# HIGH-LEVEL PARTITIONING OF DISCRETE SIGNAL TRANSFORMS FOR DISTRIBUTED HARDWARE ARCHITECTURES

By

Rafael A. Arce Nazario

A thesis submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY
in

COMPUTING AND INFORMATION SCIENCE AND ENGINEERING

UNIVERSITY OF PUERTO RICO
MAYAGÜEZ CAMPUS

June, 2007

Approved by:

| | |
|---|---|
| Domingo Rodríguez, Ph.D<br>Member, Graduate Committee | Date |
| Dorothy Bollman, Ph.D<br>Member, Graduate Committee | Date |
| Rogelio Palomera, Ph.D<br>Member, Graduate Committee | Date |
| Isidoro Couvertier, Ph.D<br>Member, Graduate Committee | Date |
| Manuel Jiménez, Ph.D<br>President, Graduate Committee | Date |
| Nazario Ramírez, Ph.D<br>Representative of Graduate Studies | Date |
| Nestor Rodríguez, Ph.D<br>Chairperson of the Department | Date |

Abstract of Dissertation Presented to the Graduate School
of the University of Puerto Rico in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

**HIGH-LEVEL PARTITIONING OF DISCRETE SIGNAL TRANSFORMS FOR DISTRIBUTED HARDWARE ARCHITECTURES**

By

Rafael A. Arce Nazario

June 2007

Chair: Manuel Jimenez, Ph.D
Major Department: Computing and Information Science and Engineering

Discrete signal transforms (DSTs) have numerous applications in a wide spectrum of scientific fields. To attain superior performance, the size and composition of these algorithms frequently require implementation to architectures involving more than one dedicated hardware device. Even though hardware implementations of signal processing algorithms are known to be orders of magnitude faster than most other generic computing platforms, they are not commonplace mainly because of the increased complexity involved in partitioning and mapping such algorithms onto distributed hardware platforms. Automated methods and tools to aid in the design and exploration of distributed implementations shall encourage adoption of dedicated hardware platforms for high-performance applications.

Traditionally, partitioning to distributed hardware architectures (DHAs) has been done either manually, or at various stages of the design process, predominantly at the behavioral and structural levels. Although these schemes have produced acceptable implementations, they do not necessarily exploit the functional properties of algorithms. Structural level techniques handle the design at an abstraction level

too low for the algorithm functionality to be effectively interpreted. Most proposed higher-level strategies target generic partitioning problems through local optimization techniques, which miss out on alternate formulations that become apparent only on a higher level of abstraction.

This dissertation presents an automated methodology specifically designed for partitioning DSTs onto DHAs. The methodology takes advantage of DST features at two levels of abstraction: the graph and algorithmic levels. At the algorithmic level, an exploration is conducted in search of equivalent transform formulations that are more suitable for the target topology. At the graph level, a series of DST-specific structural considerations are made to improve the partitioning heuristic. The developed strategy integrates several algorithms which allow exploring partitioning solutions at both abstraction levels.

A Kronecker products algebra (KPA) to dataflow graph conversion (DFG) tool was developed to allow straightforward conversion and structural visualization of KPA expressed formulations. The DFG generated by this tool is partitioned using a $k$-way deterministic algorithm with structural considerations derived from common DST features. A scheduler estimates latency of the partition solution, constrained by the available computational resources, as determined by an area estimator. Solution cost at the graph level is used to transform the current DST formulation, and thus explore alternative formulations as part of the partition optimization process. Given the exponential size of the space of equivalent formulations, a greedy, polynomial-time exploration heuristic was designed.

Results from the application of this methodology to a range of sizes of Discrete Fourier Transforms and Discrete Cosine Transforms evidence the advantages of making the partition methodology DST-aware. Latency and run-time reductions of up to 34% and 99%, respectively were obtained with respect to previously proposed, stochastic high-level partitioning approaches.

Resumen de Disertación Presentado a Escuela Graduada
de la Universidad de Puerto Rico como requisito parcial de los
Requerimientos para el grado de Doctor en Filosofía

# PARTICIONAMIENTO A ALTO NIVEL DE TRANSFORMADAS DISCRETAS DE SEÑALES A ARQUITECTURAS DE HARDWARE DISTRIBUIDO

Por

Rafael A. Arce Nazario

Junio 2007

Consejero: Manuel Jimenez, Ph.D
Departamento: Ciencias e Ingeniería de la Computación y la Información

Las transformadas de señales discretas (TSDs) tienen numerosas aplicaciones en una ámplia gama de campos científicos. Para lograr obtener un mayor rendimiento, frecuentemente se requiere la implementación de dichas transformadas en arquitecturas de múltiples dispositivos. Las implementaciones de estos algoritmos en hardware suelen obtener rendimientos significativamente mejores que otras plataformas de procesamiento general. Sin embargo, dichas implementaciones no abundan, principalmente por la complejidad necesaria para particionar estos algoritmos a plataformas de hardware distribuido y determinar las estructuras correspondientes en hardware. El desarrollo de métodos y herramientas que ayuden en el diseño y exploración de estas implementaciones facilitará la adopción de estas plataformas para aplicaciones de alto rendimiento.

Tradicionalmente, el particionamiento a arquitecturas de hardware distribuido (AHD) se ha realizado de forma manual o como parte de alguna de las etapas del diseño automatizado, tanto a alto nivel como a nivel estructural. A pesar de que dichas técnicas de particionamiento han producido resultados aceptables, en la

mayoría de los casos estas no aprovechan las propiedades de estas transformadas a nivel algorítmico. Las técnicas de particionamiento a nivel estructural trabajan con una representación a un nivel de abstracción demasiado bajo como para aprovechar cualquier aspecto funcional del algoritmo. Por otro lado, las técnicas de alto nivel han sido diseñadas, en su mayoría, para casos genéricos. Esto las limita a utilizar técnicas genéricas de optimización local y por lo tanto, obvian formulaciones alternas que solo se pueden visualizar a un nivel de abstracción más alto.

Esta disertación presenta una metodología automatizada específicamente diseñada para particionar TSDs a AHDs. La metodología aprovecha características de las TSDs tanto a nivel gráfico como a nivel algorítmico. En el nivel algorítmico, se explora el espacio de formulaciones equivalentes en busca formulaciones que resulten más apropiadas para la arquitectura. A nivel gráfico, se introdujeron una serie de consideraciones a los heurísticos de particiónamiento que atienden particularidades estructurales de las TSDs. La estrategia propuesta integra varios algoritmos que permiten la exploración de soluciones en ambos niveles de abstracción. Se desarrolló una herramienta para la conversión de expresiones en álgebra de productos Kronecker (APK) a grafos de flujo de data (GFD). Esta herramienta permite conversión rápida de formulaciones de TSDs, así como la visualización de sus estructuras computacionales. El GFD traducido es particionado usando un algoritmo determinístico de particionamiento en $k$-partes, al que se le han añadido consideraciones estructurales de las TSDs. Un planificador ("scheduler") estima la latencia de la solución de partición, restringido por los recursos computacionales disponibles, los cuales son determinados por un estimador de área. El costo de la solución a nivel gráfico se usa para transformar la formulación algorítmica actual y así explorar otras alternativas como parte del proceso de optimización. Debido al tamaño exponencial del espacio de formulaciones equivalentes, se diseñó un heurístico de exploración ambicioso ("greedy") que tiene complejidad de tiempo polinomial.

Se utilizó la estrategia propuesta para obtener resultados del particionamiento para un rango de tamaños de transformadas discretas de Fourier y transformadas discretas de coseno. Estos evidencian las ventajas que consigue nuestra estrategia gracias a ser consciente de las características de las TSDs. Se obtuvieron reducciones en latencia y tiempo de ejecución de 34% y 99%, respectivamente, en comparación con técnicas de alto nivel anteriormente propuestas, que utilizaban estrategias genéricas y estocásticas.

This work is dedicated to Maritza, my wife, for her love and understanding, for believing in me, and helping me keep my sanity. Also, to my parents Marirosa and Rafael Angel, for instilling in me the joy of learning and the drive to excel academically. To all of them for being my role models of dedication and excellence.

ACKNOWLEDGMENTS

I would like to thank my doctoral advisor Dr. Manuel Jiménez for his dedication, suggestions and guidance in the present investigation. On many occasions, it was his vision and enthusiasm that helped keep this endeavor alive. I also appreciate the intellectual support and savvy advise from Domingo Rodriguez, and the valuable suggestions from rest of my graduate committee members: Dorothy Bollman, Isidoro Couvertier, and Rogelio Palomera. My appreciation to the CISE administrative staff for their help.

I would also like to thank the organizations that funded my research work: UPR-Humacao, GEM consortium, Resource Center for Sciences and Engineering, WALSAIP project, and CISE program.

Many thanks to Carmen Irizarry for providing a welcoming home during my Mayagüez stays. My appreciation to Kathy and Jerry Takaks for helping me start off this Ph. D. venture on the right foot by being such wonderful hosts in Rochester, NY.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

## LIST OF ABBREVIATIONS

| | |
|---|---|
| CAD | Computer Aided Design for VLSI Circuits |
| DFG | Dataflow Graph |
| DHA | Distributed Hardware Architecture |
| DST | Discrete Cosine Transform |
| DST | Discrete Signal Transform |
| EDA | Electronic Design Automation |
| FCCM | Field-Programmable Custom Computing Machines |
| FPGA | Field Programmable Gate Array |
| FFT | Fast Fourier Transform |
| GPP | General Purpose Processor |
| HLS | High-Level Synthesis |
| KL | Kernighan-Lin |
| KPA | Kronecker Product Algebra |
| PDSP | Programmable Digital Signal Processors |
| RTL | Register Transfer Level |
| SA | Simulated Annealing |
| VHDL | VHSIC Hardware Description Language |
| VLSI | Very Large Scale Integration |

# CHAPTER 1

# Introduction

Applications for discrete signal transforms (DSTs), such as the Discrete Fourier Transform (DFT) and the Discrete Cosine Transform (DCT), abound in fields as diverse as communications, biomedical sciences, and astronomy. In these and many other fields, increases in the quantity and resolution of data and the need for faster processing, demand novel platforms and methodologies for the implementation of DSTs.

DSTs have been implemented to a myriad of computational platforms, from software-based General Purpose Processor systems to pure customized hardware such as Application Specific Integrated Circuits (ASICs). Throughout time, a cyclic behavior between software and hardware implementations has been observed [1] [2]. In the majority of cases, DST applications are initially implemented in software. Then, the need for real-time computation encourages their implementation to application-specific hardware. Meanwhile, advances in technology make the software implementation again competent. However, as this happens, applications evolve to demand more aggressive performance requirements, causing the cycle to repeat itself.

This observed trend highlights a major distinction between the software and hardware DST implementations: hardware implementations tend to achieve higher performance than software ones, but with a significant increase in the implementation effort. For example, it is a consistently observed fact that implementations of

DSP applications in FPGAs can achieve 10x performance over their counterparts on Programmable Digital Signal Processors (PDSPs) [3]. Similarly, it is estimated that implementation effort for an FPGA is 5x-10x that of a PDSP. In today's world, where time to market can make the difference between a commercially successful product and a failure, it is common to see how implementers shy away from hardware implementations due to their steep development costs in time, cost, and hardware design expertise. To make these dedicated hardware implementations more widespread, we need, among other considerations, to improve the methods and tools used to design them.

Efficient mapping of algorithms to hardware is a challenging task. It demands a design philosophy somewhat dissimilar to programming a software-based system and imposes a number of additional difficulties. For instance, when implementing the functionality of an algorithm to hardware, in addition to deciding the type and order of operations, the designer must determine what functional units will be implemented in the device. This expands the range of possibilities available for the implementation of each algorithm, thus widening the solution space and the number of design options to be considered.

To aggravate the case against dedicated hardware implementations, some applications may require mapping to multiple devices (FPGAs or ASICs) to attain the degree of parallelism needed to achieve acceptable performance [4][5]. Today's developers not only have to think about the efficient hardware structures to implement their algorithms, but must also partition their designs into pieces that fit to each of the devices, and make effective use of the communications and memory resources provided in a multi-chip board. Clearly, to reduce the effort required for hardware implementations in multiple device architectures, the development of automatic algorithms for mapping and partitioning is essential. Correspondingly, there

is a recently revived interest in dedicated hardware implementations using multiple intra-chip processing units, as in multi-core and System on Chip architectures, which can also benefit from effective partitioning schemes [6].

Several strategies have been proposed for automating the partitioning of algorithms to DHAs [7][8][9]. However most of the proposed automated mapping and partitioning methodologies are limited in at least one of two main aspects when used for DSTs. First, they perform optimizations at an abstraction level where it is difficult to detect functional features that could result in implementations with a higher performance. Second, they target general applications, so they only incorporate strategies that will work appropriately for the general case. Consequently, results from these strategies are often merely adequate in quality and require unnecessarily lengthened processes to converge to a solution.

DSTs represent a major energy, performance, and resource component in many modern applications, which merits the study of specialized methods for optimizing their implementation to modern computing platforms. The studies in this dissertation consider the development of a methodology for automated partitioning of discrete signal transforms to multiple-chip architectures. By targeting a specific group of algorithms we devised a partitioning methodology that exploits their functional features to attain better implementations in a more straightforward manner. The decision to focus on a certain class of algorithms allowed the incorporation of global algorithmic reformulations into the methodology, something that is not contemplated in existing methodologies.

## 1.1 Objectives and Scope of Research

One of the main objectives of this research is to make a contribution to the software tools and general philosophy that designers encountering a partitioning task can have at their service. This was accomplished through the development of a partitioning methodology specifically designed for discrete signal transforms

onto distributed hardware architectures. The developed methodology uses features and mathematical characteristics of discrete signal transforms, which would be lost or would incur in excessive effort if considered at lower abstraction levels. It encompasses several tasks that work together to accept DST algorithms, optimization objectives, constraints, and architectural specifications, and explores the solution space in search for an optimized implementation. Development of these tasks required the study, adaptation and creation of heuristic techniques to take advantage of improvement opportunities at the DST graph and formulation levels.

As a proof of concept to our methodology, its various stages were integrated into an automated tool. Individual components of the methodology were used to conduct experiments which, in turn, helped in defining the rest of the components. The integrated tool was used to partition DSTs to demonstrate its effectiveness and to compare its solutions with those of other published results. Outcomes from these evaluations confirmed the advantages of the considerations that were taken throughout the methodology's development. They validate our hypothesis that faster/higher-quality partitioning results can be obtained with a strategy that is aware of DST features.

## 1.2 Dissertation Overview

The rest of the document is organized as follows. Chapter 2 provides a review of published work related to our topic, serving to sustain the importance of the problem, analyze what other researchers have accomplished, and to survey some areas of knowledge which influenced our solution. Chapter 3 formally states the problem of high-level partitioning of DSTs to DHAs and presents the main justification for our approach. This is followed by an overview of the approach followed in finding a solution to the stated problem.

Our partitioning methodology takes advantage of dataflow graph-level constructs while exploring alternative algorithmic DST formulations. Chapter 4 presents the

development of processes to support graph-level exploration of a given DST formulation as well as the rationale behind their selection. This includes a Kronecker products algebra to DFG conversion tool, an extension of the Kernighan-Lin bipartitioning heuristic to $k$-way, and latency and hardware estimation strategies. Chapter 5 discusses how those tools were used to experiment with DST formulations and assess the effect of algorithmic level transformations. Observations from these experiments were used in the design of a formulation space exploration heuristic. The effectiveness of the various parts of our approach, as well as the unified methodology is validated in Chapter 6. Also in this chapter, results of the methodology are compared against those from a previous methodology. Chapter 7 reviews the main findings of this dissertation, highlighting the contributions of this work, and list several possible future research directions. The last chapter considers some of the ethical issues related to engineering research, in general, and to electronic design automation, in particular.

# CHAPTER 2

# Related Work

Our intended approach to the problem of partitioning of discrete signal transforms onto distributed hardware architectures requires combination of knowledge and techniques from at least three areas: electronic design automation, distributed hardware architectures, and digital signal processing. These three areas have been widely explored independently. Part of our proposed solution's novelty lies in finding how these areas of knowledge can complement each other to produce efficient hardware solutions for DST algorithms. This chapter reviews relevant technical literature to these areas, emphasizing the issues and techniques most closely related to the contributions of this dissertation.

The first section of this chapter provides a background on electronic design automation to establish the context in which automated partitioning methodologies are used. The second section reviews distributed hardware architectures and introduces the model we shall target throughout the development of our methodology. Section 2.3.2 discusses partitioning strategies for DHAs, with an emphasis on techniques at the behavioral level. The discussion highlights areas of opportunity with respect to our chosen problem and offers a glimpse into our proposed solution. Subsequently, in Section 2.4 we explore previous hardware implementations of a particular but representative signal transform, the discrete Fourier transform. Code generation methods that explore alternative DST formulations as part of their optimization

strategy are reviewed in Section 2.5. Our main intention in these later sections is to project how previously developed hardware structures and algorithm techniques influenced our solution to the problem of automated partitioning of discrete signal transforms for distributed hardware architectures. Finally, Section 2.6 summarizes the main findings of our review of previous work.

## 2.1 Electronic Design Automation

Designers looking for improved speed performance have always sought effective hardware implementations of digital signal processing applications [2]. It is commonly known that hardware implementations of these types of algorithms are faster than their software counterparts by orders of magnitude [3]. However, high-performance is achieved at an increased cost in technology price and implementation effort. Despite this, organizations seriously looking for increased performance are willing to pay the necessary price for technology. The biggest obstacle faced by designers in the process of successfully and efficiently mapping algorithms to hardware is how to handle the design complexity of these implementations. Typically, a team of designers is needed to build such project, consisting of experts in signal processing, algorithm design, and digital electronics. The required digital design expertise deters many signal processing enthusiasts and scientists from using hardware technology to test their algorithms, and achieving the promised high-performance. Clearly, an automated approach for mapping algorithms to hardware could open the opportunity of hardware implementation to the non-electronically inclined crowd and significantly ease solution exploration/optimization for digital designers.

The field of electronic design automation (EDA) has evolved to assist designers throughout the various tasks involved in the design and implementation of integrated circuits. From the highest to the lowest level of abstraction, elaborate tools have been developed to assist designers in the exploration and evaluation of the solution space to increase the odds of arriving at improved implementations. Figure

2–1 illustrates the traditional computer aided design flow for VLSI circuits (CAD) intended for a multi-device architecture. The process of transforming a digital system from a system specification into a hardware implementation is divided into four major steps: system-level synthesis, high-level synthesis, logic synthesis, and physical design, which are briefly discussed next.

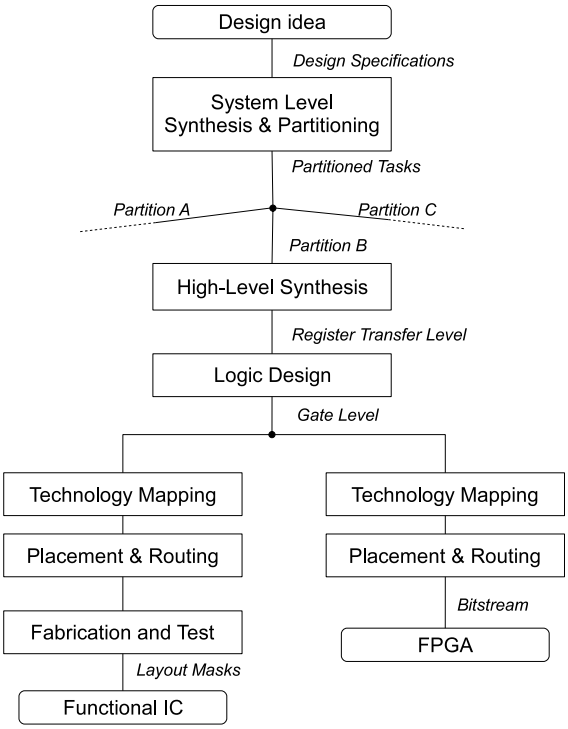



Figure 2–1: Flow for Computer Aided Design for VLSI.

At the highest level of abstraction of the EDA process, a functional system description is provided using a high-level language. C, MATLAB and hardware description languages, such as VHDL and Verilog, have all been used for this purpose [10][11][12]. A process commonly referred to as *system-level synthesis* accepts this specification and outputs a partitioned set of tasks. Each partition is handled separately throughout the rest of the design flow and ultimately implemented on a particular target architecture device.

To accomplish the partitioning objectives, a system-level synthesizer starts by converting the human-specified algorithm formulation into an intermediate graph-like format that captures the control and data flow of the algorithm's operations. Graph representation schemes with various levels of granularity have been used for such representation. Among them, data flow graphs (DFGs) [8], control-data flow graphs (CDFGs) [13] and call graphs [14] are worth mentioning. The graph representation is then partitioned using implementation objectives such as performance, area, or power utilization.

*High-level synthesis* (HLS) transforms a behavioral description of the tasks that have been assigned to each device into a technology-independent structural description, such as a register transfer level (RTL) netlist. To accomplish this, HLS also represents the behavioral description in a graph-like format. This representation is then taken through a series of optimizing subtasks, namely allocation and scheduling [13][15]. During allocation, the hardware structures that will implement the systems' functionality (e.g. functional units) are determined. The aim of scheduling is to assign each of the algorithm's operations to a given time-step using the allocated units.

The last stages of the EDA flow convert the RTL-specified circuit onto a gate or transistor level netlist and determine the circuit layout description for the final implementation. In *logic synthesis* the RTL netlist is expanded into a gate-level circuit and optimized for a variety of objectives, such as area and delay. *Technology mapping* maps the gate-level circuit into the logic structures available in the targeted device. *Placement and routing* assign each of the technology-mapped circuit components a physical location on the target device and establish how circuit connections are physically routed through the device area. In the case of FPGAs, these steps are followed by the generation of a bitstream, which is used to program the reconfigurable elements in the device. For ASICS, the geometrical shapes that

determine the various gates and connections are transcribed onto layout masks for fabrication.

Several computer aided design flows have been proposed to specifically target distributed hardware architectures [7][16][17]. All of them focus their attention on the partitioning step, either at the threshold between the system and high-level stages of the EDA process, as illustrated in Figure 2–1, or later after an RTL or logic design has been obtained. These strategies are oriented toward general algorithms and have not considered functional characteristics of signal transforms that we believe can conduct to a more straightforward exploration of the solution space. In the next sections we discuss distributed hardware architectures such as the ones we developed as part of this work. We also explain how the process of design automation, particularly at the system and high-level stages, is conducted for our architectures of interest.

## 2.2 Distributed Hardware Architectures

DSP algorithms have been implemented to multiple computing platforms, seeking to take advantage of their cost/performance features. Our research focuses on architectures consisting exclusively of homogenous dedicated hardware, e.g. multi-FPGA boards. Throughout the rest of this document the term *distributed hardware architectures* will be used to refer to computational platforms consisting of multiple homogenous dedicated hardware devices, along with memory resources and some type of communication between the devices and memory.

Documentation of distributed hardware architectures using ASICs is scarce as this type of systems is commonly proprietary. However, a number of interesting variations of multiple FPGA systems have been constructed and documented since the early 1990s. The reconfigurable nature of these systems makes their documentation important for designers who are interested in implementing their own algorithms to these architectures. Even though we foresee that we will be targeting most of our

implementation efforts to multiple-FPGA architectures, our proposed methodology for high-level partitioning is flexible enough to be useful for other architectures consisting of multiple dedicated hardware devices, such as traditional cell-based ASICs, and newer technologies like structured ASICs [18].

One of the first multiple-FPGA (MFPGA) systems to be documented was SPLASH [19]. It consisted of 32 Xilinx 3090 FPGAs with their corresponding 32 memory chips, connected in a linear array topology. A second version, SPLASH II improved on SPLASH by using higher capacity and faster FPGAs and providing a crossbar for one-hop data transfer for any two FPGAs as well as broadcast functionality [20]. Numerous DSP applications were mapped to SPLASH-2 with improvements in performance as compared to other implementation options [21][22][23]. Since then, a significant number of MFPGA systems have been constructed, both in Academia and Industry, each differing from the other in the number and types of FPGAs, the connection topology and memory capacity and arrangement [24][25]. Compton, et al., presented a thorough review of reconfigurable computer systems and issues, highlighting the fact that currently no consensus as to what combination of system characteristics, especially connection topology, are optimal for the general implementation of algorithms [26]. In spite of this, developers of these systems always seem to find a specific niche of benchmarks or metrics that make their systems superior to previous architectures.

The continued design of MFPGA systems for high performance signal processing applications evidences their acceptance as a preferred solution for low-volume or unique applications with a reasonable development time cycle. Two recent systems attest to this notion. The Serendip IV spectrometer, used as part of the Search for Extraterrestrial Intelligence (SETI) project, consists of 120 Xilinx FPGAs on 40 spectrum analyzer boards working in parallel to scan 168 million narrow-band (0.6 Hz) channels every 1.7 seconds [4]. Each SERENDIP IV board computes a

four million point FFT by breaking the computation into three smaller FFTs (128, 128, and 256 points each). Each of the smaller transforms is implemented in a single FPGA. SERENDIP V, the next-generation of spectrometer boards, is being prepared for deployment. The majority of signal processing in this board is performed by a Xilinx XC2V6000 FPGA which can sustain a 64M-point FFT in real-time.

The Berkeley Emulation Engine (BEE) is an integrated rapid prototyping and design environment for communications and digital signal processing (DSP) systems, consisting of four multi-FPGA based processing units, each consisting of 4 processing and 1 interconnection FPGA [5]. Besides neighboring inter-FPGA connections, a novel hierarchical crossbar interconnection is used to support the communications among devices and boards. BEE2, a new version of BEE, is being justified by its designers not just as a hardware emulation platform but as a cost effective option to high performance computing [27].

Figure 2–2 illustrates the general target architecture model of a DHA that will be used throughout the rest of our discussion. It consists of $k$ dedicated hardware devices with local memory, connected in a ring or linear array topology with a crossbar serving as a global communication channel. This architecture is modeled after common multi-FPGA boards produced by vendors such as Annapolis (Wildforce) and Gidel (PROC20KE), as well as high-end academic reconfigurable systems such as the Berkeley Emulation Engine 2 (BEE2) [28]. Furthermore, this architecture can be considered scalable due to the number of connections per device and its topological symmetry.

Regardless of the increasing logic capacities foreseen for next generation FPGAs and ASICs, there will always exist applications where distributed hardware systems will be either the only or the most cost/effective implementation option. Furthermore, using current manufacturing standards, single device yield diminishes dramatically as density and/or dye size are increased [29]. Power management and

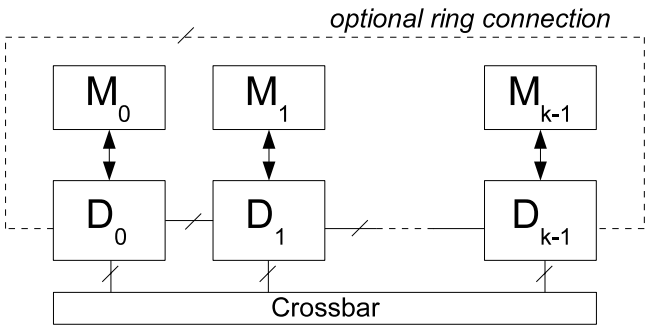



Figure 2–2: General architectural model for a distributed hardware architecture.

heat dissipation also become prominent with increased density [30]. As illustrated in Figure 2–3 for Xilinx Virtex 2 FPGAs, these factors contribute to a cost/density ratio that increases with density. For these reasons, the development of novel methods for partitioning algorithms to multiple device systems will continue to be of great importance. In the next section we discuss partitioning/mapping strategies that have been proposed specifically targeting homogeneous distributed hardware architectures.

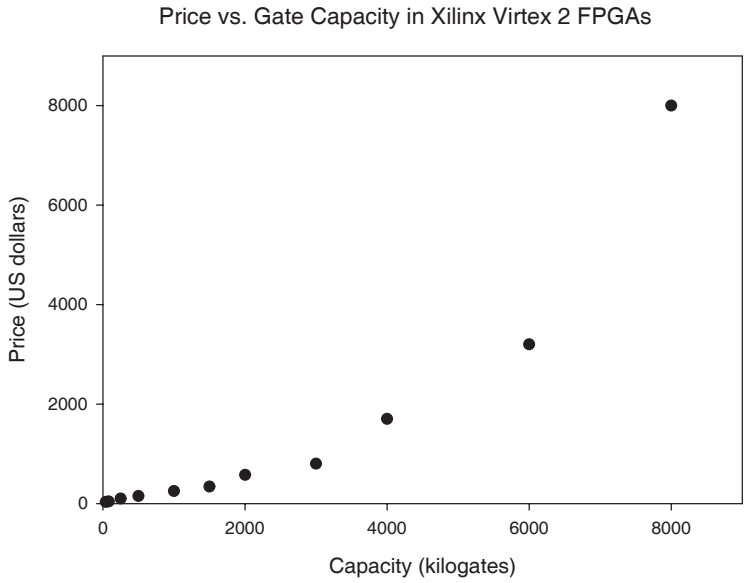



Figure 2–3: Price vs. capacity for Xilinx Virtex 2 FPGAs. Source: www.digikey.com

## 2.3 Partitioning to DHAs

We begin our discussion of DHA partitioning methodologies with some general concepts about graph partitioning algorithms that are needed to understand the basic differences between the reviewed approaches. For purposes of discussion, specific graph partitioning heuristics, such as Kernighan-Lihn's and Fiduccia Matheysses, are discussed in Chapter 4, where they are analyzed as part of our design of a DST-influenced graph partitioning strategy.

### 2.3.1 Graph partitioning algorithms

Partitioning algorithms can be classified as constructive or iterative. Constructive algorithms utilize clustering techniques to arrive at partitioning solutions. Each node initially belongs to its own cluster, and clusters are then gradually merged until a desirable partitioning is found. Thus, constructive algorithms follow a bottom-up greedy approach to partitioning.

Iterative algorithms, on the other hand, follow a top-down approach. They begin with an initial partition solution, which can be obtained in a naïve manner or from other partitioning strategies. At each iteration, the current solution is modified to obtain candidate solutions which are evaluated to determine their *goodness* according to the partitioning objectives. The *goodness* of a solution is quantified by combining the implementation objectives on an *objective function*[1] . The best candidate solution (or a randomly chosen one in the case of probabilistic algorithms) becomes the new current solution for the next iteration step.

Two main factors distinguish the various system-level iterative strategies that have been published in literature: the composition and evaluation of the objective

[1] For example, the objective function of a partitioning process aimed at minimizing area and power utilization of an implementation could be as follows: $F = k_1 \sum_i Area_i + k_2 \sum_i Power_i$ where $i$ are the partitions, and $k_1$ and $k_2$ are weight coefficients (sums are assumed to be normalized).

function, and the mechanism used to guide the exploration of the solution space. During system-level partitioning, many of the desired objectives are related to lower-level implementation issues such as logic area, latency, and power utilization. The estimation of these properties from the high abstraction levels is not trivial, which explains why the development of fast and accurate estimators for those properties has been the topic of considerable research effort [7][13][31].

The graph min-cut bisection problem was demonstrated to be NP-complete by Garey [32]. All other size-constrained formulations of the graph partitioning problem are NP-complete too, as they reduce to the min-cut bisection problem [33]. Therefore, practical partitioning strategies rely on heuristics rather than on exact algorithms. Even though heuristic decisions cannot guarantee a good (or even a feasible) solution every time and under every situation, they represent a relatively fast and effective option for finding near-optimal solutions. Heuristic partitioning mechanism explore the solution space by using either deterministic or stochastic strategies. Deterministic techniques explore the solution space by applying rules that have been observed to lead to good solutions in practice. Two widely used heuristic partitioning techniques that employ a deterministic approach are Kernighan-Lin and Fiduccia-Matheyses. They are both considered greedy heuristics, accepting at each iterative step the best candidate solution. This makes them susceptible to local minimum solutions.

In an effort to avoid local minimum solutions, stochastic algorithms randomly select the solution for the next iteration, as is done in simulated annealing (SA), or build a new *generation* of solutions by crossing and mutating previous solutions, as in genetic algorithms (GA). Given the correct coding of the partitioning problem onto the probabilistic algorithm, a wise selection of solution exploration parameters (e.g. mutation probability in GA, cooling schedule in simulated annealing) and

sufficient iterations, probabilistic algorithms can be relied on to find near-optimal partitioning solutions.

### 2.3.2 Partitioning for Distributed Hardware Architectures

Previous strategies for partitioning algorithms onto homogeneous distributed hardware architectures can be classified into two categories, according to the general abstraction level at which partitioning is performed: partitioning at the RTL-or-below-level, commonly referred to as *structural* partitioning, or partitioning at the behavioral-level, commonly known as *high-level* partitioning.

### 2.3.3 Structural partitioning

Multi-FPGA logic emulation systems served as motivation for many of the reported structural partitioning methods. These platforms are used to verify the overall functionality of a logic implementation before committing it to a silicon solution, when software simulation techniques are impractical. Since silicon admits much denser structures than FPGAs, an array of these reconfigurable devices is used to implement the circuit under validation, thus requiring a mapping strategy.

Approaches to multi-chip partitioning at RTL-levels or below typically consist of enhancements and adaptations to well-known graph partitioning algorithms such as Kernighan-Lin and Fiduccia-Mattheyses. They tend to supplement a traditional graph partitioning algorithm with additional stages of optimization mechanisms. These mechanisms help the overall partitioning strategy comply with architecture-specific constraints or limitations, such as communication channels, I/O pins and logic resources, while guiding toward more effective partitioning solutions. Additionally, techniques such as unit/module cloning and communication channel multiplexing are integrated in an effort to solve the I/O limitation problem often seen in FPGA/ASIC implementations. The cutsize minimization objective stated in *strict* graph partitioning is enhanced by considering performance and cost issues, such as area, price, delay, and pin count. Frank M. Johannes gives a concise review of logic

emulation structural partitioning methods in [34]. We discuss several representative methods to illustrate the salient issues in DHA structural partitioning.

The partitioning algorithms by Kim, Kuznar, and Chou had the objective of finding a *feasible* circuit partitioning solution while minimizing the monetary cost of an implementation; as measured by the cost of all DHA devices [35][36][37]. This objective somewhat departs from our intention in this thesis, since it assumes that the DHA can be custom-built using the results of the partitioning exploration, as opposed to targeting an established DHA platform, where device types and connections are predetermined. Nevertheless, they are representative of the types of strategies followed at the structural level.

Kuznar, et al. used Integer Linear Programming to determine the optimal distribution of Xilinx devices to partition a certain circuit in order to minimize price [35]. Then, a FM-based heuristic is recursively applied on the circuit to produce a subcircuit at each iteration that meets constraints determined by the previously determined target distribution.

Chou, et al. approached the circuit partitioning for huge logic emulation systems by using a hybrid algorithm that combined local bottom-up clustering and top-down recursive partitioning [36]. Partitioning was achieved by converting it to a Set Covering Problem and using the Espresso covering algorithm to improve results. Fang and Wu enhanced Chou's strategy by utilizing design hierarchy acquired from the circuit's pre-synthesis VHDL specification to guide clustering decisions [38]. This served to alleviate the I/O limitation problem typically encountered when partitioning to multiple FPGA platforms. Fang and Wu's work supports our general hypothesis of using higher-level information to provide better solutions to the multi-device partitioning problem. However, their approach is still essentially structural, which severely limits the amount of algorithmic information that they can use and makes their results highly dependent on programmer's style.

Kim, et al. utilized a two-phased approach based on Fiduccia-Mattheyses [37]. The first stage iteratively improves an objective function that is the weighted sum of the cut size and the delay on the critical timing paths, while not considering system constraints such as routability. The second stage further optimizes the initial solution to satisfy the constraints. Results obtained with their method required less FPGAs, and obtained a lower monetary cost than the methods by Kuznar and Chou.

Scott Hauck devised a methodology to determine an order to map the results of recursive bipartitioning to the devices in a multi-FPGA board, based on the communication properties of the later [9]. His algorithm first determines the most congested DHA channel(s) using a technique developed by Yeh, et al. [39], and establishes the order in which the resulting partitions from a recursive Fiduccia-Matheyses bipartitioning should be assigned to the DHA devices. The most significant DHA connection topologies were analyzed and the resulting partition orderings were reported in his dissertation. However, no benchmark results using the complete methodology were found in literature.

In our opinion, most of the structural partitioning methodologies accomplish their main purpose, which is to effectively map a circuit to a set of devices for hardware emulation. Multi-FPGA logic emulation systems are not the end platform for the emulated designs, thus structural partitioning methodologies do not necessarily strive for implementation performance but rather compliance. They apply generalized optimization techniques based almost entirely on netlist information. We hypothesize that use of higher-level information is beneficial in partitioning strategies, especially when performance is an important implementation objective.

### 2.3.4 Behavioral-level Partitioning

A commonly observed fact in EDA is that optimization methodologies that work at higher levels of abstraction usually achieve better performance than their

lower level counterparts. Behavioral-level partitioners typically consist of a high-level cost estimation mechanism coupled with a partitioning engine that relies on probabilistic or heuristic decisions to improve on the solution [7][8] [10][13]. A recurring characteristic of methodologies dealing with partitioning for homogenous distributed hardware architectures is that they integrate tasks that have been traditionally considered at the high-level synthesis stage, particularly allocation and scheduling. Thus, behavioral partitioning, rather than being a separate stage independent of high-level mapping synthesis tasks becomes an integrated partitioning/synthesis step.

The following three references represent, to the best of our knowledge, the best documented automated high-level partitioning strategies for DHAs. They exemplify the types of approaches that have been followed for the behavioral partitioning of general algorithms to DHAs, and highlight the combination of system-level and high-level issues throughout the optimization process:

Bringmann, et al. combined high-level synthesis and partitioning into a methodology aimed at multi-FPGA architectures [8]. They used a constructive (clustering) strategy to partition a flow-graph whose nodes represent arithmetic operations and whose edges represent data dependencies among the operations. The closeness metric used for clustering takes into consideration scheduling and allocation issues, such as the concurrency of operations (how probable it is for two operations to be executed in parallel) and the probability that an operation is on the critical path. Their algorithm is also capable of serializing data transfers among the devices in order to maximize circuit performance under the constraints of the target architecture. Their heuristic was used to partition four algorithms from the from the 1992 High-Level Synthesis Workshop [40], one of which was an 8-point DCT, to multi-FPGA platforms consisting of four Xilinx 4013 and 4025 devices.

Srinivasan worked on several methods for high-level partitioning onto FPGA-based reconfigurable computers [41]. His 'data flow graph partitioning' (DFGP) methodology starts by completely scheduling and allocating a DFG to an (imaginary) FPGA with an area equal to the cumulative area of all devices. Then, the scheduled graph is partitioned using latency as the optimization objective and constrained by the resources offered by the individual devices. Partitioning is guided by a genetic algorithm exploration engine whose fitness function is a combination of the cut-size and area penalties. His 'Block-level partitioning' (BLP) methodology utilizes a coarse-grained graph structure, called a behavioral-block graph (BBG), to represent the systems' behavior. In this case, a simulated annealing optimization technique alternates between partitioning and scheduling/allocation stages. Benchmarks, including FFTs and DCTs, were coded by Srinivasan's research group in both DFG and BBG format, and partitioned to Annapolis Systems' Wildforce, Wildchild and Wildfire multi-FPGA boards [42]. BLP obtained improved results over the DFGP in the majority of cases, both in terms of design latency and partition runtime. A 5.57% average improvement (6.87% peak) was obtained in terms of latency, while a 94.70 % (96.39% peak) reduction was seen in run-time. However, BLP requires the user to have thorough understanding of the partitioning process, synthesis and target architecture specifics. Since many variables were changed between one method and the other (e.g. optimization method and granularity of representation, among others) it is difficult to distinguish a specific feature that makes BLP superior to DFGP. The FFT and DCT benchmarks used by Srinivasan were small by todays standards. However, as seen in Chapter 6 the DFPG method's specification was sufficiently detailed, which allowed us to implement this technique and use it as comparison against our results.

Duncan, et al. proposed the COBRA-ABS High Level Synthesis System for Multi-FPGA Custom Computing Machines [10]. COBRA-ABS is a complete partitioning and HLS system for synthesis of datapath dominated applications onto multi-FPGA custom computing machines (FCCMs). It performed global optimizing high-level synthesis using simulated annealing, integrating all partitioning, scheduling, and allocation operations into one optimization step. Given an algorithm, specified in a subset of C, COBRA-ABS synthesized a custom Very Long Instruction Word (VLIW) architecture suitable for implementation on the specified FCCM. Six DFT algorithms of sizes 64 and 1024 were partitioned using this system onto a FCCM consisting of 4 Motorola MPA-1000 series fine-grained FPGAs, and their results compared to implementations in a Sun Ultra 1/140 computer [16]. A design speedup of up to 20 times is achieved as compared to the Sun Ultra implementation. Although not necessarily the authors' main intentions, the results show how different formulations of the same discrete signal transform can attain significantly different design results. For instance, all other considerations being equal, a decimation in frequency (DIF) radix-4 FFT achieved a 54% reduction in latency over a DIF radix-2 FFT. As part of the discussion of results, the authors mention that their partitioning strategy "has no knowledge of the regular structure of the FFT", probably implying that more effective results and exploration could be obtained by being aware of the FFTs regular structure. However, to the best of our knowledge, no additional automated methodologies incorporating these considerations have been proposed since.

To the best of our knowledge, the only partially-automated methodology specifically targeting the implementation of a discrete transform was documented by Pinit Kumhom [43]. He used Kronecker algebra formulations to guide mapping of a dimensionless FFT onto an Annapolis Wildforce multi-FPGA board, which has a linear array connection topology and crossbar. Taking into consideration the structure of

the FFT and the target topology, Kumhom established how computational units where to be allocated in the platform. A 2-point FFT kernel was instanced in each FPGA device with the crossbar acting as the sole communication resource among the processors. The mapping scheme between the FFT operations and the instanced kernels was accomplished by conducting an exhaustive search through the space of formulation permutations in order to minimize communication. For a given size FFT, each possible permutation reformulation was evaluated with a performance model. An optimal mapping pattern was identified after conducting an exhaustive search for various FFT sizes. Kumhom's approach acknowledges the importance of using the algorithmic regularity of FFT's to aid in their mapping. However, to extend its applicability beyond the specific case of FFTs on the Wildforce we believe that it should be enhanced to consider other transformations beyond permutations and a more effective space exploration technique.

