in the shared memory without any mutual exclusion mechanism, compliant with the OpenMP specification. At the end of the `#pragma omp parallel` directive, there is an implicit barrier causing each process to synchronize before proceeding. This barrier is implemented with D channels. In particular, P1 executes a `ReceiveDone()` while P2 and P3 execute a `WriteDone()`. The possibility of associating a value to the exchanged tokens is exploited for transmitting the thread Id (`tid` in the code) on *Start* channels and a completion code on *Done* channels. section 5.7 will show how the other OpenMP directives and clauses are translated to the elements of an SMPN.

Furthermore, in order to allow the formalization of OpenMP applications with multiple consecutive `#pragma omp parallel` constructs, which is often the case in many applications including the case-study presented later in section 5.8.1, several SMPNs can be composed. Consider for example the following code.

```
int main(int argc, char* argv[]) {
   int tid;
   int a=5;
   int c;
   int d[3];
   omp_set_num_threads(3);

   #pragma omp parallel private(tid,c)
   {
      tid= omp_get_thread_num();
      c= a+tid;
      d[tid]=c;
   }
   #pragma omp parallel private(tid,c)
   {
      tid= omp_get_thread_num();
      c= a+tid;
      d[tid]=c;
   }
   printf("%d, %d, %d\n",d[0],d[1],d[2]);
   return 0;
```

```
}
```

The code contains two consecutive parallel constructs performing the same actions. The resulting SMPN looks like the graph in figure 5.4.

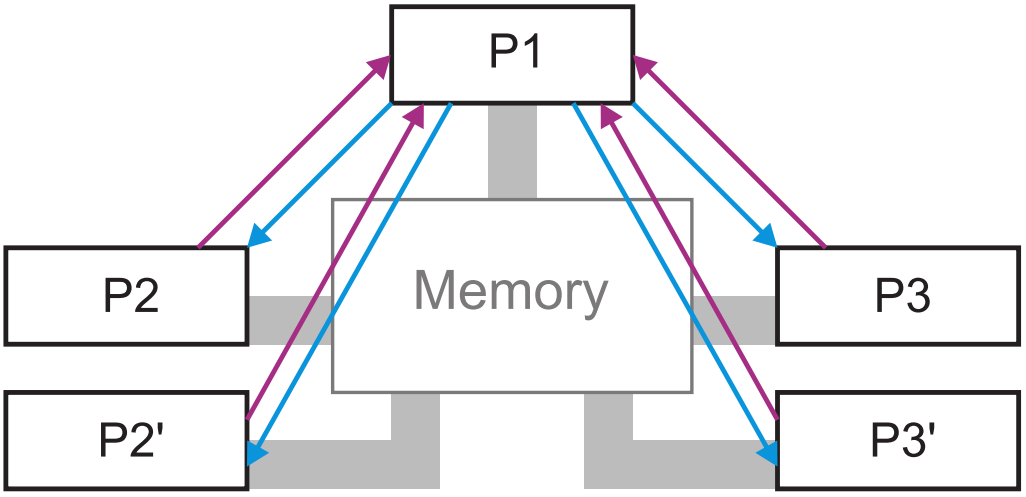


FIGURE 5.4: SMPN derived from the OpenMP code with two consecutive `parallel` constructs

The evolution of the above SMPN is determined by the process P1 sending a *Start* signal to P2' and P3' after receiving all *Done* signals from P2 and P3.

5.3.3 ILP model for automated partitioning and mapping

The SMPN directly defines the structure of design space for the OpenMP-to-MPSoC translation process. The optimization, described in this section, relies on an Integer Linear Programming (ILP) model determining the design choices in terms of partitioning and mapping of the processing components onto a hardware/software heterogeneous architecture, that will be subsequently implemented on an FPGA device through software compilation and high-level synthesis.

Following are the main definitions and elements involved in the ILP model:

- PH_i: a hardware implementation of process P_i (an element of the P set) as a hardware accelerator;

- PS_{in}: an implementation of P_i on the nth on-chip processor;
- DH_i: a hardware implementation of the D_i channel (an element of the D set);
- SH_i: a hardware implementation of the S_i channel (an element of the S set);
- MH: a hardware implementation of the memory M (either on-chip or off-chip);
- AH_i: a hardware implementation of an element of the A set.
- PI: the collection of all implementations of processes, i.e. the union of all PH_i and PS_{in} for each i and each n;
- DI: the collection of all implementations of D channels, i.e. the union of all D_i for each i;
- SI: the collection of all implementations of S channels, i.e. the union of all S_i for each i;
- AI: the collection of all implementations of memory location contained in the A set, i.e. the union of all A_i for each i.

The goal of the optimization process is defined as follows:

Given the quintuple SMPN $= (P, S, D, M, A)$, determine the vectorial mapping function $Fm(Fmp, Fms, Fmd, Fmm, Fmh)$, where:

- $Fmp : P \rightarrow PI$
- $Fms : S \rightarrow SI$
- $Fmd : D \rightarrow DI$
- $Fmm : M \rightarrow MH$
- $Fmh : A \rightarrow AI$

In particular, $Fmp(Pi) = PS_{in}$ if P_i is implemented in software on the nth processor. $Fmp(Pi) = PH_i$ if P_i is implemented in hardware as an accelerator.

We define the subsequent cost metrics:

- TIS_{in}: the software execution time needed by P_i on the nth processor;
- TIH_i: the hardware execution time of P_i as a hardware accelerator;
- $ALUT_i$: the hardware area required by P_i in Look-Up-Tables (LUTs) on the FPGA chip if it were synthesized in hardware;
- AFF_i: the hardware area required by P_i in FlipFlops (FFs) on the FPGA chip if it were synthesized in hardware;
- $ALUTP_n$: the hardware area required by the nth processor in LUTs on the FPGA chip;
- $AFFP_n$: the hardware area required by the nth processor in FFs on the FPGA chip;
- $ALUTD_i$: LUT hardware cost for D_i;
- $AFFD_i$: FF hardware cost for D_i;
- $ALUTS_i$: LUT hardware cost for S_i;
- $AFFS_i$: FF hardware cost for S_i.

The following definitions are also introduced:

- TS_i: a variable indicating the starting instant of P_i execution;
- TE_i: a variable indicating the ending instant of P_i execution;
- TD_i: a variable indicating the duration of P_i execution;
- $ALUTchip$: the maximum area available on chip in LUTs;
- $AFFchip$: the maximum area available on chip in FFs;

- $ALUTmax$: a user-specified maximum area design constraint in LUTs;
- $AFFmax$: a user-specified maximum area design constraint in FFs;
- $Tmax$: a user-specified maximum execution time constraint;
- $OnChipMemory$: the amount of on-chip memory;
- $OffChipMemory$: the amount of off-chip memory;
- $MemoryMax$: a user-specified maximum memory design constraint.

We need to specify that the maximum number of processors (n-index) be equal to $|P|$. This is to avoid the introduction of a useless limiting constraint, i.e. it should be possible to synthesize all processes in software, each on a separate on-chip processor.

We introduce the following decisional variables for the mapping problem:

- $X_i = 1$ if P_i is hardware-synthesized, 0 otherwise;
- $Y_{in} = 1$ if P_i is software-synthesized on the nth processor, 0 otherwise;
- $NY_n = 1$ if at least an element of P is mapped on the nth processor.

The general constraints are as follows:

- for each i, $Xi \leq 1$, where X_i is a binary variable;
- for each i and for each n, $Y_{in} \leq 1$, where Y_{in} is a binary variable;
- for each i, $X_i + \sum_n Y_{in} = 1$, P_i has to be synthesized in software or in hardware;
- for each n, $NY_n \leq 1$, where NY_n is a binary variable;
- for each i and for each n, $NY_n \geq Y_{in}$, by the definition of NY_n.

The resource constraints are as follows:

- $\sum_i X_i \cdot ALUT_i + \sum_n NY_n \cdot ALUTP_n + \sum_i ALUTD_i + \sum_i ALUTS_i \leq ALUTchip$
- $\sum_i X_i \cdot AFF_i + \sum_n NY_n \cdot AFF_n + \sum_i AFFD_i + \sum_i AFFS_i \leq AFFchip$
- $|M| + |A| \leq OnChipMemory + OffChipMemory$

The design constraints are as follows:

- $\sum_i X_i \cdot ALUT_i + \sum_n NY_n \cdot ALUTP_n + \sum_i ALUTD_i + \sum_i ALUTS_i \leq ALUTmax$
- $\sum_i X_i \cdot AFF_i + \sum_n NY_n \cdot AFF_n + \sum_i AFFD_i + \sum_i AFFS_i \leq AFFmax$
- for each i: $TE_i \leq Tmax$
- $|M| + |A| \leq MemoryMax$

The organization of the SMPN describing an OpenMP application, where the master process is at the root of a tree connecting all the slave process, greatly simplifies the design constraint on the overall latency of the system: $TmasterE \leq Tmax$, expressing the fact that the master process is the last process receiving the *Done* signal needed to complete the execution. The other timing constraints are as follows:

- for each i: $TE_i = TD_i + TS_i$
- for each i: $TD_i = X_i \cdot TIH_i + \sum_n Y_{in} \cdot TIS_{in}$
- for each i: $TS_i \geq ASAP(P_i)$

where $ASAP(P_i)$ is the instant of time at which the execution of P_i can start in compliance with all data dependencies induced by the structure of the OpenMP application. In the overall design flow, several actors can be mapped onto a single on-chip processor. Hence, we need a few scheduling constraints preventing concurrent processes mapped on the same processor to overlap with each other in time. The scheduling constraints are as follows. For each n and for each pair (P_i, P_j) of concurrent processes:

- $TE_i \leq TS_j + (3 - b_{ij} - Y_{in} - Y_{jn}) \cdot C_1$

- $TE_j \leq TS_i + (2 + b_{ij} - Y_{in} - Y_{jn}) \cdot C_2$

with C_1 and C_2 chosen with an appropriate size [112]. Table 5.1 clarifies the above scheduling constraints. b_{ij} is a binary variable and thus can take on only

$Y_{in} = Y_{jn} = 1$	b_{ij}	Constraint 1	Constraint 2
YES	0	$TE_i \leq TS_j + C_1$	$TE_j \leq TS_i$
YES	1	$TE_j \leq TS_i$	$TE_j \leq TS_i \cdot C_2$
NO	0, 1	$TE_i \leq TS_j + n_1 \cdot C_1$ with $n_1 \geq 1$	$TE_j \leq TS_i + n_2 \cdot C_2$ with $n_2 \geq 1$

TABLE 5.1: Interpretation of the scheduling constraints

values in 0,1. $b_{ij} = 0$ expresses the fact that P_j is executed before P_i, while $b_{ij} = 1$ corresponds to P_i being executed before P_j. Depending on the actual value of b_{ij}, only one of the two above constraints will have effect.