### 2.3.5 Limitations of Previous Methods

The state of research regarding partitioning to distributed hardware architectures evidences the need for new methods specifically focusing on algorithms where high performance is of utmost need. The following is a list of unresolved issues identified in previously reported strategies, from the perspective of our particular problem:

1. Generality of approach - Almost all current strategies target either general-purpose or an ample class of algorithms. In our opinion, this considerably impacts the effectiveness of these methods for partitioning specific algorithmic classes, and in particular those with high-connectivity and regularity such as DSTs. In trying to be 'good for the general case' these strategies are not able to incorporate specific optimization techniques which may benefit a small class of important algorithms. Furthermore, to the best of our knowledge, no automated partitioning methodologies have been reported specifically for DSTs onto DHAs. Thus, none of the

previously proposed strategies takes advantage of DST properties to expedite and improve their partitioning to distributed hardware architectures.

2. Input language - Previously proposed methods specify their inputs using high-level languages or data flow graphs. Although this is not a problem by itself, it presents a limitation as to the types of transformations that may be applied to an algorithm during the optimization process. The use of higher-level transformations is particularly important in DSTs, which have a rich set of algorithmic-level rules whose application can affect the outcome of partitioning methodologies, as evidenced by Duncan, et al [10].
3. Exploration limited to graph-level and below - Most of the inspected methodologies perform partition optimization by exploring a single formulation of an algorithm. Although this can serve to benchmark specific parts of a partitioning methodology, it fails to consider alternate formulations which may expose opportunities to obtain more effective implementations. Formulation exploration is used in strategies for automated *software* generation of DSTs [44][45]. The development of a methodology that uses formulation exploration for hardware requires a sense of the impact of reformulation on implementation, a topic which is explored as part of our work.
4. Representation granularity - Several researchers working in the behavioral-partitioning (BP) area have highlighted the effect of the graph representation granularity on the BP results [7][12][46]. Fine-grained representations, such as using a flow-graph where each node represents an arithmetic or load/store operations, allow for a more thorough exploration of the solution space than coarser representations, which already group several operations into each node. Coarse-grained representations have the advantage of having fewer nodes, which potentially results in a faster exploration of the solution space. In order to have both fast exploration and effective results, some proposed approaches have utilized a combination of both granularities. In our opinion, a common limitation to these approaches

is that they either construct coarse-grained representations based on the modules defined in the programmer's specification (for instance, defining nodes by functions or subprograms in the specification) or rely on the programmer to manually build the coarse-grain representation [7][46]. This makes the quality of results dependent on a user's programming style or requires the programmer to be reasonably familiarized with the target system in order to obtain effective results.

5. Mix of high-level synthesis tasks - The interdependence of high-level synthesis and behavioral partitioning tasks has been acknowledged by various researchers [13][47]. However, the exact relationship between the various high-level synthesis objectives and their influence on high-level partitioning is difficult to establish, especially for general algorithms. For reasons of computational cost, these tasks are handled individually by some HLP methods [7][48]. Yang and Gupta studied how the decomposition of synthesis and partition sub-tasks impact the quality of synthesis results [47]. They experimented with different orders in which to sequentially perform scheduling, binding and partitioning, as well as their concurrent performance. Of the tested orderings, the strategy that performed partitioning followed by a simultaneous scheduling and binding obtained the best combination of synthesis quality and computational cost. Surprisingly, this strategy is not followed by any of the subsequently developed HLP methodologies.
6. Comparison to other multi-device implementations - A common deficiency among documented HLP methodologies is their failure to compare results with other HLP methods. There is a lack of accepted benchmark sets for behavioral partitioning. This problem is aggravated by the fact that high-level optimization relies on other tools to produce a final result, so it is not trivial to isolate the benefits that can be directly attributed to the partitioning process itself. These other EDA tools are elaborate research projects in themselves and typically not documented explicitly enough as to allow third-party validation. Commonly, they have been developed

as part of projects for the private industry or military so they are not available for our use by the general public. Another hurdle for the comparison of results is that the EDA community has not agreed on a standardized target architecture even for benchmark purposes. As a consequence of these difficulties, most articles in high-level partitioning target very diverse architectures and only compare results to other methods from the same author/group or to implementations in alternate GPP architectures [7][8][10][11][49].

### 2.3.6 A glimpse into our approach

Even though the usefulness of behavioral partitioning has been proven for algorithms in general, we hypothesize that by introducing functional knowledge about these algorithms into the partition/synthesis process shall result in a more focused exploration of the design space than if only non-deterministic or general-case based heuristics are used. By limiting the scope of our methodology to signal transforms we were able to use characteristics inherent to DSTs to arrive at efficient partitions in a straightforward manner. DST-features are introduced in two abstraction levels as part of our methodology: at the graph level and the algorithmic-level. At the graph partitioning level, a series of DST-specific structural considerations have been taken to improve the graph partitioning heuristic. Most of these structural features are evidenced in Section 2.4, in which we review previous work related to DST hardware implementations. At the algorithm-level, an exploration is conducted in search of equivalent transform formulations that are more suitable for the target topology. Formulation-level exploration is used in several automated code generation mechanisms. However, as discussed in Section 2.5 they have not been satisfactorily used to guide automated partitioning and synthesis of multiple-device implementations.

## 2.4 Hardware implementation of signal transforms

Ultimately, our work intends to solve the problem of mapping and partitioning DST algorithms onto distributed hardware architectures. Within this context, a

review of hardware implementations of signal transforms is important for many reasons, among them:

1. Synthesis, a stage that is closely dependent on partitioning, is responsible for the actual generation of hardware structures based on algorithmic description. Thus, an adequate high-level partitioning methodology must be aware of the impact of its decisions on synthesis.
2. At high-levels of abstraction, it is essential to have a well defined target intra-device architecture to achieve acceptably accurate estimations of resource utilization and latency. A study of previous implementations helps identify structural tendencies to define a target intra-device architecture.
3. When targeting a multi-device architecture, each device will probably implement processing elements similar to the structures of single-device implementations.

Research on the implementation of discrete signal transforms has mostly focused on the discrete Fourier transform (DFT), the main reason being its usefulness in so many applications. Furthermore, formulations for other discrete signal transforms, such as the Discrete Hartley Transform (DHT) and the Discrete Cosine Transform (DCT), commonly try to emulate the regularity of the DFT [50][51]. Thus, our discussion of hardware structures for DSTs is centered around the structures proposed for the DFT.

Discussion of DST hardware implementations is organized as follows. Section 2.4.1 presents the general definition of DSTs. This is followed by a review of DFT implementations, in Section 2.4.2, highlighting the techniques that have been developed to take advantage of its structural regularity.

### 2.4.1 General DST Definition

A linear separable transform of a $d$-dimensional discrete signal $x[n]$, where each element can be specified by $d$ indexes $n_1$, $n_2$, ..., $n_d$, is defined by:

$$X[k_1, .., k_d] = \sum_{a_d=0}^{N_d-1} .. \sum_{a_1=0}^{N_1-1} x[a_1, .., a_d]\alpha_1(a_1, k_1)..\alpha_d(a_d, k_d) \tag{2.1}$$

where the $\alpha_i$'s are the transform functions. For example, for the d-dimensional DFT $\alpha_i(n_i, k_i) = e^{-j2\pi n_i k_i/N_i}$, and for the d-dimensional type-II DCT $\alpha_i(n_i, k_i) = cos\,[(2n_i + k_i)\pi/2N_i]$.

A d-dimensional transform can be expressed as a tensor (Kronecker) product of unidimensional transforms:

$$\hat{X} = (A_{N_1} \otimes \ldots \otimes A_{N_d})\hat{x} \tag{2.2}$$

where $A_{N_i}$ is the $N_i$ point discrete signal transform matrix, $\otimes$ denotes Kronecker product and $\hat{x}$, and $\hat{X}$ are vectors of size $N = N_1 \cdot \cdot N_d$ obtained by ordering $x$ and $X$ lexicographically [52].

DSTs can be reformulated into algorithms with reduced computational complexity by taking advantage of symmetries in the transform's functions (e.g. the roots of unity in the case of the discrete Fourier transform). These fast algorithms can be expressed as a multiplication of the input signal vector by a succession of sparse matrices, which can be compactly expressed in Kronecker products algebra.

### 2.4.2 DFT implementations

The differences and similarities between the various 1-D DFT implementations that have been devised over the years can be better understood by observing the 1-D DFT and fast Fourier transform (FFT) algorithm formulations. For a length $N$ complex sequence $x[n], n = 0, 1, 2, \ldots, N-1$, the DFT is defined as:

$$X[w_k] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi kn/N}, k = 0, 1, 2, \ldots, N-1 \tag{2.3}$$

The DFT formulation can be interpreted as a matrix-vector formulation and implemented as such. Periyacheri and Jones implemented DFTs and DCTs using

matrix multiplication on the WILDCHILD multi-FPGA architecture as part of the development of the MATCH compiler project. The objective of this project was to map MATLAB signal processing functions to distributed reprogrammable architectures [11][49]. The WILDCHILD platform consists of eight *slave* and one *master* FPGA interconnected by a crossbar. Each FPGA has an attached RAM. In one of their matrix multiplication ($C = A \cdot B$) schemes, the columns of $B$ were distributed among the slave FPGAs, while the elements of $A$ were broadcasted row-wise from the master to all of the slaves. Each slave FPGA carried out the multiplications and accumulations of their assigned elements of $B$ with the broadcasted elements of $A$, to obtain the elements of $C$. The complexity of the matrix-vector DFT formulation is $O(N^2)$ vs. $O(NlogN)$ for the fast Fourier transform, which explains why implementations of the former type are rare.

As illustrated in Algorithm 1, the FFT is a divide and conquer algorithm that at its kernel (instruction $\langle x_k, x_{k+p} \rangle \leftarrow \langle x_k + z^j x_{k+p}, x_k - z^j x_{k+p} \rangle$) consists of multiplications and additions between the points (or the intermediate results) and roots of one coefficients ($z^j$ terms). The instruction that we have identified as the kernel is commonly known as a radix-2 butterfly, whose dataflow graph is shown in Figure 2–4. Figure 2–5 illustrates the DFG for an 8-point decimation in time algorithm. Notice that, besides the multiply and addition operations, all that remains is a highly regular connection pattern between the different stages of the algorithm. In general, the implementation of a $N = 2^n$-point FFT requires $logN$ stages where $O(N)$ multiply/add operations are performed at each stage.

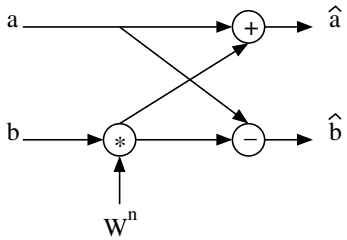



Figure 2–4: Dataflow graph representation of the radix-2 butterfly.

**Algorithm 1** FFT decimation in time algorithm.

**Input**: Signal $x$ of length-$N = mp$
**Output**: FFT($x$)

1. **For** $b \leftarrow (logN) - 1$ to 0 by $-1$
   1.1. $p \leftarrow 2^b$ ; $q \leftarrow N/p$
   1.2. $z \leftarrow w^p$ ;
   1.3. **For** $i \leftarrow 0$ to $N - 1$
      1.3.1. **If** $(kmodp) = (kmod2p)$
         1.3.1.1. $\langle x_k, x_{k+p} \rangle \leftarrow \langle x_k + z^j x_{k+p}, x_k - z^j x_{k+p} \rangle$
      1.3.2. **End If**
   1.4. **End For**
2. **End For**

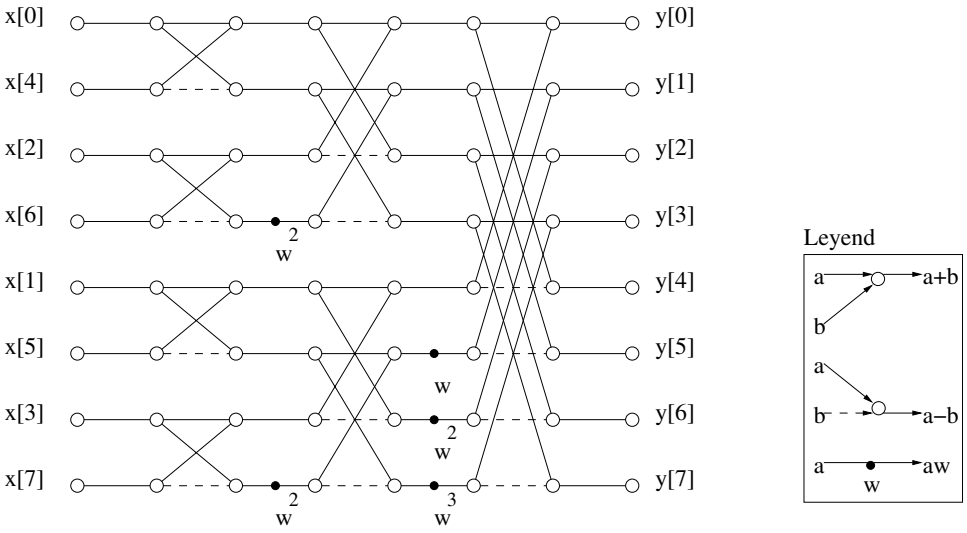



Figure 2–5: Data flow for an 8-point FFT. Dashed lines represent operands that will be subtracted. Filled dots represent multiplication by twiddle factors.

Despite the substantial amount of hardware resources provided in today's dedicated hardware devices, for most practical cases, a one-to-one mapping of DST operations to hardware functional units is not feasible. For distributed hardware architectures, the limitations in inter-device communication channels make one-to-one mappings impractical. Figure 2–6 illustrates this situation. An 8-point FFT is mapped to DHAs with three and four FPGAs connected in ring topology whose communication channels are 1-point wide. Assume that each device has enough resources to physically instance all the operations assigned to it. The limitations

in communication resources would data-starve the functional units, so most would have a very low duty cycle, which represents an ineffective use of hardware.

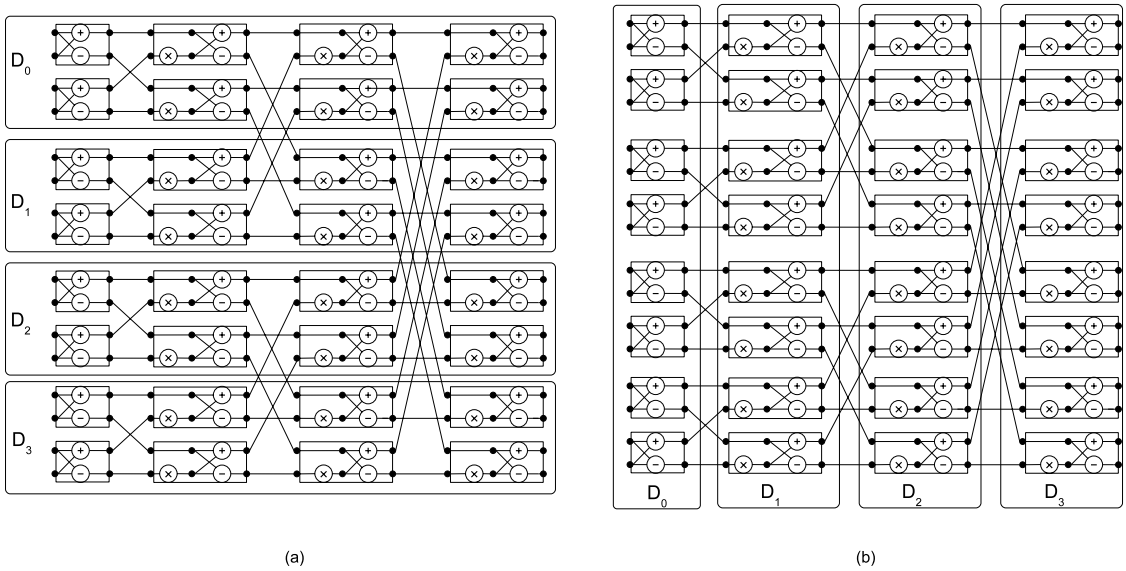



Figure 2–6: A one-to-one mapping of a 16-point FFT to a 4-FPGA platform.

These limitations call for folding the DST dataflow graph and iterating data through a limited number of functional units. Regularity of the FFT is well suited for a folding scheme, as it consists of the same kernel operations throughout its computational structure, connected by stride permutations. Kernel operations are uniform across the vertical and horizontal axis of the FFT dataflow graph. A strict *vertical* folding of the FFT, shown in Figure 2–7 maps computation as in an architectural-level pipeline, where one or more complete DST computational stages are assigned to each hardware device. In strict *horizontal* partitioning, depicted in Figure 2–7, each device carries out all stages of computation for a data subset, similar to single-instruction-multiple-data (SIMD) processing. For an FFT or Walsh-Hadamard Transform of size $2^n$ there are $H \times V$ folding strategies, where $H$ and $V$ are the number of divisors of $n$ and $2^{n-1}$, respectively [53]. Throughout the rest of this discussion, $h$ and $v$ will be the degree of horizontal and vertical mapping, respectively. For instance, Figure 2–7 illustrates the folding of an 8-point

FFT. Figure 2–7(a) shows the fully expanded DFG, (b) represents $h = 1$, $v = 8$, and (c) is a $h = 3$, $v = 1$. Both folding orientations can be combined as needed to comply with performance requirements or device resource availability. Previous FFT implementations can be classified by the degree and orientation of their folding.

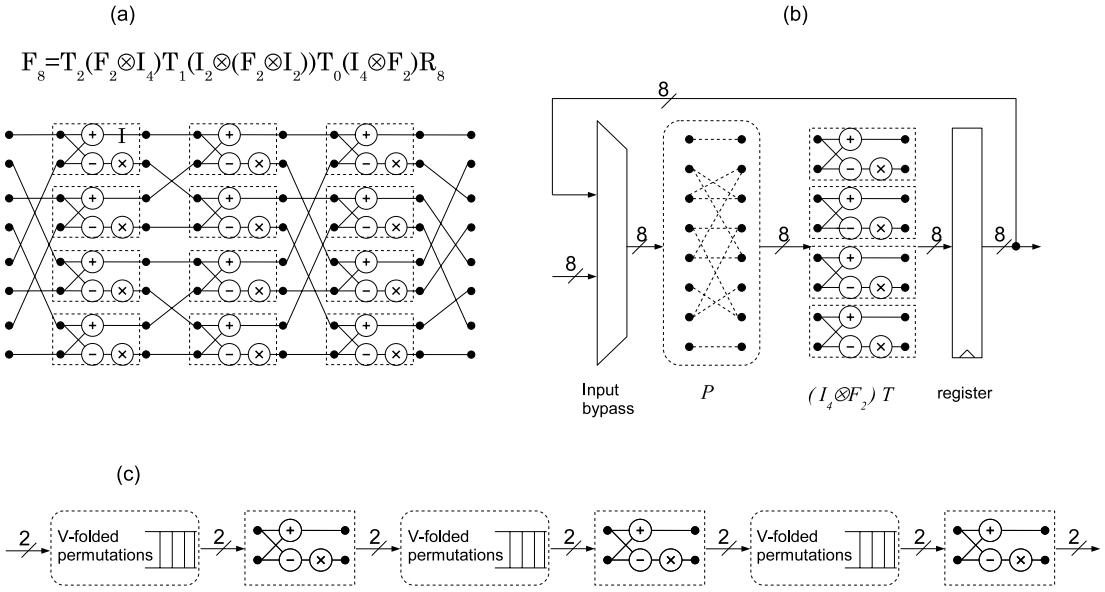



Figure 2–7: Folding of an 8-point FFT: (a) fully expanded DFG, (b) strict horizontal folding and (b) strict vertical folding.

An FFT dataflow graph which is fully folded both vertically and horizontally, requires the instancing of a single butterfly-twiddle computational kernel. Figure 2–8 illustrates this implementation. This architecture requires a memory of at least $N$ words for a length-$N$ FFT, and performs the computation *in place*. Commercially available FFT cores, such as the Xilinx LogiCore library, typically implement this type of folding [54]. Every point and intermediate result of the FFT will be read/stored in the memory at some instance. This type of scheme is similar to the one utilized by GPPs or PDSPs with the difference that the hardware is dedicated and no instructions must be read from memory. Many times, the memory interface will be the bottleneck for the performance of this system. One variation of the single memory architecture is the cached memory architecture, in which a cache is placed

between memory and the processing element in an effort to increase the effective memory bandwidth. [55]. Another option strategy to increase memory bandwidth, is to have two read/write memories connected to the processor. Data begins in one of the memories and goes back and forth between them as the transform is being processed [56].

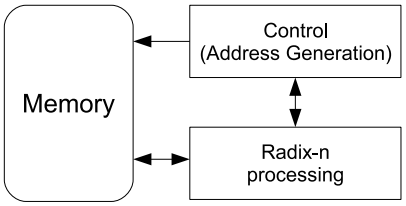



Figure 2–8: Single kernel implementation of FFT.

The pipeline is the most commonly documented architecture among hardware implementations of the FFT. Instead of requiring one memory with capacity for $N$ words, the pipeline uses smaller multiple storage elements with a simpler read/write scheme, e.g. FIFOs. Various methodologies and digital circuit techniques have been proposed for implementation of FFTs in a pipeline, especially for accomplishing the permutations needed between the computational stages. Shousheng and Torkelson surveyed some of the existent pipeline architectures and proposed a new architecture that requires a simpler connection scheme while retaining the memory and functional complexity of earlier models [57].

Depending on the amount of logic resources available and performance criteria, a combination of partial vertical and hardware foldings might represent the best implementation option. Kumhom's strategy for partitioning a dimensionless FFT allows the implementation of a single kernel for each of the four FPGA devices. An $n$-point dimensionless FFT is implemented as a $h = n$, $v = 2^{n-3}$ folding, thus behaving much like a SIMD implementation in a parallel processor system. Fang, et al. explored the effect of orientation and degree of folding on the Walsh-Hadamard transforms [53]. They conclude that folding reduces the resource requirement at

the expense of performance, with horizontal folding having the smallest impact on latency. Nordin, et al. developed a parameterizable soft core generator for FFTs which allowed them to experiment with the effect of various parameters, such as $v$, in the cost/performance of FFT implementations to FPGAs [58]. Their methodology utilizes the Pease FFT formulation to facilitate the folding of inter-stage permutations. Figure 2–9 shows the formulation and dataflow graph of an 8-point Pease FFT. Observe that all inter-stage permutations are the same ($L_{8,4}$), greatly simplifying the horizontal folding of this structure, as seen in Figure 2–9(b). Vertical folding of any power-of-two degree is implemented using an efficient v-folded implementation of the perfect shuffle permutations, illustrated in Figure 2–10 [59]. Their implementations are comparable in quality to DFT cores from the Xilinx LogiCore library.

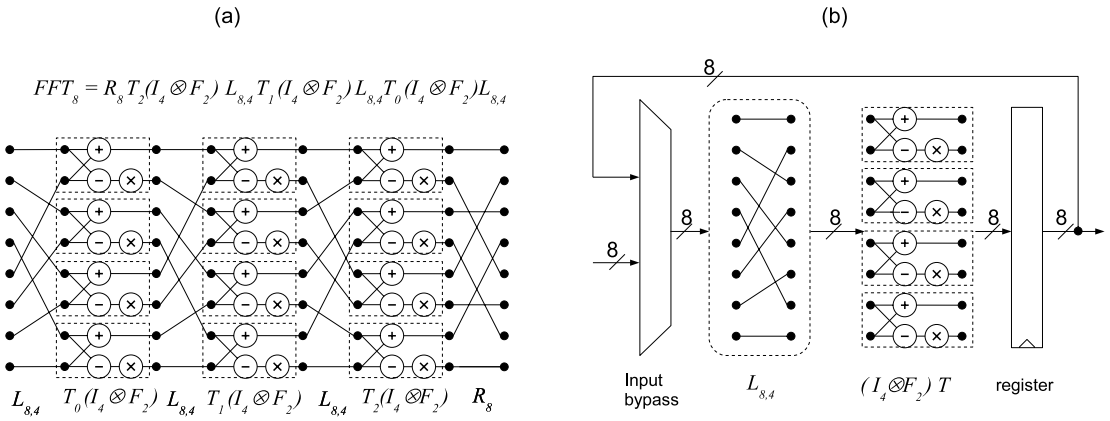



Figure 2–9: (a)An 8-point Pease FFT formulation and dataflow graph. (b) full horizontal folding [58]

Milder, et al. developed a resource estimator for Nordin's soft core generator by analyzing resource utilization of the various architectural components involved in the folded FFT structure [60]. Their estimator can be used to determine the feasibility of diverse implementations of a given FFT size to a single FPGA device. The relationship between $p$, the number of instanced $FFT_2$ kernels, and the implementation latency leads to conclude that to obtain minimal latency an FFT

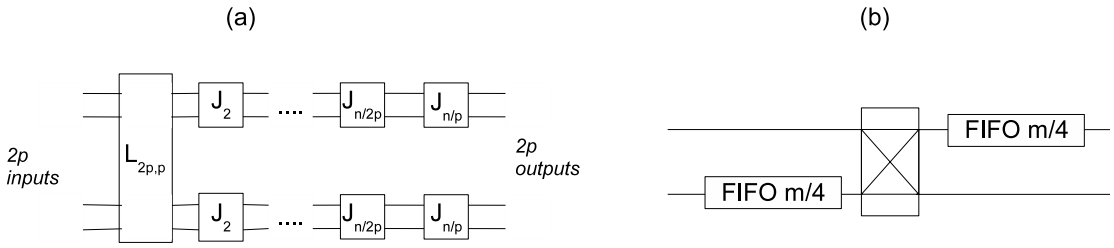



Figure 2–10: (a)A vertically-folded $L_{n,n/2}$ permutation with $2p$ ports. (b) Detail of the $J_m$ component, proposed by Takala, et al.[58][59].

should be implemented using the highest $p$ which still fits the FPGA. In most of their experiments, $p$ is limited by the available embedded multiplier resources.

In summary, manual hardware implementations take advantage of the FFT's regularity to maximize resource utilization. This is mainly accomplished by folding the computational structure.

## 2.5 Optimizing discrete signal transform implementations for specific architectures

For years, researchers looking to optimize DFT formulations were mostly concerned with minimizing the number of costly arithmetic operations, such as multiplications. Nevertheless, it has always been acknowledged that the optimality of a DFT implementation depends not only on the number of operations but also on system-specific such issues as the efficiency of data access patterns on the underlying register-cache-memory structure. Because of this, *optimizing* a DFT algorithm implementation to a specific architecture has traditionally meant hand-coding the algorithm to exploit the various features of the system [61]. Several recent efforts have dealt with the question of how to automatically produce high quality signal transform code for an arbitrary platform. FFTW [45] [62], and SPIRAL [44], two popular methodologies for this purpose, are essentially solution-space exploration engines in search for optimal signal transform implementations. To explore the solution space for a given DST, they heuristically apply algorithmic level rules, thus obtaining alternate formulations. Software code is generated for each formulation,

compiled and executed on the target system. Based on the result and the chosen optimization heuristic, further reformulations are explored.

Although the formulation of a DST is known to have an effect on its hardware implementation, the previously mentioned code-generation methods have not been properly extended to contemplate multi-device dedicated hardware implementations. Fang, et al., used AREP, a library with signal transform capabilities for the GAP computer algebra system, to generate automatic formulations for implementations of the Walsh-Hadamard transform on a single FPGA [63][53]. Kumhom used Kronecker algebra to guide his implementation of a universal FFT processor on a multi-FPGA system, but ultimately relied on an exhaustive search for permutations to optimize data ordering and communication [43]. In general, previously documented methods arrive at a point where the algorithm is expressed in a generic flow graph manner or recur to strategies that treat the signal transform as a generic algorithm. After this transition, algorithmic level characteristics can be hard to extract with the graph partitioning methods utilized by those schemes.

## 2.6 Summary

In this chapter we have reviewed some of the most relevant literature to our line of work. Distributed hardware architectures continue to be an option for high-performance implementation of discrete signal transforms. The vast majority of previous implementations to these architectures have been done manually or with generic partitioning schemes. To reduce the design complexity involved in mapping algorithms to such architectures and to improve the performance of the resulting implementation, novel methodologies are needed which consider algorithmic-level and high-level algorithm characteristics. Through a systematic study of techniques that have been used for the implementation of signal transforms to hardware devices and the exploration of algorithmic formulations, we have devised a methodology that

integrates functional knowledge about DSTs to aid in their automated partitioning and high-level synthesis.

# CHAPTER 3

# Problem Formulation

This chapter has two main objectives. The first is to define and explain the problem of high-level partitioning of discrete signal transforms to distributed hardware architectures. Second, to provide an overview of the approach we have followed in search of a suitable solution, presented in Section 3.2. The last section summarizes the ideas presented throughout the chapter.

## 3.1 Problem Statement

The problem of high-level partitioning of a discrete signal transform to a distributed hardware architecture (HLPDD) can be stated as follows. Given:

1. A high-level description $T$ of a discrete signal transform, whose functionality is implemented by a set of tasks $(T_0, T_1, \ldots T_{M-1})$. The description includes, at least, the following:
   (a) The DST's type and number of input points, or a DST algorithmic description.
   (b) The resolution (i.e. bitwidth) and format (e.g. fixed point or floating point) of the input points.
2. A description $H$ of the target multi-chip architecture, described as a weighted, annotated hypergraph[1] $H = (D, C)$, where:

[1] A hypergraph is a pair $(V, E)$ of sets, where the elements of $E$ are non-empty subsets of any cardinality of $V$ [64].

(a) Each vertex $d_i \in D$ represents one of the architectural devices. The weight $w_i$ of each vertex represents the logic capacity of device $d_i$.

(b) The set of hyperedges $C$ represent the available communication channels. Each channel $c_i \in C = \{c_0, \ldots, c_{M-1}\}$ is a subset of the vertex set $D$. Furthermore, the following two mappings are specified:

i. The *weight* $W(c_i)$ of a channel is a function $W : C \rightarrow \mathbb{Z}^+$, whose value is a relative measure of the impact on system latency of communicating a data point through $c_i$.

ii. The *bitwidth* $B(c_i)$ of a channel is a function $B : C \rightarrow \mathbb{Z}^+$, whose value is the total number of bits that may be commuicated through channel $c_i$ at any computational cycle.

Determine a mapping function $f : T \mapsto D$ that minimizes the latency of the overall transform implementation. In the context of this work, latency is defined as the number of clock steps between the beginning and end of computation. A solution to the HLPDD problem is subject to the following constraints:

1. For each device $D_j$, $0 \leq j < V$:

$$\text{Resources}\left(\bigcup_{\forall T_i | f(T_i)=D_j} T_i\right) \leq W_j$$

where $Resources\,(x)$ is the total number of logic resources required for the implementation of subset $x$ of tasks. In other words, the constraint requires that the set of tasks assigned to each device $D_j$ can be implemented with the resources provided in $D_j$. However, this does not imply that there will be a one-to-one correspondence between the tasks and the instantiated hardware functional units. Thus, each task might be assigned a computation step $t$ when a corresponding hardware unit will implement its functionality.

2. For every pair of tasks $(T_i, T_j) \in T$, at every computational step $t$

$$\text{Comm}\,(T_i, T_j, t) \leq B_{f(i)f(j)} \tag{3.1}$$

where $\text{Comm}\,(T_i, T_j, t)$ is the number of bits that are communicated between tasks $T_i$ and $T_j$ during computation step $t$. In other words, at any time the number of bits communicated from task $i$ to task $j$, does not exceed the width of the channel that connects their assigned devices, $B_{f(i)f(j)}$.

**Example 1.** *The following example illustrates the representation of a topology using a hypergraph. For the architecture depicted in Fig. 3–1, $D = \{d_0, d_1, d_2, d_3\}$, the channel $c_0 = \{d_0, d_1\}$, and the crossbar is $c_3 = \{d_0, d_1, d_2, d_3\}$. $B(c_i) = 32$ for $0 \leq i \leq 3$.*

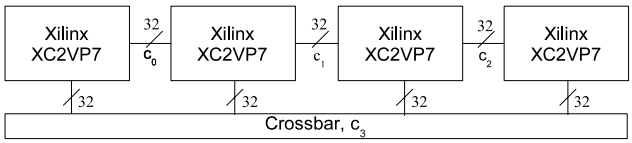



Figure 3–1: Sample topology.

Although somewhat similar in terms of inputs, constraints and objectives, our present problem is not to be confused with the classical problem of *graph* partitioning (GP). The key difference between GP and HLPDD is brought by the inputs to the problem, specially the entity which is to be partitioned and its abstraction level. GP receives a generic graph, while HLPDD receives a high-level description of a discrete signal transform. Although the high-level description will be converted to a graph for actual partitioning, a given DST has multiple equivalent algorithms, each with its corresponding graph and features that make it unique for partitioning purposes. This allows a solution to HLPDD to explore alternative formulations of a given transform rather than settling on a particular algorithm. Furthermore, working exclusively on a family of algorithms allows a solution to HLPDD to incorporate optimization techniques which would not necessarily apply to general algorithms.

GP and HLPDD also differ in their optimization objectives. The former is purely concerned with minimizing cutsize, whereas the later is interested in minimizing latency, which is a function of the distribution of communication and processing in the DHA.

## 3.2 Methodology

The main objective of this work is to develop a high-level partitioning methodology for discrete signal transforms to distributed hardware architectures. In essence, our methodology takes advantage of DST-specific features and properties to provide effective partition solutions in acceptable time.

Fig. 3–2 shows a conceptual map of the proposed partitioning methodology, called DMAGIC (DST Mapping using Algorithmic and Graph Interaction and Computation). As implied in the problem formulation, the methodology receives two inputs. The first is a DST specified as a Kronecker Products Algebra (KPA) formulation, parameterized by at least the resolution of its points. The second is a high-level specification of the target architecture, which includes the number and logic capacity of the devices and their connection topology. Based on the inputs, a series of heuristics reformulate the transform to expose characteristics that are exploited by the partition/placement process. Reformulation is accomplished by applying algorithmic-level transformations, such as factorization and permutation rules, on the original formulation.

The *Kronecker to Graph* process converts the algorithmic formulation into a dataflow graph (DFG) whose nodes denote functional primitives. These primitives are small computational components that are common throughout the formulation and have been identified as efficient procedures on the target devices. The dataflow graph is then partitioned using a deterministic graph partitioning/placement heuristic that has been enhanced to handle DST structures. The quality of results from the partition/placement process is used by a heuristic formulation-exploration

engine to guide exploration onto further formulations. Based on the partitioning results of the current formulation, factorization rules are used to generate a new formulation, which, hopefully, improves the previous results. The conversion/partitioning/reformulation process continues until no considerable improvements are detected, at which point the methodology outputs the best partition/placement scheme obtained throughout the exploration.

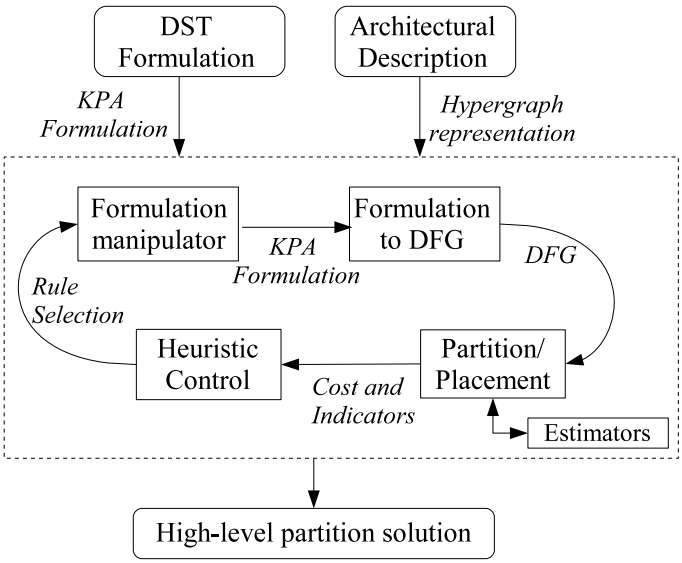



Figure 3–2: Block diagram of methodology.

The DMAGIC methodology takes advantage of DST features at two levels of abstraction: the graph and algorithmic levels. First, a series of DST-structure aware considerations have been incorporated into the partitioning/placement heuristic. These considerations help the partitioning/placement heuristic conduct a faster exploration and maintain the regularity of the original expression, thus obtaining results that can efficiently mapped to hardware structures [65]. Second, the methodology uses rules specific to the particular DST at hand to conduct an exploration of alternate formulations that might be more suitable for partitioning to the given topology. In this sense, DSTs have an advantage over other algorithms, because they a have considerable amount of properties to be used for such purposes.

DMAGIC's development was organized into four major tasks, each building upon the findings and tools of its predecessors.

**Task 1: Development of a methodology scheme.** The first task consisted of defining a general partitioning strategy and deciding the format for our methodology's inputs. An extensive literature review revealed that most reported DST implementations to distributed hardware architectures use either manual or general purpose partitioning methods. In other words, no automated partitioning strategies have been reported to take advantage of DST specifics, albeit evidence that using these properties could contribute to more effective solutions to the HLPDD problem:

- Effective manual DST implementations in hardware and distributed architectures are obtained through careful manipulation and matching between the underlying computational architecture and the algorithm [61][11]. For instance, the Pease FFT formulation is preferred for scalable single-chip hardware FFTs mainly because of its data access scheme, which requires the same data permutation throughout all computational stages [58].
- The effect of formulations on distributed hardware implementation is evident in the results of some automated methods for general purpose DHA partitioning, yet no systematic strategy has been proposed to explore formulations as part of the mapping process [16].
- Recently, automated DST code generation methods for general purpose processors have been proposed, which use algorithmic-level transformations as part of their optimization process [44][45] . However, these methods have yet to be successfully adapted for automated partitioning methodologies on dedicated distributed hardware architectures (DHAs)

The proposed methodology, shown in Figure 3–2, provides a framework to facilitate the use of DST-specific properties to aid in their partitioning. Since algorithmic-level

transformations are contemplated as part of the optimization loop, Kronecker product algebra representation was chosen as DST specification language because it can compactly capture DST functionality and allows algorithmic-level transformations, yet allows easy interpretation of the implied hardware computational structures. Additionally, weighed hypergraphs were selected as representation format for the target DHA.

**Task 2: Tool development.** The main optimization loop explores alternative formulations of a DST, partitioning each of these formulations to assess its quality. The actual partitioning is performed on a dataflow graph representation of the current formulation. To evaluate the partition of any DST formulation, we developed the tools that deal directly with graph partitioning. The KTG, graph partition/placement, resource and latency estimators were developed and integrated as part of this task. Throughout the development of these tools, opportunities were observed to introduce graph level strategies, cost functions, and resource estimation techniques especially for DSTs.

**Task 3: Experimentation and assessment to define formulation exploration strategy.** To the best of our knowledge, the effect that DST reformulations have on their partitioning to DHAs has not been adequately reported in literature. However, understanding these effects is critical to define the heuristics controlling the optimization process. In this task, we used the KTG, graph partition/placement, and estimator tools to conduct experiments to assess the effects of reformulations on solution quality. Consequently, results from these experiments allowed us to define strategies for the formulation exploration heuristics.

**Task 4: Results and validation.** In this task, several experiments were performed to validate the performance of individual methodology processes and the methodology as a whole. The effectiveness of DST-related design decisions was ascertained by comparing results with alternative general-purpose techniques.

Finally, the scaled suboptimality of the proposed heuristic was measured to evidence that quality of results is maintained throughout increasing problem sizes.

## 3.3 Summary

In this chapter we formally stated the problem of high-level partitioning a discrete signal transform to a distributed hardware architecture (HLPDD), distinguishing it from the generic graph partitioning problem. Our approach to solving HLPDD requires designing new processes and adapting existing ones, as well as their integration into a partitioning methodology. A general scheme of DMAGIC, the proposed partitioning methodology, was discussed and the main tasks toward achieving a solution to the HLPDD problem were outlined. The next chapters detail the development of these tasks.

# CHAPTER 4

# Tools

A discussion of the development of the DMAGIC methodology can be divided in two stages; (1) the development of tools to partition a given DST formulation, and (2) experimentation with these tools to determine a formulation exploration technique. In this chapter, we discuss the former, starting with the process of converting an algorithmic formulation to a dataflow graph, followed by the partition of the DFG, and the latency and area estimators.

## 4.1 Kronecker Product Algebra

Kronecker Product Algebra (KPA) is a compact and practical manner of expressing sparse linear algebra algorithms, specially those consisting of recursive derivations, such as the fast versions of DSTs. The nomenclature used throughout this thesis for the expression of DSTs is the following: Kronecker product is denoted by $\otimes$, while direct sum is denoted by $\oplus$. $I_k$ represents an identity matrix size $k$, and $L_{n,m}$ a stride-$m$ size $n$ matrix. $J_k$ is the $I_k$ matrix with the order of the columns reversed. The operation $A^{L_{n,m}}$ denotes the conjugation of matrix $A$ by permutation $A^{L_{n,m}} = L_{n,n/m} A L_{n,m}$.

The next two sections discuss rules and properties of KPA and stride-permutations, which are necessary to visualize the computational structures and dataflow of DSTs, as well as to allow their reformulation. For more details about Kronecker

product and stride permutation properties, as well as their proofs the reader is referred to [52] and [66].

### 4.1.1 Definitions and Basic Rules

The Kronecker product of two matrices $B$ and $C$ of sizes $(k, l)$ and $(m, n)$ is defined by

$$B \otimes C = \begin{bmatrix} b_{(0,0)}C & b_{(0,1)}C & \cdots & b_{(0,l-1)}C \\ b_{(1,0)}C & b_{(1,1)}C & \cdots & b_{(1,l-1)}C \\ \vdots & \vdots & \ddots & \vdots \\ b_{(k-1,0)}C & b_{(k-1,1)}C & \cdots & b_{(k-1,l-1)}C \end{bmatrix} \tag{4.1}$$

The following are important Kronecker product properties that are used throughout this work to reformulate DSTs and understand their computational insights.

**Property 1.** *Associativity of Kronecker product*

$$(A \otimes B) \otimes C = A \otimes (B \otimes C) \tag{4.2}$$

**Property 2.** *Kronecker product transpose operation*

$$(A \otimes B)^T = A^T \otimes B^T \tag{4.3}$$

**Property 3.** *Kronecker product distribution property*

*Let $A$ and $C$ be $M \times M$ and $B$ and $D$ be $N \times N$ matrices. Then,*

$$(A \otimes B)(C \otimes D) = AC \otimes BD \tag{4.4}$$

**Property 4.** *Expression of Kronecker Product with identity matrices*

*Let $A$ be $M \times M$ and $B$ be $N \times N$ matrices. Let $I_M$ and $I_N$ be $M$ and $N$ dimensional identity matrices. Then,*

$$A \otimes B = (A \otimes I_N)(I_M \otimes B) = (I_M \otimes B)(A \otimes I_N) \tag{4.5}$$

## 4.2 Stride Permutations

In DST KPA formulations, permutation matrices act mainly to reorder data arrays between computational stages.

**Definition 4.2.1.** *Permutation matrix: A permutation matrix $P_N$ is an $N \times N$ matrix with all the elements either 0 or 1, with exactly one 1 at each row and column.*

**Property 5.** *Permutation matrices are orthogonal.*

*If $P_N$ is a permutation, then $P_N^{-1} = P_N^T$. Thus,*

$$P_N \cdot P_N^T = I_N \tag{4.6}$$

Stride permutations reorder data in regular arithmetic patterns. Their use in DST formulations is advantageous, as their implementation in software or hardware is simpler than permutations that do not follow a pattern.

**Definition 4.2.2.** *Stride Permutations.*

*The application of a p-stride permutation $L_{n,p}$ on a vector $X$ of size $n$ is:*

$$L_{n,p}x = \left[x_0, x_p, .., x_{(m-1)p}, x_1, x_{1+p}, .., x_{1+(m-1)p}, .., x_{p-1}, x_{(p-1)+2p}, .., x_{(p-1)+(m-1)p}\right]^T \tag{4.7}$$

*where $n = mp$.*

For example, for $x = [x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7]^T$

$$L_{8,2}x = [x_0, x_2, x_4, x_6, x_1, x_3, x_5, x_7]^T \tag{4.8}$$

and

$$L_{8,4}x = [x_0, x_4, x_1, x_5, x_2, x_6, x_3, x_7]^T \tag{4.9}$$

A stride permutation where $p = n/2$ is called a *perfect shuffle* permutation.

**Property 6.**

$$L_{n,1} = L_{n,n} = I_n \tag{4.10}$$

**Property 7.**

$$L^T_{n,p} = L_{n,\frac{n}{p}} \tag{4.11}$$

**Property 8.**

$$L_{a,bc} = L_{a,b} L_{a,c} \tag{4.12}$$

**Property 9.**

$$L_{a,abc} = (L_{a,ac} \otimes I_b)\,(I_a \otimes L_{bc,c}) \tag{4.13}$$

**Property 10.** *Factorization of stride-2 permutation.*

$$L_{2^{n+1},2} = \prod_{q=n-1}^{0} (I_{2^{n-q-1}} \otimes L_{4,2} \otimes I_{2^q}) \tag{4.14}$$

**Property 11.** *Factorization of* $L_{2^{n+K},2^{n+1}}$

$$L_{2^{n+K},2^{n+1}} = \prod_{q=0}^{n} \left(I_{2^{n-q}} \otimes L_{2^K,2} \otimes I_{2^q}\right) \tag{4.15}$$

Stride permutations allow the commutation of the Kronecker product.

**Definition 4.2.3.** *Definition, commutative property for tensor products. Let* $A$ *be* $a \times a$ *and* $B$ *be* $b \times b$ *matrices. Then,*

$$(A \otimes B) = L_{ab,a}\,(B \otimes A)\,L_{ab,b} \tag{4.16}$$

## 4.3 From Kronecker Products Algebra to Dataflow Graph

Automated search for a partitioning solution requires us to translate the formulation representation into one that graphically expresses the relationship between data and operations. One of the shortcomings of previous high-level partitioning schemes was their dependence on user-generated dataflow graphs as input [7] [8]. Creating DFGs by hand is only practical for smaller examples, since it is a tedious and error-prone process. Manual DFG creation discourages exploration into alternative DFG granularities for a given algorithm, thus limiting the potential of finding

more suitable expressions. Since our approach contemplates experimentation with multiple sizes and forms of DSTs, an automated KPA to DFG methodology was developed.

### 4.3.1 Problem Formulation

A KPA formulation of a $n$-point DST is a product of one or more $n$-row sparse matrices. These matrices may be either permutation matrices, which alter the order of data, or computational matrices, which linearly combine points of the input data set. The operations dictated by each computational matrix conform the DFG nodes. For instance, the sparse matrix $(I_4 \otimes F_2)$ translates to four nodes of type $F_2$.

Given a KPA formulation, the Kronecker to DFG converter will output an equivalent DFG that retains *data order topology*. A data order topology DFG is a directed acyclic graph $G = (V, E)$ with two functions $f_T$ and $f_L$. The function $f_T : V \rightarrow R$ maps each node $v \in V$ to a resource type. The function $f_L : V \rightarrow \mathbb{N}$ assigns each node $v \in V$ a non-negative integer that indicates the node's level within the computational topology. The level of each node in a sparse matrix is assigned in ascending order based on the matrix position of its corresponding operations. For example, a matrix $[(I_2 \otimes A_3) \oplus (I_3 \otimes B_2)]$ would have two $A_3$ nodes with levels 0 and 1, and three $B_2$ nodes with levels 2, 3, and 4. Figure 4–1 shows several representative examples of common sparse matrices found in KPA formulations and their corresponding data order topology DFGs. Node levels are shown in italics in the lower right corner of each box.

Figure 4–2 illustrates the concept of data order topology DFG for a DST KPA formulation. Even though both (b) and (c) are equivalent DFG's that represent the formulation in (a), (b) has retained data order topology by vertically organizing the processing blocks with respect to their corresponding matrix operations. Retaining this information is important for visualization and because it helps maintain regularity in the starting partitioning solution as we shall explain in Section 4.4.

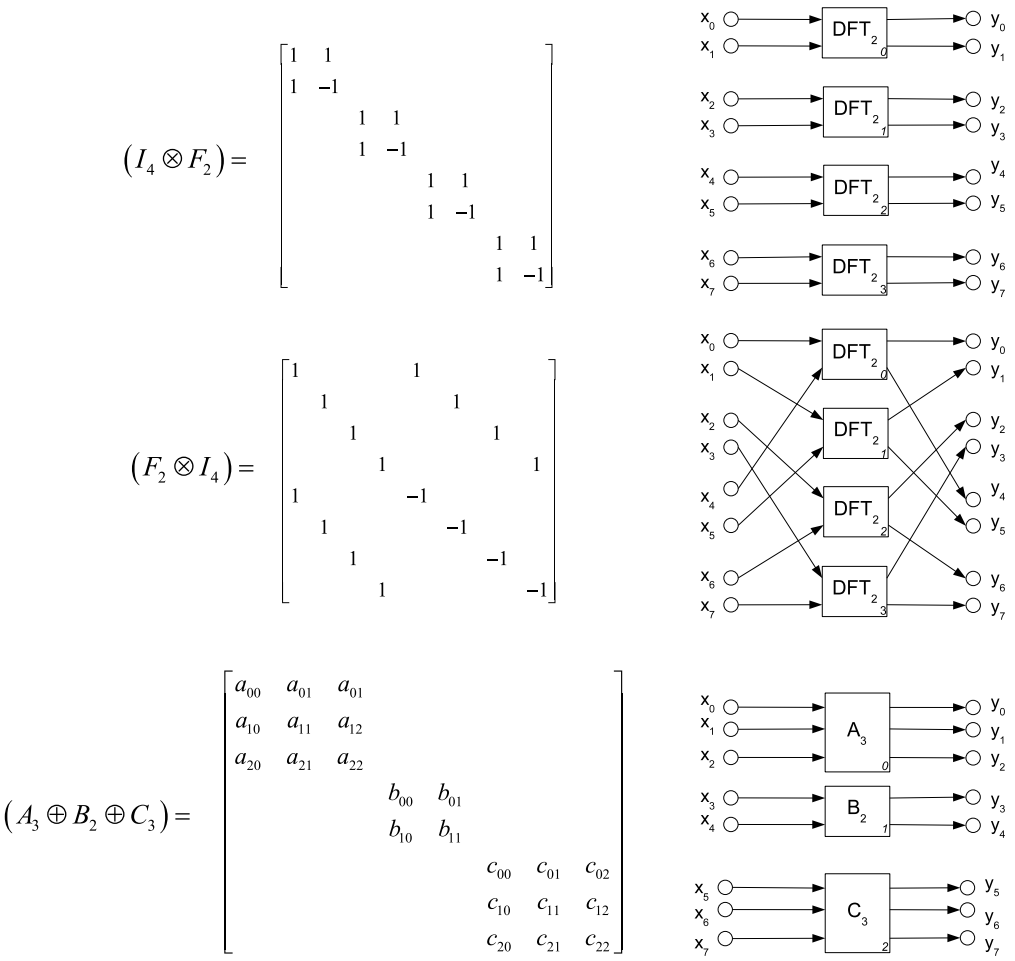



Figure 4–1: Several KPA sparse matrices and their corresponding data order topology DFGs.

### 4.3.2 Implementation

Implementation of the algorithm to translate from KPA to DFG is aided by the use of an intermediate data structure called a $KA - component$. Figure 4–3 illustrates this structure. A KA-component consists of three lists: input ports, devices, and output ports. Each input (output) port establishes a connection between a KA-component input (output) and a device input (output). Each device can be an *end* node or a pointer to another KA-component, making the KA-component a recursive data structure. The list of devices can be interpreted as being *horizontal* or *vertical*. A horizontal list symbolizes a cascade of matrix operations, such as in a sparse matrix multiplication (e.g. $P_{4,2}T_{4,2}$). A vertical list symbolizes a set of smaller

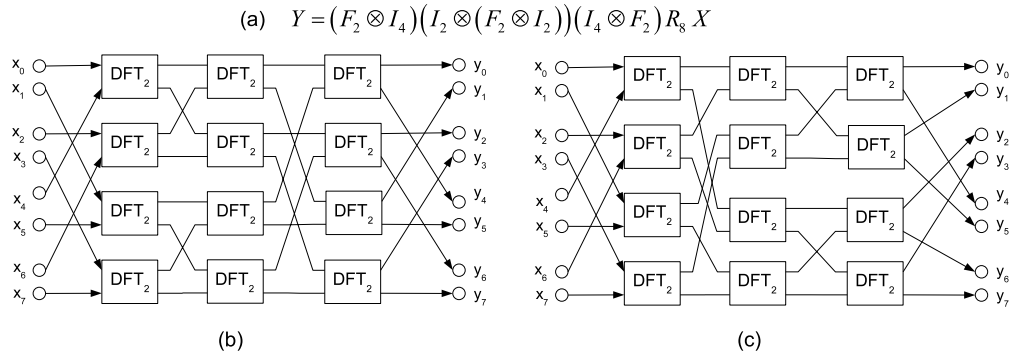



Figure 4–2: KPA formulation and two isomorphic dataflow graphs.

matrix operations acting on different subsets of a vector, such as in the operation $I_2 \otimes F_2$.

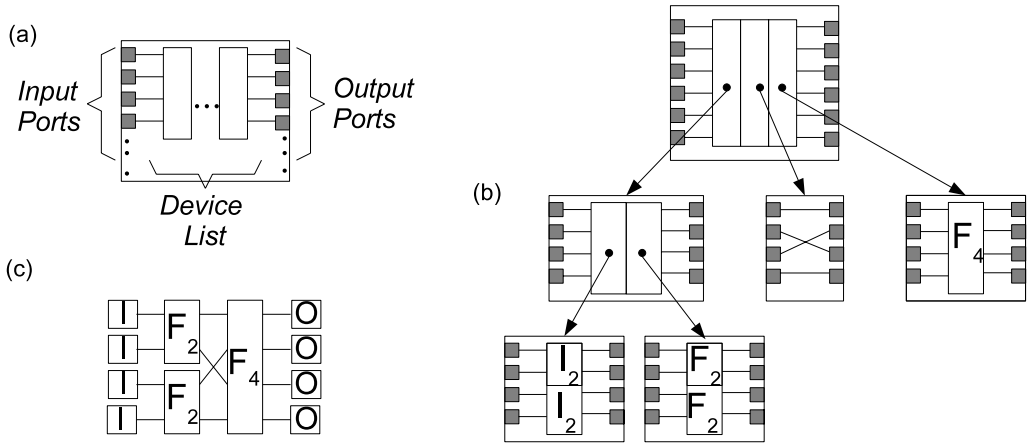



Figure 4–3: (a)The KA-component data structure, (b) KA-component representation and (c) derived DFG for sample formulation $F_4 P_{4,2}(I_2 \otimes F_2)$

Figure 4–4, illustrates how each of the common DST KPA expressions are representable using KA-components. The conversion process starts by parsing the KA formulation onto a queue. As the formulation's operands and operators are read sequentially from the queue, data structures, KA-components are created and linked to each other based on the operations and the levels of associativity in the expression. Once all operations and operands have been read from the queue, we have a hierarchical collection of KA-component that represent the expression. All that remains to create the DFG is to traverse the KA-components, creating nodes

for those components that imply hardware computation and obeying the established connections between them. For a comprehensive interpretation of tensor products and their effect on common permutation matrices please refer to Davio [66].

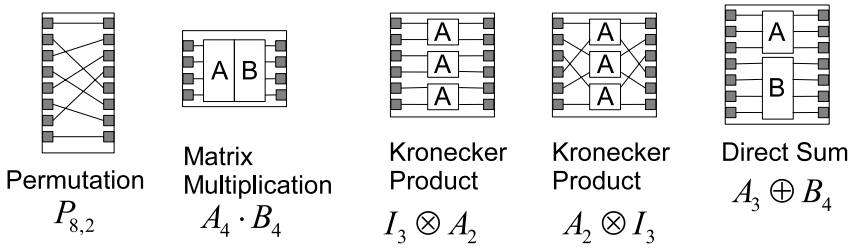



Figure 4–4: Common KPA operations and their KA-component representation.

To build the corresponding data flow graph, our algorithm performs a recursive traversal of the KA-component structures. This establishes the DFG nodes (which correspond to the end nodes in the k-components) and their connections. Each DFG node is assigned a weight that depends on the estimated area of its represented primitive. Figure 4–3c illustrates the data flow graph deduced from the KA-component representation in 4–3b. Not all components are converted to nodes, as they do not imply actual hardware computations.

## 4.4 Graph Partitioning

In this section, we introduce the graph partitioning problem, which is intimately related to our high-level partitioning objective. We begin by formally defining the graph partitioning problem and then we introduce several of the most commonly used algorithms for solving it. This is followed by a discussion of our implementation, which integrates several DST-specific properties to conduct a faster exploration of the graph-partitioning space.

### 4.4.1 Problem Formulation

The graph partitioning problem can be stated as follows. Given a graph, $G = (V, E, W_V, W_E)$, where $V = \{v_0, v_1, \ldots, v_{M-1}\}$ is the set of vertices, $E \subseteq [V]^2$ is the set of edges, $W_V = \{w_{V_0}, w_{V_1}, \ldots, w_{V_{M-1}}\}$ is the set of vertex weights, and

$W_E = \{w_{E_0}, w_{E_1}, \ldots, w_{E_{M-1}}\}$ is the set of edge weights, determine a partition $(V_0, V_1, \ldots V_{P-1})$ such that:

1. $V = V_0 \cup V_1 \cup \ldots \cup V_{P-1}$,
2. The sum of vertex weights in each partition is approximately equal, i.e. $\sum_{v_i \in V_0} w_{V_i} \simeq \sum_{v_i \in V_1} w_{V_i} \simeq \ldots \simeq \sum_{v_i \in V_{P-1}} w_{V_i}$
3. The sum of weights for the edges that join vertices in different partitions is minimized. In other words, minimize $\sum_{e_i \in C} w_{E_i}$, where $C = \{\langle v_i, v_k \rangle \,|v_i \subseteq P_a, v_k \subseteq P_b \,, a \neq b\}$.

### 4.4.2 Algorithms for Graph Partitioning

Graph partitioning is known to be a NP-complete problem [32], even for the simplest case of $V = V_0 \cup V_1$, commonly known as *bipartitioning* or *graph bisection*. Therefore, practical partitioning strategies rely on heuristics rather than on exact algorithms. Heuristic partitioning mechanisms search the solution space by using either deterministic or stochastic strategies. Sections 4.4.3 through 4.4.7 review common graph partitioning terms, as well as four commonly used graph partitioning heuristics.

### 4.4.3 Preliminaries

The Kernighan-Lin, Fiduccia-Matheyses, simulated annealing and genetic algorithms are move-based strategies: they optimize a partitioning solution by iteratively moving one or several nodes from their partitions and measuring the effect on a global cost function. Figure 4–5 illustrates some common terms related to the operation of move-based partitioning heuristics. A *cut edge* is one that connects two vertices that are currently on different partitions, for example $\langle a, b \rangle$ in Figure 4–5(b) is a cut edge. The cost function for the bi-partitioning problem is the *cut weight*, i.e. the sum of the weights of cut edges. The *gain* of a perturbation on the partitioning solution is the difference in cost before and after a move is performed,

$$gain = cut\_weight_{before} - cut\_weight_{after} \tag{4.17}$$

i.e. $gain = 5$ for swapping nodes $a$ and $b$.

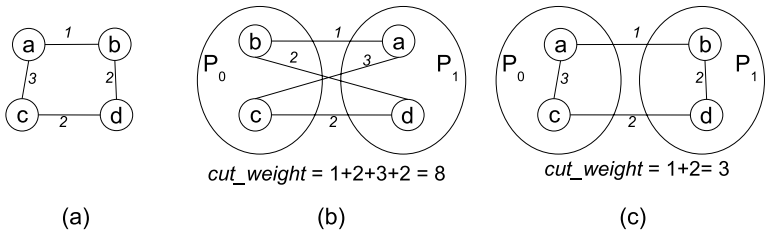



Figure 4–5: (a) Graph (b) a partition solution, (c) partition solution after swapping nodes $a$ and $b$.

### 4.4.4 Kernighan-Lin Bipartitioning Heuristic

Kernighan and Lin's bipartitioning algorithm, introduced in 1970, is considered by many to be the first "good" bi-partitioning heuristic [67] [33]. The KL algorithm, as shown in Algorithm 2, iteratively improves on the current solution by swapping pairs of nodes between partitions. Each iteration of the outer loop of the KL algorithm is called a *pass*. During each pass, each node is swapped once from its current partition. The inner loop helps to establish the order in which swaps are performed. During each iteration of the inner loop, the algorithm chooses the unlocked node pair $\langle u, v \rangle$ whose swap has the highest gain. The nodes are swapped, the node pair $\langle u, v \rangle$ and its corresponding gain are added to the sequence of performed swaps during this pass, and $u$ and $v$ are locked for the rest of the pass. The gains of any nodes connected to $u$ or $v$ are updated. At the end of each pass, the algorithm evaluates the swap sequence list to determine at which step (if any) in the swap sequence the highest positive cumulative gain was achieved. Swaps that happened after this step are reversed (i.e. $u$ and $v$ are returned to their original partitions) and the solution cost is adjusted according to the cumulative gain. Nodes are unlocked and the next pass starts. This continues until a pass does not offer a positive gain over the current cost.

**Algorithm 2** Kernighan-Lin bipartitioning heuristic

**Input**: Graph $G = (V, E)$
**Output**: Partition $\{A, B\}$

1. Obtain initial linear horizontal balanced partition $\{A, B\}$ such that $A \cup B = V$, $A \cap B = \{\}$, and $|A| \approx |B|$
2. **do**
   - 2.1. Compute $D_v$ for all $v \in V$
   - 2.2. $queue \leftarrow \phi$; $i \leftarrow 1$; $A' = A$, $B' = B$;
     - 2.2.1. **while** $A'$ and $B'$ are not empty
       - 2.2.1.1. Choose node pair $a_i \in A'$, $b_i \in B'$ with highest swap gain $g_i$
       - 2.2.1.2. $queue \leftarrow queue + (a_i, b_i, g_i)$
       - 2.2.1.3. $A' = A' - \{a\}$; $B' = B' - \{b\}$;
       - 2.2.1.4. Update $D_v$ for all $v \in A' \cup B'$ connected to $a_i$ or $b_i$
       - 2.2.1.5. $i \leftarrow i + 1$
     - 2.2.2. Find $k$ to maximize $G = \sum_{i=1}^{k} g_i$
     - 2.2.3. **if** $G > 0$ **then**
       - 2.2.3.1. Move $\{a_1, \ldots, a_k\}$ to $B$, and $\{b_1, \ldots, b_k\}$ to $A$
     - 2.2.4. **endif**
3. **while** $G > 0$

The gain computation for each swap pair is done by observing that the gain of moving a node $a \in A$ to $B$ is given by:

$$D_a = E_a - I_a, \tag{4.18}$$

where $E_a$ is the sum of weights of the edges that emerge from node $a \in A$ and terminate in $B$:

$$E_a = \sum_{e_i \in \{\langle a,b\rangle | b \in B\}} W_{E_i} \tag{4.19}$$

and $I_a$ is the sum of weights of the edges that emerge from node $a \in A$ and terminate in other nodes in $A$:

$$I_a = \sum_{e_i \in \{\langle a,v\rangle | v \in A\}} W_{E_i} \tag{4.20}$$

The swap gain of $\langle a, b\rangle$, where $a \in A$ and $b \in B$ can be computed using the following expression:

$$g_{ab} = D_a + D_b - 2W_{E_{<a,b>}} \tag{4.21}$$

Furthermore, every time a pair of nodes is swapped, the potential gains obtained for swapping other nodes may change. This change to the $D$-values can be computed in constant time:

$$D'_x = D_x + 2W_{E_{<x,a>}} - 2W_{E_{<x,b>}}, \forall x \in A - \{a\} \tag{4.22}$$

$$D'_y = D_y + 2W_{E_{<y,b>}} - 2W_{E_{<y,a>}}, \forall y \in B - \{b\} \tag{4.23}$$

A straightforward implementation of the Kernighan-Lin heuristic requires $O(n^3)$ time per pass. The selection of the node pair with the highest swap gain constitutes the most expensive step. The inner loop must be repeated $O(n)$ times, choosing among $O(n^2)$ swap pairs each iteration. In actual implementations, a per-pass complexity of $O(n^2 \log n)$ is achieved by maintaining a nonincreasing sorted list of the $D$-values for each partition. Since the number of $D$-values is linear in size, the time to update each list is $O(n \log n)$. Finding the maximum $g_{ab}$ rarely requires examination of all pairs $\left(D_{a_i}, D_{b_j}\right)$. This is because when examining the lists, once we come upon a pair $(D_{a_k}, D_{b_l})$ such that $D_{a_k} + D_{b_l}$ is less than the best gain $g_{ij}$ seen so far, there cannot be another pair $k \geq i$, $l \geq j$ with greater gain.

### 4.4.5 Fiduccia-Mattheyses

Fiduccia-Mattheyses' algorithm enhances the Kernighan-Lin heuristic in two main aspects: the ability to handle *hypergraphs* and the reduction of the per-pass time complexity [68]. A hypergraph is a pair $(V, E)$ of sets, where the elements of $E$ are non-empty subsets of any cardinality of $V$. Thus graphs are a special class of hypergraphs with cardinality of 2. Hypergraphs are a more appropriate abstraction for structural level circuit netlists since each pin of a circuit block $a \in A$ can be connected to one or more pin(s) of other blocks $b, c \in B$ [69]. As illustrated in Figure 4–6, cut weight for the graph representation implies two separate signals, while hypergraph representation correctly implies a common signal to from $a$ to $b$ and $c$.

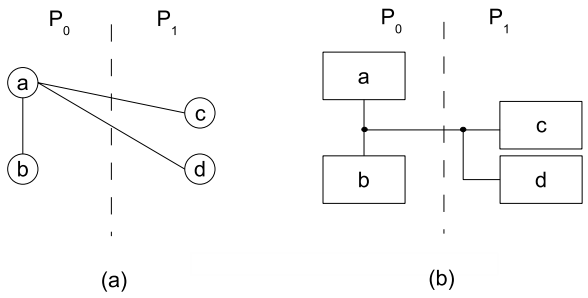



Figure 4–6: (a)Graph and (b)hypergraph representations of a circuit netlist.

Fiduccia-Mattheyses' algorithm requires $O(|E|)$ time per pass for graphs and $O(|p|)$ for hypergraphs , where $p$ is the sum of cardinalities for elements of $E$ (i.e. the total number of pins in the netlist). Similarly to KL, FM operates in passes and every node is swapped once per pass, however at each step FM exchanges one node at a time, instead of a pair. Thus, the decision of which node to exchange next can be based on single node gains ($O(n)$) instead of node pairs ($O(n^2)$). Since values for single node gains are bounded by $-deg_{max}$ and $deg_{max}$, a linked-list vector structure called a *bucket list* can be used to efficiently maintain and update the gains and detect the highest gain throughout the procedure. Figure 4–7 illustrates a gain bucket structure. At the beginning of a pass, node gains are computed in ($O(p)$) time and linked to their corresponding gain slot, meanwhile recording the maximum gain. Direct access is also kept to each bucket cell by using the CELL structure. This allows constant-time access to the maximum gain cell and to each cell for gain updates. Every time a node is changed to its complementary partition, all incident node gains are updated, which implies that a total of ($O(p)$) cells will be updated each pass.

### 4.4.6 Simulated Annealing

Simulated annealing (SA) is a combinatorial optimization technique that has been successfully applied to various tasks in electronic design automation, mainly in placement and partitioning [70] [71] [72]. Proposed by Kirkpatrick et al., SA's optimization strategy is based on the physical process of annealing, which achieves

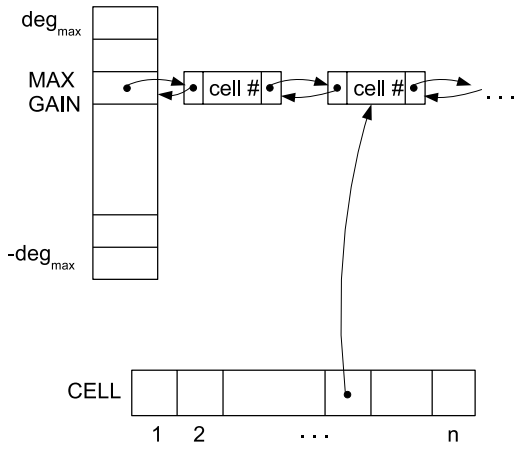



Figure 4–7: Bucket structure used in the Fiduccia-Mattheyses algorithm [68]

low energy states in metals [73]. First, the metal is heated to a *melting temperature* that is high enough to break atoms from their chemical bonds and allow them to move freely. This is followed by a *cooling schedule* in which the atoms arrange themselves in a low energy - highly ordered state. In an analogous procedure, simulated annealing explores the solution space by allowing high probability of hill-climbing during the initial stages of exploration followed by a gradual decrease in probability as the exploration progresses.

Each step in the iterative improvement strategy of SA proposes a new solution by randomly perturbing the current configuration. The cost of the new configuration is computed and the new solution is accepted or rejected according to the following criteria, commonly called the *Metropolis criteria*:

$$P_{accept} = \begin{cases} 1, & \Delta C \leq 0 \\ e^{-\frac{\Delta C}{T}}, & \Delta C > 0 \end{cases}, \tag{4.24}$$

where $T$ is the temperature and $\Delta C$ is the cost variation in the current iteration. This acceptance rule establishes the probability of hill climbing throughout the solution space exploration, which gives SA its ability for escaping local minima.

Algorithm 3 outlines the SA heuristic. An initial temperature and solution are set. The outer loop controls the monotonically decreasing temperature schedule. At each temperature, a number of moves are attempted until a certain *equilibrium* criteria has been achieved. The cooling schedule proceeds until a *stop criteria* is met. Although the core SA heuristic is conceptually simple, its actual implementation requires careful and extensive experimentation and/or a sufficient familiarity about the problem at hand to define the following specifics:

- The initial temperature, also called the *melting temperature*, determines the initial acceptance probability of hill climbing configurations. Choosing a $T_M$ that is too high will destroy any possible good qualities in an initial solution and might unnecessarily delay convergence, while choosing a value too small might not induce the necessary freedom of exploration and be prone to local minima [74].
- The inner loop (equilibrium) criterion controls the number of configurations explored at each temperature step. This criterion determines when the system has reached a steady-state, a condition necessary to guarantee appropriate solution convergence.
- The temperature decrement function establishes the rate at which the temperature is decreased. To guarantee an effective rate, either of two methods are commonly used: a predefined decrease rate schedule which is defined through experimentation, or a rate that dynamically responds to exploration statistics in previous steps.
- The stop criterion decides when the exploration has converged to a point where further reductions in temperature are not expected to have a significant impact on the solution. This is commonly deduced by measuring the rate of change in solution cost vs. rate of temperature change. When the ratio falls below a certain value, the exploration process is stopped.

**Algorithm 3** Simulated annealing heuristic.

**Input**: $S_0$: initial solution, $T_0$: initial temperature
**Output**: $S$: optimized solution
1. $S \leftarrow S_0$
2. $T \leftarrow T_0$
3. **while** stop criterion not satisfied
   3.1. **while** equilibrium not reached
      3.1.1. $S' \leftarrow Perturb(S)$
      3.1.2. $\Delta C \leftarrow Cost(S') - Cost(S)$
      3.1.3. $Probability(\Delta C) = \min\left(1, e^{-\frac{\Delta C}{T}}\right)$
      3.1.4. **if** $Random\,(0,1) \leq Probability(\Delta C)$ **then**
         3.1.4.1. $S \leftarrow S'$
      3.1.5. **end if**
   3.2. **end while**
4. $T \leftarrow Decrement(T)$
5. **end while**

### 4.4.7 Genetic Algorithms

A genetic algorithm (GA) is a general optimization technique based on the metaphor of natural evolution [75]. GA's have been used in placement, partitioning and scheduling problems [76] [77] [78] [7] [79]. Natural species evolve to become better adapted to their environment throughout generations of selective reproduction. Each individual in a population has features that determine its fitness, i.e. how well it is able to survive in its environment. Each feature is genetically controlled by a basic unit called a gene. The sets of genes controlling features are called chromosomes. On each generation, pairs of individuals mate and reproduce, inheriting a combination of their genes to their offspring. Better fitted individuals have a greater chance of reproducing, thus new generations are potentially more fit than the previous.

GA maps an optimization problem to that of finding the most fit individual after several generations of an evolution process [13]. In GA, candidate solutions to the optimization problem are analogous to the individuals of a population. Each candidate solution is called a *chromosome* and is composed of a string of *genes* representing the values for the various problem variables. For example, as shown in

Figure 4–8, a common way of encoding the solution of a bipartitioning problem is an integer array where each cell indicates the partition assigned to a certain node.

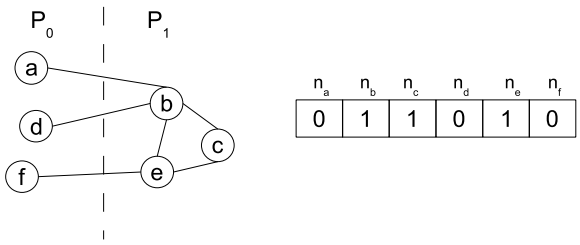



Figure 4–8: Example of a chromosome encoding for a bipartitioning solution.

Algorithm 4 shows the GA heuristic. During each iteration, a new generation is created by selecting members of the current population, crossing pairs of them to generate offspring, and randomly introducing mutations to chromosomes of the new generation. The selection process probabilistically selects some of the members of the current population to serve as parents for the next one. It emulates the survival of the fittest principle by mostly selecting highly fit individuals. Crossover creates offspring with a combination of genes from both parents. Mutation randomly modifies the structure of some chromosomes. This introduces variations that can help the optimization heuristic in hill-climbing from local minimum.

**Algorithm 4** Generic algorithm optimization heuristic.

**Input**: , $P_S$: Selection percentage, $P_C$: Crossover percentage, $P_M$: Mutation percentage
**Output**: $S$: optimized solution

1. $P \leftarrow$ **CreateInitialPopulation**()
2. **while** stop criterion not satisfied
   - 2.1. **for** every member $m \in P$
     - 2.1.1. compute **Fitness**($m$)
   - 2.2. Sort $P$ in decreasing order of fitness
   - 2.3. *best_solution* $\leftarrow$ first member of sorted $P$
   - 2.4. $b_s \leftarrow$ **Select**$(P, P_M)$;
   - 2.5. $b_c \leftarrow$ **CrossOver** $(b_s, P_C)$;
   - 2.6. $P \leftarrow$ **Mutate**$(b_s \cup b_c, P_M)$
3. **end while**

### 4.4.8 k-way Partitioning

In their original paper, Kernighan and Lin suggested two adaptations of their bipartitioning heuristic for $k$-way partitioning [67]. The first method required the creation of an initial $k$-way partition, followed by the repeated application of the bipartitioning heuristic to pairs of subsets. As observed by the authors, although this method might achieve pairwise optimality, this is only one condition for *global* optimality. The second method relied on the recursive use of the bipartitioning algorithm. For example, for a 4-way partition, KL is used to bipartition the original graph. Then, each of the resulting partitions is split using KL, for a total of four partitions. As pointed by the authors, this scheme might suffer from several problems, especially because earlier partitions have an impact on later ones. For instance, the first split tries to minimize the number of cuts by tending to maximize the connections inside the partitions. This, in turn, can have an adverse impact on the quality of subsequent partitions.

L. Sanchis proposed one of the first widely cited and compared adaptations of the Fiduccia-Mattheyses' heuristic for multi-way partitioning [80]. Rather than using a recursive or pairwise approach, the author's adaptation improves the partition uniformly at each step. During a pass, all possible moves of a free cell from its current partition to any other partition are considered, choosing the move that represents the highest gain. The uniform partitioning algorithm by Sanchis found lower cutsets than the recursive (hierarchical) approach under similar execution time constraints. Extension of FM bipartitioning to a uniform k-way implementation, requires the adaptation of gain computations, gain vector maintenance, balancing and the bucket array data structures, resulting in additional space and time complexity.

Hauck used the recursive bipartitioning method for structural partitioning of circuit netlists to multi-FPGA topologies [9]. His algorithm determines the order to perform bipartitioning by finding the critical bottlenecks in the topology, while

ensuring that all partitions created are connected. In his dissertation, the computed bipartitioning ordering is given for several typical multi-FPGA topologies. Topologies with crossbar connecting all components are identified as an opportunity for using uniform multi-way partitioning. However, Hauck's method does not ultimately use uniform partitioning for these topologies. His argument is that typical uniform algorithms do not consider individual channels but instead choose objective functions such as the total number of nets connecting logic in two or more partitions (the net-cut metric), or the total number of partitions touched by each of these cut nets (the pin-cut metric). Our method utilizes an objective function that instead of counting the total number of cuts utilizes a vector of cuts.

Even though for bipartitioning FM and KL may produce similar results, when used for $k$-way partitioning the differences in their optimization decisions become more evident. In FM, when only two partitions are present, if balance is to be kept throughout the optimization process, every move of a node from $P_0$ to $P_1$ is followed by moving a node in the opposite direction. Thus, for all effective purposes, node pairs are being swapped, as in KL. On the other hand, when dealing with more than two partitions in FM, the effect of swapping node pairs is not always maintained. For instance,think of 4-way partition and the D values for all nodes. The first node chosen, say $a \in P_1$ may have its greatest gain when moved to partition $P_2$. However, none of the nodes in $P_2$ might have a good gain. Thus, may be a node from $P_3$ could be moved, and so forth. The differences between $k$-way KL and FM also extend to the move commitment rules. As previously explained, on every pass of KL and FM all nodes are moved once. At the end of each pass, the sequence of moves that produced the highest cumulative cost gain is committed, the remaining moves are reversed. Thus, for example, in FM if the only move that contributed gain is moving $a$ from partition $P_1$ to partition $P_2$ we might not be able to commit this swap since it

will leave the solution with an unbalance. On the contrary, in KL a swap involving node $a$ would be committed, as long as it represents a positive gain.

## 4.5 $k$-way Implementation

In this section, our implementation of a k-way partitioning heuristic is discussed. We begin by discussing the cost function. Then, we discuss several DST considerations that were taken into account for the implementation. Finally, the complexity for the heuristic is analyzed.

### 4.5.1 Cost Function

A limitation of previous $k$-way extensions of deterministic partitioning heuristics is that their objective function does not consider individual channels but instead chooses cumulative objective functions such as the total number of nets connecting logic in two or more partitions (the net-cut metric), or the total number of partitions touched by each of these cut nets (the pin-cut metric). These cumulative functions are useful in the context of ASICs where the designer might want to minimize the total number of physical wires crossing chip boundaries. However, for partitioning to distributed hardware architectures, a cumulative function presents several problems:

- *It assumes complete connectivity between partitions*: In a DHA, not all devices have a direct connection to each other. A DHA will commonly have different types of connections between various devices. Adjacent devices might have a direct point to connection, whereas distant devices might rely on a communication channel such as a crossbar or multiple-device data hopping for data interchange. Thus, the cost of communicating varies across certain partition boundaries. Traditional cumulative objective functions do not take this fact into account and simply count the number of cut nets regardless of the partitions they join.
- *DHAs have 'hard' interconnect resources*: When communication resources are hardwired and scarce, as they are in most DHAs, a *balanced distribution* of cuts throughout the offered resources is a better objective than the minimization of the

sum of cuts. This becomes more evident when we consider that communication of data will be scheduled throughout time. A cumulative function might minimize total cuts by concentrating all cuts on a few partition boundaries. These partition boundaries will then become the implementation bottleneck. Figure 4–9 exemplifies this situation. The *cumulative* objective function in Figure 4–9(a) is smaller than in (b), i.e. 29 vs. 39. However, distribution of cuts is more balanced in (b), promoting a better communication distribution throughout execution, and thus decreasing latency.

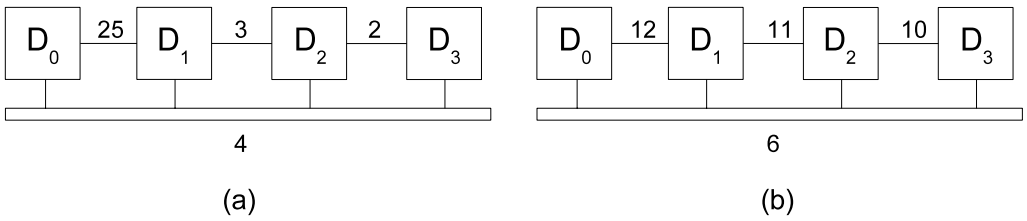



Figure 4–9: Effect of cut distribution on a DHA.

Our P/P process estimates solution latency using a cost function that measures the impact of communications on each individual DHA communication resource. The cost of a solution is represented by a vector:

$$P = \langle \Phi_0, \Phi_1, \ldots \Phi_{M-1} \rangle \,, \tag{4.25}$$

where $\Phi_i$ is the cost of communications through channel $i$. Let $P_1$ and $P_2$ be costs, we say $P_1 < P_2$ if the nonincreasing ordering of $P_1$ is lexicographically smaller than the ordering of $P_2$. For example, $P_1 = \langle 1, 3, 1, 4 \rangle$ is smaller than $P_2 = \langle 0, 2, 3, 4 \rangle$, since, lexicographically 4,3,1,1 is smaller than 4,3,2,0. Each channel cost communication $\Phi_i$ is obtained as follows:

$$\Phi_i = W(c_i) \sum_{e \in E} R(e, c_i), \tag{4.26}$$

where $W(c_i)$ is the weight of channel $c_i$; a function $W : C \rightarrow \mathbb{Z}^+$, whose value is a relative measure of the impact on system latency of communicating a data

point through $c$. The communication flag $R(e, c_i)$ of an edge $e$ through a channel $c_i$ determines if the communication represented by $e$ will be done through $c_i$.

$$R(e, c_i) = \begin{cases} 1 & \text{if } c_i = \mu(e), \\ 0 & \text{else.} \end{cases} \tag{4.27}$$

Let $a(x)$ represent the partition to which node $x$ is assigned. Let $J_e$, where $e = \langle u, v \rangle$ be the set of channels that can be used to communicate data from $a(u)$ to $a(v)$. The minimum weight channel $\mu(e)$ is the channel $c_i \in J_e$ with the minimum weight.

The described changes to the cost function require modifications to the gain computation and maintenance. The $D$ value for moving a node $x$ is no longer a scalar, as in bipartitioning, but a collection of $k-1$ vectors each representing the effect of moving $x$ to one of the remaining $k-1$ partitions. The $D$ vector for moving a node $x$ from $P_x$ to another partition $P_y$ is given by:

$$D_{x,P_y} = \left\langle D_{x,P_y,c_0}, D_{x,P_y,c_1}, \ldots, D_{x,P_y,c_{M-1}} \right\rangle , \tag{4.28}$$

where $D_{x,P_y,c_i}$ is the cost effect on channel $c_i$ of moving node $x$ from $P_x$ to another partition $P_y$, and can be computed as follows:

$$D_{x,P_y,c_i} = F_{x,c_i} - T_{x,P_y,c_i} \tag{4.29}$$

Assuming that channel $c_i$ is the minimum weight channel between partitions $P_a$ and $P_b$:

$$F_{x,c_i} = \begin{cases} \sum_{\langle x,u \rangle | u \in P_b} W(c_i) & \text{if } P_a = P_x, \\ 0 & \text{else.} \end{cases} \tag{4.30}$$

$$T_{x,P_y,c_i} = \begin{cases} \sum_{\langle x,u\rangle | u \in P_a} W(c_i) & \text{if } P_b = P_y, \\ 0 & \text{else.} \end{cases} \tag{4.31}$$

The gain obtained by swapping two nodes $x$ and $y$ between partitions $P_x$ and $P_y$ on a channel $c_i$ is computed by:

$$G_{x,y,c_i} = D_{x,P_y,c_i} + D_{y,P_x,c_i} - \Theta, \tag{4.32}$$

where

$$\Theta = \begin{cases} 2W_{E_{\langle a,b\rangle}} & \text{if } P_a = P_x \text{ and } P_b = P_y\ , \\ 0 & \text{else.} \end{cases} \tag{4.33}$$

In this manner, each of the values for the gain vector of node pair $x, y$ are obtained:

$$G_{x,y} = \left\langle G_{x,y,c_0}, D_{x,y,c_1}, \ldots, D_{x,y,c_{M-1}} \right\rangle\ , \tag{4.34}$$

On each iteration, the next chosen swap pair will be the one whose gain vector combined with the current cost vector obtained the lowest cost vector (as previously defined).