Finally, the objective function can be one of the following two, depending on whether we want to optimize area or latency:

- minimize: $f = \sum_i X_i \cdot ALUT_i + \sum_n NY_n \cdot ALUTP_n + \sum_i X_i \cdot AFF_i + \sum_n NY_n \cdot AFFP_n$
- minimize: $f = TmasterE$

5.4 OpenMP support

The first component of the prototypical environment is an ad-hoc lightweight compiler dealing with source-to-source transformation, called SOPHIE (Source-to-source OpenMP to Portable code compiler for HIgh-Level-SynthEsis). The compiler, built on top of a standard C grammar with OpenMP extensions, was implemented in C/C++ and relied on the well-known *Flex* and *Bison* tools [113] for the generation of the lexical scanner and the parser, respectively. It takes C source code with OpenMP 2.0 `#pragma` statements [105] as input, generating source files suitable for high-level synthesis, for the compilation on the target embedded microprocessors, and for functional simulation. In that respect, it substitutes the OpenMP clauses with ad-hoc structures as explained later in this section.

In the following, we provide the technical details of how the most relevant OpenMP clauses and functions are implemented.

5.4.1 `private` and `shared` variables

The code is analyzed to build a Data Scope Table (DST) containing all information related to variable scopes. The DST will be accessed each time we need to know whether a variable is shared or private. The entry corresponding to each variable is modified whenever a scope modifier is encountered (i.e. private, shared clauses) and is initialized with the default value according to the OpenMP standard.

Private variables are not required to be located in an addressing space accessible to all computing elements. This is particularly relevant for specialized hardware accelerators where private memory is made of limited elements, such as flip-flops or BRAMs, which should be used very carefully. When generating the source files for processors or HLS, the compiler accesses the DST for each variable it encounters.

The compiler provides each processing unit with a pointer to the globally shared address space and each access to a shared variable is replaced by a pointer dereferentiation and an offset. To this aim, the compiler keeps a map of the shared memory and builds the offsets accordingly. Notice that current HLS tools do support pointers and pointer arithmetic as long as arrays are mapped to random-access memory blocks, i.e. they do not involve wire, handshake, or FIFO interfaces, since these do not allow out of order accesses [36].

5.4.2 `parallel` directive

The `parallel` directive is fundamentally a way of starting threads and synchronizing them after they all have executed the piece of code marked by the directive. The corresponding overhead involves broadcasting synchronization information to the threads. The implementation of this directive thus greatly benefits from the tree-based scheme and the synchronization signals presented above and inherent to the SMPNs.

An additional key point concerns the assignment of thread IDs. To this aim, we exploit the capability of synchronization signals to convey integer values. The *start* signal that goes to the left child carries the value $tid \cdot 2 + 1$ while the other one carries $tid \cdot 2 + 2$, where *tid* is the thread ID assigned to the node. The master thread is assigned a thread ID equal to 0. Propagating the thread IDs through the tree is of fundamental importance, as it is the standard run-time technique used to identify threads. This is another example of a scalable mechanism put in place by the proposed flow. The assignment of thread IDs is in fact completely distributed, and there is no additional delay incurred by their computation apart from a shift, i.e. a multiplication by two, and an addition.

5.4.3 `for` directive

The `#pragma omp for` directive constitutes the most important work sharing construct of OpenMP and is the most common construct together with the `parallel` directive. Like the `parallel` directive, at the end of the work performed by each thread there is an implicit barrier unless a `nowait` clause is used.

The most relevant clause supported by the `for` directive is the scheduling type. In the case of *static* scheduling, iterations are trivially divided among processing elements as specified by the OpenMP standard. Basically, at the source code level, each processing element is provided with a code having modified indices in the loop to be partitioned. The way the new indices are calculated is compliant with the OpenMP specification and depends on the original bounds, the increment factor, and the chunk size. In the case of *dynamic* scheduling, the iterations each thread must perform are identified at run-time. This is of paramount importance for load balancing in heterogeneous architectures. In fact, threads should be assigned a number of iterations depending on the actual computational power of the unit where they are executing as well as the different loads they happen to handle. Again, the proposed implementation of the `dynamic` clause is fully distributed. Each hardware unit implements a state machine that delivers iterations to the computing resources synchronizing with the others. These state machines, also implemented by means of HLS, are described by the following code.

```
index = 0;
```

```
while (index  < total_iterations_number){
  temp = iter;                    //  lock on 'iter'
  if (temp != index ){
  index = temp;
  }
  if (temp != error_code && temp < total_iterations_number){
      prev_iter = temp;
      iter = prev_iter + chunk_size;  //  unlock on 'iter'
      for (i=prev_iter; i<=prev_iter+chunk_size;prev_iter++){
         original code of iterations
      }
      index = temp + chunk_size;
  }
}
```

where `iter` is an atomic memory location (also known as atomic register, see section 5.7) and `prev_iter`, `temp`, and `index` are local variables. The use of an atomic memory location allows the threads to synchronize for the update of the number of assigned iterations. In fact, in case of contention, only one thread will be able to read the number of already issued iterations (i.e. the `error_code` will not be returned), and consequently it will take the next available `chunk_size` of them. This code is compiled and executed by the software units as well.

5.4.4 `critical` directive

SOPHIE first counts the number of `critical` directives and then, for each output file, it inserts a macro defining the memory-mapped address of a number of atomic registers (see section 5.7) equal to the number of critical directives. The memory-mapped addresses are chosen in order not to conflict with all other addresses of physical memories and other peripherals. Then, the code contained into the critical directive is wrapped by a `lock/release` pair implemented accessing the atomic register corresponding to that critical directive. For example:

```
#pragma omp critical
{
    //CODE
}
```

is translated into:

```
#define atomic_register_offset_0 0xC00000
/*  Lock operation */
while (mem[atomic_register_offset_0] == -1);
//CODE
/*  Release operation */
mem[atomic_register_offset_0] = 1;
```

5.5 Functional simulation

As mentioned in section 5.3.1 and depicted in figure 5.1, the functional simulation relies on the PtolemyII framework. PtolemyII is mainly composed of polymorphic actors [107] that always exhibit the same behavior independent of the used MoC. The presence of several "directors" ensures the scheduling of the model according to the rules of a certain model of computation. Among the others, PtolemyII supports processes networks, in which a thread is executed for each actor and the channel operations obey KPN rules. To enable the functional simulation of software components extracted from OpenMP descriptions:

- We defined a new "director" extending the "Shared Memory Process Network Director". Before executing all actors, this director creates a pool of shared memory locations, whose starting address is passed to all actors in the system. This required hacking some parts of the PtolemyII Java source code.
- We devised a plug-and-play mechanism for the definition of the behavior of each SMPN actor, based on the JNA library [114], not complying with the PtolemyII actor class mechanisms for defining the functional behavior on the actor firing. As an example, the actor number 1 of the model automatically links the shared library `actor1` found in the same directory searching for a function with the following signature `void processOnFire(int start, char *sharedMemory, int *done)`. The parameter passing is embedded in the new "SMPN director", relying on the JNA library.

5.6 Early cost estimation

A central tool supporting automated DSE in the design flow is provided by EASTER (EArly coST EstimatoR), included in figure 5.1. In fact, early prediction of hardware complexity is essential in driving hardware/software partitioning and the automated generation of HDL descriptions from high-level code, as it helps estimate the "hardware cost" of a given high-level code segment *before* the actual synthesis takes place, dramatically reducing the time required for an exhaustive exploration of different design choices. An essential aspect of EASTER is that it can infer early hardware estimates directly from the source C / OpenMP code by applying a linear regression model to a set of metrics extracted from the high-level code. EASTER is developed on top of the LLVM compiler infrastructure [115] along with the R statistical package [116] used to perform regression analysis. It can be pre-configured for specific HLS engines by collecting extensive experimental results on a large body of C benchmarks. Relying on the custom LLVM-based compiler, the C source files are processed deriving a representation of the program in static single assignment (SSA) form. This representation captures a number of essential aspects of the program structure that can be used to perform quantitative evaluations on the application structure. As an example, the complexity of the control flow graph is likely to influence the amount of resources needed to implement the control logic of the hardware block. We thus consider, among the others, a set of metrics that capture such aspects, such as the cyclomatic complexity. Similar considerations can be done for the basic building blocks of the control flow graph. These correspond to basic operations such as additions, comparisons, multiplications, logic operations etc, which impact the amount of logic resources required to implement the given program in hardware. A few relevant software metrics are listed in Table 5.2.