### 4.5.2 DST Considerations in Graph Partitioning

Our partition/placement algorithm, KL-MH, is a uniform $k$-way partition heuristic for heterogeneous-channel topologies. It complements the basic Kernighan-Lin bipartition heuristic with several considerations derived from DST flow-graph characteristics, in an effort to improve optimization convergence and solution quality. The main considerations are discussed next:

**Linear partitioning**: Fast DST algorithms, because of the regularity of their dataflow and inter-stage data dependence, have traditionally been partitioned vertically or horizontally [81][58]. *Vertical* partitioning maps computation as in an

architectural-level pipeline, where one or more complete DST computational stages are assigned to each hardware device. In *horizontal* partitioning, each device carries out all stages of computation for a data subset, similar to single-instruction-multiple-data (SIMD) processing. We argue that, for the type of architecture we are targeting, horizontal partitioning will obtain lower latencies than the vertical scheme. To begin with, in vertical partitioning, all data enter through the first device and leave through the last. Thus, the potential of using each device as an input (reading new data from its own local memory) is lost. Furthermore, in a pipeline scheme, *every* data point must cross through each of the channels, while in a horizontal scheme even a naïve linear partitioning can reduce data communications requirements in half. When communication channels are the system's bottleneck, as they are in DHA systems, the excess communication in pipeline implementations hinders performance. Based on this reasoning, our partition/placement algorithm explores the solution space by exclusively considering horizontal partitioning schemes.

**Linear initial partitions**: The concept of balanced linear horizontal partitions is illustrated in Figure 4–10 by the dashed horizontal lines. The initial linear horizontal partitions are obtained using Algorithm 5. Given a DST's data-flow graph, the initial partition scheme is determined by dividing the structure horizontally into $k$ equally weighted partitions, respecting the DST's data order topology.

Common formulations of DSTs have corresponding DFGs which cluster highly interconnected subgraphs in such way that a balanced linear horizontal partition represents a good solution. For this reason, in KL-MH we use this method, rather than randomly obtained initial solutions. As later presented on Chapter 6, on average, better solution quality was achieved when using this initial partition mechanism as compared to random initial solutions.

**Algorithm 5** Algorithm to obtain linear horizontal partitions.

**Input**: DFG $G = (V, E)$ with data order topology, $k$: the number of partitions
**Output**: Horizontal linear partition $\{V_0, V_1, .., V_{k-1}\}$, where $V_i \in V$ and $V_0 \cup V_1 \cup \ldots \cup V_{k-1} = V$

1. $V_0 \leftarrow \emptyset$, $V_1 \leftarrow \emptyset$, $\ldots V_{k-1} \leftarrow \emptyset$
2. **For every** computational column $C \in V$
   - 2.1. Sort $C$ in order of increasing node levels.
   - 2.2. $W_{CC} = \sum_{c \in C} w(c)$
   - 2.3. Determine indexes $(p_0, \ldots, p_{k-2})$ such that $\sum_{i=0}^{p_0} w(c_i) \approx \sum_{i=p_0+1}^{p_1} w(c_i) \approx \ldots \approx \sum_{i=p_{k-2}+1}^{|C|-1} w(c_i) \approx \frac{W_{CC}}{k}$
   - 2.4. $V_0 = V_0 \cup \bigcup_{i=0}^{p_0} c_i$, $V_1 = V_1 \cup \bigcup_{i=p_0+1}^{p_1} c_i$, $\ldots$, $V_{k-1} = V_{k-1} \cup \bigcup_{i=p_{k-2}+1}^{|C|-1} c_i$
3. **End For**

**Schedule compactness**: A common graph structure found in fast DST algorithms is the butterfly network (BN) [82]. As illustrated in Figure 4–10, canonical formulations of the FFT are isomorphic to BNs when each node represents a 2-point butterfly ($F_2$) and successive twiddles. Other DSTs such as the DCT and the DHT are commonly formulated using BN-like structures [81][83][1]. Schedule-wise the BN is a completely rigid structure, since all paths of its computational stages are isometric and a delay in the computation of any node means a delay to the completion of processing as a whole. We take this into account as part of the partitioning process by only considering solutions that exchange nodes from the same computational stage. This consideration entails a more focused partition solution exploration and a simplified subsequent scheduling.

Algorithm 6 shows how the KL-MH algorithm integrates the cost function and DST-specific considerations. Step 1 obtains a balanced linear partition as an initial solution. Inside each internal iteration, step 2.1 chooses a swap pair of nodes from the same computation stage. Throughout the rest of the algorithm, the decisions

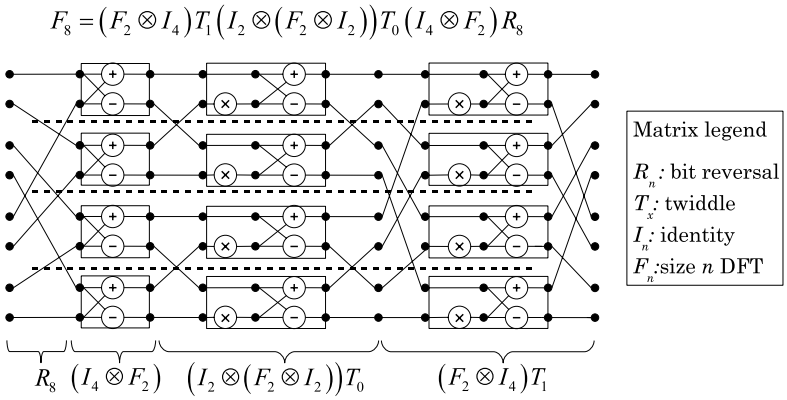



Figure 4–10: 8-point DFT Cooley-Tukey formulation, showing initial linear horizontal partition.

as to which node pairs to swap and which sequence of swaps to commit are taken based on the cost function defined in Section 4.5.1.

### 4.5.3 Complexity

Similarly to the bipartitioning KL heristic, selection of the node pair with the highest swap gain constitutes the most expensive step of KL-MH. Assuming that $n$ is the number of nodes in the DST dataflow graph, the inner loop must be repeated $O(n)$ times, choosing among $O(n^2)$ swap pairs each iteration. This amounts to a time complexity of $O(n^3)$.

The impact on time complexity of choosing nodes from the same computational stage can be visualized by analyzing complexity in terms of DST points. The dataflow graph of an $N$-point fast DST consists of $O(N \log_2 N)$ nodes. If arbitrary pairs of nodes are allowed, a total of $O(N^2 (\log_2 N)^2)$ pairs must be considered, resulting in a complexity of $O(N^3 (\log_2 N)^3)$. When node pairs can only be formed by nodes in the same computational stage, each node can be paired with $O(N)$ others, resulting in a total complexity of $O(N^3 (\log_2 N)^2)$.

In both complexity approaches just presented, actual running time depends on the number of DFG nodes, which for a given DST size can vary depending on the *granularity* of the formulation. A coarser formulation will be represented by a smaller

**Algorithm 6** KL-MH: Adapted KL algorithm for $n$-way heterogeneous channel architectures.

**Input**: Data Flow Graph $G(V, E)$, Info about target architecture: devices $D = \{P_0 \dots P_{M-1}\}$, communication channels $C = \{C_0, \dots, C_{N-1}\}$ and their weights $W = \{W_0 \dots W_{N-1}\}$

**Output**: Partition assignment for each $v \in V$.

1. Initial balanced partition $|P_0| \cong |P_1| \cong \dots |P_{M-1}|$
2. **while** not all nodes are locked
   - 2.1. Determine $a_i, b_i \in V$ s.t. $a_i$ and $b_i$ are in same computational stage, and $p(a_i) \leftrightarrow p(b_i)$ minimizes $cost()$
   - 2.2. Perform swap $p(a_i) \leftrightarrow p(b_i)$. $C_i \leftarrow cost()$
   - 2.3. Lock $a_i, b_i$
   - 2.4. $[a_i, b_i, C_i] \rightarrow queue$
   - 2.5. $i \leftarrow i + 1$
3. **end while**
4. Choose $k$ s.t. $C'_k = \min_{0<j<i} (C_j)$
5. Reverse all swaps $a_j, b_j$ where $j > k$
6. **if** $C'_k < \mathrm{C_k}$ **then**
   - 6.1. $C'_k \leftarrow \mathrm{C_k}$, unlock nodes, goto step 2
7. **else** stop

$cost() = \max_{i \in [0, N-1]} [R_i \times W_i]$,
where $R_i$ = number of required communications through $C_i$

number of nodes and will require less exploration time. However, the quality of the partition solution might be compromised as the coarse nodes encapsulate certain nodes which might be needed to obtain better solutions. This situation is exemplified in Figure 4–11, which shows partitioning solutions for two formulations of the 16-point FFT. Figure 4–11(a) has fewer nodes and required a smaller execution time, whereas (b) obtained a better solution quality (as measured by the cost function) thanks to the mobility of certain nodes which had been clustered as coarser nodes in (a). It must be mentioned that coarser granularity does not necessarily imply poorer results. In fact, as reflected by results presented in Chapter 5, a good match between the granularity of a formulation and the decisions made by the partitioning heuristic can allow coarser formulations to have better solutions than finer ones.

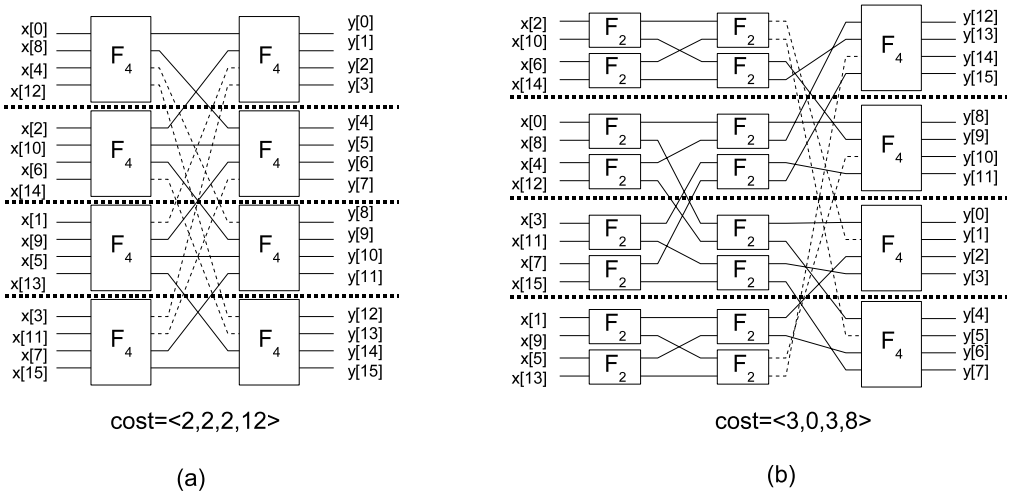



Figure 4–11: Two formulations for an 16-point FFT, representing different granularities. Horizontal dashed lines represent the partition boundaries.

## 4.6 Scheduling

In regular/scheduled-constrained structures such as DSTs, a general sense of partition quality can obtained by looking at the distribution of cuts through the channels and stages. However, the only way to have an clear estimate is by actually scheduling the nodes assigned to each partition to the architectural resources. For scheduling purposes, communication channels are architectural resources. In fact for

highly connected structures, such as DSTs, communication channels are commonly the most strained resource. Thus, to schedule a partition solution, channel resources must be first inserted into the DFG as determined by the solution. This is, for every connected node pair $\langle u, v\rangle$ such that $u \in P_x$ and $v \in P_y$ ($P_x \neq P_y$) a third node is inserted between $u$ and $v$. The inserted node represents utilization of the communication channel that connects $P_x$ and $P_y$.

Once the communication nodes have been introduced into the dataflow graph, the *resource-constrained scheduling* problem can be solved to obtain a latency estimate. This problem can be stated as follows. Given:

1. A directed acyclic graph $G = \langle V, E\rangle$, where $V$ is a set of nodes and $E$ is a set of edges. Each edge is expressed as a sequence $\langle u, v\rangle$ where $u, v \in V$.
2. A set $K = \{K_0, \ldots, K_{m-1}\}$ of $m$ functional unit types.
3. A type function $f : V \rightarrow K$, i.e. a function that maps each node to a functional type.
4. A resource constraint function $h : K \rightarrow \mathbb{N}$ that states the number of available resources of each functional unit type, where $\mathbb{N}$ is the set of non-negative integer numbers.
5. A function $w : K \rightarrow \mathbb{N}$ that states the latency of each functional unit type.

Determine a mapping function $t : V \rightarrow \mathbb{Z}^+$ that minimizes $\max\limits_{\forall v \in V} |t(V)|$ and subject to the following constraints:

1. For all $\langle u, v\rangle$, $\max\limits_{\forall u} |t(u) + w(h(u))| < t(v)$, i.e. precedence relations are respected.
2. For each time step $t_i$ and functional type $k_j$,

$$\sum_{\forall v \in V \,|\, t(v)=t_i \text{ AND } f(v)=k_j} 1 \leq h(k_j) \tag{4.35}$$

In other words, the number of nodes of type $k_j$ assigned at any time $t_i$ does not exceed the number of available resources of type $k_j$.

In our implementation, we chose an As Soon As Possible (ASAP) scheduling heuristic, shown in Algorithms 7 and 8 [84]. First, a sorted list is constructed which contains the nodes in order of the earliest starting times. This is done in two steps: first, the depth of each node is determined, i.e. longest path from any starting node the node in question. Then, DFG nodes are sorted in a list $L_T$ on non-descending order according to their depth. Following the order of the list, nodes are assigned to the allocated resources as the resources become available.

**Algorithm 7** Topological sort heuristic.

**Input**: DFG $\{V, E\}$ and $V_S \in V$, the set of all starting nodes.
**Output**: Topologically list $L_T$

1. $V_{Current} \leftarrow V_{Start}$, $V_{Next} \leftarrow \{\}$, $level \leftarrow 0$
2. **while** $V_{Current} \neq \{\}$
   - 2.1. **For all** $v \in V_{Current}$
     - 2.1.1. $V_{Next} \leftarrow V_{Next} \cup \{\text{children of } v\}$
     - 2.1.2. $depth\,(v) \leftarrow level$
   - 2.2. **end for**
   - 2.3. $V_{Current} \leftarrow V_{Next}$
   - 2.4. $V_{Next} \leftarrow \{\}$
   - 2.5. $level \leftarrow level + 1$
3. **end while**
4. $L_T \leftarrow$ sort $V$ in nondescending $depth\,()$ order.

The topological sort heuristic requires $O(|V| + |E|)$ time to determine earliest start step, and $O\left(|V|\, log\left(|V|\right)\right)$ to sort the list. The assignment of nodes in $L_T$ to the various resources takes $O(|V|)$. Thus, the ASAP algorithm has $O\left(|V|\, log\left(|V|\right)\right)$ time complexity.

## 4.7 Resource Estimation

In order for a DFG partition solution to be implementable in a DHA, it must be mapable to the available resources offered by the architecture, without exceeding them. The exact number of resources used by a partition can only be known by synthesizing the design, but such an approach requires too much time if it is to be part of an iterative improvement heuristic. To ensure resource feasibility of solutions during high-level exploration, an estimation technique must be developed

**Algorithm 8** ASAP scheduling heuristic.

**Input**: Resources set $R = \{\langle r_{type_i}, r_{lat_i} \rangle\}$, and DFG $\{V, E\}$ and $f_t : V \rightarrow R_{type}$
**Output**:

1. $L_T = Topological\_Sort\,(G)$
2. **for all** $r \in R$, $r_{free} \leftarrow$ true, $r_{done} \leftarrow \infty$
3. **for all** $n \in V$, $n_{done} \leftarrow \infty$
4. $c\_step = 0$
5. **while** $L_T \neq \{\}$
   - 5.1. **for** every $r \in R$
     - 5.1.1. **if** $r_{free} =$ false
       - 5.1.1.1. **if** $r_{done} = c\_step$ **then** $r_{free} \leftarrow$ true
     - 5.1.2. **if** $r_{free} =$ true
       - 5.1.2.1. select topmost $n \in L_T$ whose $n_{type} = r_{type}$ **AND** all parents $m$ of $n$ are such that $m_{done} \leq c\_step$
       - 5.1.2.2. $n_{done_x} = c\_step + r_{lat}$
       - 5.1.2.3. $r_{done} = c\_step + r_{lat}$
       - 5.1.2.4. $L_T \leftarrow L_T - n_x$
       - 5.1.2.5. **if** $n_{done_x} > max\_c\_step$ **then** $max\_c\_step \leftarrow n_{done_x} >$
   - 5.2. **end for**
   - 5.3. $c\_step{+}{+}$
6. **end while**

that computes resource utilization in a fast and relatively accurate manner. One of the challenges for accurate resource estimation at high levels of abstraction is that there are multiple ways in which a dataflow graph can be implemented in hardware. Each of the structures has a different resource utilization. To overcome this problem, high-level optimization techniques define a specific target architecture model [10]. This establishes the general structural style of the solutions envisioned by the optimization mechanism and allows the deduction of mathematical models that establish a relationship between DFG properties and resource utilization.

In this section, we discuss the development of a high-level resource estimation technique for the DMAGIC methodology. First, the target architectural model is presented, followed by the justification and general overview of our chosen target technology. Based on the architectural model and target technology, a resource estimation model is developed.

### 4.7.1 Architectural Model

Figure 4–12 shows DMAGIC's target architectural model for each dedicated DHA device. The computational load of a partition is implemented by a set of interconnected modules that function as a customizable vertical folding structure. Each module consists of a functional primitive common throughout a particular DST's structure, as well as the necessary storage, control and data path options to implement the various stages of the transform. For instance, Figure 4–12 shows a module for the implementation of an FFT. It consists of a two input *kernel* implementing a size-2 DFT with twiddle multiplication, data memories to store intermediate results, a twiddle table that contains the various coefficients for multiplication, and data path/control elements to establish data movement throughout execution.

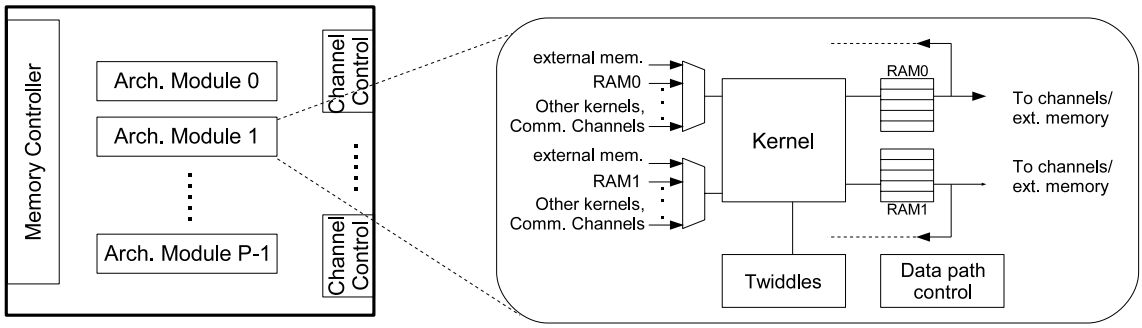



Figure 4–12: Device-level architectural model and block diagram for an FFT module.

The regularity exhibited by fast DST algorithms allows them to be entirely implemented by the proposed modular architecture. After a partitioning solution has been obtained, each node in a partition is mapped to one or more structural modules. Mapping proceeds in a balanced manner, equally distributing computational load throughout the modules. For a size $N = 2^n$ DST consisting of $\lambda N \log_2 N$ functional primitive operations, partitioned to a DHA consisting of $M = 2^m$ devices, each implementing $P = 2^p$ modules, each module implements the operation of $(\lambda N \log_2 N) / (M \cdot P)$ primitives. Figure 4–13 shows a possible mapping for an example partition to a device with two modules.

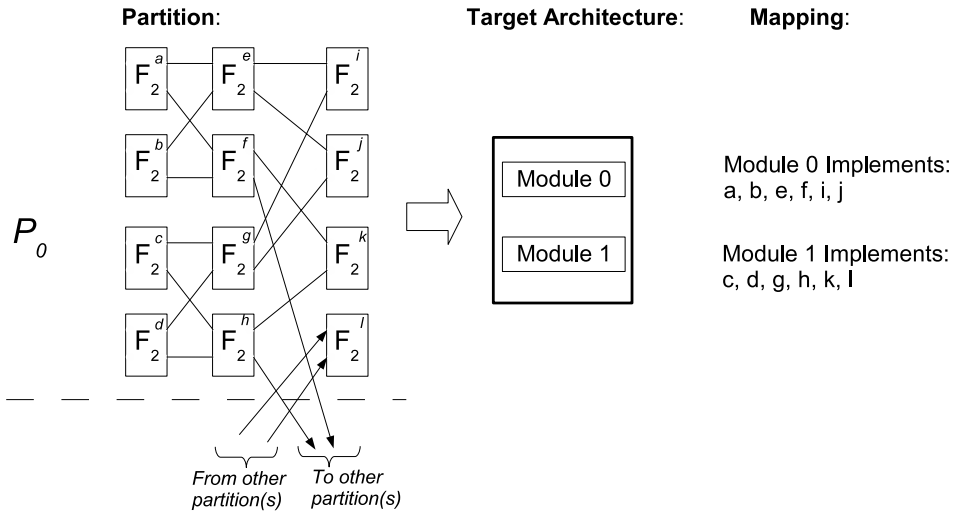



Figure 4–13: Example mapping of a DFG partition to a device with two architectural modules.

The number of modules per device determines the computational latency of the implementation. A higher number of modules requires less folding of the original DST DFG, each architectural module needs less cycles to complete all its computations, thus reducing latency. For instance, the DFG partition presented in Figure 4–13 could be mapped to a single module. However, this would negatively impact the implementation latency, as each module would be responsible for implementing a higher number of DFG operations. Given that latency minimization is the main objective in our partitioning strategy, our principal intention when estimating resources is to determine the maximum number of modules that can fit to the DHA devices.

### 4.7.2 Target Technology

The DMAGIC methodology is oriented toward platforms with dedicated hardware devices. This includes devices such as ASICs and FPGAs, where the implemented internal circuitry is customized for the specific application at hand. As previously mentioned, documentation about implementation and experimental results for distributed ASIC architectures is scarce. Moreover, the availability of such

architectures is significantly more restrictive than for distributed FPGA architectures. For these reasons, our developed resource estimation model targets FPGA devices, specifically those of newer families which include embedded elements, such as multipliers and RAMs. Nevertheless, the development of the FPGA resource-estimation models is presented in a modularized manner to simplify their adaptation to alternate technologies, if needed.

As illustrated in Figure 4–14, modern FPGAs are internally organized as an array of configurable logic blocks (CLBs), embedded functional units and programmable interconnections. Configurable logic blocks are called *slices* and *adaptive logic modules* in Xilinx and Altera devices, respectively. A configurable logic block usually consists of two look-up tables (LUTs), each capable of implementing any 4-to-1 function, two bits of storage for registering the LUTs outputs, and data path-related gates for implementing simple functionalities, such as carry propagation and multiplexing. Embedded functional blocks implementing more elaborate operations such as multipliers, block RAMS, and microprocessor cores are also included in most of today's FPGAs. The logic array, embedded elements, and device pins are connected via several levels of programmable interconnect.

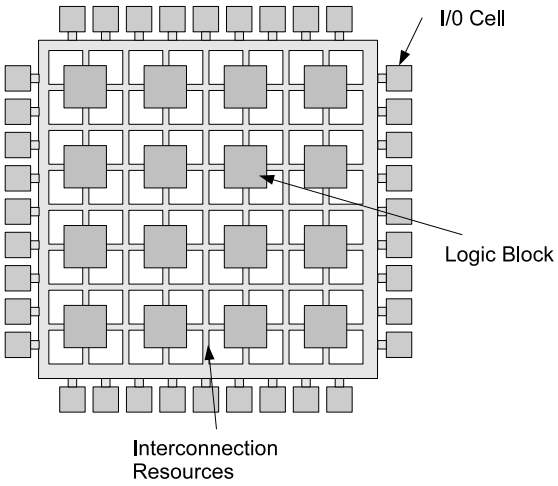



Figure 4–14: FPGA components.

Resource estimation for FPGAs is essentially concerned with determining the number of compromised CLBs and embedded functional units for an application [60][85]. Programmable interconnects are harder to estimate at higher-levels since they depend on decisions made much later in the EDA design flow at the physical placement and routing stages. Furthermore, in practice, designs implemented to modern FPGAs typically run out of logic resources before routing resources.

### 4.7.3 Resource Estimation Model

Generally speaking, the mathematical model for resource utilization in the hardware implementation of an algorithm is the sum of the resources required for the following: functional units, storage units, datapath, control and connections. Our discussion for the resource estimation model follows the hierarchical nature of the chosen architectural model. In other words, discussion begins with a device-level model consisting of modules and data communication components, followed by the development of resource estimation models for the modules' internal components. Estimation for user logic and embedded units is handled separately at all levels, since they are physically distinct resources in the FPGA.

At the device-level, resources are estimated by Equations 4.36 and 4.37.

$$TOTAL_{CLB} = MCNT_{CLB} + \sum_{m \in Modules} MODULE_{CLB}(m) + \sum_{c \in Channels} CHCNT_{CLB}(c) \tag{4.36}$$

$$TOTAL_{E} = MCNT_{E} + \sum_{m \in Modules} MODULE_{E}(m) + \sum_{c \in Channels} CHCNT_{E}(c), \tag{4.37}$$

where $TOTAL_{CMB}$ and $TOTAL_{E}$ are the total estimated CLB and embedded units, respectively. $MCNT_{CLB}$/is the number of CLBs required for the implementation of the memory controller, $MODULE_{CLB}(m)$ is the number CLBs for module

$m$, and $CHCNT_{CLB}(c)$ is the number CLBs for channel controller $c$. Quantities for embedded units follow a similar naming convention.

The number of CLBs in each module is estimated by:

$$MODULE_{CLB} = K_{CLB} + IM_{CLB} + DMEM_{CLB} + TT_{CLB} + CTRL_{CLB}\,, \quad (4.38)$$

where $K_{CLB}$ is the number of CLBs used by the kernel, $IM_{CLB}$ is the number of CLBs used by input multiplexers units, $DMEM_{CLB}$ is the number of CLBs used by data memory, $TT_{CLB}$ is the number of CLBs used by twiddle tables, and $CTRL_{CLB}$ is the number of CLBs used for control logic. Since all of these components, except the input multiplexers, can be completely or partially implemented using embedded units, the number of embedded units per module is estimated by:

$$MODULE_E = K_E + DMEM_E + TT_E + CTRL_E \quad (4.39)$$

The rest of our discussion pertains to how the individual terms of Equations 4.38 and 4.39 are estimated. This requires some understanding on how certain operations are synthesized to embedded units.

**Mapping to Embedded Multipliers and BRAMs**

Among the embedded units provided by modern FPGAs, embedded multipliers and block RAMs (BRAMs) play an essential role in the dedicated hardware implementation of DSP algorithms. Most synthesis tools can automatically synthesize multiplications in the HDL specification to embedded multipliers, and likewise, storage functionality to embedded BRAMs. Embedded multipliers in modern Xilinx and Altera families are hardwired 18-bit by 18-bit multipliers. Multiplications wider than 18 bits are synthesized from several 18-by18 modules. The exact number of embedded multipliers needed to perform a $w_a$-bit by $w_b$-bit real multiplication follows a simple formula:

$$EM(w_a, w_b) = \lceil w_a/18 \rceil \cdot \lceil w_b/18 \rceil \tag{4.40}$$

BRAMS have a total capacity of 16 kbits, which can be configured to be between 16Kx1-bit and 512x32-bit [86][87]. Memories with capacities that exceed this range of configurations are synthesized using several BRAMs. The number of BRAMs needed to implement a memory of height $h$, width $w$ is given by the equation

$$BRAMS(h, w) = \begin{cases} \lceil w/32 \rceil & \text{if } (w'h) \leq \text{BRAM\_capacity} \\ 2^{\lceil \log_2(w'h/BRAM\_capacity) \rceil} & \text{otherwise.} \end{cases}, \tag{4.41}$$

where $w'$ is the effective bit-width of the twiddle factors, obtained as

$$w' = 2^{\lceil \log_2(w) \rceil} \tag{4.42}$$

### 4.7.4 Module Components Resource Estimation

**Kernels**

A kernel contains the physical implementation of the functional primitive common throughout a given DST. Commonly, a DST functional primitive performs a combination of addition/subtractions and multiplications on its inputs. In order to represent the functionality of two similar operations, a kernel can also include simple data path redirections. For instance, the kernel in Figure 4–15 could implement the functionality of similar DFG blocks (a) and (b). Kernel resource utilization can be understood by considering the impact of implementing its most common structures: multipliers, adders/subtractors, and multiplexers.

**Multiplications**

Multipliers are one of the main resource spenders in DSP applications. This is especially true for DSTs using complex number representation, where each complex multiplier implementation could use up to three or four real multipliers and up

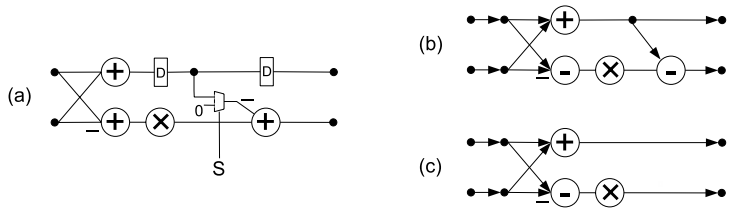



Figure 4–15: Functional primitive (a) implements the functionality of DFGs (b) and (c).

to three real additions. In modern FPGAs, multipliers can be implemented using either of two resources: user logic or embedded multipliers. As a rule of thumb, a fixed-point multiplier implemented with user logic blocks requires a number of CLBs proportional to the square width of its inputs, and operates at a slower rate than those implemented with dedicated logic. Because of this, recent implementations exclusively use embedded units for multiplication. The number of embedded multipliers for a kernel can be computed by adding the embedded multipliers required by each of the multiplication operations.

$$EM_{Kernel} = \sum_{u \in multiplications} EM(BW(u)) \,, \tag{4.43}$$

where $BW\,(u)$ is the operand bit-width of multiplication node $u$, and $EM()$ is computed according to Equation 4.40.

**Additions**

In FPGAs, additions are implemented with user logic, requiring a number of CLBs proportional to the bit-width of its operands. The CLB utilization of kernel additions can be appropriately estimated by:

$$CLB_{Kernel_{Add}} = \lambda_A \sum_{u \in additions} BW(u) \,, \tag{4.44}$$

where $\lambda_A$ is the proportionality factor relating the operand bit-width to the actual number of CLBs required. For instance, in Xilinx Virtex-2 devices $\lambda_A = \frac{1}{2}$, e.g. requiring 8 CLBs to implement a 16-bit adder.

**Multiplexers**

Multiplexers are also implemented with user logic in FPGAs. The number of CLBs used by a multiplexer is proportional to the product of the data input bit-width and the number of inputs from which the multiplexer must select.

$$CLB_{Kernel_{Mux}} = \lambda_M \sum_{u \in multiplexers} \left( BW(u) \times 2^{\lceil \log_2(NI(u)) \rceil} \right) , \qquad (4.45)$$

where $\lambda_M$ is the proportionality factor relating the product of operand bit-width ($BW$) and number of inputs ($NI$), with the actual number of CLBs required. For instance, in Xilinx Virtex-2 devices $\lambda_M = \frac{1}{4}$, e.g. requiring 16 CLBs for a 4-input multiplexer with 16-bit inputs.

**Twiddle Factor Tables**

The typical structure of a fast DST involves the multiplication of data by different twiddle factors throughout the various columns and rows of its DFG. Each architectural module implements several DFG functional primitives. Thus, every module must be provided with a different twiddle factor for each iteration. In previous DST hardware implementations twiddle factors were either computed 'on the fly' by a dedicated generator or accessed from a precomputed table [60][88]. The twiddle factor generator approach is too resource expensive for a multi-kernel architectural model since it requires the use of multipliers, which are used abundantly in the DST kernels. The remaining option is to store twiddle factors in a table and access them as needed by the each iteration.

Milder, et al. proposed two methods for implementing twiddle tables in an FPGA multi-kernel implementation [60]. The first method instantiates a table containing all the transform's twiddles per kernel multiplier. Each kernel multiplier requires a twiddle address generator to access the correct factor in each computational stage. Typically for fast DSTs the number of different twiddle factors is half the size of the transform. Therefore, this method requires that each table contain $N/2$

entries. The second method instances in each table only the twiddle factors needed by that specific kernel multiplication throughout computation. In this method, the factors can be sequentially stored in the table, allowing the table to be implemented as a FIFO and saving in memory address logic. Typically, for fast DSTs the total number of different functional primitives is proportional to $\lambda N \log_2(N)$. Thus in this method the number of twiddles per table corresponds to the number of times a kernel will be used throughout computation: $\lambda N \log_2(N) / (M \cdot P)$. The second method, although requiring more *storage* resources for the twiddle factors, will be less expensive for our architectural model, since it will save significantly on control logic for memory addressing.

If implemented using BRAMs, twiddle tables resource utilization per module can be estimated with the following expression:

$$TT_E = t \cdot BRAM(h, w) \tag{4.46}$$

where $BRAM(h, w)$ is as defined in Equation 4.41, $t$ is the number of kernel multipliers, $w$ is the twiddle factor bit-width, and $h = \lambda N \log_2(N) / (M \cdot P)$.

When implementing a table of constant values using user logic, synthesis tools use logic minimization heuristics which considerably reduce the number of required CLBs. Figure 4–16 shows the results of an experiment to determine how compression behaves across different ROM sizes. A wide range of ROM sizes with various lengths and widths were generated in Verilog and synthesized using Xilinx ISE 6.3 targeting a Xilinx Virtex 2P device. The graph shows the relationship between the achieved compression ratio vs. the uncompressed slice utilization. The dominating trend follows an exponential rise ($y = y_0 + a\left(1 - e^{-bx}\right)$) to a maximum of approximately 95%, as indicated by the trend line.

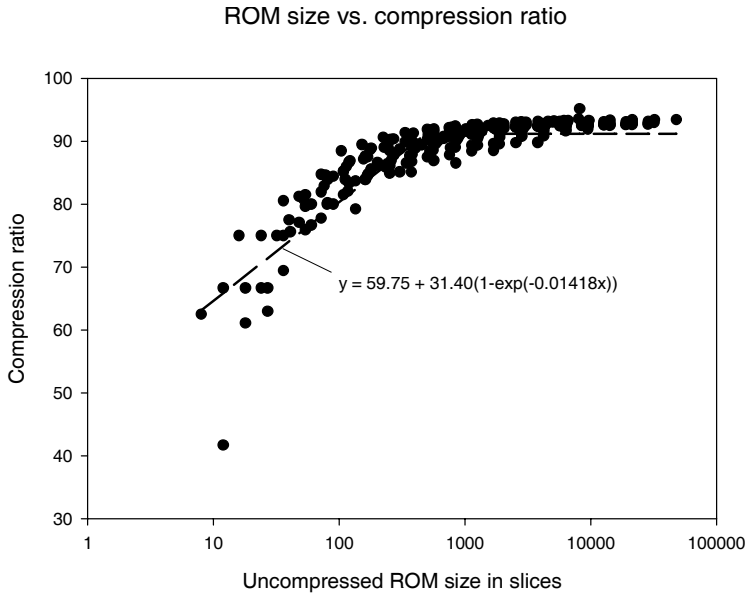



Figure 4–16: Slice compression ratio vs. ROM (uncompressed) slice utilization.

Thus, the number of CLBs used in the ROM implementation of table of values, such as the twiddle factor table is:

$$TF_{CLBs} = ROM\_size \cdot (1 - K\,(ROM\_size))\ , \tag{4.47}$$

where $ROM\_size$ is the uncompressed size of the data table

$$ROM\_size = \frac{k\lambda N \log_2(N)}{M \cdot P}\,, \tag{4.48}$$

where $\lambda$ is the proportionality factor between $n$ and the actual number of functional primitives. $K$ is the ROM compression factor function achieved by the synthesis tool, e.g. in our empirical results:

$$K\,(ROM\_size) = y_0 + a\left(1 - e^{-bx}\right)\ , \tag{4.49}$$

with $y_0 = 59.79$, $a = 31.40$, and $b = 0.01418$.

**Data Memory**

Data memory is needed as part of each module to store the intermediate results of the transform's computation. Each kernel memory is responsible for storing part

of the data set. Since partitions and mappings are balanced, each of the DHA's devices is responsible for handling $\lambda N \log_2(N)/M$ data points, where $\lambda$ is the proportionality factor between the number of functional primitives and $N \log_2(N)$. Each module is responsible for storing $\lambda N \log_2(n)/(M \cdot P)$ data points. Since $t$ data points are produced simultaneously by each kernel, $t$ data memories each with height $\lambda N \log_2(n)/(M \cdot P \cdot t)$ are needed. Data memories can be implemented with user logic or using the embedded BRAMs. The first option requires a number of CLBs proportional to the memories' width $w$ and height $h$.

$$DM_{CLB} = \frac{\lambda N \log_2(n) \cdot w}{M \cdot P}, \tag{4.50}$$

where $w$ is the bit-width of data points.

It was experimentally corroborated that data memory structures do not represent the slowest modular component, even when implemented using CLBs. Therefore, they can be completely or partially implemented using BRAMS or CLBs with no significant impact on the system performance. When implemented using BRAMs, each data memory, no matter how small, will occupy at least one of these structures. The total BRAMs for implementing data memory in a module is computed by:

$$DM_E = t \cdot BRAM(h, w), \tag{4.51}$$

where $h = \lambda N \log_2(n)/(M \cdot P \cdot t)$ is the height of each memory.

**Input Multiplexers**

Input multiplexers select the input data source for the next kernel iteration. Since the proposed architectural model implements a partitioned vertical folding of a DST structure, kernel inputs for the next iteration may arrive from external memory, kernel memories, or communication ports. From a partitioning-level perspective, resource estimation for input multiplexers is not as straightforward as for the previously discussed components because information is needed about the actual

mapping of DFG nodes to architectural modules. In a worst case scenario, a kernel would need to input data from a different source during each iteration. In this situation, each of the input multiplexers would be a $Q$ to 1 multiplexer, where $Q$:

$$Q = \min\left\{t \cdot P, \frac{\lambda N}{P \cdot M}(\log(N) - 1)\right\} + (M - 1) + 1 \tag{4.52}$$

Since there are $P$ kernels, there are $t \cdot P$ possible data outputs. The term $\frac{\lambda N}{P \cdot M}(\log(N) - 1)$ accounts for the number of processing stages that the kernel will go through in which it could require data from sources other than external memory. The later terms account for the rest of hardware devices $(M - 1)$ and external memory. The upper bound hypothesized by equation 4.52 could probably only be achieved by intentionally looking for a bad partitioning solution followed by a bad mapping scheme, which go against the intention of the partitioning methodology. In practice, the number of multiplexer inputs is significantly lower than the upper bound, thanks mainly to the following consideration in the proposed methodology. Without any loss of generality, assume that an FFT is being partitioned and that the architectural kernels each implement an $F_2$ and twiddle factor multiplication operation.

First, the FFT's KPA formulation is converted to an equivalent DFG. Each $F_N$ operator in the KPA formulation becomes a node in the DFG. The DFG is partitioned as is, without further breaking or clustering the nodes. Thus, after each node has been assigned to a partition, its constituting operations will be mapped to the kernels in that specific device. Nodes that represent FFT operations larger than 2 can be mapped to the available kernels in a manner that minimizes multiplexer requirements. To illustrate this concept, consider a particular KP formulation that contains a stage of DFTs of size n=8. During the P/P each of them will be assigned to a certain device. The DFT2 operations that make up the DFT8 can be assigned in a manner which lessens the impact on multiplexer resources. Usually this would

imply a folded Pease implementation in which the input of every kernel receives data from only one other kernel.

In order to obtain a more realistic estimate of the number of inputs to the input multiplexers, a simple mapping mechanism was implemented. Using a simulated annealing optimization heuristic, the nodes assigned to each partition are mapped to the device's modules in an effort to minimize the number of inputs required by the input multiplexers. Several FFT sizes were partitioned and then mapped to architectures containing different module quantities. Figure 4–17 shows the results for a 4_RING topology. The results indicate that the number of multiplexer inputs required after mapping is bounded by a number that is considerably smaller than implied by Equation 4.52. Furthermore, this experimental upper bound increases only slightly across various $P$ sizes. The resource estimation model for input multiplexers was changed to reflect the empirical results.

$$IM_{CLB} = t \cdot P \cdot \Upsilon \cdot \lambda_{MUX} \cdot w \tag{4.53}$$

where $\Upsilon$ is an empirically established value.

**Control Logic**

Control logic accounts for all the additional FPGA logic that is needed to control the dataflow throughout the rest of the components. To achieve proper computation in the architectural model, at each computational-step a module's control logic must specify: the source of data inputs, the address of the data inputs (if from one of the data memories), as well as provide any other signals to specify the kernel's functionality. One approach to implement control would be to provide a table of instructions, where each instruction contains the necessary control signals for all the controlled components. This approach is illustrated in Figure 4–18(a). As the DST computation advances through time, a control word is read from the table on each c-step, and the corresponding signals sent to the controlled components. A

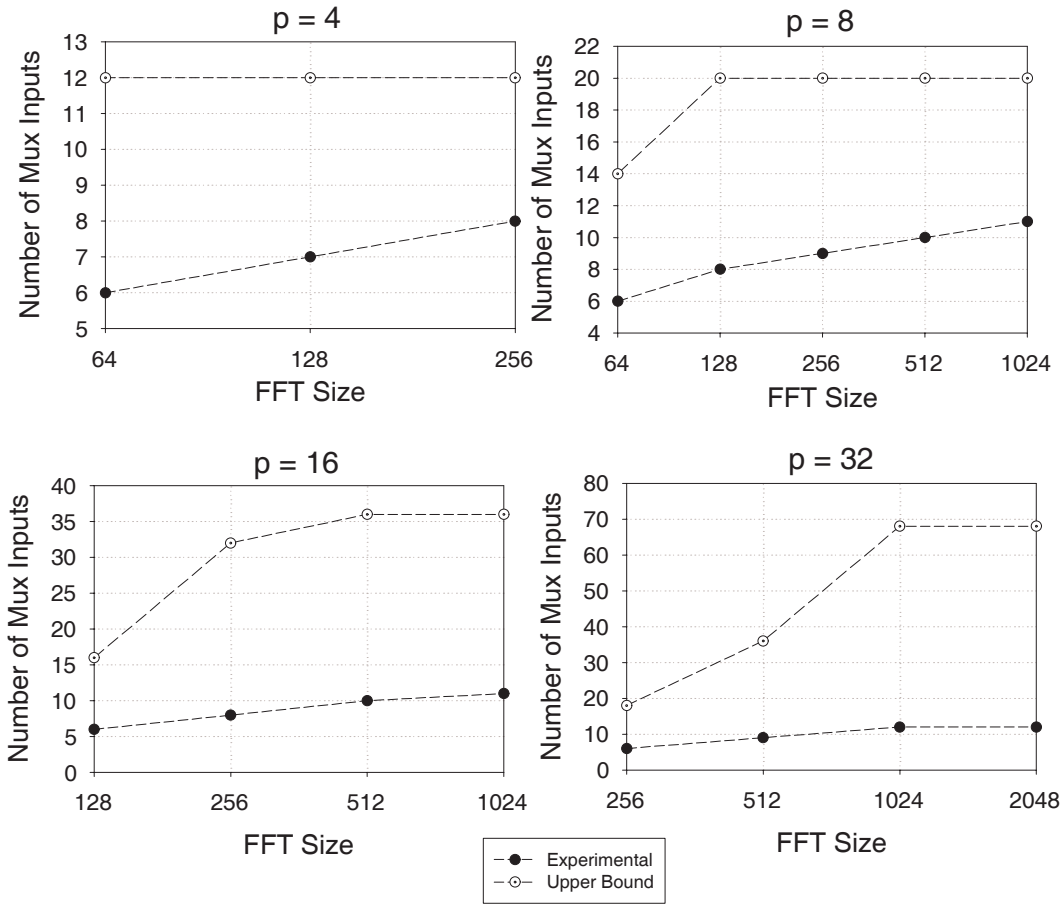



Figure 4–17: Experimental results for mapping of several FFT sizes to architectures with 4, 8, 16, and 32 modules.

table with as many entries as c-steps would be required. This is bound to waste logic resources, since modules are actually performing computation only during part of the total latency. The total logic required for a CLB implementation using this method can be estimated as:

$$CLB_{control} = Latency \cdot W_{instruction} \cdot (1 - K\,(Latency \cdot W_{instruction}))\ , \qquad (4.54)$$

where $W_{instruction}$ is the width of the control word that specifies the control signals for all controlled components, and can be computed as follows:

$$W_{instruction} = \sum_{i \in InputMuxes} 2^{\lceil \log_2(NumInputs(i)) \rceil} + \sum_{i \in DataMems} 2^{\lceil \log_2(MemHeight(i)) \rceil} + Kernel\_Signals \quad (4.55)$$

and $K()$ is the function relating the uncompressed ROM size to the synthesis compression ratio (Equation 4.49) and $Kernel\_Signals$ are any additional control signals that must be provided to the computational kernel, e.g. signal $S$ in Figure 4–15.

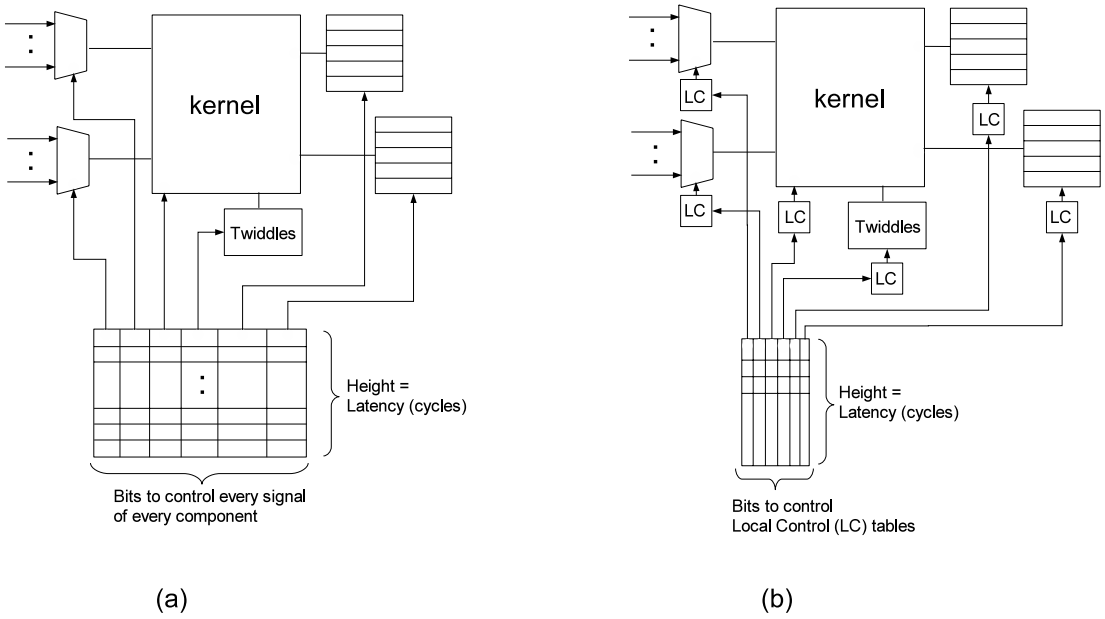



Figure 4–18: Two approaches for implementing module-level control logic (a)integrated, (b)distributed.

A less resource-intensive approach would be to provide each of the controlled components with its own instruction table. Figure 4–18(b) illustrates this approach, which would additionally require a module-level unit to coordinate the work of the individual tables. Each of the distributed tables needs to have as many entries as there are actual computational steps involving the component. Each kernel implements the functionality of $\lambda N \log(N) / (M \cdot P)$ DFG functional primitives. This implies, for example, that the control table for each input multiplexer needs to have

$\lambda N \log(N) / (M \cdot P)$ entries of width $2^{\lceil \log_2(IM_{inputs}) \rceil}$, where $IM_{inputs}$ is the number of inputs to the multiplexer. The module-level table would still need to have as many entries as c-steps in the total latency, yet the width of each entry is significantly reduced to one bit per controlled component plus one for the kernels signals.

$$CLB_{control} = (1 - K(ROM\_SIZE))\, ROM\_SIZE\,, \tag{4.56}$$

$$\begin{aligned} ROM\_SIZE &= L_{Total} \cdot W_{instruction} + L_{Comp} \cdot \left( \sum_{i \in InputMuxs} 2^{\lceil \log_2(NumInputs(i)) \rceil} + \right. \\ &\left. \sum_{i \in DataMems} 2^{\lceil \log_2(MemHeight(i)) \rceil} + KSignals \right), \end{aligned} \tag{4.57}$$

where $W_{instruction}$ now refers to the global control table, which includes only the necessary signals to specify the state of the controlled components.

### 4.7.5 Resource estimation scheme validation

The proposed resource estimation scheme was validated by constructing Verilog HDL models of different sizes, synthesizing them using Xilinx ISE 6.3.03i, and comparing the synthesis results against our estimates. Figure 4–19 shows the actual vs. estimated slice utilization for various FFT sizes and modules per device. As indicated by the proximity of the points to the line of $slope = 1$, there is a high degree of correspondence between the estimated and actual slice utilization. A 5.79% average estimation error with a maximal error of 21.15% was measured among the results. Estimates for the embedded components were exactly the same as the synthesis results.

As predicted by our model, resource utilization is directly proportional to the number of modules per device. Since our optimization objective is latency, which has an inversely proportional relation to $P$, our methodology essentially chooses to instance as many modules as can be fit to the device.

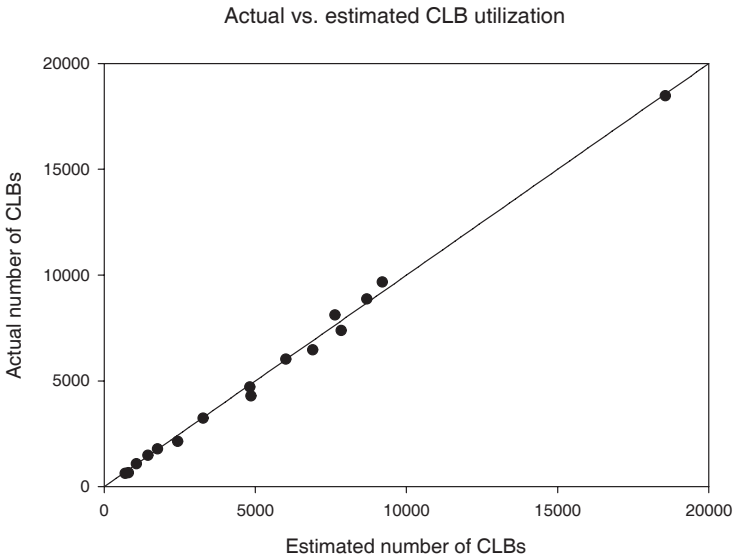



Figure 4–19: Actual vs. estimated slice utilization for various FFT sizes.

The resource estimator is not consulted at each graph partitioning optimization step, since in each iteration the only changes introduced would be to the connections between the input multiplexers. Thus, we establish a $P$ in the beginning and stick with it throughout the whole optimization process.

## 4.8 Summary

In this chapter we have introduced the various tools used by our methodology to partition a given formulation of a DST. A KPA to DFG tool was developed to facilitate representation of DSTs and their subsequent graph partition. The incorporation of DST considerations into a graph partitioning heuristic was also presented. Lastly, a resource estimation model was derived for the chosen architectural model. Next chapter discusses how these tools were used in order to determine a heuristic for formulation exploration.

# CHAPTER 5

# Formulation Exploration

Our partitioning methodology contemplates the use of DST algorithmic properties to conduct an exploration of equivalent DST formulations, in search of formulations that may be more suitable for the target distributed hardware architecture. Even the application of a small set of algorithmic-level transformations can result in the combinatorial explosion of the exploration space. Thus, some level of awareness of the effect of these rules on solution quality is needed to design an exploration strategy that provides good results in a practical amount of time. In this chapter, the development of our formulation exploration strategy is described.

Section 5.1 compares our intended exploration with that performed by other DST code optimization methods, highlighting the special considerations needed in our case. Sections 5.2.1, 5.2.2 and 5.2.3 describe the experiments conducted to gain insight into the effect of FFT reformulations on solution quality. Results from these experiments were used to devise a greedy exploration heuristic, which is presented in Section 5.3. The extension of our methodology to the discrete cosine transform is detailed in section 5.4. The last section presents a chapter summary.

## 5.1 General Considerations

The impact of formulation on single and distributed hardware implementations has been documented in some isolated DST cases [16]. However, to the best of our knowledge, no methodologies have been proposed that will automate the task

of exploring the formulation space for hardware implementations. Algorithmic-level rules have been successfully integrated into methodologies for automated DST (software) code generation. FFTW and SPIRAL, two such methodologies, are essentially solution-space exploration engines which utilize factorization rules to generate and evaluate DST formulations for general-purpose processor architectures [45][89].

The nature of our problem, i.e partitioning rather than code generation, as well as the characteristics of our target architectures, significantly differentiate our approach from DST code generation methodologies. Current DST code generation methodologies target single processor and bus-connected multi-processor platforms, which have a one-dimensional address space (i.e. cache levels, and main memory). In our target architecture, each processing element can obtain its data from a variety of communication channels or its own memory. This constitutes a multidimensional address space, which has been shown to require alternative strategies for evaluation and optimization [90]. The software-oriented nature of code-generation approaches also allows them to make certain strategical assumptions which would not apply to our case. For example, both SPIRAL and FFTW can speed their exploration process by assuming that DSTs have an *optimal substructure*: if an optimal implementation for a size $n$ is known, this implementation is still optimal when size $n$ is used as a subproblem of a larger transform [62]. This assumption is in principle false, yet it allows them to obtain good results using faster exploration methods such as dynamic programming.

The *dedicated* nature of the target devices, also introduces a key difference between our approach and code generation schemes. As part of their optimization loop, code generation strategies compile and execute each solution to measure its runtime on the architecture. On the other hand, the implementation process for dedicated devices is considerably more time consuming than GPP source code compilation.

This implies that our methodology must rely on high-level models and assumptions to estimate solution quality.

There are many Kronecker Product Algebra and DST specific rules and algorithms that can be applied to a given discrete transform. If all of these rules were considered equally capable of affecting partition quality, then all of them would have to be considered as part of a formulation exploration. One problem with this scenario is the considerable growth of the exploration space. For instance, the SPIRAL code generation program can consider a total 1,639,236,012 formulations for a size-32 DCT [44]. The huge number of formulations for this relatively small transform is caused by the number of rules considered as part of the formulation exploration. In SPIRAL's case, using an extensive number of rules is justified by the fact that the target devices are General Purpose Processors. Formulation changes ultimately translate to changes in program code, which might exploit some concealed architectural advantage of the chosen GPP. This is not the case for dedicated hardware, where we favor DST algorithms that are as regular as possible [1]. Thus, the type of rules that we allow in exploration is limited to those that can maintain a structure with regular functional primitives.

## 5.2 Experiments to Assess Effect of Transformations on Partition Quality

In order to use DST functional properties to improve their partitioning process, we need to first understand their effect on partition solution quality and exploration efficacy. To this end, several experiments were carried to assess the effect of properties that can be algorithmically controlled on DST formulations: inter-stage permutations, kernel granularity and breakdown strategy. The resulting observations were used to devise the heuristics employed throughout the DMAGIC methodology. Diverse FFT formulations of various sizes and characteristics were partitioned using KL-MH to target architectures consisting of four and eight devices. For all the

experiments we assumed adjacent and crossbar channels weights of 1 and 2, respectively. For compactness, solution costs were expressed as the highest channel cost in the solution cost vector. For example, the reported value of a solution with cost vector $P = \langle 12, 11, 14, 10 \rangle$ is 14.

### 5.2.1 Inter-stage Permutations

The solution quality of deterministic partitioning methods, such as Kernighan-Lin, is highly dependent on the initial solution. Some popular graph partitioning algorithms actually run several times with different randomly created initial solutions and then chose the best result [91]. In an effort to salvage regularity through the final solution, DMAGIC generates initial linear horizontal partitions. Thus, algorithmic formulations with different inter-stage permutations represent distinct initial solutions. To observe the effect of permutations, and possibly detect heuristic strategies to be applied in partitioning, KPA formulations for a range of sizes of five common FFT formulations were converted to DFGs and partitioned/placed using KL-MH. The chosen formulations were Cooley-Tukey (CT), Gentleman-Sande (GS), Pease, Stockham, and Transposed Stockham.

Fig. 5–1 shows the results from this experiment. The graphs show the percent difference in solution cost for various formulations on the target architectures. Average solution costs for a randomly determined initial partition were also included. Two main observations can be drawn from these results. First, randomly generated initial solutions yield inferior results than when starting with linear partitioning initial solutions. Secondly, none of the formulations exhibit a consistent advantage over others for all sizes and/or topologies, even though formulations CT and GS, consistently start with a lower initial cost.

### 5.2.2 Kernel Granularity

Clustering is commonly used during graph partitioning to help prune the solution space while improving solution quality and reducing time of convergence. When

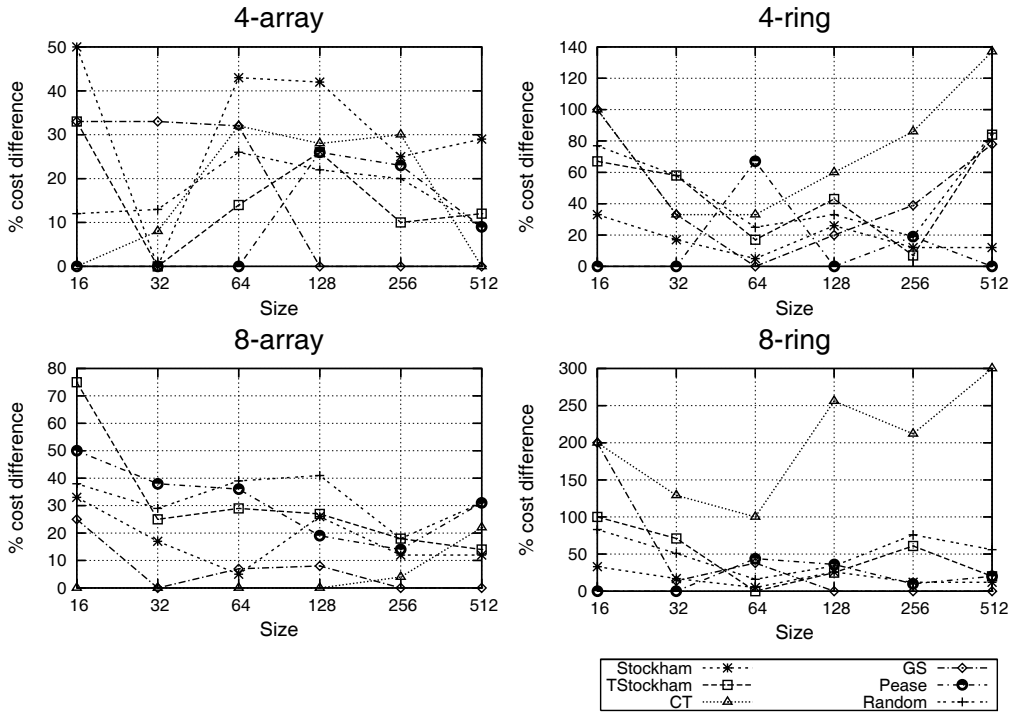



Figure 5–1: Results from the permutation experiment.

dealing purely with graphs, clustering techniques determine the formation of clusters based on graph qualities such as connectedness [91]. In generic HLP methods, information extracted from the high-level language algorithm specification or manually added information has been used to cluster DFG operations to form coarse-node graphs [7][12]. At the formulation level, the granularity of a DST can be manipulated by decomposing larger sized DST kernels into combinations of smaller ones. We used a Cooley-Tukey-like factorization formula to study the effect of granularity in the partitioning of FFTs. This formula states that, if $n = pm$, then:

$$F_n = (\beta_p \otimes I_m)\,(I_p \otimes \beta_m)\,P_{n,p} \tag{5.1}$$

where $F_n$ represents a size $n$ DFT, $I_n$ is an identity matrix, $T_{n,m}$ is a diagonal matrix of weights, $P_{n,p}$ is a stride permutation matrix.

$$\beta_{2^t} = \prod_{q=t,t-1}^{1} \left(I_{2^{t-q}} \otimes (\beta_2 \otimes I_{2^{q-1}})\right) , \tag{5.2}$$

where is a butterfly-twiddle operation

$$\beta_2 = F_2 \begin{bmatrix} 1 & 0 \\ 0 & \alpha \end{bmatrix} \tag{5.3}$$

and $\alpha$ is a twiddle factor. The derivation of Equations 5.1 and 5.2 is given in Appendix B.

Using Equation 5.1 we generated KPA formulations for every combination of stage granularities for a range of sizes of FFTs. For instance, for an FFT size $n$=8, three formulations were generated: 2×2×2, 2×4, and 4×2, where each number corresponds to the size of kernels in each stage (e.g. the 2×2×2 formulation corresponds to that in Fig. 4–10). The formulations were converted to their corresponding dataflow graphs and partitioned using KL-MH. Table 5–1 summarizes our results by showing the formulations that achieved minimum cost for each of the FFT sizes 16 through 512. In this table, *k-Array* and *k-Ring* denote architectures with $k$ devices connected in a linear array and ring topologies with an additional crossbar. Cases where multiple formulations achieved the minimum cost are identified by asterisks. For these cases, we show the minimum cost formulation with coarsest granularity.

Generally, the results demonstrate the effect of topology on solution cost. Average solution cost reductions of 37% and 57% were achieved when comparing linear arrays vs. ring topologies. This represents significant reductions, since the difference between the two topologies is only an additional communication channel. It is also evident that, for a fixed topology, having more devices does not decrease solution cost. This is due to the high connectivity in FFT algorithms and the fact that when scaling from 4 to 8 devices, each device's degree of connectivity doesn't increase.

Table 5–1: Results for granularity experiment.

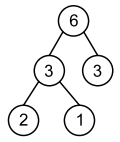
| FFT Size | Cost for Topologies: Array 4 Cost | Formulation | Ring 4 Cost | Formulation | Array 8 Cost | Formulation | Ring 8 Cost | Formulation |
|---|---|---|---|---|---|---|---|---|
| 32 | 11 | 2×2×2×4* | 7 | 2×2×2×4 | 32 | 8×2×2* | 16 | 2×4×2×2 |
| 64 | 22 | 8×2×4* | 14 | 2×2×8* | 48 | 2×2×2×2×4 | 20 | 4×2×2×4 |
| 128 | 43 | 8×2×8* | 26 | 16×2×2×2* | 92 | 2×2×2×2×2×4 | 32 | 2×2×2×2×2×4 |
| 256 | 86 | 4×2×32* | 55 | 16×8×2* | 132 | 4×2×2×2×2×4 | 58 | 2×2×2×2×2×2×4 |
| 512 | 171 | 4×2×64* | 106 | 64×4×2* | 276 | 2×2×2×2×2×2×4×2 | 116 | 2×2×2×2×2×2×8 |

As evidenced by the results, for the general case we cannot easily establish a correspondence between granularity scheme, architectural topology, and quality of solution. A practical fact that we can observe is that the finest grained formulations do not necessarily obtain the best results, so in many cases it would be wise to avoid these formulations as they also represent an increased exploration time.

### 5.2.3 Breakdown Strategy

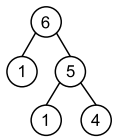
As evidenced in the previous experiments, the independent consideration of permutation and granularity did not reveal definite relations with solution quality. For this reason, an additional experiment was conducted in which we explored the effect of *breakdown strategy* on the partitioning results. A breakdown strategy describes the order and divisors with which the decomposition rule such as (5.1) is applied to obtain a formulation. It ultimately has an effect on both granularity and permutations, as it completely determines a DST's formulation. *Split trees* are a common graphical representation of decomposition strategies [92]. Fig. 5–2 shows two split trees for an FFT size $n = 2^6$ and their corresponding KPA formulations. Each node in the split tree is labeled with the $log_2$ of the size of the DFT that it represents. For discussion purposes, we shall call this label the *size* of a node. The children of a node indicate how the node's DFT is recursively computed.

(a)




$$F_{64} = \left(\left(\beta_4 \otimes I_2\right)\left(I_4 \otimes \beta_2\right) \otimes I_8\right)\left(I_8 \otimes \beta_8\right) R_{64}$$

(b)




$$F_{64} = \left(\beta_2 \otimes I_{32}\right)\left(I_2 \otimes \left(\left(\beta_2 \otimes I_{16}\right)\left(I_2 \otimes \beta_{16}\right)\right)\right) R_{64}$$

Figure 5–2: Two split trees for FFT size $n = 2^6$ and their formulations.

Breakdown strategies have been used to search the space of DST formulations, mainly in processor code generation/optimization methodologies [89]. The main

idea can be summarized as follows. A DST is decomposed by recursively applying heuristically or randomly-chosen decomposition rules, such as that in Equation 5.1, to obtain a split tree. The corresponding formulation is then coded, code-optimized, compiled and run on the target processor architecture or alternatively evaluated using an architectural model. The execution time (or performance measure) is registered and used to drive an optimization strategy which explores alternative split-trees in search of an optimal formulation.

In our experiment, all possible split trees were generated using Equation 5.1 for a range of FFT sizes from $n$ =16 to 256 and partitioned for architectures with linear array and ring topologies. For analysis purposes, breakdown strategies and their partitioning results were represented as *breakdown mega-trees* where the root corresponds to the unfactored DFT and children of a node $n$ symbolize the various ways in which a single application of Equation 5.1 can be used to further factorize $n$. Figure 5–3 illustrates the breakdown strategy mega-tree for a 32-point FFT. Each node is labeled by the breakdown tree that it represents, as well as the partition latency. The highlighted nodes illustrate the relationship between children and parent in this tree representation. For instance, given a 32-point DFT which has been factorized into leaves with exponents 1 and 4, Equation 5.1 can be used to further factorize the '4' leaf into $(1, 3)$, $(2, 2)$, and $(3, 1)$. Nodes of each level $k$ of the mega-tree correspond to split trees where the breakdown rule has been applied $k$ times. The maximum level of a mega-tree for a $2^n$-point DFT using Equation 5.1 is $n - 1$. Mega-tree leaves correspond to fully factorized formulations where all operands cannot be further factorized.

An essential observation within the created mega-trees is that the top-down paths leading to high-quality solutions follow most of the time a systematic improvement, noticeable since the beginning breakdown stages. The following behaviors are seen throughout the studied mega-trees:

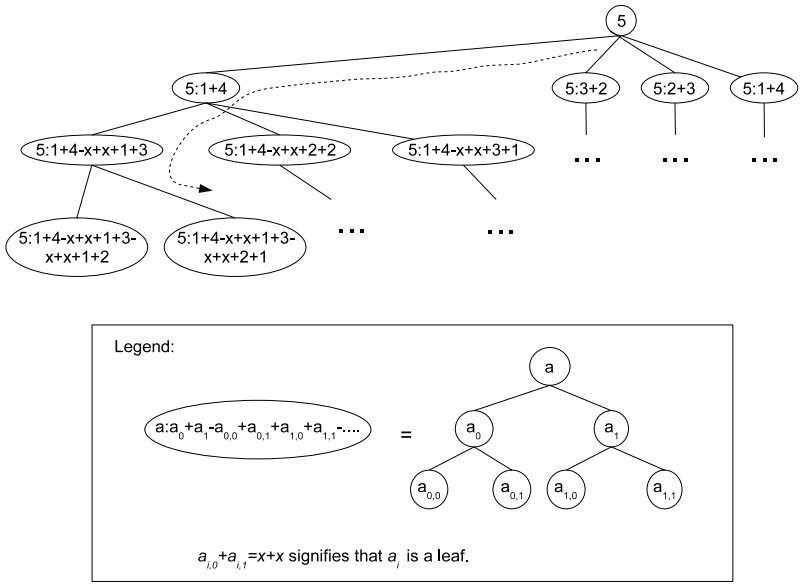



Figure 5–3: Part of a breakdown strategy mega-tree for 32-point FFT.

1. The quality of level 1 formulations is not indicative of the quality of their ascendants. A formulation $u$ which has the best latency of level 1 does not imply that best quality formulations will be their ascendants. For this reason, the first stages of our exploration strategy are based on common behaviors seen for the various DFT sizes for a given topology. For instance, ascendant formulations from split trees where a first level equally distributes size among its children tend to obtain better partitioned solutions when targeting topologies consisting of 4 devices. For example, in Fig. 5–2, formulations that follow a breakdown strategy similar to tree (a) generally have better results than those of tree (b).
2. For level 3 and above, if a node $u$ corresponds to a breakdown tree that has minimal latency among all the nodes of the mega-tree, it is highly probable that its parent is one of the best latencies among the nodes of its' level.
3. There is a wide spread of solution qualities throughout the nodes of the mega tree and the number of highest-quality solutions is scarce. Thus, randomly searching for a good formulation would not be practical.

4. In most of the studied mega-trees, there is more than one formulation that results in the lowest latency when partitioned. Thus even if we found a strategy to search the formulation-space which always resulted in the best formulation, this strategy would not be unique.
5. In many cases, formulation(s) with the lowest latency can be found in levels below the leaves of the mega-tree. In cases where leaves contain the lowest latency formulations, there exist nodes in lower levels the attain that same lower latency. This is a key observation for exploration time since graph partition complexity depends on the number of DFG nodes. Mega-tree nodes in lower levels have DFGs with less nodes and thus can be partitioned faster than mega-tree nodes in higher levels. Formulation-exploration strategy should be oriented to find the lower-level high-quality nodes.
6. A *leaf-bound* path is a sequence of nodes $\langle u_0, u_1, \ldots, u_j \rangle$ where $u_0$ is the mega-tree root and each node $n_i$ is a node with level $i$. One such path is illustrated in Figure 5–3 with a dashed line. The wide majority of leaf-bound paths had the characteristic that given $0 < i < j$, if the latency of $u_i$ was higher than that of $u_{i-1}$ then latency of all children of $n_i$ is greater than or equal to that of $u_{i-1}$. In other words, the mega-tree showed very few instances of hill-climbing (when explored in the root-leaf direction), and in these few occasions the path did not lead to the best quality solutions. Thus, a formulation-exploration strategy can use increase in latency as a condition to halt exploration down a certain formulation path.

All these observations suggest the use of a bottom-up greedy heuristic exploration in which starting from coarse formulations of a DFT, increasingly finer-grained formulations are explored until a leaf or a condition similar to observation #6 is found. The general strategy would be as follows. Given a DST, perform the first few factorizations by using common rules observed for that target topology. Partition the resulting formulation $u$ and all its children $c_0, c_1, \ldots c_j$. If the best children

$c_i$ has a latency lower than the parent formulation $n$, then choose $c_i$ and explore its children. This process repeats until the chosen formulation is a leaf or no children are found with lower latency than the parent. For a $2^n$-point DFT, this greedy approach evaluates $O(n)$ formulations at each level of exploration, thus $O(n^2)$ formulations would be evaluated in total.

## 5.3 FFT Formulation Exploration Heuristic

Algorithm 9 shows our proposed heuristic for formulation exploration using breakdown properties. Figure 5–4 illustrates a technique used by our algorithm to reduce the average number of explored formulations. Instead of exploring all children of a given formulation in order to decide which exploration path to follow, our algorithm explores only children obtained by factoring a specific split tree leaf. The chosen leaf represents the DST computation stage which can benefit most by factorization. The rationale for choosing this leaf is as follows.

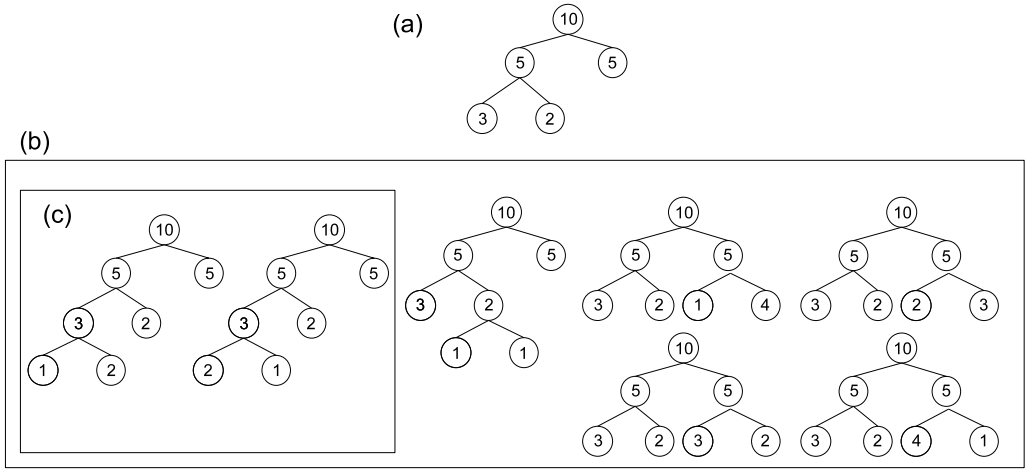



Figure 5–4: (a) A split tree for a $2^{10}$-point FFT. (b) All possible children split trees of (a). (c) children split trees exclusively factoring leaf '3'.

FFT formulation dataflow graphs are composed of a series of computational columns of smaller FFT kernels and corresponding inter-column connections. For instance, Figure 5–5 illustrates a split tree, formulation, and dataflow graph for a 16-point FFT. Notice that each split tree *leaf* corresponds to a computational column

in the DFG. Thus a split tree with $m$ leaves will have $m$ DFG computational columns and $m-1$ inter-column permutation columns.

Our graph partitioning algorithm can determine the most *cut-congested column*: the inter-column permutation column with the greatest contribution to overall partition cost. For instance, in Figure 5–5 column $A$ has the greatest contribution to total cut cost. The most cut-congested column lies between two computational columns corresponding to two split-tree leaves, e.g. 2 and 3 in our example. Through analysis of mega-trees it was determined that a strategy which lead to high-quality partitions is to factorize the split tree leaf adjacent to the most cut-congested column and with the highest size, e.g. leaf 3 in our example.

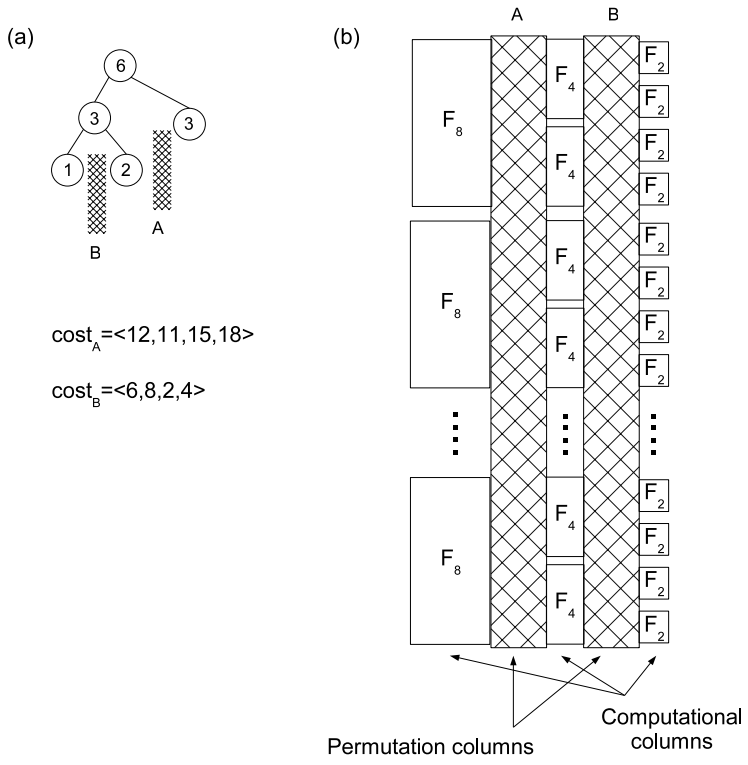



Figure 5–5: A split tree for a $n = 2^6$-point DFT and part of its corresponding DFG.

The algorithm starts by factoring the transform to a breakdown tree with a distribution of children's sizes that has been observed to lead to partition friendly formulations in smaller cases. This formulation is partitioned and its communication

costs are measured. Partitioning cost information is used to determine which leaf of the current formulation to split. The chosen leaf is split exhaustively into its children and the cost of each resulting formulation is measured. If any of the new formulations has a better cost than the current solution, te best among them is chosen as the current solution for further iterations. Exploration continues until no further improvement is obtained or the current solution allows no more factorization.

**Algorithm 9** Heuristic for formulation exploration based on top-down breakdown using CT-like factorization.

**Input**: Discrete signal transform Kronecker product expression $D$
**Output**: Optimized expression $D'$

1. $Cost \leftarrow \infty$
2. $D' =$ **InitialBreakdown**$(D)$;
3. $Cost' =$ **Partition**$(D')$
4. **While** ( $Cost' < Cost$ )
   - 4.1. $Cost \leftarrow Cost'$
   - 4.2. $Cost' \leftarrow \infty$
   - 4.3. $H =$ **NextChildToSplit**(D')
   - 4.4. **For every** split $(a, b)$ of $H$
     - 4.4.1. $D\_split =$ **Split**$(D,H,(a,b))$
     - 4.4.2. $Split\_Cost =$ **Partition**$(D\_split)$
     - 4.4.3. **If** $(Split\_Cost < Cost')$ **Then** $Best\_D \leftarrow D\_Split$
   - 4.5. **End For**
   - 4.6. $D' \leftarrow Best\_D$);
   - 4.7. $Cost' =$ **Partition**$(D')$
5. **End While**

Tables 5–2 and 5–3 show the results obtained for the formulation exploration method compared against the best result obtained with a Simulated Annealing DFG partition heuristic. Latency improvements of up to 13.3% are obtained over SA with a dramatic reduction of up to 99.4% in exploration time. Reduction in exploration time can be attributed to the fact that our method begins exploration by considering coarser FFT formulations, which represent coarser DFGs and thus require less time to partition. The SA heuristic lacks the capability to guide formulation exploration,

and must rely on exploring a *single* formulation that can expose all partition optimization opportunities. This corresponds to the finest-grained FFT formulation, whose large number of DFG nodes significantly impacts convergence-time.

The 4-ring topology benefits more than the 4-array from the deterministic exploration scheme. We believe that this can be attributed to the fact that the 4-ring topology, like the FFT, is symmetric, and thus may require less random/hill-climbing decisions to reach acceptable results.

Table 5–2: Results of FFT formulation exploration for various FFT sizes targeting a 4-Ring topology.

| | Latency (c-steps) | | | Exploration Time | | |
|---|---|---|---|---|---|---|
| Size | Form. Exp. | SA | Improvement | Form. Exp. | SA | Improvement |
| 512 | 122 | 136 | 10.3% | 50s | 10m 16s | 91.9% |
| 1024 | 235 | 253 | 7.1% | 5m 35s | 3h 16m | 97.2% |
| 2048 | 458 | 528 | 13.3% | 1h 34m | 1d 22h 2m | 96.6% |
| 4096 | 913 | 1047 | 12.8% | 1h 40m | 11d 7h 37m | 99.4% |

Table 5–3: Results of FFT formulation exploration for various FFT sizes targeting a 4-Array topology.

| | Latency (c-steps) | | | Exploration Time | | |
|---|---|---|---|---|---|---|
| Size | Form. Exp. | SA | Improvement | Form. Exp. | SA | Improvement |
| 512 | 191 | 183 | -4.4% | 5m | 10m 26s | 52.1% |
| 1024 | 370 | 361 | -2.5% | 26m 14s | 5h 20m | 91.8% |
| 2048 | 722 | 757 | 4.6% | 2h 18m | 1d 12h 34m | 93.7% |
| 4096 | 1437 | 1528 | 6.0% | 2d 2h 35m | 13d 22h 10m | 84.9% |

## 5.4 Partitioning the Discrete Cosine Transform

The encouraging results obtained with the exploration of FFTs using CT factorization rule (CT-FR) motivated our search for similar rules for discrete cosine transforms (DCT) [65]. CT-FR is an effective algorithm for the exploration of FFT formulations for two main reasons. First, it is capable of decomposing a size $n = mp$ FFT into the combination of *arbitrary* sized FFTs sized $m$ and $p$. This allows the generation of multiple breakdown strategies, each corresponding to a unique formulation and potentially having characteristics that enable its improved partitioning

for a given DHA. Second, regardless of the decomposition factors, the resulting formulation can still be implemented using the same basic functional primitives as the original, i.e. size-2 butterfly-twiddles. Functional primitive regularity is essential for effective hardware implementation since it simplifies control and allows a more effective use of hardware resources for computation.

In general, fast algorithms for the DCT do not posses the regularity found in FFT algorithms [1]. This is especially true for fast DCT algorithms that have not been developed with hardware implementation in mind. These algorithms concentrate on the reduction of operations; even at the cost of non-regular computational structures [93][94][95]. Several regular fast DCT algorithms have been reported over the years, yet none of them inherently comply with both of the features that make CT-FR desirable for FFTs. Püschel, et al. proposed several Cooley-Tukey like algorithms for the DCT which factor instances of size $n = mp$ into a combination of size-$m$ and size-$p$ terms [96]. However, the resulting structures do not naturally encourage hardware implementation. Other algorithms, such as those reported by Wang, Takala, Hsiao, and Nikara imply effective hardware structures, but lack the arbitrary decomposition capability [97] [98] [81] [1].

As part of our search for a decomposition algorithm for DCT that offers both features, we studied the several well-known regular DCT formulations. This analysis led is to the identify Nikara's perfect shuffle DCT (NPS-DCT) algorithm as the most suitable among those studied for distributed implementation and the development of a CT-like decomposition rule based on NPS-DCT.

## 5.5 DCT Regular Algorithms

The N-point 1-D DCT type-II transform matrix is defined as:

$$\left[\mathrm{DCT}_n^{\mathrm{II}}\right]_{mn} = \sqrt{\frac{2}{N}}\left[b_m \cos\left(\frac{m\left(n+\frac{1}{2}\right)\pi}{N}\right)\right],\ m,n = 0,1,\ldots,N-1 \qquad (5.4)$$

where $b_m$ is a scaling factor defined as:

$$b_m = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } m = 0 \text{ or } m = N \\ 1, & \text{if } m \neq 0 \text{ and } m \neq N \end{cases} \tag{5.5}$$

The redundancies found in the DCT matrix are exploited to obtain algorithms which reduce its computational complexity. For hardware implementation, it is essential that the fast DCT algorithm not only have a reduced number of expensive operations, such as multiplication, but must also have an overall regular structure. Regular computational structures facilitate the mapping of DCT operations to the limited resources available in hardware, meanwhile keeping a simplified control structure. In general, DCT algorithms have not obtained the regularity found in Cooley-Tukey FFT algorithms. However the regularity of some proposed DCT algorithms is enough to implement efficient hardware pipeline structures that meet performance requirements [1] [81]. The following sections discuss the four candidate formulations that were evaluated for their potential use within the DMAGIC methodology. Section 5.5.1 presents Pu̇schel's Cooley-Tukey-like DCT algorithm, which allows arbitrary decomposition at the expense of structural irregularities. Sections 5.5.2, 5.5.3, and 5.5.4 discuss three regular fast DCT algorithms, emphasizing their suitability for distributed implementation.