During the pre-configuration of EASTER, performed only once for each different HLS engine to be used, estimations are computed for all supported metrics on the input high-level benchmarks, and are then collected in a database, along with the actual data coming from the synthesis and place&route process. The data are then examined by means of a linear regression analysis, yielding a linear model

Name	Meaning
`ops_div_flt`	floating point divisions
`ops_mul_flt`	floating point multiplication
`ops_add_flt`	floating point sum and subtractions
`ops_div_i`N	integer division (N bits wide) operations
`ops_mul_i`N	integer multiplication (N bits wide) operations
`ops_add_i`N	integer sum (N bits wide) operations
`ops_bra_cond`	conditional branches
`cfg_cyc`	cyclomatic complexity of the CFG
`blk_dep_tot`	weighted sum of the nesting level of all basic blocks
`loops`	number of loops in the CFG
`mem_flt_tot`	memory consumed by floating point variables
`mem_int_tot`	memory consumed by integer variables

TABLE 5.2: The most relevant software metrics identified by EASTER

describing the dependence between some of the metrics extracted and the real synthesis data.

Once the pre-configuration is completed, EASTER can be used to process the source files generated by SOPHIE, providing an efficient and convenient way to populate the ILP model of section 5.3.3 with reliable performance parameters (e.g. $ALUT_i$, AFF_i, etc). The computation of the estimates is as fast as a software compilation of the high-level code, followed by the application of a linear equation to the extracted metrics. The accuracy of the estimates is always within an acceptable threshold, being less than 20% for LUTs and less than 10% for flip-flops.

Notice that, in addition to hardware cost estimation, EASTER also provides a tool to evaluate the latency of the single components instantiated in the system. The tool relies on an Instruction Set Simulator (ISS) for each of the used general-purpose core. To give the designer an early estimate of the global execution time, the single latencies are combined by EASTER with the SMPN, describing how the components interacts with each other, and the results provided by the ILP solver.

5.7 System architecture

In the last steps of the design flow, the optimized design choices are translated into a physical implementation, as shown in the bottom part of figure 5.1. For the platform-based system composition we adopted the Embedded Development Kit (EDK) [117] as currently the prototypical environment only supports Xilinx FPGA devices and Microblaze processors for the implementation of the software subsystems. Figure 5.5 shows the reference architectural model adopted for the generation of the physical system from the initial OpenMP application.

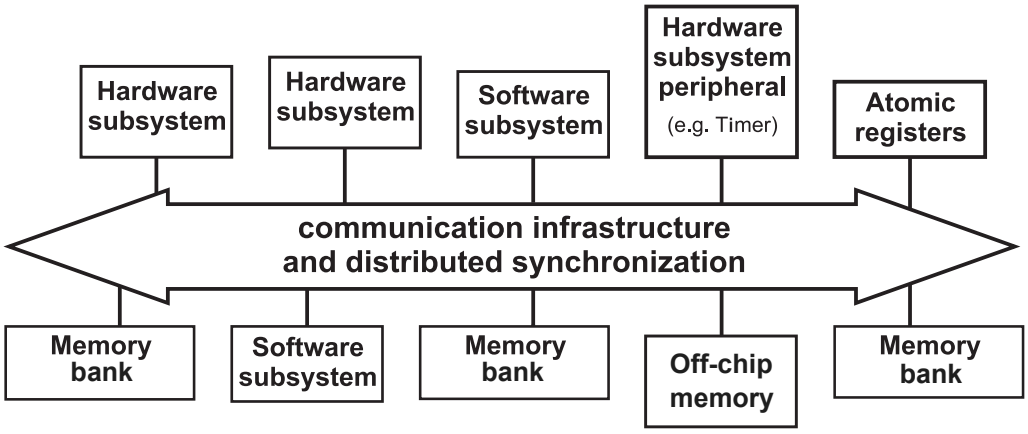


FIGURE 5.5: Architecture of a heterogeneous MPSoC derived from an OpenMP program

The system is structured in hierarchical interconnection domains since it derives directly from what seen in chapter 4. Each cluster containing a subset of the application components communicating by means of shared memory mechanisms. Software subsystems are processors for which a portable C code (derived from the OpenMP application) can be compiled. Hardware subsystems are generated with HLS in order to execute parts of the original parallel code, or they can be ordinary peripherals such as a non-volatile memory block to boot the code or a timer. Each subsystem represents an OpenMP thread, executing a certain portion of the original OpenMP code. The figure also depicts the memory infrastructure. Every part of the application memory can be implemented in a different technology and indeed be mapped to on-chip banks or off-chip memory. As explained before, the

choice is made by the designer explicitly. If a specific array must be mapped to on-chip memories it is subject to the memory partitioning step. In fact, each on-chip memory subsystem corresponds to a memory bank as explained in chapters 2 and 3. Special emphasis was put on the support for the `shared` and `private` clauses, which are essential in OpenMP because of its model inherently based on shared memory. As explained previously, shared variables are detected during compilation by tracking each access and replacing it with suitable memory operations accessing the appropriate areas.

The atomic registers, accessible on a particular address by all subsystems, allow the implementation of *read-only-after-write* primitives: if a read is performed without a previous write, an error value is always returned. This device acts as the basic building block for implementing synchronization directives. The number of instantiated atomic registers is equal to the number of `barrier`, `atomic`, and `critical` constructs plus the numbers of calls to `omp_init_locks()` (that creates a lock) found in the original OpenMP code.

Hardware subsystems generated by HLS are memory mapped, have each their own thread id (tid), have DMA master access capabilities, and include a *Start*/*Done* synchronization port, in compliance with the SMPN model. The connection of synchronization ports must obey the SMPN model. This is a fully distributed approach where the OpenMP threads involved in a fork-join structure form a tree. Each node in the tree, i.e. a thread, forwards a *Start* signal to its subtrees before starting its own computation, and forwards a *Done* signal to its parent node only after receiving all the subtree *Done* signals and completing its own task. As a consequence, the implicit barrier at the end of OpenMP work-sharing constructs takes a time corresponding to the worst-case propagation delay through the tree, which is logarithmic in the number of threads.

5.8 A step by step example: parallel JPEG encoding

This section demonstrates the proposed methodology by presenting a case study of moderate complexity and going through the overall design flow from the specification down to the FPGA synthesis. The memory partitioning and interconnect synthesis stages are not discussed because many examples have been already presented in previous chapters.

5.8.1 Parallel JPEG encoding

We chose a parallel JPEG encoder [118], where the image is decomposed into several parts in order to speed-up the compression operation by distributing it among several threads. We selected this specific case-study because:

- it is a moderately complex system enabling us to emphasize all the main aspects of the flow;
- it is a well-established and widely treated case-study in the literature;
- it is a data intensive application that may require a large amount of memory concurrently accessed by different components, which makes it well suited to an OpenMP-based implementation.

Figure 5.6 contains the block diagram of the system.

The application compresses a 320x240 pixel bitmap image encoded with 8 bits per pixel and stored into an external file-system. The compression uses a variable quality rate chosen by the user, compliant with the JPEG standard. The compressed image is stored into the same external filesystem. In the case the image is not grayscaled, it is converted by the *RGB2Gray* converter. The original image is displayed on a DVI screen during processing. The block diagram includes the typical components of a JPEG encoder. A key aspect is that some of them are parallelized, e.g. the bi-dimensional DCT is distributed among several threads to

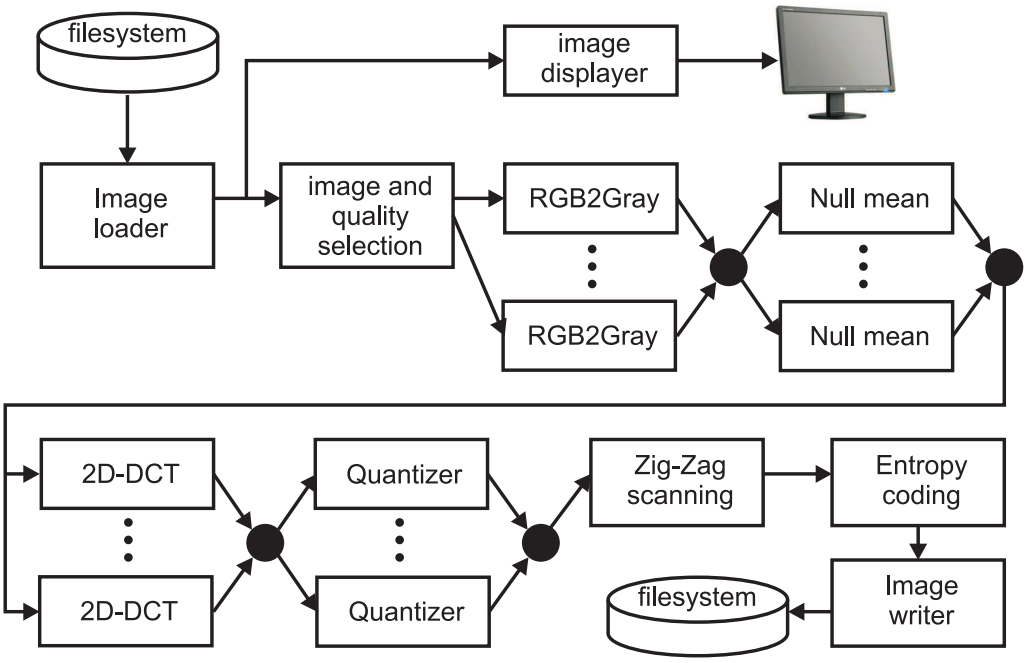


FIGURE 5.6: The block diagram of the case-study: JPEG encoding

improve the processing rate. Other components, on the other hand, are not parallelized because of the difficulties of extracting task-level parallelism from them. The implementation of a few blocks dependent on the specific platform, i.e. the filesystem, the image loader, the encoded image writer, the image displayer, and the module for user-interaction, relied on third-party IP cores.