### 5.5.1 Püschel's Cooley-Tukey-like DCT Algorithms

Püschel, et al. reported several algorithms where DCTs of size $N = M \cdot P$ can be synthesized as the composition of DCTs size $M$ and $P$ along with interfacing permutations and additions. Equation 5.6 shows one of the proposed algorithms for the DCT-II.

$$\mathrm{DCT}_n^{\mathrm{II}}(r\pi) = C_{n,k} L_{n,k} \left(I_m \otimes \mathrm{DCT}_k^{\mathrm{II}}(r\pi)\right) \left(\bigoplus_{0 \leq i < k} \mathrm{DCT}_m^{\mathrm{II}}(r_i \pi)\right)^{L_{n,k}} R_{n,m} \tag{5.6}$$

where

$$C_{n,m} = \begin{bmatrix} I_k & Z_k & & \\ & I_k & \ddots & \\ & & \ddots & Z_k \\ & & & I_k \end{bmatrix}, \tag{5.7}$$

$$Z_n = \begin{bmatrix} & & & 0 \\ & & 0 & 1 \\ & \iddots & \iddots & \\ 0 & 1 & & \end{bmatrix}, \tag{5.8}$$

and

$$R_{n,m} = (I_k \oplus J_k \oplus I_k \oplus J_k \oplus \ldots) \ . \tag{5.9}$$

Here $J_k$ is $I_k$ with the order of the columns reversed.

In particular,

$$DCT_2^{(II)}(r_i\pi) = \text{diag}\left(1, {}^{1}\!/\!_{(2\cos({}^{r_i\pi}\!/\!_{2}))}\right) \cdot F_2 \tag{5.10}$$

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{5.11}$$

The $DCT_n^{II}(r_i\pi)$ term is called a *skew* DCT of type II, where $r_i$ has an effect on the multiplication coefficients but not on the structure of the DCT flowgraph.

The essential limitation of this algorithm when targeting a distributed implementation resides in matrix $C$. This matrix represents a varying number of additions and permutations depending on its indexes. Figure 5–6 shows the DFG interpretation of two different $C$ matrices. Repetitive functional primitives can be seen in each

of the graphs. However, since different $C$ matrices may be used throughout decomposition; the functional primitives will differ from stage to stage, making difficult its hardware implementation through uniform modules.

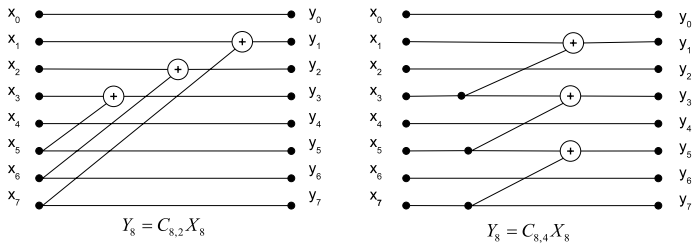



Figure 5–6: Dataflow graphs of $C_{8,2}$ and $C_{8,4}$ matrices.

Experimental results using our partitioning heuristics revealed latency increases of up to 100% when comparing formulations containing heterogenous $C$ matrices to formulations using a unique $C$ matrix. Thus, practical instances for hardware implementations using Equation 5.6 could be obtained as factorizations that produce $C$ matrices of the same type throughout all stages. This represents a very limited set of formulations. For example, Figure 5–7 shows the split trees of the DCT formulations for size=64.

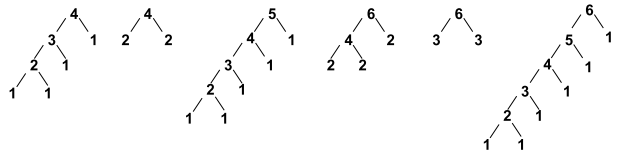


Figure 5–7: Practical split trees for 16, 32 and 64-point DCT when using Equation 5.6 for hardware implementation.

### 5.5.2 Hsiao and Tseng's DCT Algorithm

Hsiao and Tseng reported a 1-D DCT decomposition algorithm (HT-DCT) that computes a size $N = 2^n$ transform as $n$ stages of a butterfly-with-multiplication (BM) stages followed by $n-1$ post processing (PP) stages [81]. The PP stages add several of the results from the BM stages. Figure 5–8 shows the dataflow graph for

the type-II formulation of an 8-point 1D HT-DCT. The regularity obtained throughout both stages allowed its implementation as resource-efficient VLSI pipeline consisting of variations of two kernels: one to implement each of the butterfly stages and another for each of the post-processing stages. The explicit separations into two stages performing distinct operations preclude merging both functionalities into one functional primitive, since this would lead to a rather non-regular structure, as shown in Figure 5–9. This discourages the development of an arbitrary factorization scheme based on HT-DCT.

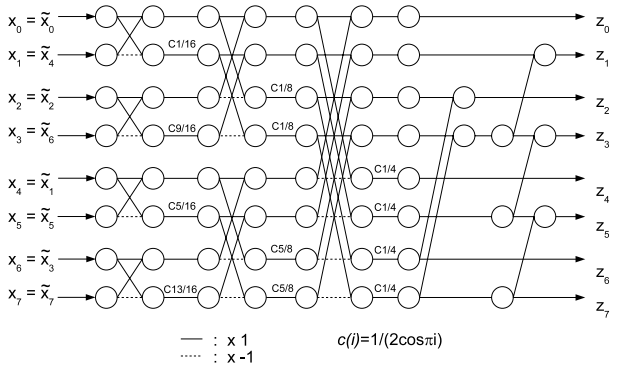



Figure 5–8: 8-point HT-DCT [81] data flow graph.

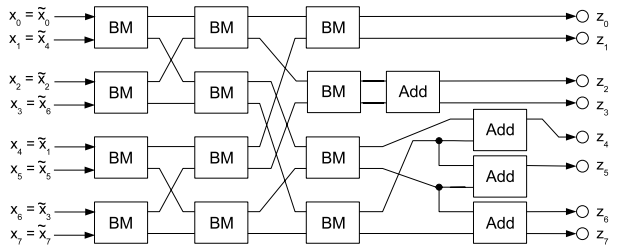



Figure 5–9: 8-point HT-DCT using a single functional primitive that performs both the BM and post-processing functionalities.

### 5.5.3 Morikawa's Simple Structured Fast DCT algorithm

As part of the development of his Pruning Discrete Cosine Transform algorithm, Wang used Morikawa's simple structured fast DCT (SS-FCT) algorithm [97]. SS-FCT is similar to HT-DCT in that it involves both butterfly-multiply and adding

structures. However, as shown in Figure 5–10, SS-DCT intermixes the adding structures and with the butterfly-network structures, making it more feasible to utilize a merged BM/Add functional primitive. Figure 5–11 shows a dataflow graph for a 8-point WP-DCT using a unified functional primitive. Notice that if all the successive BM and add stages were like in the second BN column we could implement a size-2, common-data integrated functional primitive, i.e. a functional primitive that would perform BM followed by addition on the same two points. However, this doesn't happen throughout the rest of the structure, as BMs are followed by permutations. This eliminates the practicality of integrating BM and adds as common-data unified functional primitive. Thus, we have a $2n - 1$ computational column structure, as shown in Figure 5–11. Once again, the lack of regularity and the explicit separation of functionalities limit SS-FCT chances for being factorized in a CT-like manner.

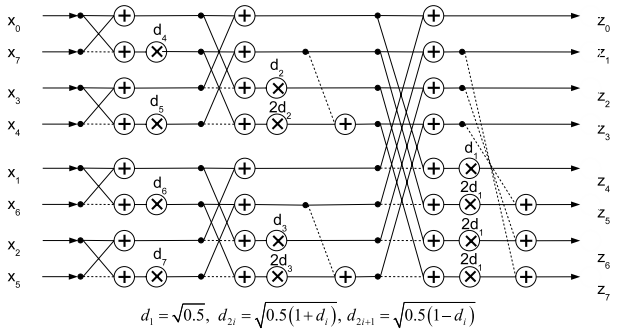



Figure 5–10: 8-point SS-FCT [97].

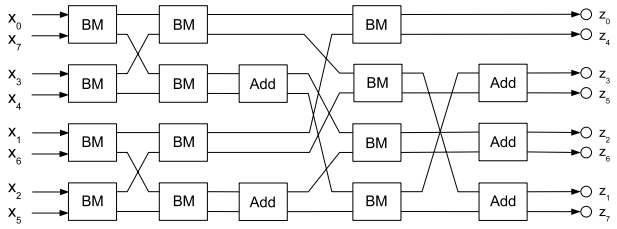



Figure 5–11: 8-point SS-FCT using functional primitive blocks.

### 5.5.4 Nikara's Perfect Shuffle DCT Algorithm

Nikara's Perfect Shuffle DCT algorithm (NPS-DCT) has several features that facilitate its pipelined hardware implementation [1]. Figure 5–12 shows a dataflow

graph of an 8-point NPS-DCT. First, it is almost perfectly regular across each of its computational columns, an essential characteristic to vertically fold its columns into a pipeline. Second, irregularities have been distributed across the computation in such manner that they operate on the same data sets as the previous BM structure. Third, data permutations between successive computational columns are kept to varying sizes of perfect-shuffle permutations, for which efficient pipeline structures have been proposed [59]. From the DFG, we can begin to identify a functional primitive that is common throughout the complete PS-DCT structure (identified in Figure 5–12 by dashed boxes). This serves as basis for the development of an arbitrary clustering/factorization technique, developed in the next Section.

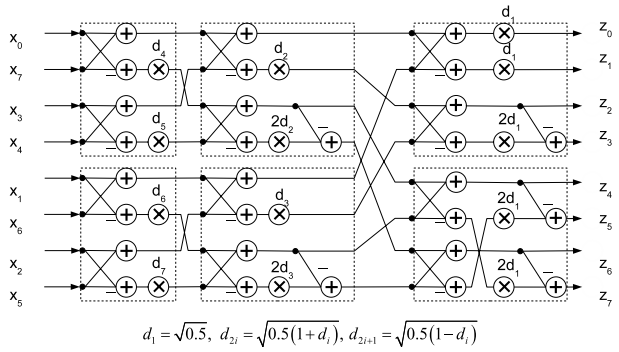



Figure 5–12: DFG for an 8-point NPS-DCT [1].

## 5.6 CT-like Decomposition for NPS-DCT

NPS-DCT is formulated as a product of sparse matrices using Kronecker Algebra operators. Let $C_{2^n}^{II}$ be a $2^n$-point DCT type-II formulation,

$$C_{2^n}^{II} = \sqrt{\frac{2}{2^n}} U_{2^n}^{(n-1)} \prod_{s=n-1}^{1} \left[ A_{2^n}^{(s)} \left( I_{2^{n-s-1}} \otimes L_{2^{s+1},2^2} \right) \right] A_{2^n}^{(0)} P_{2^k}^{H}, \tag{5.12}$$

where

$$A_N^{(s)} = M_N^{(s)} D_N^{(s)} H_N^{(s)} F_N, \tag{5.13}$$

$$M_{2^k}^{(s)} = \bigoplus_{i=0}^{2^{k-1}-1} \begin{pmatrix} 1 & 0 \\ -\mu_s(i) & 1 \end{pmatrix}, \tag{5.14}$$

$$D_{2^k}^{(s)} = \text{diag}\left(g_k\left(i,s\right)\right),\ i = 0, 1, ..., 2^k - 1\,, \tag{5.15}$$

$$g_k\left(i,s\right) = \left(2^{\mu_s(\lfloor i/2 \rfloor)} d\left(2^{k-s-1} + \left\lfloor i/2^{s+1} \right\rfloor\right)\right)^{f_k(i,s)}, \tag{5.16}$$

$$f_k\left(i,s\right) = \left(i \bmod 2\right) + \left(1 - \tau_0\left(i\right)\right)\left(1 - \tau_{k-1}\left(s\right)\right), \tag{5.17}$$

$$\tau_i\left(s\right) = \begin{cases} 0,\ s = i \\ 1,\ s \neq i \end{cases}, \tag{5.18}$$

$$H_{2^k}^{(s)} = \bigoplus_{i=0}^{2^{k-2}-1} \left(Q_4 R_4 Q_4\right)^{\mu_{s-1}(i)}, \tag{5.19}$$

$$F_{2^k} = I_{2^{k-1}} \otimes F_2\,, \tag{5.20}$$

and $P_{2^k}^H$ is a Hadamard permutation matrix of order $N$. To allow us to concentrate on the main computational components, we rewrite Equation 5.12 as follows:

$$C_{2^n}^{II} = \sqrt{\frac{2}{2^n}} U_{2^n}^{(n-1)} \Gamma_{2^n} P_{2^k}^H \tag{5.21}$$

where

$$\Gamma_{2^n} = \prod_{s=n-1}^{1} \left[A_{2^n}^{(s)} \left(I_{2^{n-s-1}} \otimes L_{2^{s+1},2^s}\right)\right] A_{2^n}^{(0)}\,, \tag{5.22}$$

As evidenced by Equation 5.22, actual arithmetic operations are performed by the $A_n^{(s)}$ terms. Upon closer examination, it was noticed that the sparse matrices involved in the computation of these $A_n^{(s)}$ terms are all composed of kernels operating

on 2 or 4 points. Thus, a complete formulation can be represented in terms of 4-input functional primitives, as illustrated by the dashed lines in Figure 5–12. Furthermore, these 4-input functional primitives can be systematically combined into clusters with $4p$ inputs. If we define the operation performed on each 4-input group to be a functional primitive $\Phi_4$, we can rewrite Nikara's formulation as follows:

$$\Gamma_{2^n} = \prod_{s=n-2}^{1} \left[ \left( I_{2^{n-2}} \otimes \Phi_4 \right) \left( I_{2^{n-s-2}} \otimes L_{2^{s+2},2^{s+1}} \right) \right] \left( I_{2^{n-2}} \otimes \Phi_4 \right) \left( I_{2^{n-2}} \otimes L_{4,2} \right) \left( I_{2^{n-2}} \otimes \Phi_4 \right) \tag{5.23}$$

In other words, PSDA consists of $n$ processing columns, each consisting of $2^{n-2}$ $\Phi_4$ components interconnected using the perfect-shuffle sequence. Let us define $\Phi_{2^n}$ as:

$$\Phi_{2^n} = \prod_{s=n-2}^{1} \left[ \left( I_{2^{n-2}} \otimes \Phi_4 \right) \left( I_{2^{n-s-2}} \otimes L_{2^{s+2},2^{s+1}} \right) \right] \left( I_{2^{n-2}} \otimes \Phi_4 \right) \tag{5.24}$$

Equation 5.24 can be split into three products of length $m-1$, 1, and $k-1$, where $n = m + k + 1$:

$$\begin{aligned} \Phi_{2^n} \quad = \quad & \prod_{s=n-2}^{n-m} \left[ \left( I_{2^{n-2}} \otimes \Phi_4 \right) \left( I_{2^{n-s-2}} \otimes L_{2^{s+2},2^{s+1}} \right) \right] \\ & \left( I_{2^{n-2}} \otimes \Phi_4 \right) \left( I_{2^{n-k-2}} \otimes L_{2^{k+2},2^{k+1}} \right) \\ & \prod_{s=k-1}^{1} \left[ \left( I_{2^{n-2}} \otimes \Phi_4 \right) \left( I_{2^{n-s-2}} \otimes L_{2^{s+2},2^{s+1}} \right) \right] \left( I_{2^{n-2}} \otimes \Phi_4 \right) \end{aligned} \tag{5.25}$$

For discussion purposes Equation 5.25 is rewritten as follows:

$$\Phi_{2^n} = A \left( I_{2^{n-2}} \otimes \Phi_4 \right) \left( I_{2^{n-k-2}} \otimes L_{2^{k+2},2^{k+1}} \right) B \tag{5.26}$$

, where,

$$A = \prod_{s=n-2}^{n-m} \left[ \left( I_{2^{n-2}} \otimes \Phi_4 \right) \left( I_{2^{n-s-2}} \otimes L_{2^{s+2},2^{s+1}} \right) \right] \left( I_{2^{n-2}} \otimes \Phi_4 \right) \tag{5.27}$$

$$B = \prod_{s=k-1}^{1} \left[ \left( I_{2^{n-2}} \otimes \Phi_4 \right) \left( I_{2^{n-s-2}} \otimes L_{2^{s+2},2^{s+1}} \right) \right] \left( I_{2^{n-2}} \otimes \Phi_4 \right) \tag{5.28}$$

Since, $n-2 = m+k-1$, we factor out $I_{2^m}$

$$B = I_{2^m} \otimes \prod_{s=k-1}^{1} \left[ \left( I_{2^{k-1}} \otimes \Phi_4 \right) \left( I_{2^{k-s-1}} \otimes L_{2^{s+2},2^{s+1}} \right) \right] \left( I_{2^{k-1}} \otimes \Phi_4 \right) = I_{2^m} \otimes \Phi_{2^{k+1}} \tag{5.29}$$

Expanding the $A$ term exposes the permutations between the computational columns (permutation terms are shown underlined):

$$\begin{aligned} A &= \prod_{s=n-2}^{n-m} \left[ \left( I_{2^{n-2}} \otimes \Phi_4 \right) \left( I_{2^{n-s-2}} \otimes L_{2^{s+2},2^{s+1}} \right) \right] \\ &= \left( I_{2^{n-2}} \otimes \Phi_4 \right) \underline{\left( L_{2^n,2^{n-1}} \right)} \left( I_{2^{n-2}} \otimes \Phi_4 \right) \underline{\left( I_{2^{n-2}} \otimes L_{2^{n-1},2^{n-2}} \right)} \ldots \\ &\quad \left( I_{2^{n-2}} \otimes \Phi_4 \right) \underline{\left( I_{2^{m-2}} \otimes L_{2^{n-m+2},2^{n-m+1}} \right)} \end{aligned} \tag{5.30}$$

The underlined terms in Equation 5.30 are expanded using the permutation property shown in Equation 4.14, yielding the underlined terms in Equation 5.31.

$$\begin{aligned} A &= \left( I_{2^{n-2}} \otimes \Phi_4 \right) \underline{\left( I_2 \otimes L_{2^{n-1},2^{n-2}} \right) \left( L_{4,2} \otimes I_{2^{n-2}} \right)} \left( I_{2^{n-2}} \otimes \Phi_4 \right) \\ &\quad \underline{\left( I_{2^{n-1}} \otimes L_{2^{n-2},2^{n-3}} \right) \left( I_2 \otimes L_{4,2} \otimes I_{2^{n-3}} \right)} \ldots \left( I_{2^{n-2}} \otimes \Phi_4 \right) \\ &\quad \underline{\left( I_{2^{m-1}} \otimes L_{2^{n-m+2},2^{n-m+1}} \right) \left( I_{2^{M-1}} \otimes L_{2^{n-2},2^{n-3}} \otimes I_{2^{n-2-(m-2)}} \right)} \end{aligned} \tag{5.31}$$

The $\left( I_X \otimes L_{4,2} \otimes I_Y \right)$ expressions are moved to the end of the formulation and rewritten as a multiplication series:

$$\begin{aligned} A &= \left( I_{2^{n-2}} \otimes \Phi_4 \right) \left( I_2 \otimes L_{2^{n-1},2^{n-2}} \right) \left( I_{2^{n-2}} \otimes \Phi_4 \right) \left( I_{2^{n-1}} \otimes L_{2^{n-2},2^{n-3}} \right) \\ &\quad \ldots \left( I_{2^{n-2}} \otimes \Phi_4 \right) \left( I_{2^{m-1}} \otimes L_{2^{n-m+2},2^{n-m+1}} \right) \prod_{q=m-2}^{0} \left( I_{2^{m-2-q}} \otimes L_{4,2} \otimes I_{2^q} \right) \otimes I_{2^{n-m}} \end{aligned} \tag{5.32}$$

Then, using Propery 10:

$$\begin{aligned} A &= \left(I_{2^{n-2}} \otimes \Phi_4\right)\left(I_2 \otimes L_{2^{n-1},2^{n-2}}\right)\left(I_{2^{n-2}} \otimes \Phi_4\right)\left(I_{2^{n-1}} \otimes L_{2^{n-2},2^{n-3}}\right)... \\ &\quad \left(I_{2^{n-2}} \otimes \Phi_4\right)\left(I_{2^{m-1}} \otimes L_{2^{n-m+2},2^{n-m+1}}\right) L_{2^m,2} \otimes I_{2^{n-m}} \end{aligned} \tag{5.33}$$

The same procedure is repeated, yielding the underlined term:

$$\begin{aligned} A &= \left(I_{2^{n-2}} \otimes \Phi_4\right)\left(I_{2^2} \otimes L_{2^{n-2},2^{n-3}}\right)\left(I_{2^{n-2}} \otimes \Phi_4\right)\left(I_{2^3} \otimes L_{2^{n-2},2^{n-3}}\right)... \\ &\quad \cdot \left(I_{2^{n-2}} \otimes \Phi_4\right)\left(I_{2^{M-2}} \otimes L_{2^{n-M+1},2^{n-M}}\right)\left(\underline{I_2 \otimes L_{2^M,2} \otimes I_{2^{n-M-1}}}\right)\left(L_{2^M,2} \otimes I_{2^{n-M}}\right) \end{aligned} \tag{5.34}$$

After $k$ iterations of this procedure, exhibited in Equations 5.30-5.34, another multiplication series is obtained:

$$\begin{aligned} A &= \left(I_{2^{n-2}} \otimes \Phi_4\right)\left(I_{2^k} \otimes L_{2^{n-k},2^{n-k-1}}\right)\left(I_{2^{k+1}} \otimes \Phi_4\right)\left(I_2 \otimes L_{2^{n-2},2^{n-3}}\right)... \\ &\quad \cdot \left(I_{2^{n-2}} \otimes \Phi_4\right)\left(I_{2^{k+M-2}} \otimes L_{2^{n-m+1},2^{n-m}}\right) \prod_{q=0}^{k-1} \left(I_{2^{k-1-q}} \otimes L_{2^m,2} \otimes I_{2^q}\right) \otimes I_{2^2} \end{aligned} \tag{5.35}$$

Property 11 is used, yielding:

$$A = I_{2^k} \otimes \prod_{s=m-1}^{0} \left[\left(I_{2^{n-2}} \otimes \Phi_4\right)\left(I_{2^{n-s-2}} \otimes L_{2^{s+2},2^{s+1}}\right)\right]\left(L_{2^{m+k-1},2^k} \otimes I_{2^2}\right) \tag{5.36}$$

Since $m + k - 1 = n - 2$, we finally obtain:

$$A = I_{2^k} \otimes \prod_{s=m-1}^{0} \left[\left(I_{2^{n-2}} \otimes \Phi_4\right)\left(I_{2^{n-s-2}} \otimes L_{2^{s+2},2^{s+1}}\right)\right]\left(L_{2^{n-2},2^k} \otimes I_{2^2}\right) \tag{5.37}$$

Thus,

$$\begin{aligned}\Phi_{2^n} &= A\left(I_{2^{n-2}} \otimes \Phi_4\right)\left(I_{2^{n-k-2}} \otimes L_{2^{k+2},2^{k+1}}\right) B \\ &= \underbrace{I_{2^k} \otimes \prod_{s=m-1}^{0} \left[\left(I_{2^{n-2}} \otimes \Phi_4\right)\left(I_{2^{n-s-2}} \otimes L_{2^{s+2},2^{s+1}}\right)\right]\left(I_{2^{n-2}} \otimes \Phi_4\right)}_{I_{2^k} \otimes \Phi_{2^{m+1}}} \\ &\quad \left(L_{2^{n-2},2^k} \otimes I_{2^2}\right)\left(I_{2^{n-k-2}} \otimes L_{2^{k+2},2^{k+1}}\right)\left(I_{2^m} \otimes \Phi_{2^{k+1}}\right)\end{aligned} \tag{5.38}$$

Finally,

$$\Phi_{2^n} = \left(I_{2^k} \otimes \Phi_{2^{m+1}}\right)\left(L_{2^{n-2},2^k} \otimes I_{2^2}\right)\left(I_{2^{n-k-2}} \otimes L_{2^{k+2},2^{k+1}}\right)\left(I_{2^m} \otimes \Phi_{2^{k+1}}\right) \tag{5.39}$$

For example, the following two decompositions of $DCT_{2^4}$ can be obtained using Equations 5.40 and 5.41. Figures 5–13a and 5–13b illustrate the two decompositions.

$$\Gamma_{2^4} = \left[\left(I_2 \otimes \Phi_8\right)\left(L_{4,2} \otimes I_4\right)\left(I_2 \otimes L_{8,4}\right)\left(I_4 \otimes \Phi_4\right)\right]\left(I_4 \otimes L_{4,2}\right)\left(I_4 \otimes \Phi_4\right) \tag{5.40}$$

$$\Gamma_{2^4} = \left[\left(I_4 \otimes \Phi_4\right) L_{16,8}\left(I_2 \otimes \Phi_8\right)\right]\left(I_4 \otimes L_{4,2}\right)\left(I_4 \otimes \Phi_4\right) \tag{5.41}$$

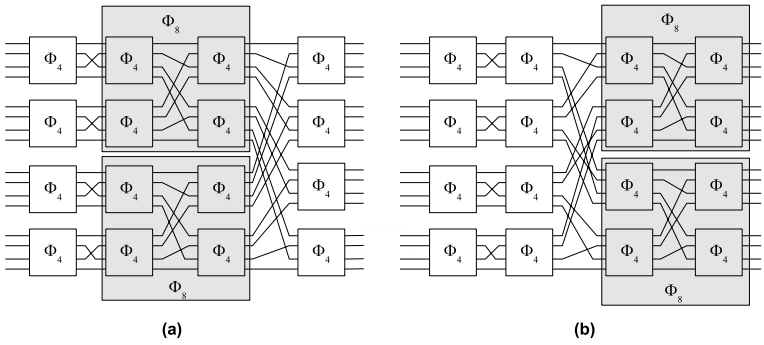



Figure 5–13: DFG for Equations 5.40 and 5.41.

## 5.7 Experiments

In order to assess the suitability of the various DCT formulations for distributed implementation, we partitioned them using the DMAGIC methodology. For DCT

formulations where only one algorithm is available per size, such as HT-DCT and SS-FCT, the methodology only conducts a graph partitioning without formulation exploration. When using the developed formulation, the CT-like property is used to explore alternative formulations, as directed by Algorithm 1. In all cases, a scheduling algorithm using the As-Soon-As-Possible heuristic is run after partitioning to map the various DCT nodes to the available hardware kernels in each device.

Figure 5–14 shows the target topology for our experiments. It consists of Virtex-2 Pro XC2VP7 FPGAs connected in ring topology with a crossbar which mainly serves to communicate non-adjacent devices. Latency for communication through the direct channels and the crossbar is 1 and 2 cycles, respectively. Width for all communication channels is 16 bits. Latency for operations (addition, subtraction and multiplication) is 1 cycle.

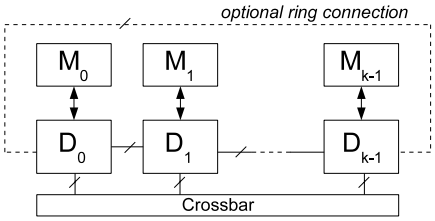



Figure 5–14: Target topology for experiments.

Tables 5–4 and 5–5 summarize our results. In the majority of cases, the use of the PSDA-CT algorithm within the formulation-exploration methodology obtained better latency values than the rest of formulations, with up to 18% improvement over the best of the other cases. DMAGIC did not achieve the best latency overall for the 128-point DCT, however, its result (57 c-steps) was only surpassed by the fully fine-grained PSDA formulation. Execution time was also significantly reduced when compared to the rest of formulations with reductions up to 83%. These results evidence the advantage of exploring different formulations of a given transform as part of the partitioning process. They also demonstrate that formulation-exploration can be performed in a non-exhaustive manner and yield acceptable results. Run

time reduction can be attributed to the fact that our formulation exploration strategy starts by considering coarser-granularity formulations. This requires a smaller number of nodes and consequently a reduced graph partitioning vs. the rest of formulations, whose format requires them to be in their finest-granularity. Among the previously proposed formulations, PSDA consistently obtained the best results, both in latency and execution time, highlighting the importance of regularity on distributed hardware implementation of DCT.

Table 5–4: Latency in c-steps for various sizes of DCT formulations.

| | PSDA | Hsiao | Wang | Pueschel | Best of rest | PSDA-CT | Latency decrease |
|---|---|---|---|---|---|---|---|
| 64 | 50 | 48 | 46 | 40 | 40 | **35** | 12.50% |
| 128 | 54 | 80 | 72 | 65 | 54 | **57** | -5.56% |
| 256 | 86 | 161 | 112 | 118 | 86 | **75** | 12.79% |
| 512 | 160 | 385 | 219 | 199 | 160 | **131** | 18.13% |
| 1024 | 318 | 669 | 387 | 404 | 318 | **267** | 16.04% |

Table 5–5: Execution time in seconds for various sizes of DCT formulations.

| | PSDA | Hsiao | Wang | Pueschel | Best of rest | PSDA-CT | Time decrease |
|---|---|---|---|---|---|---|---|
| 64 | 0.2 | 1 | 1 | 1 | 0.2 | **0.2** | 0.00% |
| 128 | 0.2 | 11 | 5.2 | 17 | 0.2 | **0.2** | 0.00% |
| 256 | 6 | 123 | 99 | 142 | 6 | **1** | 83.33% |
| 512 | 14 | 1234 | 934 | 1321 | 14 | **3** | 78.57% |
| 1024 | 189 | 18701 | 21509 | 16015 | 189 | **45** | 76.19% |

## 5.8 Summary

This chapter described the development of heuristics to guide exploration of the DST formulation space, which embodies an essential feature of our partitioning algorithm. The effect of various types of reformulations on the partition quality of FFTs was experimentally assessed. Observations were synthesized into a heuristic strategy using breakdown trees. The extension of such strategy for the discrete cosine transform was also detailed.

The results presented in this chapter evidence the impact of algorithmic formulation on partition quality, and support our decision to perform reformulations

as part of our partition optimization strategy. The developed heuristics achieved fast exploration of the formulation-space along with high-quality partitioning results. Further experimentation, following the guidelines set forth in this chapter, may allow the identification of further exploration strategies and the enhancement of our methodology.

# CHAPTER 6

# Results and Analysis

In this chapter we validate the effectiveness of the various parts of our approach as well as the unified methodology. First, the advantage of DST graph-level considerations is evaluated by comparing results of the graph partitioning heuristic with and without these features. Second, the impact of formulation exploration is appraised by quantifying the quality of results obtained by our formulation-space exploration heuristic. The complete methodology is compared to a previously published high-level partitioning heuristic to confirm its competitiveness. Finally, asymptotic behavior is measured to evidence that quality is maintained throughout problem scaling.

## 6.1 Graph Considerations

Graph considerations were integrated into our $k$-way extension of the Kernighan-Lin heuristic to improve partition solution quality and run time. Sections 6.1.1 and 6.1.2 discuss the rationale behind these considerations and present results to evidence their benefits.

### 6.1.1 Initial Partitioning Solution

Our partition/process algorithm uses initial linear horizontal partitions (described in Section 4.5.2), in an effort to preserve the data-access regularity of a DST throughout the partition process. Common formulations, when partitioned in this manner generate an initial solution that allows effective partition-space search and

improved solution quality with KL-H, as compared to randomly generated initial solutions.

Figure 6–1 shows the average and range of costs obtained by using initial linear horizontal partitions vs. initial random partitions for several FFT sizes. For visualization purposes, these graphs present cost as the sum of scalars in the cost vector $\Phi$ (presented in Section 4.5.1). For the majority of sizes, initial linear horizontal partitions obtained a lower average and range values than random partitions, with an average 10.34% and peak 15.51% cost reduction.

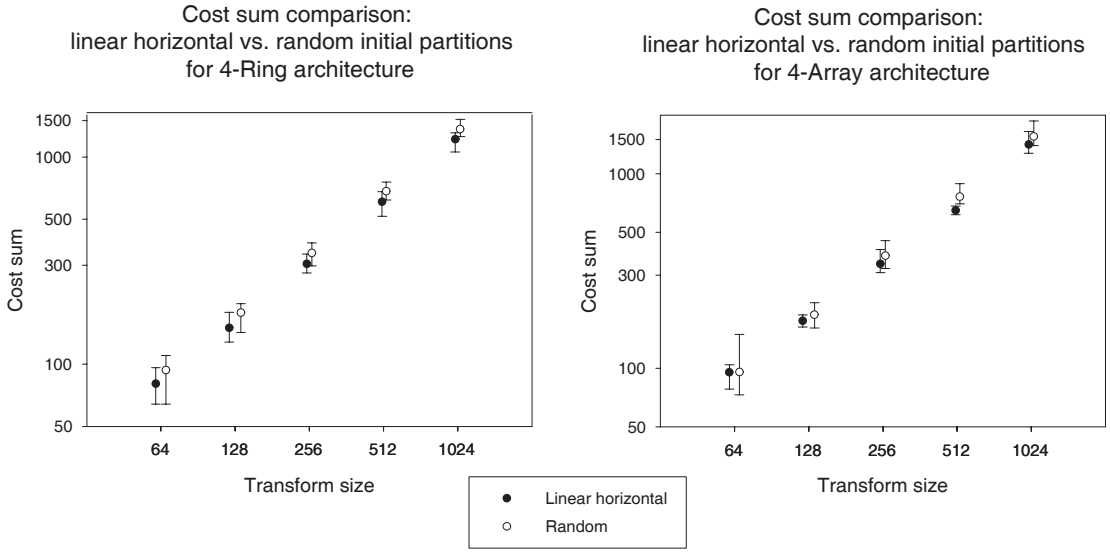



Figure 6–1: Comparison of cost sum for initial horizontal partitions vs. random for 4-Ring and 4-Array architectures.

Figure 6–2 compares initial linear horizontal partitions vs. initial random partitions in terms of iterations. A lower number of average iterations was required when using initial linear horizontal partitions, reducing iterations by an average of 16.12% and up to 34.00%.

### 6.1.2 Stage-limited Node Swapping

An additional DST-oriented consideration taken in KL-H was the limitation of node swaps to nodes in the same computational stage. It could be conjectured that limiting swap opportunities during optimization could have a negative effect on

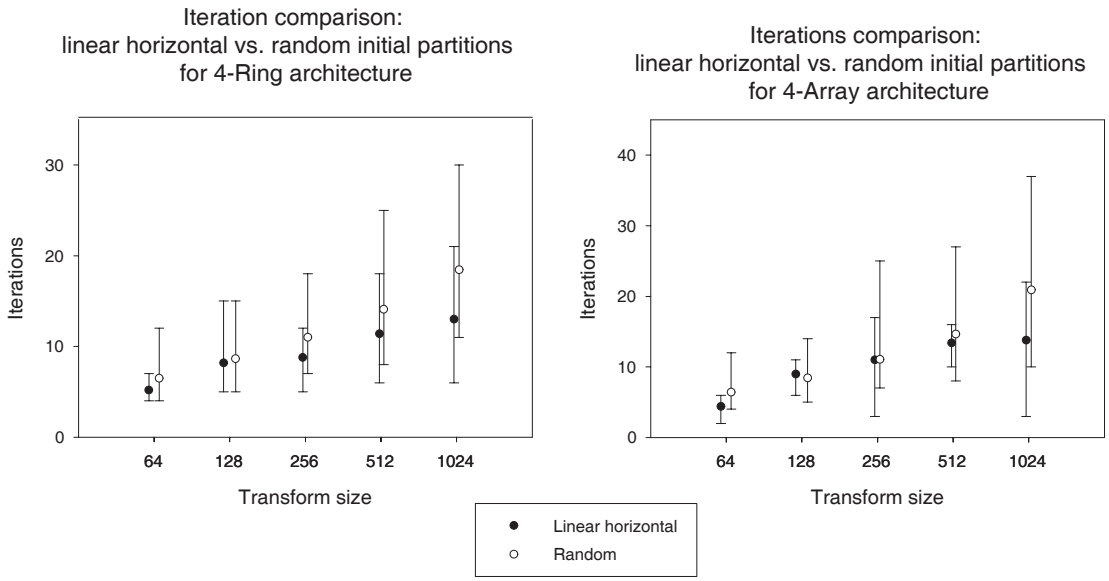



Figure 6–2: Comparison of iterations for initial horizontal partitions vs. random for 4-Ring and 4-Array architectures.

partitioning results, since a considerable part of the solution space is left unexplored. Nevertheless, this impact was found to be minimal. Furthermore, exploring swaps outside the computational stage significantly increased exploration time.

Figures 6–3 and 6–4 show latency and run time graphs for various FFT sizes, comparing the results of activating and deactivating node-swap restrictions throughout the graph partitioning process. Average latency was slightly higher when using these restrictions (an average increase of 4.63% and peak increase of 13.87%), however run time was significantly lower (an average decrease of 84.49% and peak decrease of 88.55%). Our methodology uses the graph partition/placement process as part of a major formulation-exploration optimization loop. Thus, any additional run time incurred in the graph partitioning heuristic will be amplified by the formulation-level iterations. In the case of the stage-limited swapping consideration, the run time advantage overpowers the increase in latency. Regardless, this slight increase in latency is counterbalanced by the rest of the graph considerations and the formulation-level exploration. This justifies our decision to utilize the stage restrictions within our methodology.

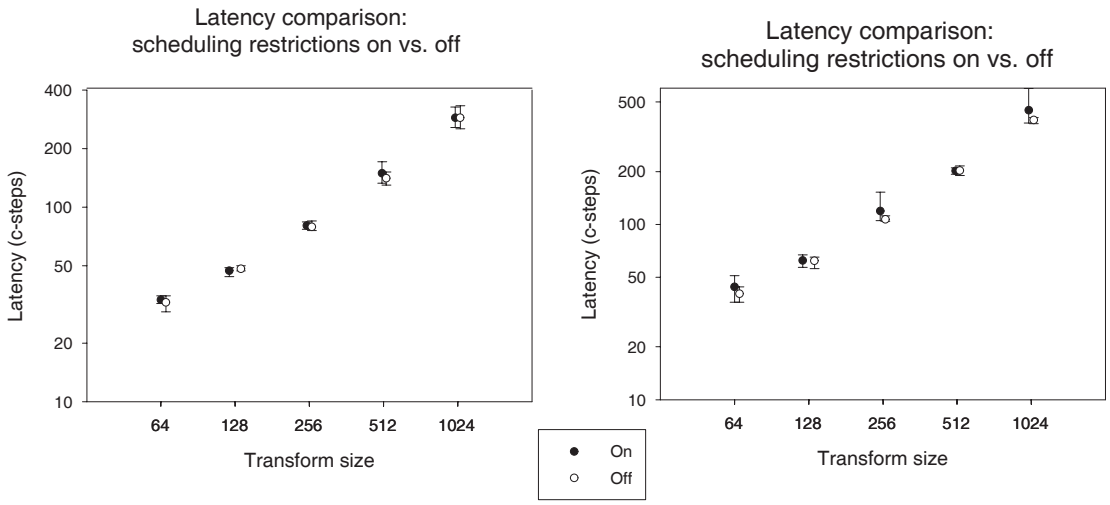



Figure 6–3: Comparison of latency with and without stage-restricted swaps.

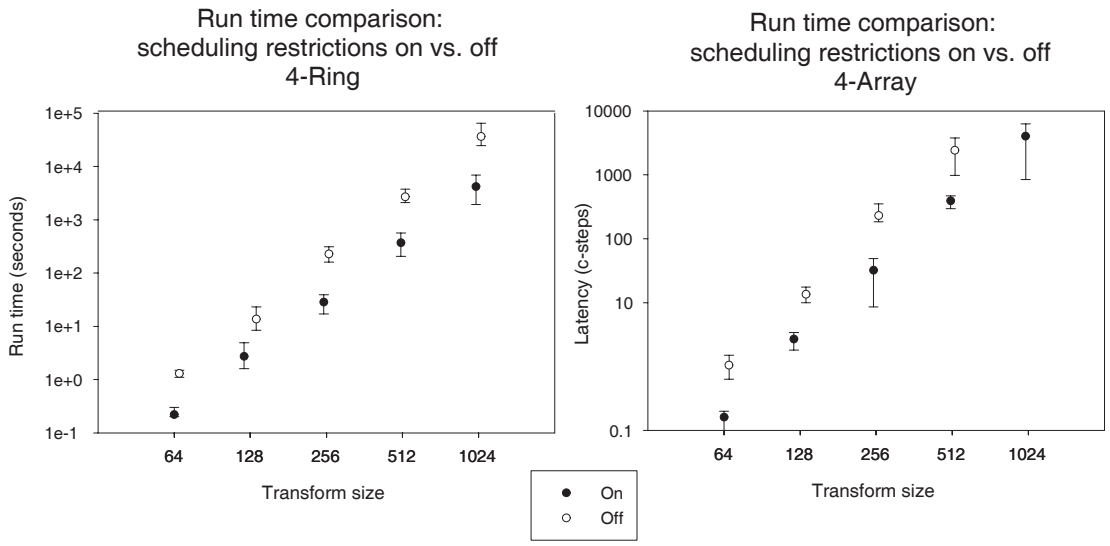



Figure 6–4: Comparison of run time with and without stage-restricted swaps.

## 6.2 Effect of Formulation Exploration

In terms of run-time, it is impractical to exhaustively search the solution space of equivalent formulations of a DST, as it grows exponentially with transform size. Our proposed search heuristic, presented in Section 5.3, performs a greedy exploration, starting with a coarse formulation and moving onto increasingly finer ones, running in $O(n^2)$ time.

Table 6–1: Formulation exploration heuristic performance - 4-Ring topology.

| Size | Formulation universe | | Heuristic | Percentile | % better |
|---|---|---|---|---|---|
| | Range Bottom | Range Top | Latency | | |
| 16 | 18 | 21 | 18 | 50.00% | 0.00% |
| 32 | 23 | 32 | 24 | 66.00% | 4.00% |
| 64 | 29 | 52 | 29 | 97.33% | 0.00% |
| 128 | 43 | 90 | 43 | 99.86% | 0.00% |
| 256 | 70 | 159 | 71 | 98.34% | 0.07% |

Table 6–2: Formulation exploration heuristic performance - 4-Array topology..

| Size | Formulation universe | | Heuristic | Percentile | % better |
|---|---|---|---|---|---|
| | Range Bottom | Range Top | Latency | | |
| 16 | 19 | 26 | 20 | 57.14% | 28.57% |
| 32 | 26 | 41 | 27 | 58.00% | 28.00% |
| 64 | 36 | 68 | 38 | 87.70% | 4.28% |
| 128 | 58 | 119 | 59 | 90.82% | 1.64% |
| 256 | 102 | 218 | 103 | 93.32% | 2.14% |

Tables 6–1 and 6–2 show the results of the exploration heuristic as compared to exhaustive results from formulation-space of FFT sizes 32 through 256. To highlight the performance of the heuristic, the tables show the percentile of heuristic results as well as the percent of exhaustive results that are better than those obtained by the heuristic. The heuristic achieved the best overall result in several cases, while maintaining a superior percentile, specially in the larger cases.

Interestingly, as for the other graph considerations, the 4-Ring architecture benefits more than the 4-Array from our methodology's features. We conjecture that this could be caused, among other things, by the pairing of the DHA connection symmetry and the symmetry implied by the moves (swaps) done as part of the KL heuristic. Each device in the 4-Ring topology has two neighbor connections and a crossbar connection to access non-neighbors. Thus, a move from one partition to other may have less impact on the nodes' connectivity than in the 4-Array, where the first an last devices have one connection less than the rest. It could also be conjectured that topologies with low symmetry could benefit from a certain degree of non-deterministic decisions as part of the optimization process.

## 6.3 Comparison Against Established Methodology

A significant drawback of previously reported high-level partitioning methods is their failure to compare results with other proposed techniques. The lack of accepted high-level partitioning benchmark sets and the diversity of target platforms makes it difficult to perform a one-to-one comparison. To further complicate matters, a complete algorithm-to-hardware mapping solution requires the interaction of several tools (e.g. partition engine, scheduler, estimators) which may not be available or properly documented to allow third party validation.

We shall compare the results of our methodology to the 'data flow graph partitioning' (DFGP) strategy proposed by Srinivasan, et al. [7]. Even though the original DFGP prototype is not available, the method is sufficiently well documented and close in objective to ours as to allow an acceptable implementation.

### 6.3.1 Srinivasan's DFGP Methodology

In DFGP, input algorithms are specified as fine grained dataflow graphs, where each node represents a single operation, such as a multiplication or addition. Additional specification files establish the target architecture: its connection topology and the capacity of its devices. The methodology begins by performing scheduling/allocation of the DFG onto a single (virtual) device with as much resources as all DHA devices combined. Then, the DFG is partitioned with a genetic algorithm-based partition engine, which relies on the genetic operations of selection, crossover and mutation to evolve a population of mostly random initial partitioning solutions. The most relevant features of DFGP genetic algorithm are as follows:

1. Encoding: Each population member is represented as an integer array in which the value of location $i$ represents the partition assigned to node $i$.
2. Population: Population is kept constant throughout optimization at 100 to 200 members, depending on the problem size.

3. Fitness function: The fitness of a solution $x$ is computed as follows:

$$fitness(x) = \frac{1}{1 + AreaPenalty\,(x) + InterconnectPenalty\,(x)}\,, \tag{6.1}$$

where $AreaPenalty\,(x)$ is proportional to the ratio of excess area needed by $x$ over the total area provided by the device. $InterconnectPenalty\,(x)$ is proportional to the number of cut edges in $x$.

4. Selection mechanism: The selection operator probabilistically selects highly fit solutions in the current generation and crosses them to obtain future generations. The DFGP's selector operator tosses a biased-coin and, depending on the result of the toss, recursively branches either to the lower or upper half of the array. The process continues until the size of the array is one, at which point the single solution in the array is returned. A call to the selection mechanism with a highly biased probability, e.g. 0.9, strongly biases selection to the fittest members.
5. Crossover: The selection mechanism is used to select two parents for the crossover. To guarantee better population diversity, the selection function is called with probability 0.8 for the first parent and 0.55 for the second. Once the parents have been chosen, a uniform crossover operation generates two offspring. The content from chromosome $i$ in each offspring is copied from the same position in either of its parents. The parent that donates each chromosome to each offspring is determined randomly by an unbiased coin toss.
6. Mutation: Mutation randomly changes a small percentage (5%) of chromosomes of a small percentage (10%) of members. This introduces changes that allow generation member's to possibly improve their fitness and/or escape local minima.
7. Stop criterion: No specific exit conditions are explicitly documented for DFGP. In our implementation, iterations are stopped when the running average fitness of 100 generations has not improved above 0.10%.

After a partitioning solution has been obtained, latency is estimated in DFGP by the formula:

$$L = L_{sched} + L_{I/O} + L_{data} + L_{synch} \tag{6.2}$$

where $L_{sched}$ is the *schedule* latency, i.e. the number of time steps in the scheduled DFG. Recall that in DFGP scheduling assumes a single FPGA, thus $L_{sched}$ does not include time steps needed for transferring data between devices. $L_{I/O}$ is the number of cycles for reading input data from memory and writing output data to memory. In the Wildforce architecture reading takes three cycles, while writing takes one cycle. $L_{data}$ is the *data transfer* latency, i.e. number of cycles to communicate data from FPGA to FPGA using channels in the interconnection network or through shared memory. The Wildforce platform requires two cycles and four cycles for these transfers, respectively. Finally, $L_{synch}$ is the number of cycles needed for synchronization/handshaking among devices interchanging data.

As implied by Equation 6.2, communications in DFGP are assumed to occur non-concurrently with execution. In our methodology, a post-partitioning schedule is capable of arranging concurrent communications and executions. This difference is highly influential in terms of the final latency achieved by each methodology. Thus, in order to properly compare both methodologies, each has been adapted to the assumption of the other before comparison. For instance, a comparison is done between our methodology with no concurrent communication/execution vs. DFGP. Another comparison is done with DFGP assuming concurrency vs. our methodology.

### 6.3.2 Results Comparison

Table 6–3 compares the best result obtained with the DFGP implementation to those of our methodology for various sizes of FFTs, assuming no concurrency. The target distributed hardware architecture for the original SBPH FFT partitioning results was a Wildforce 4013 by Annapolis Systems, which had enough resources for a small FFT (size 16) [7]. To be fair in our comparison, we assume that for

bigger FFT sizes, the target DHA is a scaled-up version of the Wildforce 4013, providing enough resources to accommodate bigger FFTs. When no concurrency is assumed, latency is essentially based on the cumulative size of the cutsize. Table 6–3 shows that in most cases our methodology produced better latencies, even though its objective was not necessarily the minimization of the cumulative cutsize. This advantage can be mainly attributed to the granularity in the DFGP input graphs. DFGP has to partition a fine-grained graph and is completely unaware of compact structures, e.g. butterflies, whose partition is counterproductive in most instances. On the other hand, our methodology relies on coarser graphs which tend to cluster highly connected structures, thus avoiding many high-cutsize solutions.

It must be stressed that these comparison are being done against the *best* DFGP results. The stochastic nature of the GA optimization engine gives variability to their results. Furthermore GA relies on convergence over a much larger size of iterations. On the contrary, our methodology is deterministic and requires much less iterations.

Table 6–3: FFT Results of SBPH vs. our methodology, assuming no concurrency.

| Size | DFGP | | | Ours | | | Latency Improvement |
|---|---|---|---|---|---|---|---|
| | $L_{exec}$ | $L_{comm} + L_{synch}$ | Latency | $L_{exec}$ | $L_{comm} + L_{synch}$ | Latency | |
| 16 | 14 | 130 | 144 | 15 | 116 | 131 | 9.0% |
| 32 | 24 | 260 | 284 | 24 | 234 | 258 | 9.2% |
| 64 | 54 | 442 | 496 | 51 | 508 | 559 | -12.7% |
| 128 | 123 | 1001 | 1124 | 115 | 916 | 1031 | 8.3% |
| 256 | 275 | 2490 | 2765 | 259 | 1712 | 1971 | 28.7% |

Table 6–4 presents results for partitioning various FFT sizes assuming that communications and execution can occur concurrently. Our methodology has a clear advantage over the best DFGP results. This advantage can be mainly attributed to two considerations. First, our methodology explicitly emphasizes the balance of communications among the available channels and execution among the various processing units. This improves the efficacy with which the scheduler is able to

accommodate concurrent operations and communications. Second, it has been demonstrated that scheduling after partitioning achieves better results than the other way around [47].

Table 6–4: FFT Results of SBPH vs. our methodology, assuming concurrency.

| Size | DFGP | Ours | Improvement |
|---|---|---|---|
| 16 | 53 | 45 | 15.1% |
| 32 | 116 | 85 | 26.7% |
| 64 | 226 | 175 | 22.6% |
| 128 | 501 | 373 | 25.5% |
| 256 | 1047 | 847 | 19.1% |

Tables 6–5 and 6–6 show a comparison between the best DFGP results and our methodology for various DCT sizes. Interestingly, even though DCTs are not as regular as FFTs, our method offers larger improvements as compared to DFGP than for FFTs.

Table 6–5: DCT Results of SBPH vs. our methodology, assuming no concurrency.

| | DFGP | | | Ours | | | Latency |
|---|---|---|---|---|---|---|---|
| Size | Lexec | Lcomm + Lsynch | Latency | Lexec | Lcomm + Lsynch | Latency | Improvement |
| 16 | 16 | 144 | 160 | 16 | 128 | 144 | 10.0% |
| 32 | 21 | 336 | 357 | 18 | 184 | 202 | 43.4% |
| 64 | 32 | 592 | 624 | 40 | 412 | 452 | 27.6% |
| 128 | 69 | 976 | 1045 | 91 | 836 | 927 | 11.3% |
| 256 | 156 | 1768 | 1924 | 211 | 1548 | 1759 | 8.6% |

Table 6–6: DCT Results of SBPH vs. our methodology, assuming concurrency.

| Size | DFGP | Ours | Improvement |
|---|---|---|---|
| 16 | 85 | 60 | 29.4% |
| 32 | 135 | 89 | 34.1% |
| 64 | 213 | 161 | 24.4% |
| 128 | 389 | 330 | 15.2% |
| 256 | 749 | 595 | 20.6% |

## 6.4 Scaling the Suboptimality

To demonstrate the scalability of our heuristics we use Hagen's suboptimality principle [99]. This approach states that given a heuristic $H$, an instance $I$ and the solution cost for the instance $c_H(I)$, if a new instance $kI$ is constructed by scaling

Table 6–7: Run times for DCT

| | Run time (seconds) | | |
|---|---|---|---|
| Size | DFGP | Ours | Improvement |
| 16 | 3.70 | 0.01 | 99.7% |
| 32 | 17.50 | 0.02 | 99.9% |
| 64 | 23.50 | 0.16 | 99.3% |
| 128 | 89.00 | 0.90 | 99.0% |
| 256 | 190.00 | 12.30 | 93.5% |

the original by $k$, then $c_H(kI) > k \cdot c_H(I)$. The suboptimality of a heuristic can be measured by how much it departs from the optimal scaling cost $k \cdot c_H(I)$:

$$\eta_H(k) = \frac{c_H(kI)}{kc_H(I)} - 1 \tag{6.3}$$

where a value $\eta_H(k) \geq 0$.

The scalability of our partitioning methodology in terms of cutsize can be readily verified using Hagen's principle by using FFTs as the scalable instance $I$. Bornstein, et al. demonstrated that the minimum bipartition cutsize of a butterfly network of input size-$n$ is $2\left(\sqrt{2}-1\right)n + o(n) \approx 0.82n$ [82]. If we think of a $k$-way partitioning solution as being composed of multiple bipartitions, then we can conjecture that the minimum cutsize of a $k$-way partitioning solution also grows at a rate $O(n)$. The FFT DFG is an example of a butterfly network, therefore, scaling an FFT by $2^m$ should optimally have a $2^m$ effect on its partition cutsize.

Tables 6–8 and 6–9 show the results of our partitioning methodology, in terms of cutszie, for a range of FFT sizes partitioned to 4-Ring and 4-Array topologies. Cost is measured as the sum of all scalars in the cost vector $\Phi$. Results for the 4-Ring evidence the near-optimal behavior of the heuristic with respect to the base case: 32-point FFT. Results for the 4-Array show more variation and serve to demonstrate the importance of base-case results when using Hagen's principle. The third and fifth columns of Table 6–9 show suboptimality factors ($\eta_H$) based on the 32-point FFT and 256-point FFT results, respectively. The 32-point FFT result was not

particularly good, hence the negative suboptimality factor deviations. On the other hand, choosing a good result as base case, e.g. 256-point FFT, gives a worst-case scaling scenario. Anyhow, the suboptimality factors of 13.22% and below are maintained throughout, evidencing an excellent asymptotic behavior of the heuristic.

Table 6–8: Suboptimality comparison based on cost sums for 4-Ring topology .

| Size | Cost Sum | $\eta$ | Deviation |
|---|---|---|---|
| 32 | 32 | 1 | - |
| 64 | 64 | 2 | 0.00% |
| 128 | 128 | 4 | 0.00% |
| 256 | 256 | 8 | 0.00% |
| 512 | 512 | 16 | 0.00% |
| 1024 | 1024 | 32 | 0.00% |
| 2048 | 2070 | 64.6875 | 1.07% |

Table 6–9: Suboptimality comparison based on cost sums for 4-Array topology .

| Size | Cost Sum | $\eta_{32}$ | Deviation | $\eta_{256}$ | Deviation |
|---|---|---|---|---|---|
| 32 | 42 | - | - | - | - |
| 64 | 85 | 2.02381 | 1.19% | - | - |
| 128 | 164 | 3.904762 | -2.38% | - | - |
| 256 | 295 | 7.02381 | -12.20% | - | - |
| 512 | 659 | 15.69048 | -1.93% | 2.233898 | 11.69% |
| 1024 | 1326 | 31.57143 | -1.34% | 4.494915 | 12.37% |
| 2048 | 2672 | 63.61905 | -0.60% | 9.057627 | 13.22% |
| 4096 | 5158 | 122.8095 | -4.06% | 17.48475 | 9.28% |

Although minimizing the cost function is the objective of partition/placement process within our methodology, the global objective is latency. The latency of a partitioning solution is bounded as follows:

$$\max\left[L_{Exec}, L_{Comm}\right] \leq L \leq \left(L_{Exec} + L_{Comm}\right) , \tag{6.4}$$

where $L_{Exec}$ is the execution latency of the implementation, i.e. the number of cycles spent in the actual processing operations of the DST. $L_{Comm}$ is the communication latency, i.e. the number of cycles spent communicating data points between the DHA devices. The lower bound represents a situation where execution and communication

cycles can be perfectly overlapped. The upper bound represents nonoverlapping execution and communication processes. Although $L_{Exec}$ can be known exactly for each architectural module and $L_{Comm}$ can be estimated from the cost function, exact latency value is only known after scheduling. Although conceptually simple, the ASAP scheduler may introduce non-linearity with respect to $L_{Exec}$, $L_{Comm}$, and $L$. For instance, it was experimentally assessed that in some occasions, two different partitioning solutions with the same cost had slightly different latencies. This is due, among other things, to the fact that our chosen cost function does not store information how the cuts are distributed throughout the DST computational stages. A more uniform distribution of cuts might make it easier for the scheduler to overlap execution and communication latencies.

Estimating suboptimality in terms of latency requires us to take into consideration the effect of the scheduler. Scheduler optimization was not the main purpose of this work. Thus, no explicit efforts were taken to make it scale properly. Nevertheless, we can exactly compute the bounds of a scheduled partition solution to have an idea of how well scheduled results scale. Once again, the FFT is chosen as instance because of its proven lower bounds on cutsize. On a DHA with $2^q$ devices, where computation is evenly distributed, the scaling of a $2^n$-point FFT to a $2^{n+m}$-point implies an increase from $(n \cdot 2^n)/2^{q+1}$ to $((n+m) \cdot 2^{n+m})/2^{q+1}$ butterfly+twiddle operations. It also implies an increase in cutsize of $O(n)$. Using these two considerations, a $2^m$ scaling in FFT size implies an optimal scaling bounded by:

$$\frac{\max\left[\frac{(n+m)2^{n+m}}{2^{q+1}}t_b, 2^{n+m}t_c\right]}{\frac{n2^n}{2^{q+1}} + 2^n t_c} \leq k(2^m) \leq \frac{\frac{(n+m)2^{n+m}}{2^{q+1}}t_b + 2^{n+m}t_c}{\max\left[\frac{n2^n}{2^{q+1}}, 2^n t_c\right]} \tag{6.5}$$

clock steps, where $t_b$ is the number of c-steps necessary for the execution of a butterfly+twiddle element, and $t_c$ is the latency for communicating each point that is transfered through the interconnection resources. The lower bound represents a situation where the $2^n$-point FFT has no concurrent operations/communications,

while the $2^{n+m}$-point FFT is perfectly concurrent. The upper bound represents the inverse situation. Results of the latency suboptimality experiments are summarized in Tables 6–10 and 6–11. Observe that, given the effect of the scheduler and the available resources, latency is not expected to increase at the same exact rate that FFT sizes are scaled.

Table 6–10: Suboptimality comparison based on latency for 4-Ring topology .

| Size | Latency | $\eta$ | Optimal Scaling | | Deviation from L.B. |
|---|---|---|---|---|---|
| | | | Lower Bound | Upper Bound | |
| 64 | 29 | - | - | - | - |
| 128 | 43 | 1.48 | 0.51 | 9.44 | 189.49% |
| 256 | 71 | 2.45 | 1.17 | 19.56 | 109.12% |
| 512 | 131 | 4.52 | 2.63 | 40.44 | 71.49% |
| 1024 | 264 | 9.10 | 5.85 | 83.56 | 55.52% |
| 2048 | 553 | 19.07 | 12.88 | 172.44 | 48.07% |
| 4096 | 1130 | 38.97 | 28.10 | 355.56 | 38.68% |

Table 6–11: Suboptimality comparison based on latency for 4-Array topology .

| Size | Latency | $\eta$ | Optimal Scaling | | Deviation from L.B. |
|---|---|---|---|---|---|
| | | | Lower Bound | Upper Bound | |
| 64 | 40 | - | - | - | - |
| 128 | 65 | 1.63 | 0.51 | 9.44 | 217.26% |
| 256 | 103 | 2.58 | 1.17 | 19.56 | 119.95% |
| 512 | 204 | 5.10 | 2.63 | 40.44 | 93.61% |
| 1024 | 407 | 10.18 | 5.85 | 83.56 | 73.82% |
| 2048 | 786 | 19.65 | 12.88 | 172.44 | 52.59% |
| 4096 | 1130 | 28.25 | 28.10 | 355.56 | 0.54% |

Results indicate appropriate behavior of our heuristic, as evidenced by the fact that the suboptimality factor is consistently closer to the lower bound than to the upper bound. In fact, deviation with respect to the optimal lower bound is monotonically decreasing, which promises competitive results for larger FFT sizes.

## 6.5 Summary

In this chapter, the validity of the graph partition/placement and formulation exploration processes, as well as our methodology as a whole have been evidenced empirically. Graph level considerations significantly helped in speeding partition/placement heuristic convergence while offering small improvements to partition quality. The formulation-exploration heuristic achieved competitive results as compared to the universe of solutions, with reduced exploration time. Finally, an analysis of the methodology's scalability reveled that quality of results is maintained throughout a range of input problem sizes.

# CHAPTER 7

# Conclusions

This thesis presented a high-level partitioning methodology for discrete signal transforms onto distributed hardware architectures. Multiple opportunities were found throughout the methodology's design to improve the effectiveness of the partitioning process by taking advantage of DST and DHA features. For instance, the graph partitioning heuristic's cost function was adapted to more adequately represent the main implementation constraint when partitioning to DHAs. Furthermore, the initial solution generation and solution exploration functions were adjusted based on structural DFG qualities of fast regular DSTs. An intra-level architectural model was tailored to the typical computational/data-path requirements of these transforms, allowing for accurate resource and latency estimation. The development of graph-level partitioning tools and a KPA to DFG converter allowed an assessment of the impact of formulation-level transformations on partition-solution quality. A formulation-level greedy heuristic was derived through systematic analysis of assessment results.

The integration of graph-level and algoritmic-level strategies, allowed DMAGIC to obtain improved results over a generic DFG-based reported method. Results were improved both in terms of implementation latency and run-time required for solution exploration.

The work presented in this thesis represents, to the best of our knowledge, the first attempt at combining the algorithmic and graphic abstraction levels to improve partition of DSTs to DHAs. Besides evidencing that DMAGIC fuses both levels appropriately, it is the intent of this thesis to foster further research in this area. The improved understanding of the interaction between these two abstraction levels and their impact on hardware implementations promises to have a powerful impact on the design-time and performance of future systems.

The following sections summarize the main aspects discussed throughout this dissertation, salient contributions and future research directions.

## 7.1 Contributions

The design of the DMAGIC methodology required fusing two otherwise disjoint strategies: the treatment of DSTs at the algorithmic level, common in the signal processing arena, and the automated treatment of algorithms at the graph level, typical in EDA. The main contributions of this work will be useful in each of those areas and toward the original goal of facilitating the implementation of DSTs on distributed hardware architectures. We consider the following to be the main contributions of this work.

- **An automated and extensible methodology for conversion of Kronecker product algebra (KPA) formulations into DFGs**: This tool proved highly valuable for the practical study of formulation effect on partition quality. KPA formulations have been converted to code as part of automated code generation techniques [44]. However, to our knowledge, no tool has been documented that will output a dataflow graph from a KPA formulation. Moreover, our KTG tool maintains data order throughout the DFG by using the concept of levels. Data order information can be used to maintain regularity throughout partitioning, as evidenced when initial horizontal partitions were used. The practicality of KTG is not limited to its use within a partitioning methodology. Additional tasks that can

be aided with such tool include hardware synthesis, code generation and manual analysis of DSTs.

- **An architectural model for the implementation of distributed DSTs**: Section 4.7 introduced a flexible architectural model which extends previously proposed folded FFT models. The model allows distributed implementation of signal processing transforms in general. Mapping between DFG and architectural components is simplified by encapsulating operator's functionality inside kernels. Data communication is mapped to simple data path components and control structures. The model's ease of scalability and modularity allowed precise estimation of its resources using linear (mathematical) model approach.
- **Extension of Kernighan Lin bipartitioning algorithm into a $k$-way partition/placement heuristic for DHAs with considerations to DST graph features**: The cost function of our graph partitioning heuristic emphasizes distribution of costs rather than cumulative cost, as many previous adaptations. This has the effect of better balancing communication load over interconnection channels and achieving lower latency schedules. Several graph considerations that take advantage of DST DFG features were introduced into the heuristic and their advantages in time and solution quality were experimentally assessed.
- **Study of the effect of formulations on their distributed implementation**: To our knowledge, our work is the first to explicitly assess the impact of formulation-level transformations, such as permutations and factorizations, on the partition of FFTs and DCTs. The effect of formulation on implementation is commonly taken into account during manual DST mappings. The exhaustive experimentations reported in this work, albeit for smaller DST sizes, shed some light as to what transformations may be used to guide DST onto more suitable mappings. These results also highlight the importance of clearly specifying formulation when using DSTs as part of a benchmark set.

- **A decomposition method for exploring the formulation space**: Experimental assessment of the impact of formulation on partition quality exposed strategies for effective exploration of the formulation space. A heuristic was developed which takes advantage of granularity features of DSTs to conduct a coarse-to-fine exploration that saves run-time while arriving at competent solutions.
- **Study of current DCT formulations and their appropriateness for DHA's**: Most literature on DCT hardware implementations concentrates on 8-point unidimensional and 8×8 bidimensional cases, which can be currently achieved in single devices while meeting performance criteria. In contrast with FFTs, not much attention has been given DCT distributed implementations in literature. The extension of automated partitioning strategies for DCT's required us to examine various popular regular DCT formulations and their suitability for distributed implementation.
- **Derivation of a Cooley-Tukey like factorization algorithm for DCTs**: Previously reported DCT regular formulations lack the ability to be arbitrarily factorized into multiple equivalent algorithms. The Cooley-Tukey-like DCT factorization algorithm developed in this work was successfully used in DMAGIC to explore alternate DCT expressions and find improved partitioning solutions. It could be useful for future DCT implementers as it may expose folding or partitioning opportunities in other architectures.
- **DMAGIC, an automated high-level partitioning methodology for DSTs onto distributed hardware architectures**: The DMAGIC methodology integrates a series of processes and supporting tools to conduct an optimizing search that benefits from the interaction of the algorithmic and graphic abstraction levels. To the best of our knowledge, this methodology represents the first reported effort to engage the information and opportunities available at these two levels to solve the problem of algorithm mapping to distributed hardware architecture. Results

were superior both in quality and time to general-purpose partitioning strategies. Thus, this approach certainly merits further development into other types of target platforms (e.g. SoCs), algorithm families (e.g. image processing) and optimization objectives (e.g. energy utilization).

## 7.2 Limitations

DMAGIC's ability to exploit DST features during the partitioning process inevitably imposes some limitations to its applicability to algorithms beyond DSTs. In a sense, DMAGIC cedes its generality in exchange for effectiveness of exploration and quality of solution of DSTs, a fundamental component in many modern applications. However this does not mean that the general philosophy of using graph and algorithmic opportunities during partitioning cannot be extended to other classes of algorithms.

Within the scope of DST to DHA partitioning DMAGIC has yet to address some specific limitations, some of which could become significant problems in their own.

- The experiments and results were limited to $2^n$-point DSTs. Nevertheless, this covers a significant percentage of common application sizes. Experiments and heuristics such as the ones conducted throughout this work could be used to verify DMAGIC's effectiveness for non-$2^n$-point DSTs.
- The DHA representation method, architectural model, and cost functions may prove to be limited for connection schemes beyond the point-to-point/crossbar topologies. For instance, in Network-on-Chip topologies communications are handled by a mesh of routers. In this case, communicating any two processing elements might imply solving a routing optimization problem, something that is simplified in our current methodology by the availability of point-to-point connections.

- Formulation exploration for further DSTs might require specialized factorization methods, such as the CT-like factorization algorithms for the FFT and DCT mentioned in Chapter 5.

Regardless of these limitations, the heuristic and results presented throughout this thesis are sufficiently comprehensive to prove our hypothesis.

## 7.3 Future Work

Several enhancements and research paths have been envisioned throughout the development of this work. The following is a description of related issues that might be undertaken in the near future.

- **Partition-aware scheduling heuristics**: This work focused on high-level partitioning strategies, a task within the high-level synthesis of digital circuits. As critical as partitioning may be to the final implementation, high-level synthesis consists of additional tasks, e.g. scheduling and allocation, which influence performance parameters such as latency. In DMAGIC's prototype implementation we used a simple scheduling heuristic. Further development of an integrated DST mapping tool could contemplate the evaluation of additional scheduling heuristics that take advantage of the decisions made by the graph partition/placement process.
- **High-level partitioning benchmarks**: One of the shortcomings uncovered throughout our literature review was the lack of formally established benchmarks for high-level partitioning: both for the input problems and the target architectures. This has the unfortunate consequence that the results in documented methodologies can't be appropriately compared against each other. The development of this work helped us identify desirable characteristics in the definition of benchmarks. For instance, DST benchmarks should specify formulation, as it has an impact on implementation. For general algorithms, benchmarks should also specify how they

are to be scaled. This shall help benchmarks avoid obsolescence as target device density and/or data requirements increase.

- **Computer-driven search of heuristics**: The effect of algorithmic-level transformations was evidenced throughout this work. Determining heuristics for efficiently arriving at good formulations is not trivial even when limiting transformations to a reduced set of rules. Even then, development of these heuristics relies on user knowledge and insightfulness rather than on a systematic (automated) method. For instance, the proposed formulation exploration heuristic was solely based on humanly detectable patterns' on results for smaller sized transforms. Future work could be focused on using techniques such as genetic programming or computer learning for extracting improved formulation-exploration heuristics.
- **Exploration of hardware structures for data permutation**: Folding schemes for DSTs take advantage of their regularity to achieve implementations using a reduced number of functional units. An important part for achieving effective folding is the implementation of data permutation structures. Efficient hardware structures have been found for isomorphic permutations such as those found in the Pease FFT formulation, and have been used in automated and manual hardware implementations [59] [58]. A research idea that stems from the work presented in this dissertation is the study of efficient hardware data-permutation structures for transforms with non-isomorphic permutation sequences. A methodology similar to the one proposed in this work could be used to explore alternate permutation representations in search for optimal structures, given knowledge about the underlying hardware resources.
- **Study of partitioning in System-on-chip architectures** : The high-level partitioning techniques described in this work could also be applicable when targeting on-chip distributed architectures. The Network-on-chip connection topology would be an interesting study case, as it incorporates considerations similar to those of

the DHAs studied herein. For example, not all processing elements are directly connected to each other, and the cost of communication varies according to the position of elements in the network mesh.

- **Extension of DMAGIC for a power efficiency objective**: Power dissipation is one of the main limiting factors in digital VLSI performance. Its importance will certainly increase thanks to the current global environmental/energetic concerns. One of the most promising ways to approach this problem is by introducing power-saving modifications at the architectural level. A practical extension of the current work would study its application for power optimization as a performance objective.
- **Development of a production-quality tool**: The current prototype tool is adequate for experimental purposes and achieved results which sustain the initial hypothesis. Nevertheless, the various methodology components as well their integration would require further fine code tuning/testing to capture isolated cases which may be encountered by users not familiar with the detailed program workings. Further work is also required in terms of data structures to represent the application of decomposition/synthesis rules to the formulations. The current implementation can only represent the repeated application of a single rule. This was sufficient for the studied cases, since the application of further rules de-regularized the expressions.
- **Study of further DSTs**:The two most common DST's, the DFT and DCT, have been studied throughout this dissertation. An evident extension of the presented methodology is to adapt it to other discrete transforms. Similar to the DCT, use of the DMAGIC methodology can motivate the search for regular formulations and folding techniques, which in turn benefit dedicated hardware implementations.

# CHAPTER 8

# Ethics

Ethical concepts must be considered even in predominantly conceptual engineering works, such as the one we have developed. Moriarty distinguishes three elements of ethics involved in the engineering enterprise [100]:

1. virtue ethics: deal with the moral sense of the *engineer*
2. conceptual ethics: are related to the goodness of the engineering process: means, methods and procedures (how engineers can do good engineering)
3. material ethics: apply to the result of engineering process, i.e. the *engineered*.

The lawful motivations behind our research guided our work in a way where *virtue* and *material* ethics were respected at all times. The tools and results obtained throughout this research contribute to the better understanding of partitioning in EDA for distributed hardware architectures, a key process in achieving better and faster computation to these computation platforms. We envision our synthesized and newly discovered knowledge being used for the direct benefit of humanity in fields such as biomedical image processing, genomics and digital communications. Nevertheless, good intentions do not exempt any work from having possible ethical implications, both direct and indirect.

Throughout the development of our research we maintained awareness of its possible ethical implications, using research integrity guidelines such as the ones suggested by the Health and Human Services Commission on Research Integrity

[101]. In the publications generated as part of this research we have been careful to respect the intellectual property of other researchers. In cases where ideas included in this document have been adapted from previous works, we have clearly referenced their originators. Furthermore, all the information included in this document has been obtained from publicly available sources.

The element of *conceptual* ethics could be the most controversial aspect in respect to the technical fields impacted by this work. In electronic design automation, high-level abstractions of an ultimately physical implementation depend on many variables and tools, and are not always exactly representable. This is aggravated by the fact that many intermediary steps are handled by heuristic and non-deterministic processes. Thus, it is hard to account exactly for the contribution of each step, making an exact, third-party validation practically impossible. All these difficulties not withstanding, during the development of our methodology and the experimentation phase we were conscientious to produce results in a manner where no details that might unfairly give us an advantage were hidden.

Another ethical aspect related to the higher-levels of EDA is that by handling ever-higher levels of abstraction in the design processes, some might argue that we are attempting to substitute human knowledge with synthetic computational processes. Nevertheless, even if this perception might have some validity in some humanistic context, it fails to consider what, in our opinion, is the general idea behind EDA and automation. Freeing our minds from repetitive, implementation-related and other menial problems liberates our intellects to solve more challenging ventures. Besides, technology moves at such fast pace that we need advances in automation just to keep current. Furthermore, although the proposed methodology aspires to be completely automated, the need for human intervention is still evident.

Throughout the development of this work, our best efforts focused to generate and synthesize knowledge which does not have a negative impact in the well being,

privacy, or dignity of other people, organizations, or the advance of research in a direct or indirect manner.

# APPENDICES

# APPENDIX A

# Prototype Documentation

The results discussed throughout this work were obtained using a prototype software implementation of the DMAGIC methodology, henceforth referred to as *Revision 1 (R1)* . This implementation consists of the various graph tools and formulation exploration strategies discussed in Chapters 4 and 5. This Appendix documents the *R1* user interface and most relevant data structures used in its tools.

## A.1 Kronecker to dataflow graph tool (KTG)

One of the main advantages of using Kronecker products algebra as a framework for the representation of DSTs is that it facilitates the deduction of computational structures directly from formulation. Although manual KPA to DFG conversion is practical for small examples and illustration purposes, bigger DST sizes demand an automated conversion method. Furthermore, a KPA to dataflow graph conversion software tool is crucial to the fully-automated operation of our proposed scheme. The creation of KTG allowed us to experiment with various sizes and formulations of the FFTs and DCTs, which ultimately defined the strategies used in DMAGIC's formulation exploration component. The following section explains the use of the implemented KTG tool.

### A.1.1 KTG Usage

The command to execute KTG is the following:

$$\text{ktg.exe} - \text{exp } \text{“}expression\text{”} - \text{name } output\_file\_name \ ,$$

where *expression* is a KPA expression which may consist of the operands and operators listed below, and *output_file_name* will be the name of the generated output files.

The Backus-Naur form for a KPA expression accepted by the current ktg version is as follows:

⟨KPA_expression⟩ ::⇒ (⟨KPA_expression⟩ ⟨operation⟩ ⟨KPA_expression⟩) |
(⟨KPA_expression⟩ ⟨operation⟩ ⟨matrix⟩) |
(⟨matrix⟩ ⟨operation⟩ ⟨KPA_expression⟩) |
(⟨matrix⟩ ⟨operation⟩ ⟨matrix⟩)

⟨operation⟩ ::⇒ \otimes | \times | \osum

⟨matrix⟩ ::⇒ I_{size} | ⟨DST_{size}⟩ | M_{⟨row_list⟩} |
G_{⟨vector⟩} | ⟨permutation_matrix⟩ | U_{size} | UT_{size}

⟨DST_{size}⟩ ::⇒ DFT_{size} | DCT_{size}

⟨row_list⟩ ::⇒ (⟨row⟩) | (⟨row⟩)(⟨row_list⟩)

⟨row⟩ ::⇒ real | real , ⟨row⟩

⟨vector⟩ ::⇒ integer | integer, ⟨vector⟩

⟨permutation_matrix⟩ ::⇒ L_{stride,size} | P1_{stride,size} | P3_{size} | P11_{size} |
⟨arbitrary_permutation⟩

⟨arbitrary_permutation⟩ ::⇒ [ ⟨cycle_list⟩ , size ]

⟨cycle_list⟩ ::⇒ ⟨cycle⟩ | ⟨cycle⟩ ⟨cycle_list⟩

⟨cycle⟩ ::⇒ integer | integer, ⟨cycle⟩ ,

where *size* is a positive integer, and *stride* is a divisor of *size*.

**Expression operands**

The current version accepts the following operand matrices:

1. Identity matrix: The format is `I_{size}`.
2. Discrete signal transform: Currently supports DFT and DCT. The format is `DFT_{size}` or `DCT_{size}`. Each DST is represented by a graph node whose weight is assigned according to the transform's implementation area.
3. Custom matrix: Format `M_{(row1)(row2)..(row N)}`, where each row consists of a sequence of comma-separated real numbers. For example,`M_{(1.0,1.0)(0,1.0)}`.
4. Diagonal matrix: Format `G_{n[0],n[1],..,n[N-1]}`, where each $n$ is an integer number. Represents a diagonal matrix

$$\mathrm{G\_}\{\mathrm{n}\,[0]\,,..,\mathrm{n}\,[\mathrm{N}-1]\} = \mathrm{diag}\,[\mathrm{n}[0],\ldots,\mathrm{n}[\mathrm{N}-1]] = \begin{bmatrix} n[0] & & \\ & \ddots & \\ & & n[N-1] \end{bmatrix}$$

5. Permutation matrix: Permutation matrices are specified using permutation cycles. The format is `[(cycle0),(cycle1),..., size]`, where each cycle is a sequence of comma-separated indexes. Indexes are assumed to start at 1. For example, `[(1,2,4),(3,6,5),8]` represents the size-8 stride-by-2 permutation matrix, $L_{8,2}$. The format is not limited to stride permutations.

   Additional compact formats are provided for the specification of common permutations used in DSTs:

   (a) Stride permutations: Format `L_{size,stride}`.

   (b) Inverse identity: This permutation matrix is used as part of Püschel's Cooley-Tukey-like DCT formulations [96]. The format is `P1_{size}`.

   (c) Perfect shuffle : Format `P3_{size}`.

   (d) Bidiagonal matrix: This matrix is utilized in Hsiao and Tseng's DCT algorithm and Püschel's Cooley-Tukey-like DCT formulations [81][96]. The bidiagonal matrix is defined as:

$$S_k = \begin{bmatrix} 1 & 1 & & & \\ & 1 & 1 & & \\ & & \ddots & \ddots & \\ & & & 1 & 1 \\ & & & & 1 \end{bmatrix}$$

The format is `P11_{size}`.

6. $U$ and $U^T$ matrices: $U$ is a matrix of the form $U_p = [u_0, u_1, \ldots, u_{p-1}]$, $u_0 = u_1 = \ldots = u_{p-1} = 1$. The formats are `U_{size}` and `UT_{size}` for the $U$ and $U^T$ matrices, respectively.

### Expression operators

The following operations are accepted:

1. Kronecker product ($\otimes$), specified as `\otimes`.
2. matrix-matrix multiplication, specified as `\times`.
3. direct summation ($\oplus$), specified as `\osum`.

### Program output

The output consists of two files:

1. a `.fig` file with the data order topology representation of the dataflow graph. This file can be viewed using the Xfig program [102].
2. a `.gph` file with a graph representation of the expression. The format of this file is similar to the one used by METIS, a family of multilevel graph partitioning algorithms [103]. The first line contains three integers $n$, $m$, and $fmt$, where $n$ and $m$ are the number of nodes, respectively, and $fmt$ indicates that the graph has weights associated with both the nodes and edges. The remaining $n$ lines specify information about the nodes and edges. Each line has the following structure:

$$w, v_1, e_1, \ldots, v_k, e_k \ ,$$

where $w$ is the weight of node corresponding to the line number, $e(1), \ldots, e(k)$ are the nodes adjacent to this node, and $v(1), \ldots, v(k)$ are the weights of the adjacent nodes. Additional information and examples are available in [104].

The following example shows how to formulate a KPA expression using the KTG format, as well as illustrations of the generated output files.

**Example 2.** *A user would like to use KTG to obtain the dataflow graph for the expression:*

$$(F_2 \otimes I_4)\,(I_2 \otimes (F_2 \otimes I_2))\,(I_4 \otimes F_2)\,R_8$$

*The KTG command is as follows:*

```
ktg.exe  −exp "(DFT_{2} \otimes I_{4} \times
          (I_{2} \otimes (DFT_{2} \otimes I_{2})) \times (I_{4} \otimes DFT_{2})
          \times R_{8}"  − name CT
```
(A.1)

*This command generates two files:* `CT.fig` *and* `CT.gph`*. Figure A–1 shows a* `CT.fig` *when opened using the Xfig program. Figure A–2 shows the content of* `CT.gph`*.*

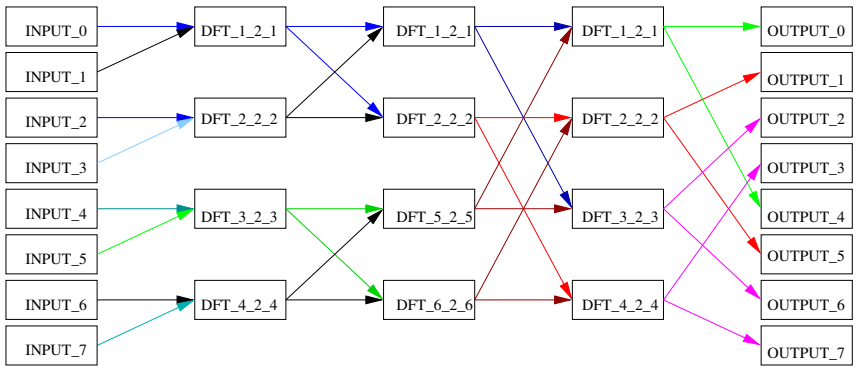



Figure A–1: Visualization of `CT.fig` using the Xfig program.

*The* `.gph` *output file contains an 'dummy' node in addition to the derived dataflow graph. The purpose of this 'dummy' node is to provide a connected graph structure,*

```
29 40 11 1
100 5 1 6 1 13 1 14 1
100 5 1 6 1 15 1 16 1
100 7 1 8 1 17 1 18 1
100 7 1 8 1 19 1 20 1
100 9 1 11 1 1 1 2 1
100 10 1 12 1 1 1 2 1
100 9 1 11 1 3 1 4 1
100 10 1 12 1 3 1 4 1
100 21 1 25 1 5 1 7 1
100 22 1 26 1 6 1 8 1
100 23 1 27 1 5 1 7 1
100 24 1 28 1 6 1 8 1
0 1 1 29 0
0 1 1 29 0
0 2 1 29 0
0 2 1 29 0
0 3 1 29 0
0 3 1 29 0
0 4 1 29 0
0 4 1 29 0
0 9 1
0 10 1
0 11 1
0 12 1
0 9 1
0 10 1
0 11 1
0 12 1
0 13 0 14 0 15 0 16 0 17 0 18 0 19 0 20 0
```

Figure A–2: `CT.gph` file contents.