5.8.2 Design steps

The first step in the flow is the C/OpenMP description of the application. This is required for those parts of the system that need to be mapped to hardware as accelerators or to software executed by on-chip processors. Third-party IP Cores are of course not described in the high-level application. Below an example code snippet is given:

```
unsigned char image[320*240]

omp_set_num_threads(4);

#pragma omp parallel private (all_the_other_variables) shared(image){
```

```
   RGB2Gray code
}
#pragma omp parallel  {
   NULL_MEAN code
}
#pragma omp parallel  {
   2D-DCT code
}
#pragma omp parallel  {
   Quantizer code
}
ZIG-ZAG SCANNING code
ENTROPY CODING code
```

Those components, where task-level parallelism can be extracted a-priori, are described by means of OpenMP directives. They are processed by four concurrent threads, as specified by the `omp_set_num_threads(4)` directive. Other components are described in plain C.

The next step resorts to the source-to-source compiler, SOPHIE, to analyze the C OpenMP code and generate two sets of source files for software compilation and for high-level synthesis, respectively. It was chosen because of its robustness and its comprehensive hardware/software synchronization mechanisms.

The range of the `for` cycle is partitioned according to the overall number of threads allocated for the RGB-to-Gray conversion. The generated hardware accelerator stops on a blocking read on the input *Start* channel and performs a non-blocking write on the *Done* channel as specified by the SMPN model. A similar source module, not shown here, is generated by SOPHIE to support the functional simulation with PtolemyII and the software compilation for on-chip processors. Its interface is `void processOnFire(int start, unsigned char *sharedMemory, int *done)`, while the code is very similar to example given above except for the HLS-specific library function calls.

The generated source files are compiled as a shared library, so that the function can be called by the simulator through the JNA library. Figure 5.7 shows the PtolemyII depiction of the system derived from the SMPN specification of the

system. The shared memory is not displayed but each actor has a pointer to it. The barrier actors in the model are used to simulate the behavior of the master

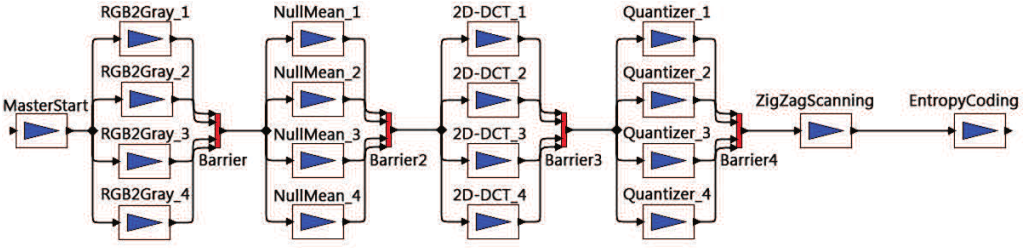


FIGURE 5.7: PtolemyII simulation of the case-study

process. They can be used to build a SMPN whenever an OpenMP application has multiple consecutive parallel regions. The representation is isomorphic with the actual SMPN model of the application. Figure 5.8 shows the simulation results.

FIGURE 5.8: An example of execution of the case-study application

Following the source generation, the EASTER tool performs early cost function estimation taking as input the same source files used for software compilation and HLS. The design space exploration process is based on the LPSolve solver [119] relying on a branch-and-bound algorithm to solve the ILP model described in section 5.3.3. Constraining area and optimizing latency (or viceversa), yields the typical Pareto-like curve. Finally, we proceed with system synthesis. The components are separately synthesized and evaluated to verify the early cost function estimations made by EASTER. Specifically, hardware accelerators are synthesized by means of HLS, while software components are executed on Xilinx Microblaze

soft-cores in order to evaluate their execution times relying on the *xps-timer* IP Core for time measurement. The timer was also used for measuring hardware latencies. Xilinx XPS is used for system composition and Xilinx SDK for software compilation and linking. The supporting components not directly involved in the bulk processing (image and quality selection, filesystem, image loader, encoded image writer, and image displayer) rely on Xilinx IP cores and the related device drivers. For the physical implementation, we used a development board equipped with a Xilinx XUPV5-LX110T device, a solid-state memory device containing the filesystem, a DVI connector, and a few switches exploited for user interaction.

5.8.3 Quantitative results for the JPEG encoder

Table 5.3 shows the results of the hardware cost early estimation performed by EASTER, targeting a xcv5-lx110t Virtex-5 FPGA device. To evaluate the accu-

Component	Estimated LUTs	Estimated Flip-flops
RGB2Gray(single thread)	619	436
NullMean(single thread)	702	510
2D-DCT(single thread)	1175	622
Quantizer(single thread)	987	583
ZigZag Scanning	1078	557
Entropy Coding	1321	808

TABLE 5.3: Results of the hardware cost early estimation

racy of the estimates derived by the tool, Table 5.4 provides the actual results of the synthesis process, which match very closely those obtained during the fast early estimation step.

Component	Placed LUTs	Placed Flip-flops
RGB2Gray(single thread)	678	478
NullMean(single thread)	652	558
2D-DCT(single thread)	1345	701
Quantizer(single thread)	901	532
ZigZag Scanning	978	513
Entropy Coding	1525	938

TABLE 5.4: Actual results in terms of hardware cost

The ILP model specified for this experiment counts 1507 constraints and 396 variables. Model solving took approximately 1 second or less, depending on the chosen objective function.

The design space exploration performs a comprehensive evaluation of the area/-time trade-off, indicating various choices for hardware/software partitioning based on the given area/time constraints. Some solutions in the design-space are depicted in figure 5.9, which shows both early estimated and actual results. These are the solutions obtained by constraining LUT occupation and minimizing global execution latency, which yields a familiar Pareto-like curve, as shown in the figure. For

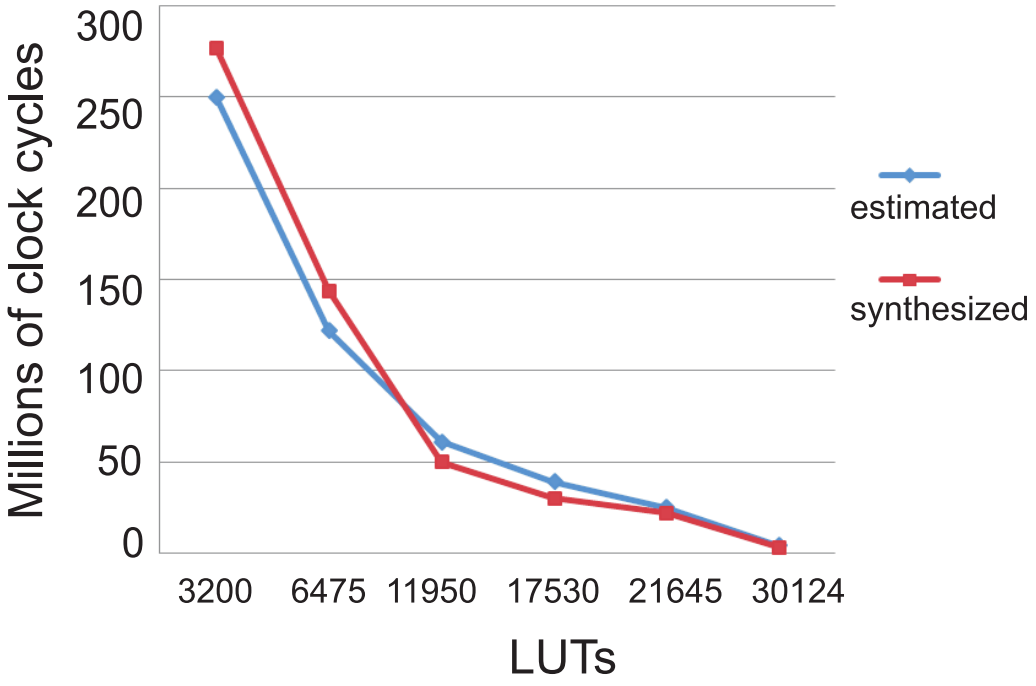


FIGURE 5.9: DSE results for the case-study

example, considering a constraint of 12000 LUTs, the solver allocates the four 2D-DCT components to hardware accelerators and instantiates four processors. The $RGB2Gray_i$, $NullMean_i$, and $Quantizer_i$ components are allocated to the ith processor as software routines and scheduled sequentially. The *Zig-Zag Scanning* and *Entropy coding* components are also synthesized as software routines and allocated to the first processor. The pure area overhead related to both synchronization and communication OpenMP constructs and clauses in this example is 350 LUT and 200 FF per accelerator that are respectively the 0.5% and the 0.2% of the available chip area.

5.9 Experimental results

This section demonstrates the improvements over a previous work enabled by some architectural techniques (section 5.9.1) as well as the scalability of the approach (section 5.9.2).

5.9.1 Comparisons with previous work

We compared the results with [2], that is the only OpenMP-to-hardware translator presenting a working tool and some quantitative performance evaluation. However, the work in [2] does not generate a heterogeneous system. Rather, it generates only hardware components (basically a state machine for each OpenMP thread) which communicate using a centralized controller. As a consequence, it is inherently suitable only for small systems.

Figure 5.10.a and figure 5.10.b compare the results with [2] in terms of speedup and system frequency as the number of threads increases. Figure 5.10.a has been obtained by an OpenMP description of the Sieve of Eratosthenes algorithm, while figure 5.10.b concerns a program running an infinite impulse response filter, as in [2]. As can be noticed, the approach compares favourably in terms of scalability. In fact, the approach in [2] suffers from frequency degradation due to the presence of centralized components that become more and more complex as the number of threads increases. Moreover, its speedup is limited by the presence of such centralized components that act as performance bottlenecks. On the contrary, being highly distributed, the approach yields a satisfactory trend concerning both clock frequency and speedup.

In conclusion, the approach achieves up to a 3.25x improvement concerning speedup and a 4x improvement of the clock frequency.

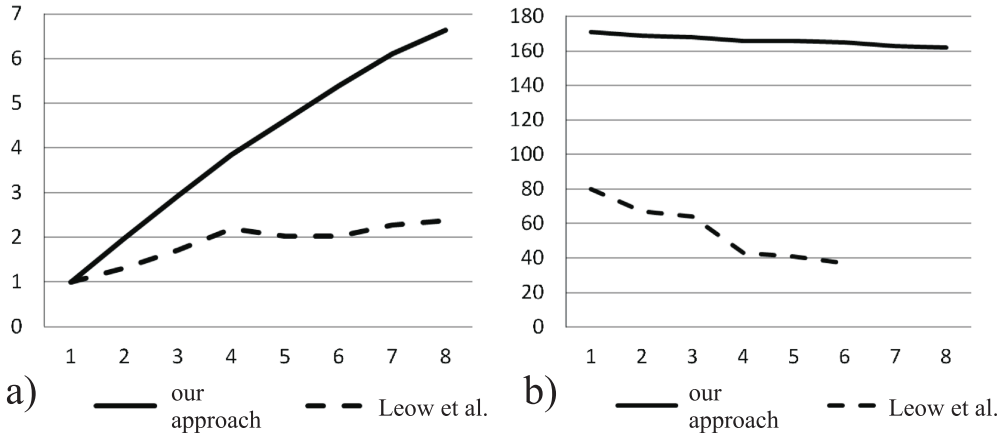


FIGURE 5.10: Comparisons with [2] (denoted *Leow et al.*) for the implementation of of the Sieve of Eratosthenes algorithm. (a) Speed-up vs. number of threads. (b) System frequency (MHz) vs. number of threads.

5.9.2 Overheads and comparisons with software approaches

We also measured the run-time overhead incurred by the implementation of OpenMP directives. Precisely, we evaluated the *normalized overhead*, i.e. the overhead/execution time ratio, so that we could figure out which portion of the execution time was taken by the runtime mechanisms for supporting constructs. To this aim, we resorted to the well-known EPCC benchmarks [120] that provides a way to evaluate performance overheads and compare OpenMP implementations. For the experiments, we relied on a Virtex5-LX110T FPGA device.

The EPCC suite is composed of three applications:

1) `Array_bench`, related to the overhead of handling variables scopes;

2) `Sched_bench`, evaluating scheduling strategies;

3) `Sync_bench`, running several synchronization mechanisms e.g. `barrier`, `single`, `critical`, `lock`.

We benchmarked the implementation against a purely software version running on an Intel i7 processor. However, due to normalization, technological details were abstracted away favouring a fair comparison of scalability and efficiency. The

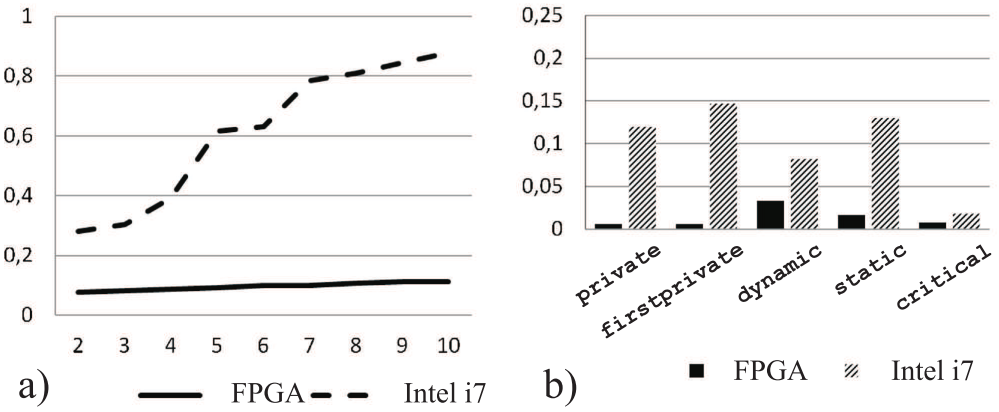


FIGURE 5.11: Experimental results. (a) Normalized overhead vs. number of threads. (b) Average overhead slopes for several OpenMP constructs.

measures were conducted for many directives using 2 to 10 threads. 10 measures per experiment were carried out and, then, averaged.

Figure 5.11.a presents the normalized overhead as the number of threads is increased for the `#pragma omp parallel` directive. As suggested by the plot, the normalized overhead (depicted by the solid line) is low in absolute values and tends to be horizontal. This confirms the efficiency and the scalability of the presented approach. Looking at the `#pragma omp parallel private(var)` construct, the FPGA system synthesized with the proposed approach had an average of 0.096 on the average whereas the i7 implementation of 0.616. The overhead in the presented methodology was only related to the activation of the threads by the *start* signals and the barrier at the end of the construct controlled by the *done* signals.

The `#pragma omp parallel firstprivate (var)` clause had an average overhead of 0.701 versus 0.841 on the i7. The overhead was the same of the `private` clause plus a constant value due to the variable being copied from the shared memory of the master thread to the private memory locations of the accelerators. The `#pragma omp for schedule (static)` clause corresponds to the default setting for the thread scheduling strategy. It is thus representative of the `#pragma omp for` construct. The average overhead was 0.069 versus 0.725 on the i7. The `#pragma omp for schedule (dynamic)` clause is supported by the proposed environment in spite of its dynamic nature, unlike the existing solutions for OpenMP-to-hardware translation. The average overhead was 0.283 versus 0.895

on the i7. Some constructs like `#pragma omp single` and `#pragma omp master` are supported with no overhead since are solved totally at run-time.

Figure 5.11.b shows the overhead trends, showing the *slopes* of the overhead obtained in the experiments. As an example, the overhead of the proposed methodology concerning the `#pragma omp parallel firstprivate (var)` directive increases by 0.006 per thread, while it increases by 0.147 per thread in the software implementation.

5.10 Related works

A large body of research works addressed embedded systems co-design methodologies, also including design space exploration techniques. The Daeadalus framework [121] synthesizes C applications onto a programmable platform by an automated parallelization step based on Kahn Process Networks (KPNs) and an optimization step based on a Strength Pareto Evolutionary Algorithm. Daedalus architectures are built from a library of predefined and preverified IP components, including a variety of programmable processors, dedicated hardwired IP cores, memories, and interconnects, allowing the implementation of a wide range of heterogeneous MPSoC platforms. Although Daedalus is one of the most complete academic approaches to ESL design, it does not provide standard forms for specifying explicit parallelism, preventing the reuse of a large body of parallel code, e.g. multicore applications, based on multithreading. Moreover, the design flow does not make use of high-level synthesis for hardware design, being a purely platform-based approach. SystemCoDesigner [122] is another academic environment for ESL. The SystemCoDesigner design flow begins by describing the application in form of a SystemC behavioral model to be written using the SYSTEMOC library [122]. Each SystemC module is defined by a finite state machine specifying the communication behavior and methods controlled by the finite state machine. These methods are executed atomically and data consumption and production is only done after computing a method. Hence, SYSTEMOC realizes a rule-based model of computation. The application domain in SystemCoDesigner is restricted to multimedia and networking, i.e. streaming applications. Unlike

Daedalus, SystemCoDesigner relies on high-level synthesis, allowing each actor in the application to be implemented either as a hardware component on the FPGA or a software routine. Furthermore, its design-entry formalism expresses explicit task-level parallelism by incorporating a specific MoC. The design-entry language, a subset of SystemC constrained by a particular MoC and incorporated into the language by means of a library, limits the range of supported applications as it does not support parallel multi-threaded code written for multicore applications and high performance computing platforms, unlike the design flow presented here. Other approaches include Metropolis [111], Koski [123], PeaCE [124]. All these approaches rely on well defined MoCs: Koski uses KPNs, Metropolis allows the use of various MoCs, and PeaCE uses a combination of two MoCs, one for control specification and another for data processing specification. None of them takes into account the possibility of using a design-entry language with support for explicit parallelism. Another example of an ESL design flow relying on a KPN model is Compaan [125]. It requires input applications to be specified as parameterized static nested loop programs in the Matlab language. A KPN specification is automatically derived and mapped onto a target platform composed of a microprocessor and an FPGA.

As emphasized above, none of the previous ESL environments supports high-level languages and formalisms borrowed from the parallel computing domain, particularly OpenMP [105], in spite of its popularity and its powerful model allowing an easy description of parallel high-performance applications. The only aspect recently explored by some works is related to the generation of hardware components from OpenMP descriptions, along with the integration of hardware accelerators in purely software OpenMP applications. The work in [2] created backends that generate synthesizable VHDL or Handel-C [126] code from OpenMP C programs, targeting an FPGA platform and presenting some quantitative results from their tool. The approach is basically oriented to pure hardware synthesis where each thread corresponds to a finite state machine. There is no explicit memory hierarchy, limiting the available memory to the resources available on chip and making it difficult to support the shared memory in a scalable and efficient way. Few details are provided concerning synchronization, while nested parallelism and dynamic scheduling, a part of the OpenMP specification, are not supported. The authors

of [127] presented the possibilities and limitations of the hardware synthesis of OpenMP directives. The work concludes that OpenMP is relatively "hardware friendly" due to the formalisms allowing an easy description of explicit parallelism. However, it excludes the full support for the OpenMP standard, dropping dynamic scheduling of threads, an essential aspect of OpenMP, and typically software routines such as those related to time. The related work in [128] presented a methodology for transforming an OpenMP program into a functionally equivalent SystemC description. The emphasis is mainly on the source-to-source transformation process, as the output code is not necessarily guaranteed to be compliant with the OSCI SystemC synthesizable Subset. The authors in [129] present some extensions to OpenMP and a related runtime system implementation to specify the offloading of tasks to reconfigurable devices and the interoperability with OpenMP. Their contribution essentially relies on a host-based model where one or more FPGA boards act as accelerators providing functions to an OpenMP application.

5.11 Conclusions

The research in the area of ESL design is increasingly targeting languages and formalisms borrowed from the software domain for the specification of complex embedded systems, including paradigms used for parallel computing. In line with this trend, this chapter investigated an ESL design methodology based on the widespread OpenMP parallel programming language for design entry and specification. As memory and communication are known to be the real bottlenecks in parallel embedded systems, taking advantage of the memory partitioning and interconnect design techniques presented in the previous chapters ensures high levels of scalability. Unlike a few previous attempts to using OpenMP as a pure hardware specification language [2, 127, 128], the proposed toolchain takes a system-oriented approach targeting the automated generation and optimization of complex, heterogeneous MPSoCs, fully supporting the complete OpenMP specification based on a range of techniques for the automated generation of the hardware/software system components. One of the biggest benefits using the proposed approach along with its prototypical environment is the inherent support for dynamic scheduling,

it is vital for heterogeneus architecture in which load balancing strongly affects the overall performance; this is a unique feature that is not found in other related works. Furthermore, unlike other approaches, the final architecture is fully distributed both in hardware and in software also in the memories distribution; a partitioned memory architecture is key to scalability. While several research works aim at defining design methodologies encompassing design space exploration and behavioral application specification [111, 121–124], none of them supports high level parallel programming paradigms, unlike the approach proposed here. In particular, to support the specification of high-level multi-threaded applications specified through OpenMP, this work explored a new model of computation that ideally suits OpenMP, allowing a direct formalization of high-level multithreaded applications and underpinning an effective approach to design space exploration. The architecture optimization relies on an ad-hoc ILP formulation of the hardware/-software partitioning and mapping problems, supported by an original technique for early cost estimation to dramatically reduce the time required by the solution space exploration. As a conclusion, thus, the presented ESL methodology creates a direct path form high-level multi-threaded OpenMP applications down to automatically synthesized, heterogeneous hardware/software systems implemented onto FPGA devices, extending the spectrum of ESL design to high-level OpenMP applications.

Chapter 6

Conclusions

The decades of frequency scaling as the highroad for improving performance has definitely come to an end [130]. As the era of heterogeneous many-core computing approaches, the memory and interconnect subsystems are becoming the overwhelming bottleneck in the evolution of performance. In that respect, two kinds of innovations appear to have the potential to change the future of computing. On the one hand, there are key innovative technologies, such as 3D integration and optical interconnects. For example, 3D integration implies shorter on-chip wires and optical interconnects provide a promising low-power alternative. On the other hand, electronic system level approaches aim to improve designs quality by raising the abstraction layer and filling semantic gaps with intelligent design automation techniques. The work presented in this thesis can be ascribed to the latter category and, in particular, deals with the issues of memories and interconnects customization because the way they are designed impacts scalability more than anything else. Throughout the thesis we have adopted a bottom-up approach by which we have first explained our solutions for memory and interconnects and then integrated them in a comprehensive design flow along with other architectural optimizations. Although the presented techniques apply to both ASICs and programmable logic, the experiments have been carried out by using FPGAs and their accompanying programming tools. High-end FPGAs can be provided with up to several thousands independent memory banks for an on-chip capacity up

to around 10 Megabytes [58]. The presence of a fine-grained distributed memory, together with the ease of programmability, makes them an ideal platform for experimenting customized solutions.

During chapter 2 we have seen how, in presence of many small and independent memory blocks, memory partitioning constitutes the essence of memory design and it is one of the most important factors for the actual parallelism achieved by a synthesized circuit. Current high-level synthesis tools implement basic techniques [36] by means of cyclic or block partitioning. Moreover, they partition accesses at statements granularity instead of instances; this means that a certain access in a certain statement is always made from the same memory bank. However, this limitation can be overcome by the adoption of polyhedral compilation tools that look at instances, not just statements. As explained in the related work section of chapter 2, the literature offers works already using polyhedral techniques for memory partitioning, however they use a limiting mathematical abstraction for representing partitioning solutions, namely hyperplanes. Hyperplanes are limiting from two perspectives. First, they restrict the design space cutting off many possibilities potentially good for performance; second, the excluded possibilities are often the most area efficient. This second effect is the subject of chapter 3; spatially irregular partitioning solutions lead to inefficient datapaths as they incur the bank switching effect. In that respect, while chapter 2 has introduced the partitioning methodology, chapter 3 has shown how to improve the circuit starting from an imposed partitioning solution providing a technique for reducing bank switching. The mathematical abstraction used throughout the treatment are integer sparse lattices. As lattices geometrically include hyperplanes, they enlarge the solution space. Furthermore, they are provided with a high degree of spatial regularity that directly reflects into more compact datapaths. The adopted benchmarks have shown that the adoption of lattices can save up to nearly half of the area of the synthesized circuit while achieving the same latency. One more contribution of the thesis, still concerning the memory subsystem, is having formalized for the first time the interplay between memory partitioning, bank switching and loop unrolling. Unrolling has been shown to potentially improve the latency-area product of the synthesized circuit considerably in presence of partitioning, three

times on average for the considered benchmarks in case of eight independent memory banks. As a future development for the partitioning methodology, liveness of memory locations might be taken into account. If a certain memory location is not meant to be accessed anymore it can be overwritten with a different one, leading to a reduction of the overall footprint.

Throughout chapters 2 and 3 there was an implicit assumption: the interconnection between the processing elements and the memory banks was always a partial crossbar. This limitation stemmed only from relying on HLS tools in order to generate the circuit after having transformed the source code introducing several array names one-to-one mapped to banks. In chapter 4 we have removed this limitation presenting a methodology for automating the communication architecture for complex systems as in the case of numerous independent memory banks. As for the memory partitioning technique, the methodology aims to customize the interconnect to the system's requirements. As a consequence, the solutions depend on how the application memory has been partitioned across the available banks; different partitioning solutions imply different traffic profiles. The outcome of the methodology is a heterogeneous interconnection infrastructure made of buses and crossbars linked by means of ad-hoc configured bridges. The architecture is clusterized so that highly interacting subsystems can be placed close to each other. In particular, three levels of parallelism have been identified: global-parallelism across different independent clusters, intra-domain parallelism, relying on inherently concurrent interconnect components such as crossbars, and inter-domain parallelism, where multiple concurrent paths across different local domains are exploited. After the design space has been characterized, an automated methodology to search it was presented, aimed at maximizing the exploitation of those forms of parallelism. The approach differentiates from the current literature because it performs communication scheduling and interconnect synthesis concurrently. This leads to a better communication locality within the same cluster and to a more natural exploitation of the inter-cluster parallelism inherent to the architecture. For the considered benchmarks, the presented methodology allows an area reduction ranging between 30% and 70% compared to full crossbar implementations while keeping the latency degradation negligible.

In the final chapter, the thesis has proposed an automated design flow that well

fits in the current ESL panorama and tendencies. As the experimental results confirm, it offers high levels of scalability and performance. This comes as the result of having integrated automated techniques for memory partitioning and interconnection design. The adoption of OpenMP as design entry enables the reuse of a large body of OpenMP code and kernels developed by the parallel computing community. However, adopting a very high-level language creates a big semantic gap that must be filled in order to derive a synthesizable system. Throughout chapter 5 several techniques are presented in order to fill this gap partly reusing the techniques introduced in the previous parts of the thesis. However, besides interconnection and memory structures, other issues have been tackled such as hardware/software partitioning and the support of OpenMP constructs. In that respect, the implementation of proper dynamic scheduling of loops constitutes a key innovation because it enables load balancing, traditionally regarded as an essential aspect for heterogeneous systems. Experimental data have highlighted a speed-up improvement of roughly three times against similar approaches for the considered case study. The design flow executes sequentially the several steps that are memory partitioning, interconnect synthesis, hardware/software task partitioning and system synthesis. Undoubtedly, the separation of those steps simplifies the flow considerably, but on the other hand it represents an opportunity for improvements and future developments. Ideally, all the design choices should be carried out in one single design exploration stage that takes into account every possible decision.

Bibliography

[1] Minje Jun, Sungjoo Yoo, and Eui-Young Chung. Mixed integer linear programming-based optimal topology synthesis of cascaded crossbar switches. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pages 583–588. IEEE, 2008.

[2] Y.Y. Leow, C.Y. Ng, and W.F. Wong. Generating hardware from OpenMP programs. In *IEEE International Conference on Field Programmable Technology (FPT 2006)*, pages 73–80, December 2006.

[3] Sudeep Pasricha and Nikil Dutt. *On-Chip Communication Architectures: System on Chip Interconnect*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 012373892X, 9780123738929.

[4] *AMBA Specification (Rev 2.0)*. ARM, 1999.

[5] IBM. Coreconnect interconnect standard, 2012.

[6] *AMBA AXI and ACE Protocol Specification*. ARM, 2011.

[7] *STBus Communication System: Concepts And Definitions*. STMicroelectronics, 2007.

[8] Arteris. From bus and crossbar to network-on-chip. White Paper, 2009.

[9] Giovanni De Micheli and Luca Benini. *Networks on chips: technology and tools*. Morgan Kaufmann, 2006.

[10] B. Bailey, G. Martin, and A. Piziali. *ESL Design and Verification. A Prescription for Electronic System-Level Methodology*. Elsevier Inc, 2007. ISBN 978-0-12-373551-5.

[11] P. Coussy and A. Morawiec (Eds.). *High-Level Synthesis from Algorithm to Digital Circuit*. Springer, 2008. ISBN 978-1-4020-8587-1.

[12] M. Fingeroff. *High-Level Synthesis Blue Book*. Xlibris, Corp., 2010. ISBN 978-1-4500-9724-6.

[13] Andr R. Brodtkorb, Trond R. Hagen, and Martin L. Stra. Graphics processing unit (gpu) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4 – 13, 2013. ISSN 0743-7315. Metaheuristics on GPUs.

[14] Jia-Jhe Li, Chi-Bang Kuan, Tung-Yu Wu, and Jenq Kuen Lee. Enabling an opencl compiler for embedded multicore dsp systems. In *Proceedings of the ICPP'12 Workshops*, pages 545–552. IEEE Computer Society, 2012. ISBN 978-1-4673-2509-7.

[15] Alessandro Cilardo, Luca Gallo, and Nicola Mazzocca. Design space exploration for high-level synthesis of multi-threaded applications. *J. Syst. Archit.*, 59(10):1171–1183, November 2013. ISSN 1383-7621.

[16] Gautam Gupta and Sanjay Rajopadhye. The Z-polyhedral model. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–248, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8.

[17] Ethan E. Danahy, Sos S. Agaian, and Karen A. Panetta. Algorithms for the resizing of binary and grayscale images using a logical transform. In Jaakko Astola, Karen O. Egiazarian, and Edward R. Dougherty, editors, *Proceedings of SPIE 2010*, volume 6497, page 64970, 2007.

[18] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1-55860-286-0.

[19] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Trans. Des. Autom. Electron. Syst.*, 16(2):15:1–15:25, April 2011. ISSN 1084-4309.

[20] Jonathan Kelner. Lecture 18, an algorithmists toolkit, 2009.

[21] Jürgen Teich and Lothar Thiele. Partitioning of processor arrays: A piecewise regular approach. *Integr. VLSI J.*, 14(3):297–332, February 1993. ISSN 0167-9260.

[22] H. Le Verge. Recurrences on lattice polyhedra and their applications, 1995.

[23] Patrice Quinton, Sanjay Rajopadhye, and Tanguy Risset. On manipulating z-polyhedra. Technical report, 1996.

[24] Rachid Seghir. Zpolytrans: A library for computing and enumerating integer transformations of z-polyhedra. In *Proceedings of the IMPACT 2012, Second International Workshop on Polyhedral Compilation Techniques*, 2012.

[25] Albert Cohen, Sylvain Girbal, and Olivier Temam. A polyhedral approach to ease the composition of program transformations. In *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 292–303. Springer, 2004. ISBN 3-540-22924-8.

[26] Benot Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. R-stream compiler. In *Encyclopedia of Parallel Computing*, pages 1756–1765. Springer, 2011. ISBN 978-0-387-09765-7.

[27] V. Loechner. Polylib: A library for manipulating parameterized polyhedra, 1999.

[28] Sven Verdoolaege and Kevin M. Woods. Counting with rational generating functions. *J. Symb. Comput.*, 43(2):75–91, 2008.

[29] Martin Griebl and Christian Lengauer. The loop parallelizer loopo-announcement. In David C. Sehr, Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *LCPC*, volume 1239 of *Lecture Notes in Computer Science*, pages 603–604. Springer, 1996. ISBN 3-540-63091-0.

[30] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, October 2007.

[31] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992. ISSN 0885-7458.

[32] Alexander Barvinok. *A Course in Convexity*. American Mathemtical Society, 2002.

[33] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005. ISSN 0018-9340.

[34] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986. ISBN 0-471-90854-1.

[35] Morris Newman. *Integral Matrices*, volume 45 of *Pure and Applied Mathematics*. Academic Press, 1972.

[36] Xilinx Inc. *Vivado Design Suite User Guide High-Level Synthesis*, 2012.

[37] Guillame Iooss and Sanjay Rajopadhye. A library to manipulate z-polyhedra in image representation. In *IMPACT 2012*, 2012.

[38] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2229-7.

[39] Eric W. Weisstein. *CRC Concise Encyclopedia of Mathematics*. CRC Press, 2003.

[40] Alain Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *Integration*, 12(3):293–304, 1991.

[41] Alain Darte, Robert Schreiber, B. Ramakrishna Rau, and Frederic Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Trans. Des. Autom. Electron. Syst.*, 7(1):159–172, January 2002. ISSN 1084-4309.

[42] Manish Gupta. Automatic data partitioning on distributed memory multicomputers, 1992.

[43] Siddhartha Chatterjee, John R. Gilbert, Fred J. E. Long, Robert Schreiber, Shang-Hua Teng, Dr. John, and R. Gilbert. Generating local addresses and communication sets for data-parallel programs, 1995.

[44] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: A tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems*, 2007(5): 1–13, 2007.

[45] Peng Li, Yuxin Wang, Peng Zhang, Guojie Luo, Tao Wang, and Jason Cong. Memory partitioning and scheduling co-optimization in behavioral synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 488–495, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1573-9.

[46] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. Memory partitioning for multidimensional arrays in high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference*, pages 12:1–12:8, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2071-9.

[47] Yuxin Wang, Peng Li, and Jason Cong. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, pages 199–208, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2671-1.

[48] Qiang Liu, George A. Constantinides, Konstantinos Masselos, and Peter Y. K. Cheung. Combining data reuse with data-level parallelization for FPGA-targeted hardware compilation: a geometric programming framework. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(3):305–315, 2009. ISSN 0278-0070.

[49] Song Chen and Adam Postula. Synthesis of custom interleaved memory systems. *IEEE Trans. VLSI Syst.*, 8(1):74–83, 2000.

[50] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A Rountev, P. Sadayappan, Yongjian Chen, Haibo Lin, and Tin-Fook Ngai. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *Parallel Architectures and Compilation Techniques,*

2009. PACT '09. 18th International Conference on, pages 348–357, Sept 2009.

[51] Qiang Liu, George A. Constantinides, Konstantinos Masselos, and Peter Y. Cheung. Automatic On-chip Memory Minimization for Data Reuse. In *Proceedings of the 2007 Field-Programmable Custom Computing Machines.*, pages 251–260, 2007.

[52] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 29–38, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1887-7.

[53] Samuel Bayliss and George A. Constantinides. Optimizing sdram bandwidth for custom fpga loop accelerators. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 195–204, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1155-7.

[54] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for fpga. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 575–580, San Jose, CA, USA, 2013. EDA Consortium. ISBN 978-1-4503-2153-2. URL http://dl.acm.org/citation.cfm?id=2485288.2485430.

[55] Claudia Leopold. On optimal temporal locality of stencil codes. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 948–952, New York, NY, USA, 2002. ACM. ISBN 1-58113-445-2. doi: 10.1145/508791.508975. URL http://doi.acm.org/10.1145/508791.508975.

[56] A. Cilardo and L. Gallo. Improving multi-bank memory access parallelism with lattice-based partitioning. *ACM Transactions on Architecture and Code Optimization*, 11(4), 2014. ISSN 1544-3566.

[57] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *Computers, IEEE Transactions on*, 54(10):1242–1257, Oct 2005. ISSN 0018-9340.

[58] Xilinx Inc. 7 series fpgas overview. Data sheet, 2014.

[59] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, October 1991. ISSN 1045-9219.

[60] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press.

[61] Umit Y Ogras, Jingcao Hu, and Radu Marculescu. Key research problems in noc design: a holistic perspective. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 69–74. ACM, 2005.

[62] Partha Pratim Pande, Cristian Grecu, Michael Jones, Andre Ivanov, and Resve Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *Computers, IEEE Transactions on*, 54(8): 1025–1040, 2005.

[63] Alessandro Cilardo, Edoardo Fusella, Luca Gallo, and Antonino Mazzeo. Joint communication scheduling and interconnect synthesis for fpga-based many-core systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*. IEEE, 2014.

[64] Alexander Strehl and Joydeep Ghosh. Cluster ensembles—a knowledge reuse framework for combining multiple partitions. *The Journal of Machine Learning Research*, 3:583–617, 2003.

[65] Jack Edmonds. *Optimum branchings*. National Bureau of standards, 1968.

[66] Nikhil R Devanur and Uriel Feige. An o (n log n) algorithm for a load balancing problem on paths. In *Algorithms and Data Structures*, pages 326–337. Springer, 2011.

[67] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition, 1994. ISBN 0070163332.

[68] Keki M. Burjorjee. Explaining optimization in genetic algorithms with uniform crossover. In *Proceedings of the twelfth workshop on Foundations of genetic algorithms XII*, FOGA XII '13, pages 37–50, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1990-4. doi: 10.1145/2460239.2460244. URL http://doi.acm.org/10.1145/2460239.2460244.

[69] Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Jürgen Teich. Opt4j: a modular framework for meta-heuristic optimization. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 1723–1730, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0557-0. doi: 10.1145/2001576.2001808. URL http://doi.acm.org/10.1145/2001576.2001808.

[70] Egbert Mujuni and Frances Rosamond. Parameterized complexity of the clique partition problem. In *Proceedings of the fourteenth symposium on Computing: the Australasian theory - Volume 77*, CATS '08, pages 75–78, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc. ISBN 978-1-920682-58-3. URL http://dl.acm.org/citation.cfm?id=1379361.1379375.

[71] R Duncan Luce and Albert D Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, 1949.

[72] *Zynq-7000 All Programmable SoC Overview*. Xilinx, 2012.

[73] *LogiCORE IP AXI Interconnect (v1.06.a)*. Xilinx, 2012.

[74] Robert P Dick, David L Rhodes, and Wayne Wolf. Tgff: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101. IEEE Computer Society, 1998.

[75] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):679–698, 1986.

[76] Cuong Pham-Quoc, Zaid Al-Ars, and Koen Bertels. A heuristic-based communication-aware hardware optimization approach in heterogeneous multicore systems. In *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, pages 1–6. IEEE, 2012.

[77] Partha Pratim Pande, Cristian Grecu, Michael Jones, André Ivanov, and Res Saleh. Effect of traffic localization on energy dissipation in noc-based interconnect. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 1774–1777. IEEE, 2005.

[78] Hyung Gyu Lee, Naehyuck Chang, Umit Y Ogras, and Radu Marculescu. On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3):23, 2007.

[79] Praveen Bhojwani and Rabi Mahapatra. Interfacing cores with on-chip packet-switched networks. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 382–387. IEEE, 2003.

[80] Srinivasan Murali and Giovanni De Micheli. Bandwidth-constrained mapping of cores onto noc architectures. In *Proceedings of the conference on Design, automation and test in Europe-Volume 2*, page 20896. IEEE Computer Society, 2004.

[81] Jae Young Hur. *Customizing and hardwiring on-chip interconnects in FPGAs*. PhD dissertation, TU Delft, 2011.

[82] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *INTEGRATION, the VLSI journal*, 38(1):69–93, 2004.

[83] Aline Mello, Leonel Tedesco, Ney Calazans, and Fernando Moraes. Virtual channels in networks on chip: implementation and evaluation on hermes noc. In *Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 178–183. ACM, 2005.

[84] Sungchan Kim and Soonhoi Ha. Efficient exploration of bus-based system-on-chip architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(7):681–692, 2006.

[85] Kanishka Lahiri, Anand Raghunathan, and Sujit Dey. Design space exploration for optimizing on-chip communication architectures. *Computer-Aided*

Design of Integrated Circuits and Systems, IEEE Transactions on, 23(6): 952–961, 2004.

[86] Junhee Yoo, Dongwook Lee, Sungjoo Yoo, and Kiyoung Choi. Communication architecture synthesis of cascaded bus matrix. In *Design Automation Conference, 2007. ASP-DAC'07. Asia and South Pacific*, pages 171–177. IEEE, 2007.

[87] Junhee Yoo, Sungjoo Yoo, and Kiyoung Choi. Topology/floorplan/pipeline co-design of cascaded crossbar bus. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(8):1034–1047, 2009.

[88] Minje Jun, Sungjoo Yoo, and Eui-Young Chung. Topology synthesis of cascaded crossbar switches. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(6):926–930, 2009.

[89] Minje Jun, Deumji Woo, and Eui-Young Chung. Partial connection-aware topology synthesis for on-chip cascaded crossbar network. *Computers, IEEE Transactions on*, 61(1):73–86, 2012.

[90] William J Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684–689. IEEE, 2001.

[91] Luca Benini and Giovanni De Micheli. Networks on chips: A new soc paradigm. *Computer*, 35(1):70–78, 2002.

[92] John D Owens, William J Dally, Ron Ho, DN Jayasimha, Stephen W Keckler, and Li-Shiuan Peh. Research challenges for on-chip interconnection networks. *IEEE micro*, 27(5):96, 2007.

[93] Radu Marculescu, Umit Y Ogras, Li-Shiuan Peh, Natalie Enright Jerger, and Yatin Hoskote. Outstanding research problems in noc design: system, microarchitecture, and circuit perspectives. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(1):3–21, 2009.

[94] Jingcao Hu and Radu Marculescu. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In

Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings, volume 1, pages 234–239. IEEE, 2004.

[95] Srinivasan Murali, Paolo Meloni, Federico Angiolini, David Atienza, Salvatore Carta, Luca Benini, Giovanni De Micheli, and Luigi Raffo. Designing application-specific networks on chips with floorplan information. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 355–362. ACM, 2006.

[96] Andreas Hansson, Kees Goossens, and Andrei Rădulescu. A unified approach to mapping and routing on a network-on-chip for both best-effort and guaranteed service traffic. *VLSI design*, 2007, 2007.

[97] Vladimir Todorov, Daniel Mueller-Gritschneder, Helmut Reinig, and Ulf Schlichtmann. A spectral clustering approach to application-specific network-on-chip synthesis. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 1783–1788. IEEE, 2013.

[98] Sudeep Pasricha, Nikil Dutt, and Mohamed Ben-Romdhane. Constraint-driven bus matrix synthesis for mpsoc. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 30–35. IEEE Press, 2006.

[99] Srinivasan Murali, Luca Benini, and Giovanni De Micheli. An application-specific design methodology for on-chip crossbar generation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(7):1283–1296, 2007.

[100] Vesa Lahtinen, Erno Salminen, Kimmo Kuusilinna, and T Hamalainen. Comparison of synthesized bus and crossbar interconnection architectures. In *Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on*, volume 5, pages V–433. IEEE, 2003.

[101] Alessandro Cilardo, Edoardo Fusella, Luca Gallo, and Antonino Mazzeo. Automated synthesis of fpga-based heterogeneous interconnect topologies. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8. IEEE, 2013.

[102] A Cilardo, E Fusella, L Gallo, A Mazzeo, and N Mazzocca. Automated design space exploration for fpga-based heterogeneous interconnects. *Design Automation for Embedded Systems*, pages 1–14, 2014.

[103] Neal K Bambha and Shuvra S Bhattacharyya. Interconnect synthesis for systems on chip. In *System-on-Chip for Real-Time Applications, 2004. Proceedings. 4th IEEE International Workshop on*, pages 263–268. IEEE, 2004.

[104] Neal K. Bambha and Shuvra S. Bhattacharyya. Joint application mapping/interconnect synthesis techniques for embedded chip-scale multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):99–112, 2005.

[105] OpenMP Architecture Review Board. OpenMP application program interface, v3.1, 2011. URL www.openmp.org.

[106] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2002.805829.

[107] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java (volume 3: Ptolemy ii domains). Technical Report UCB/EECS-2008-37, University of California at Berkeley, April 2008. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-37.html.

[108] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal models for embedded system design. *IEEE Design & Test archive*, 17(2):14–27, April 2000.

[109] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 1974*, pages 471–475. North-Holland Publishing Co., 1974.

[110] Amir Hossein Ghamarian. *Timing Analysis of Synchronous Data Flow Graphs*. PhD thesis, Technische Universiteit Eindhoven, July 2008.

[111] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, April 2003.

[112] R. Niemann and P. Marwedel. An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems*, 2(2):165–193, 1997.

[113] V.V. Das. *Compiler Design Using FLEX and YACC*. Prentice-Hall of India Pvt.Ltd, 2007. ISBN 8120332512.

[114] Java native access (jna) v. 3.5.1. Technical report, 2012. URL https://github.com/twall/jna#readme.

[115] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the international symposium on Code generation and optimization (CGO'04)*, page 75, 2004.

[116] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. URL http://www.R-project.org. ISBN 3-900051-07-0.

[117] Xilinx. Platform studio and the embedded development kit (EDK), 2012. URL http://www.xilinx.com/tools/platform.htm.

[118] Joint Photographic Experts Group. Recommendation t.81. Technical report, International Communication Union, September 1992. URL http://www.itu.int/rec/T-REC-T.81-199209-I/en.

[119] Introduction to `lp_solve` 5.5.2.0. Technical report, 2012. URL http://lpsolve.sourceforge.net/5.5.

[120] EPCC. EPCC OpenMP benchmarks. 2012. URL http://www.epcc.ed.ac.uk/software-products/epcc-openmp-benchmarks/.

[121] H. Nikolov, M. Thompson, T. Stefanov, A. D. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. F. Deprettere. Daedalus: Toward composable multimedia MP-SoC design. In *Proceedings of Design Automation Conference (DAC)*, pages 574–579, June 2008.

[122] J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith. SystemCoDesigner - an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Transactions Design Automation of Electronic Systems*, 14(1):1–23, January 2009.

[123] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, J. Riihimäki, and K. Kuusilinna. UML-based multi-processor SoC design framework. *ACM Transactions on Embedded Computer Systems*, 5(2):281–320, May 2006.

[124] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo. PeaCE: A hardware-software codesign environment of multimedia embedded systems. *ACM Transactions Design Automation of Electronic Systems*, 12(3), August 2007.

[125] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System design using Kahn Process Networks: The Compaan/Laura approach. In *Proceedings of Design Automation and Test in Europe Conference (DATE) 2004*, pages 340–345, 2004.

[126] Mentor Graphics. Handel-C synthesis methodology, August 2012. URL http://www.mentor.com/products/fpga/handel-c/.

[127] P. Dziurzanski and V. Beletskyy. Defining synthesizable OpenMP directives and clauses. In *Proceedings of the 4th International Conference on Computational Science - ICCS 2004*, volume 3038, pages 398–407. Springer, 2004.

[128] P. Dziurzanski, W. Bielecki, K. Trifunovic, and M. Kleszczonek. A system for transforming an ANSI C code with OpenMP directives into a SystemC description. In *Design and Diagnostics of Electronic Circuits and Systems, 2006*, pages 151–152. IEEE, apr 2006.

[129] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, and D. Jimenez-Gonzalez. OpenMP extensions for FPGA accelerators. In *International Symposium on Systems, Architectures, Modeling, and Simulation, 2009 - SAMOS '09*, pages 17–24, July 2009.

[130] Yuan Xie. Future memory and interconnect technologies. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 964–969, 2013. URL http://dl.acm.org/citation.cfm?id=2485520.