*which is a constraint in some graph partitioning software. Both the additional node and the edges connecting it to the rest of the structure are weightless, to null their effect on the partition solution.*

### A.1.2 KTG implementation functions and data structures

The current version of the KTG program was implemented in Microsoft Visual C++. Figure A–3 shows KTG's pseudocode, while Figure A–4 illustrates the effect of KTG's functions on a sample command. Figures A–5 through A–7 show the most relevant classes utilized in KTG. The `Eq_Node` class, shown in Figure A–5, is used during the conversion from a KPA expression string to a tokenized postfix queue. An `Eq_Node` object is created for each operation or operand matrix

in the KPA expression. The `CComp` class (Figure A–6) is an implementation of the *k-comp* concept presented in Section 4.3.2. As shown in Figure 4–4, this data structure expedites the deduction of computational structures and their interconnections from the expression by facilitating the implementation of the basic KPA operations. Figure A–7 shows the `Node` class, which is used to implement the data order topology DFG nodes and edges.

```
Input: Command line arguments (CLA)
Output: .fig and .gph files
--------------------------------------------------------------------------------
// Extracts the expression and output file name strings from
// the command line argument string.
(KPA_expression, output_filename) = parse_command_line_arguments(CLA);

// Extract all tokens (matrices and operations) from the expression string
// and return a vector of strings, where each element is a token.
vector <string> expression_tokens  = parse_KPA_expression(KPA_expression);

// Organize the expression tokens into a queue so that they can be read
// and interpreted as a postfix expression.
queue <Eq_Node*>  expression_queue  = expression_to_postfix(expression_tokens);

// Read the token queue, interpret as a postfix expression, and
// create the corresponding CComp structures.
vector <Ccomp*> ccomp_list = expression_to_ccomp(expression_queue);

// Generate a data order topology dataflowgraph based on the list of
// CComp structures generated in the previous step.
vector <Node*> node_list = ccomp_to_nodes(ccomp_list);

// Generate the .fig and .gph files, based on the DFG contained in node_list;
generate_fig_file(node_list, output_filename + ".fig");
generate_gph_file(node_list, output_filename + ".gph");
```

Figure A–3: Pseudocode for the KTG prototype implementation.

## A.2 Graph partitioning heuristic

DMAGIC's graph partitioning heuristic was inspired by the Kernighan-Lin graph bipartition heuristic. It incorporates DST and DHA graph level considerations to provide effective partition solution space exploration. *DGP*, the current version of the graph partitioning tool integrates the rest of DMAGIC graph-related tools, i.e.

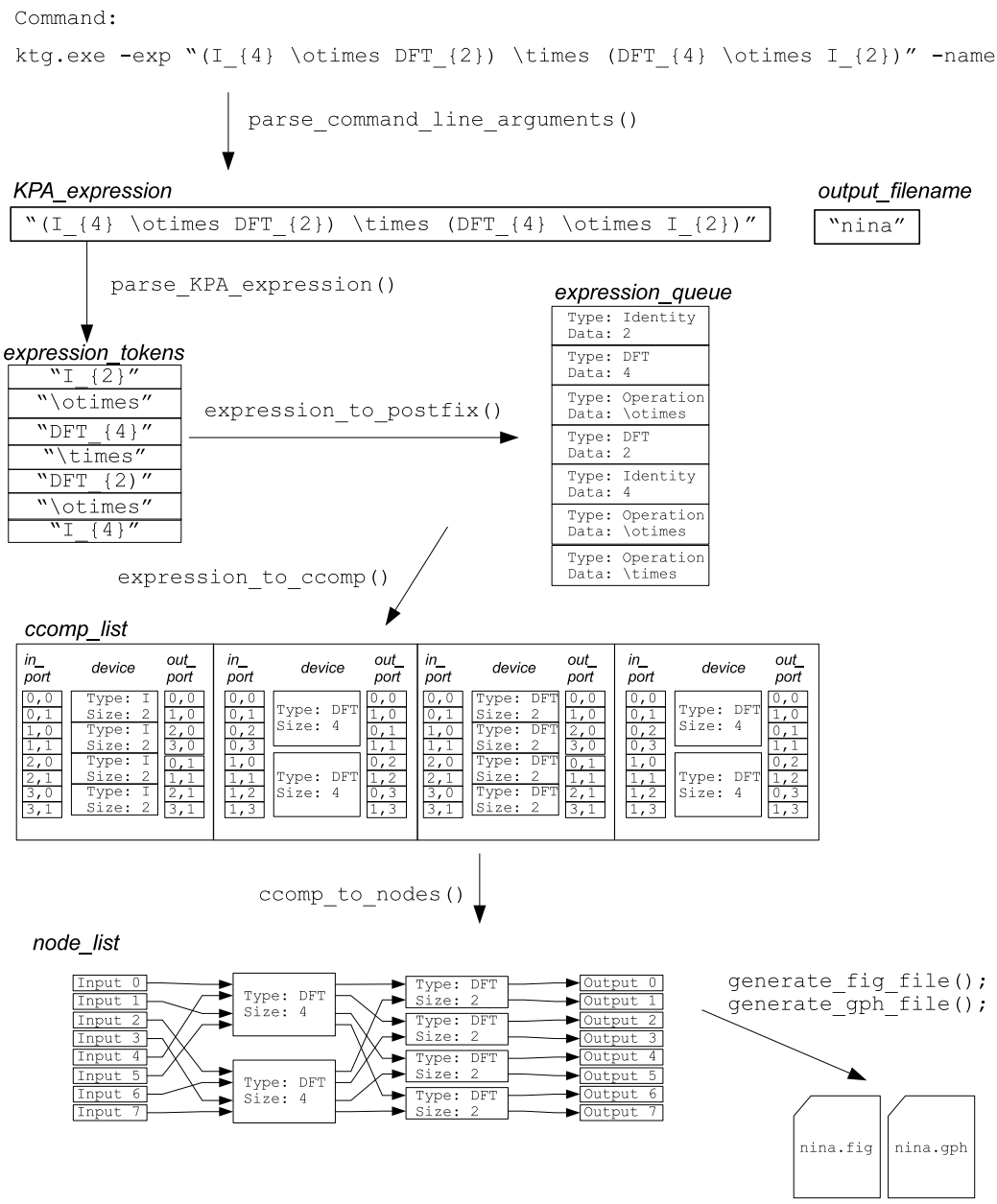



Figure A–4: Illustration of the effect of KTG functions.

the KPA to DFG converter, the graph partitioning heuristic, and the resource and latency estimators.

```
class Eq_Node {
   Node_Type type;
   void *data;  // rest of data depending on type
   public:
      // Functions
      void set_type(Node_Type);
      Node_Type get_type();
      // Several constructors for various types
      Eq_Node(int);
      Eq_Node(DST_Type, int);
      Eq_Node(Operation_Type);
      Eq_Node(Node_Type, std::string st);
      . . .
      ~Eq_Node();     };

enum Node_Type {
   I,
   Transf,
   Perm,
   Twiddle,
   Operation,
   . . . . };
```

Figure A–5: Extract of `Eq_Node` class.

### A.2.1 Usage

The command to execute DGP is the following:

dgp.exe −exp "*expression*" −name *output_file_name*
−top *topology_description_file_name* −fam *device_family_file_name*

where:

- *expression* is a KPA expression, using the same specifications as stated in Section A.1.1
- *output_file_name* will be the name of the generated output files.
- *topology_description_file_name* is the name of the file that specifies the distributed hardware architecture as a hypergraph structure. The first line of this file contains three integers $e$, $n$, and $fmt$, where $e$ and $n$ are the number of hyperedges and nodes, and $fmt = 11$ indicates that the graph has weights associated with both the nodes and edges. The next $e$ lines specify

```
class CComp {
   public:
      std::vector<Port> in_port, out_port;
      std::vector<Device> device;
      bool vert;
      unsigned short stage, level;

      // Functions
      // Several constructors for various types
      CComp();
      CComp(int,int);
      . . .
      // Operations
      CComp *mult(CComp *m); // matrix mult operation
      CComp *clone();
      CComp *kron(CComp*); // kron product operation
      CComp *kron_by_U(CComp*);
      CComp *direct_sum(CComp*);  // direct sum operation
      . . .
      ~CComp(); };


class Port {
   public:
      short dev;
      unsigned int pin;
      Node *gnode;
      void set(int d, int p);
      ~Port(); };

enum Device_Type { I_dev,   Perm_dev, Twiddle_dev, . . . . };

class Device {
   public:
      Device_Type type;
      short size, index;
      CComp *desc; // pointer for recursive structure
      Node *gnode; // pointer to DFG node during DFG creation

      // Functions
      void set(char* st, CComp *d);
      void set(char*, CComp*, Node*);
      // Several constructors for various types
      Device ();
      . . . .
      ~Device(); };
```

Figure A–6: Extract of `CComp`, `Port`, and `Device` classes .

```
class Node {
   public:
      std::vector<Node*> child, parent;  // pointers to other nodes
      short stage;                       // computational column
      short level;
      Device_Type type;
      int size;
      int index;
      short part;                        // partition
      bool visible;
      bool locked;                       // for KL-MH purposes
      unsigned short c_step;

      // Functions
      void add_dest_edge(Node *to_node);
      void add_source_edge(Node *to_node);
      void delete_edge(Node *node);
      std::vector<Node*> get_children();
      std::vector<Node*> get_parents();
      // Creators for various types
      Node();
      Node(char *st);
      Node(Device_Type,int);
      Node(Device_Type,int,int);
      . . .   };
```

Figure A–7: Extract of the `Node` class .

the hyperedges in the following format:

$$(\text{line p}) : \ w_{p-1}, v_1, \ldots, v_r$$

where $w_{p-1}$ is the weight of hyperedge $p-1$, and $v_1, \ldots, v_r$ are the nodes connected to hyperedge $p-1$. The next $n$ lines specify the type of each architectural device. The last line specifies the *width* in terms of points of each of the communication channels. Figure A–8 shows the topology description file for a DHA with four Xilinx XC2VP7 devices connected in a ring topology with a crossbar. Latencies for the point-to-point channels and crossbar are 1 and cycles, respectively. The width of all channels is 1 point.

- *device_family_file_name* is the name of a file that specifies the resources offered by a family of devices. Each line in this file specifies the device name, number

```
5 4 11
1 1 2
1 2 3
1 3 4
1 4 1
2 1 2 3 4
xc2vp7
xc2vp7
xc2vp7
xc2vp7
1 1 1 1 1
```

Figure A–8: Example of a topology description file.

of slices, embedded multipliers and BRAMs for the device. For example, Figure A–9 shows the contents of the file `virtex2pro.res`, which provides resource information for Xilinx Virtex 2 Pro devices.

```
#device_name clbs emb_mults bram
xc2vp4 3008 28 28
xc2vp7 4928 44 44
xc2vp20 9280 88 88
xc2vp30 13696 136 136
xc2vp40 19392 192 192
xc2vp50 23616 232 232
xc2vp70 33088 328 328
xc2vp100 44096 444 444
```

Figure A–9: Example of a device family resource file.

Additional runtime parameters include:

- `-type` $[CT|P|GS|S|TS]$: this option can be used along with the `-size` option to specify common FFT formulations, instead of using the `-expression` option.
- `-size` $s$: where $s$ is the FFT size, when using the `-type` option.
- `-rnd`: This option generates a random initial partition instead of a linear horizontal initial partition.
- `-nsr`: This option allows arbitrary swapping of nodes in the partitioning heuristic. If this option is not specified, the default swapping is limited to nodes within the same computation column.

The output of this program is provided in the standard output stream. The following information is reported throughout execution:

(a) Resource estimation

(b) The final partition assignment for each node and the solution cost function obtained by the partitioning heuristic.

(c) Scheduling information, followed by the latency estimate.

### A.2.2 DGP prototype functions and data structures

The current prototype of the DGP tool was implemented in Visual C++. Figure A–10 shows the pseudocode for this implementation. Figure A–11 illustrates the most relevant data structures used in the DGP graph partitioning heuristic implementation. Figure A–11(a) shows that each vertex of the dataflow graph is implemented as a node in a double linked-list. Each node's associated information includes its weight, level, partition and lists of pointers to its preceding and succeeding nodes, i.e. parents and children. Additionally, a vector $v$ of pointers to each node is maintained for $O(1)$ access. Cost (Figure A–11(b)) is maintained as a size-$M$ vector, where $M$ is the number of available channels in the topology. Figure A–11(c)) illustrates how DGP maintains communication channel information. Communication channel properties such as its weight and width are maintained using a vector of nodes $CCP$. Available communication between devices is implemented using the upper triangular of a $k \times k-1$ bidimensional matrix of pointer vectors, where the content of cell $i, j$ points to the available channels that communicate devices $i$ and $j$.

## A.3 DMAGIC

The current version of DMAGIC explores the DFT or DCT formulation space using the CT-like rules presented throughout this thesis and guided by Algorithm 9 in Chapter 5.

```
Input: Command line arguments (CLA)
Output: Estimation, partitioning, and scheduling information.
-------------------------------------------------------------------------------
// Extracts strings from the command line argument.
(KPA_expression, output_filename, topo_filename, fam_filename, type, size) =
                                      parse_command_line_arguments(CLA);

// If a DFT type and size was provided instead of an expression,
// build the DFT expression string.
if (empty(KPA_expression))
     { KPA_expression = build_formulation(type,size);}

// Convert the expression string to a data order topology DFG
DFG = ktg(KPA_expression);

// Interpret the topology file
topology = read_topology_file(topo_filename);

// Estimate the number of kernels available in each device
estimated_resources =
     resource_estimate(KPA_expression, topology.device_type, fam_filename);

// Perform KL-MH algorithm on the DFG subject to the specified topology.
// See Section 4.5, Algorithm 6.
(min_cost, partition_mapping_function) = KL-MH(DFG, topology);

// Insert channel resource nodes in partitioned DFG (discussed in Section 4.6)
DFG = insert_channel_nodes(DFG);

// Resource-constrained ASAP schedule the DFG based on the estimated resources.
// See Section 4.6, Algorithms 7 and 8.
latency = ASAP_schedule(DFG, estimated_resources);
```

Figure A–10: Pseudocode for the DGP prototype implementation.

### A.3.1 Usage

dmagic.pl −type *DST_type* −size *DST_size*
−top *topology_description_file_name* −fam *device_family_file_name*

where

- *DST_type* is either DFT or DCT.
- *DST_size* is power of two integer number.

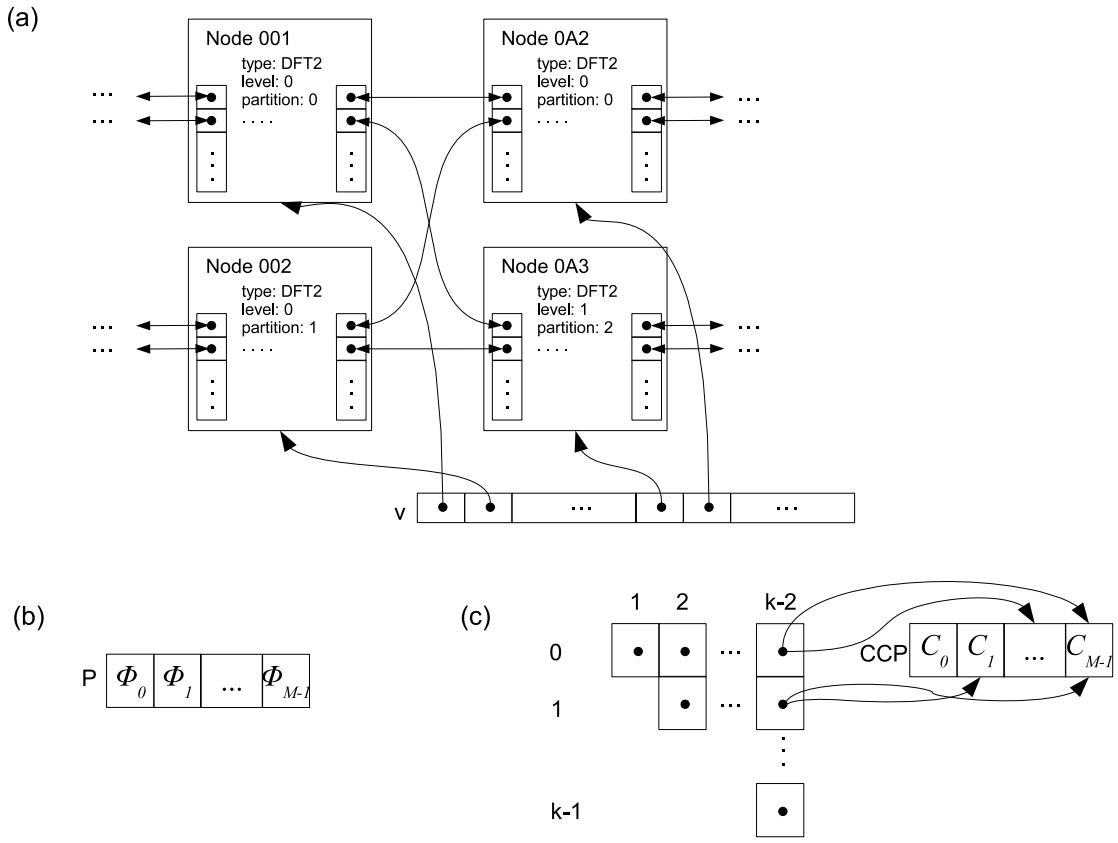



Figure A–11: Main DGP data structures.

- *topology_description_file_name* and *device_family_file_name* are as in the previous section.

**Example 3.** *A 128-point DFT is to be partitioned to a DHA that consists of Xilinx XC2VP4 devices connected in a ring topology with crossbar. The DMAGIC command is the following:*

dmagic.pl −type DFT − size 128
−top 4_ring_xbar_XC2VP4.top −fam virtex2pro.res ,

*Figure A–12 shows part of DMAGIC's output as the previous command is executed. An initial coarse tree (74+3-x+x+2+1-x+x+x+x) is generated based on experimentally observed heuristics. This tree is partitioned and further decomposed*

*based on the partitioning results. Finally, the program outputs the formulation with the minimal latency found through exploration.*

### A.3.2 DMAGIC functions and data structures

The current prototype of the DMAGIC formulation exploration algorithm was implemented in the Perl programming language. DMAGIC explores the formulation space using FFT and DCT Cooley-Tukey-like equations 5.1 and 5.40. The DGP executable is called every time a new formulation is to be partitioned. Figure A–13 shows the pseudocode of the current implementation. Since only one rule is used throughout exploration, at any moment the formulation can be fully represented by its split tree. The current implementation uses binary tree data structures to represent split trees. Extension of DMAGIC to handle multiple rules throughout the exploration of a given DST would require a data structure that can store information about the rule used in each decomposition step. Readers interested in extending DMAGIC in this manner are referred to [44] for related data structures.

```
perl dmagic.pl -type DFT -size 128 -top 4_ring_xbar_XC2VP4.top  -fam virtex2pro.res

Performing first level split
Current split tree: 7@4+3-x+x+x+x
Performing second level split
Current split tree: 7@4+3-x+x+2+1-x+x+x+x
The final expression is: 7@4+3-x+x+2+1-x+x+x+x

Partitioning ...
Results:
Cost = <16,16,16,16,64>

Cost by stage: <0,0,0,0,0>
Cost by stage: <16,16,16,16,32>

The execute latency was: 89

Most congested stage: 1
Will split between leaves 4 and 2

Exploring children 4 (1,3)
Current split tree: 7@4+3-1+3+2+1-x+x+x+x+x+x+x+x

Partitioning ...
Results:
Cost = <24,24,24,24,32>

Cost by stage: <0,0,0,0,0>
Cost by stage: <24,8,24,8,16>
Cost by stage: <0,16,0,16,0>

Most congested stage: 1

The execute latency was: 57
Better latency than parent!!!

Exploring children 4 (2,2)
Current split tree: 7@4+3-2+2+2+1-x+x+x+x+x+x+x+x

Partitioning ...
......

This exploration has found that the best formulation is:
7@4+3-2+2+2+1-x+x+x+x+x+x+x+x with latency = 46
```

Figure A–12: Extract from DMAGIC's output.

```
Input: Command line arguments (CLA)
Output: Exploration information, best formulation found.
-------------------------------------------------------------------------------
// Extracts strings from the command line argument.
(DST_type, DST_size , topo_filename, fam_filename) =
                                      parse_command_line_arguments(CLA);

// Initialize split tree structure (create the split tree root).
split_tree = init_split_tree(DST_size);

// Perform the initial split tree splits based on experimentally
// obtained heuristics.
best_split_leaf_tree = perform_initial_splits(split_tree, DST_type, topo_filename);
best_latency = Infinite;
best_split_leaf_latency = 0;

// Explore the formulation space, as presented in Section 5.3, Algorithm 9.
while (best_split_leaf_latency < best_latency  && split leaves != 0) {
   best_latency =  best_split_leaf_latency;
   split_tree = best_split_leaf_tree;

   // Based on the latency results determine which leaf to split.
   leaf_to_split = analyze(latency, split_tree);

   // Determine the DST formulation for the split tree.
   expression = tree_to_expression(split_tree, DST_type);

   // Partition using dgp and obtain inter column and global latencies
   latency = dgp(expression, topo_filename, fam_filename);

   best_split_leaf_latency = Inf;

   // Explore the possible split combinations for the chosen split leave.
   for each possible split_combination of leaf_to_split {
  // Create a candidate split tree by splitting the chosen leaf
      candidate_split_tree =
            split_leaf_of_tree(split_tree, leaf_to_split, split_combination);

      expression = tree_to_expression(candidate_split_tree);
      latency = dgp(expression, topo_filename, fam_filename);

      if (latency < best_split_leaf_latency) {
        best_split_leaf_latency =  latency;
        best_split_leaf_tree =  candidate_split_tree;
      }
   }
}
output(split_tree);
```

Figure A–13: Pseudocode for the DMAGIC prototype implementation.

# APPENDIX B

# CT-like FFT formulation derivation

The FFT Cooley-Tukey-like formulation used in our experiments and formulation exploration derives from the Cooley-Tukey version of the FFT. For hardware implementation and formulation exploration purposes, our formulation has the advantage of requiring the same amount of multiplications for any breakdown strategy of a given size FFT. Furthermore, the derived formulation consists of recursively decomposable functional primitives which are ultimately composed of the same butterfly·twiddle primitive. This appendix details the derivation of the CT-like FFT formulation.

The Cooley-Tukey version of a $2^t$-point FFT corresponds to the factorization

$$F_{2^t} = \left( \prod_{q=t,t-1}^{1} \left( I_{2^{t-q}} \otimes B_{2,2^{q-1}} \right) \right) R_{2^t} \ , \tag{B.1}$$

where

$$B_{p,m} = \left( F_p \otimes I_m \right) T_{p,m} \, , \tag{B.2}$$

$R_{2^t}$ is the size-$2^t$ bit reversal permutation matrix, and

$$T_{p,m} \left( j, j \right) = \omega_{pm}^{(j \bmod m) \lfloor j/m \rfloor} \text{ for } j = 0, \ldots, pm-1 \, , \tag{B.3}$$

where $\omega_n = \exp\left(\frac{2\pi pqi}{n}\right)$.

In Equation B.3, if $n = pm$ and $p = 2$ or $m = 2$, then at least half of the twiddles factors have a value of 1. For example,

$$\begin{aligned} T_{2,n/2}\,(j,j) &= \omega_n^{(j \bmod (n/2))\lfloor j/(n/2)\rfloor} \text{ for } j = 0,\ldots,n-1 \\ &= \text{diag}\left[\underbrace{\omega_n^0, \omega_n^0, \ldots, \omega_n^0}_{n/2 \text{ coefficients}}, \underbrace{\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{(n/2-1)}}_{n/2 \text{ coefficients}}\right] . \end{aligned} \tag{B.4}$$

Thus, in the expression:

$$B_{2,m} = (F_2 \otimes I_m)\, T_{2,m}\,, \tag{B.5}$$

each $F_2$ will be preceded by at most a single (non-trivial) multiplication. The functionality of each butterfly·twiddle combination can be encapsulated and represented by a size-2 matrix $\beta_2$. The computational structure of Equation B.5 can then be written as:

$$B_{2,m} = (\beta_2 \otimes I_m)\ , \tag{B.6}$$

and

$$F_{2^t} = \left(\prod_{q=t,t-1}^{1} (I_{2^{t-q}} \otimes (\beta_2 \otimes I_{2^{q-1}}))\right) R_{2^t}\,. \tag{B.7}$$

Let

$$\beta_{2^t} = \prod_{q=t,t-1}^{1} (I_{2^{t-q}} \otimes (\beta_2 \otimes I_{2^{q-1}}))\ , \tag{B.8}$$

then

$$F_{2^t} = \beta_{2^t} R_{2^t}\,, \tag{B.9}$$

If $t = r + s$, $\beta_{2^t}$ can be expanded

$$\beta_{2^t} = \underbrace{\left(\prod_{q=t,t-1}^{t-r+1} (I_{2^{t-q}} \otimes (\beta_2 \otimes I_{2^{q-1}}))\right)}_{C} \underbrace{\left(\prod_{q=r,r-1}^{1} (I_{2^{t-q}} \otimes (\beta_2 \otimes I_{2^{q-1}}))\right)}_{D}, \tag{B.10}$$

$$
\begin{aligned}
C &= \left( \prod_{q_1=s}^{1} \left( I_{2^{t-(q_1+r)}} \otimes \left( \beta_2 \otimes I_{2^{q_1+r-1}} \right) \right) \right) \\
&= \left( \prod_{q_1=s}^{1} \left( I_{2^{s-q_1}} \otimes \left( \beta_2 \otimes I_{2^{q_1-1}} \right) \right) \right) \otimes I_{2^r} \qquad \text{(B.11)} \\
&= \beta_{2^s} \otimes I_{2^r} \ , \\
D &= \left( \prod_{q_2=r,r-1}^{1} \left( I_{2^{t-q_2}} \otimes \left( \beta_2 \otimes I_{2^{q_2-1}} \right) \right) \right) \\
&= I_{2^s} \otimes \left( \prod_{q_2=r,r-1}^{1} \left( I_{2^{r-q_2}} \otimes \left( \beta_2 \otimes I_{2^{q_2-1}} \right) \right) \right) \qquad \text{(B.12)} \\
&= I_{2^s} \otimes \beta_{2^r} \ .
\end{aligned}
$$

Therefore,

$$\beta_{2^t} = \left( \beta_{2^s} \otimes I_{2^r} \right) \left( I_{2^s} \otimes \beta_{2^r} \right) \ , \qquad \text{(B.13)}$$

and

$$F_{2^t} = \left( \beta_{2^s} \otimes I_{2^r} \right) \left( I_{2^s} \otimes \beta_{2^r} \right) R_{2^t} \ . \qquad \text{(B.14)}$$

REFERENCE LIST

[1] Jari Nikara. *Application-Specific Parallel Structures for Discrete Cosine Transform and Variable Length Coding*. PhD thesis, Tampere University of Technology, 2004.

[2] J. M. Rabaey, W. Gass, R. Brodersen, T. Nishitani, and Tsuhan Chen. VLSI design and implementation fuels the signal-processing revolution. *IEEE Signal Processing Magazine*, 15(1):22–37, January 1998.

[3] Kevin Morris. DSP heats up. *FPGA and Programmable Logic Journal*, May 2004.

[4] Tom Durkin. SETI researchers sift interstellar static for signs of life. *Xcell Journal*, Spring 2004.

[5] Chen Chang, Kimmo Kuusilinna, Brian Richards, and Robert W. Brodersen. Implementation of BEE: a real-time large-scale hardware emulation engine. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 91–99. ACM Press, 2003.

[6] Dac Pham, et al. Key features of the design methodology enabling a multi-core SoC implementation of a first-generation CELL processor. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 871–878, New York, NY, USA, 2006. ACM Press.

[7] Vinoo Srinivasan, Sriram Govindarajan, and Ranga Vemuri. Fine-grained and coarse-grained behavioral partitioning with effective utilization of memory and design space exploration for multi-FPGA architectures. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(1):140–159, 2001.

[8] O. Bringmann, C. Menn, and W. Rosenstiel. Target architecture oriented high-level synthesis for multi-FPGA based emulation. In *Proceedings of the European Design and Test Conference 2000*, pages 326–332, 2000.

[9] Scott Hauck. *Multi-FPGA Systems*. PhD thesis, University of Washington, 1995.

[10] A. A. Duncan, D. C. Hendry, and P. Gray. An overview of the COBRA-ABS high level synthesis system for multi-FPGA systems. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 106–115, Napa Valley, CA, April 1998.

[11] A. Jones, A. Nayak, and P. Banerjee. Parallel implementation of matrix and signal processing libraries on FPGAs. In *Proc. Intl. Conf. on Parallel and Distrib. Computing and Systems*, 2001.

[12] Frank Vahid. Partitioning sequential programs for CAD using a three-step approach. *ACM Trans. Des. Autom. Electron. Syst.*, 7(3):413–429, 2002.

[13] Petru Eles, Krzysztof Kuchcinski, and Zebo Peng. *System Synthesis with VHDL*. Kluwer Academic Publishers, 1997.

[14] Frank Vahid. A three-step approach to the functional partitioning of large behavioral processes. In *ISSS*, pages 152–157, 1998.

[15] Michael C. McFarland, Alice C. Parker, and Raul Camposano. Tutorial on high-level synthesis. In *Proceedings of the 25th ACM/IEEE conference on Design automation*, pages 330–336. IEEE Computer Society Press, 1988.

[16] A.A. Duncan, D.C. Hendry, and P. Gray. The COBRA-ABS high-level synthesis system for multi-FPGA custom computing machines. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):218–223, 2001.

[17] Scott Hauck and Gaetano Borriello. Logic partition orderings for multi-FPGA systems. In *Proceedings of the International Symposium on Field-Programmable Arrays*, 1995.

[18] Ron Wilson. Structured ASICs arrive. *EETimes*, May 5, 2003.

[19] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and using a highly parallel programmable logic array. *Computer*, 24(1):81–89, 1991.

[20] Jeffrey M. Arnold, Duncan A. Buell, and Elaine G. Davis. Splash 2. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 316–322. ACM Press, 1992.

[21] J.M. Arnold, D.A. Buell, D.T. Hoang, D.V. Pryor N. Shirazi, and M.R. Thistle. The SPLASH 2 processor and applications. In *Proceedings IEEE International Conference on VLSI in Computers and Processors*, pages 482–485, 1993.

[22] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[23] M. Rencher and B.L. Hutchings. Automated target recognition on SPLASH 2. In *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, pages 192–200, 1997.

[24] Liang Xuejun and J.S.-N. Jean. Mapping of generalized template matching onto reconfigurable computers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(3):485–498, 2003.

[25] Steve Guccione. List of FPGA-based Computing Machines . http://www.io.com/ guccione/HWlist.html, 2000.

[26] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[27] Bob Brodersen, Chen Chang, John Wawrzynek, Dan Werthimer, and Melvyn Wright. BEE2: a multi-purpose computing platform for radio telescope digital signal processing applications. In *International Square Kilometre Array*

*Meeting*, 2004.

[28] C. Chang, J. Wawrzynek, and R.W. Brodersen. BEE2: a high-end reconfigurable computing system. *IEEE Design and Test of Computers*, 22(2):114–125, 2005.

[29] I. Koren and Z. Koren. Defect tolerance in VLSI circuits: techniques and yield analysis. *Proceedings of the IEEE*, 86(9):1819–1838, September 1998.

[30] Kaustav Banerjee, Massoud Pedram, and Amir H. Ajami. Analysis and optimization of thermal issues in high-performance vlsi. In *ISPD '01: Proceedings of the 2001 international symposium on Physical design*, pages 230–237, New York, NY, USA, 2001. ACM Press.

[31] Jason Cong, Yiping Fan, Guoling Han, Xun Yang, and Zhiru Zhang;. Architectural synthesis integrated with global placement for multi-cycle communication. In *ICCAD-2003 - International Conference on Computer Aided Design*, pages 536–543, Nov. 2003.

[32] M. Garey, D. Johnson, and L. Stockmeye. Some simplified NP-complete graph problems. *Theoretical Computer Science*, (1):237–267, December 30, 1976.

[33] Charles J. Alpert and Andrew B. Kahng. Recent directions in netlist partitioning: a survey. *Integr. VLSI J.*, 19(1-2):1–81, 1995.

[34] Frank M. Johannes. Partitioning of VLSI circuits and systems. In *Proceedings of the 33rd annual conference on Design automation conference*, pages 83–87. ACM Press, 1996.

[35] Roman Kuznar, Franc Brglez, and Krzysztof Kozminski. Cost minimization of partitions into multiple devices. In *DAC '93: Proceedings of the 30th international conference on Design automation*, pages 315–320, New York, NY, USA, 1993. ACM Press.

[36] Nan-Chi Chou, Lung-Tien Liu, Chung-Kuan Cheng, Wei-Jin Dai, and Rodney Lindelof. Circuit partitioning for huge logic emulation systems. In *DAC '94: Proceedings of the 31st annual conference on Design automation*, pages 244–249, New York, NY, USA, 1994. ACM Press.

[37] Chunghee Kim and Hyunchul Shin. A performance-driven logic emulation system: FPGA network design and performance-driven partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(5):560–568, May 1996.

[38] Wen-Jong Fang and Allen C.-H. Wu. Multiway FPGA partitioning by fully exploiting design hierarchy. *ACM Trans. Des. Autom. Electron. Syst.*, 5(1):34–50, 2000.

[39] Ching-Wei Yeh, Chung-Kuan Cheng, and Ting-Ting Y. Lin. A probabilistic multicommodity-flow solution to circuit clustering problems. In *ICCAD '92: Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 428–431, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[40] N. Dutt and C. Ramachandran. Benchmarks for the 1992, high-level synthesis workshop. In *Tech. Report #92-107 - Information and Computer Science Department, Irvine*, 1992.

[41] Vinoo Srinivasan. *Partitioning for FPGA-based reconfigurable computers.* PhD thesis, University of Cincinnati, 1999.

[42] Annapolis Microsystems Inc. http://www.annapmicro.com/.

[43] Pinit Kumhom. *Design, Optimization, and Implementation of a Universal FFT Processor.* PhD thesis, Drexel University, 2001.

[44] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL:

Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2), 2005.

[45] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[46] E.D. Lagnese and D.E. Thomas. Architectural partitioning for system level synthesis of integrated circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(7):847–860, 1991.

[47] Zhang Yang and Rajesh K. Gupta. A case analysis of system partitioning and its relationship to high-level synthesis tasks. In *VLSID '98: Proceedings of the Eleventh International Conference on VLSI Design: VLSI for Signal Processing*, pages 442–448, Washington, DC, USA, 1998. IEEE Computer Society.

[48] F. Vahid and D. D. Gajski. Specification partitioning for system design. In *DAC '92: Proceedings of the 29th ACM/IEEE conference on Design automation*, pages 219–224, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[49] S. Periyacheri, A. Jones, A. Nayak, D. Zaretsky, P. Banerjee, N. Shenoy, and A. Choudhary. Library functions in reconfigurable hardware for matrix and signal processing operations in MATLAB. In *International Conference on Parallel and Distributed Computing and Systems (PDCS 1999)*, November 1999.

[50] C. Chakrabarti and J. JaJa. VLSI architectures for multidimensional transforms. *IEEE Transactions on Computers*, 40(9):1053–1057, Sept. 1991.

[51] U. Meyer Baese. *Discrete signal processing with field programmable gate arrays*. Springer, 2004.

[52] Charles VanLoan. *Computational frameworks for the fast Fourier transform*. SIAM, 1992.

[53] Fang Fang, James C. Hoe, Markus Pueschel, and Smarahara Misra. Generation of custom DSP transform IP cores: Case study Walsh-Hadamard transform. In *HPEC 02 Workshop*, September 2002.

[54] Xilinx FFT Logic Cores. http://support.xilinx.co.jp/ipcenter/coregen/41i-1-datasheets.htm.

[55] B.M. Baas. A low-power, high-performance, 1024-point FFT processor. *IEEE Journal of Solid-State Circuits*, 34(3):380–387, March 1999.

[56] S. Magar, S. Shen, G. Luikuo, M. Fleming, and R. Aguilar. An application specific DSP chip set for 100 MHz data rates. In *Int. Conf. Acoustics, Speech, and Signal Processing*, volume 4, pages 1989–1992, April 1988.

[57] He Shousheng and M. Torkelson. Design and implementation of a 1024-point pipeline FFT processor. In *Proceedings of the IEEE 1998 Custom Integrated Circuits Conference*, pages 131–134, 1998.

[58] Grace Nordin, Peter A. Milder, James C. Hoe, and Markus Püschel. Automatic generation of customized discrete Fourier transform IPs. In *Proceedings of the 2005 Design Automation Conference*, June 2005.

[59] J.H. Takala, T.S. Jarvinen, P.V. Salmela, and D.A. Akopian. Multi-port interconnection networks for radix-r algorithms. In *Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '01)*, 2001.

[60] Peter A. Milder, Mohammad Ahmad, James C. Hoe, and Markus Püschel. Fast and accurate resource estimation of automatically generated custom DFT IP cores. In *FPGA'06: Proceedings of the internation symposium on Field programmable gate arrays*, pages 211–220, New York, NY, USA, 2006. ACM Press.

[61] Alex Chow. Unleashing the power: A programming example of large FFTs on Cell. In *European Power.org Community Conference*, 2005.

[62] Matteo Frigo. *Portable High-Performance Programs*. PhD thesis, MIT, 1999.

[63] Sebastian Egner, Jeremy Johnson, David Padua, Markus Pschel, and Jianxin Xiong. Automatic derivation and implementation of signal processing algorithms. *ACM SIGSAM Bulletin Communications in Computer Algebra*, 35(2):1–9, 2001.

[64] Reinhard Diestel. *Graph Theory*. Springer-Verlag, 2005.

[65] R.A. Arce-Nazario, M. Jimenez, and D. Rodriguez. Functionally-aware partitioning of discrete signal transforms for distributed hardware architectures. In *Proceedings of the 49th Midwest Symposium on Circuits and Systems*, pages 1438–1441, 2006.

[66] Mark Davio. Kronecker products and shuffle algebra. *IEEE Trans. on Computers*, 30(2):116–125, Feb. 1981.

[67] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.*, 49(2):291–307, 1970.

[68] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC '82: Proceedings of the 19th conference on Design automation*, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.

[69] Sadiq M. Sait and Habib Youssef. *VLSI Physical Design Automation: Theory and Practice*. World Scientific Pub Co., 1999.

[70] J. Rose, W. Klebsch, and J. Wolf. Temperature measurement and equilibrium dynamics of simulated annealing placements. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(3):253–259, March 1990.

[71] Manuel Jimenez. *A placement methodology for low power VLSI circuits*. PhD thesis, Michigan State University, 1999.

[72] Y. C. Zhao, L. Tao, K. Thulasiraman, and M. N. S. Swamy. An efficient simulated annealing algorithm for graph bisectioning. In *Proceedings of the*

*Symposium on Applied Computing*, pages 65–68, Kansas City, MO, April 1991.

[73] M. P. Vecchi S. Kirkpatrick, C. D. Gelatt. Optimization by simulated annealing. *Science*, 220(4958):671–680, 1983.

[74] Steve R. White. Concepts of scale in simulated annealing. In *Proceedings ICCD*, pages 646–651, 1984.

[75] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press.

[76] V. Schnecke and O. Vornberger. Hybrid genetic algorithms for constrained placement problems. *IEEE Transactions on Evolutionary Computation*, 1(4):266–277, November 1997.

[77] S.M. Sait, H. Youssef, K. Nassar, and M.S.T. Benton. Timing driven genetic algorithm for standard-cell placement. In *Computers and Communications, 1995. Conference Proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference on*, pages 403–409, Scottsdale, AZ, March 1995.

[78] J.P. Cohoon and W.D. Paris. Genetic placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(6):956–964, November 1987.

[79] V. Krishnan and S. Katkoori. A genetic algorithm for the design space exploration of datapaths during high-level synthesis. *IEEE Transactions on Evolutionary Computation*, 10(3):213–229, June 2006.

[80] L.A. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, 1989.

[81] Shen-Fu Hsiao and Jian-Ming Tseng. Parallel, pipelined and folded architectures for computation of 1-D and 2-D DCT in image and video codec. *The Journal of VLSI Signal Processing*, 28(3):205–220, 2001.

[82] C. F. Bornstein, A. Litman, B. M. Maggs, R. K. Sitaraman, and T. Yatzkar. On the bisection width and expansion of butterfly networks. In *Proceedings of the 12th International Parallel Processing Symposium*, pages 144–150, March 1998.

[83] R.N. Bracewell. The fast Hartley transform. *Proceedings of the IEEE*, 72(8):1010–1018, 1984.

[84] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[85] C. Brandolese, W. Fornaciari, and F. Salice. An area estimation methodology for FPGA based designs at SystemC-level. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 129–132, 2004.

[86] Xilinx, Inc. *Xilinx Virtex-II Pro Platform FPGA Data Sheet*. 2005.

[87] Altera, Inc. *Stratix II Device Handbook* . 2006.

[88] Jen-Chuan Chi and Sau-Gee Chen. An efficient FFT twiddle factor generator. In *Proceedings of the 12th European Signal Processing Conference*, 2004.

[89] Marcus Püschel, et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), 2005.

[90] M. Kandemir. 2D data locality: definition, abstraction, and application. In *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*, pages 275–278, 2005.

[91] George Karypis. *Multilevel Hypergraph Partitioning*, chapter 1. Kluwer Academic Publishers, 2002.

[92] Bryan Singer and Manuela Veloso. Learning to construct fast signal processing implementations. *J. Mach. Learn. Res.*, 3:887–919, 2003.

[93] W.H. Chen., C.H. Smith, and S. C. Fralick. A fast computational algorithm for the discrete cosine transform. *IEEE Transactions on Communications*,

25(9):1004–1009, 1977.

[94] C. Loeffler, A. Ligtenberg, and G. S. Moschytz. Practical fast 1-d DCT algorithms with 11 multiplications. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 988–991, Glasgow, May 1989.

[95] Zhongde Wang. Reconsideration of "a fast computational algorithm for the discrete cosine transform". *IEEE Transactions on Communications*, 31(1):121–123, January 1983.

[96] M. Puschel. Cooley-Tukey FFT like algorithms for the DCT. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 501–504, April 2003.

[97] Zhongde Wang. Pruning the fast discrete cosine transform. *IEEE Transactions on Communications*, 39(5):640–643, May 1991.

[98] J. Takala, D. Akopian, J. Astola, and J. Saarinen. Constant geometry algorithm for discrete cosine transform. *IEEE Transactions on Signal Processing*, 48(6):1840–1843, 2000.

[99] Lars W. Hagen, Dennis J. H. Huang, and Andrew B. Kahng. Quantified suboptimality of VLSI layout heuristics. In *Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 216–221, New York, NY, USA, 1995. ACM Press.

[100] G. Moriarty. Three kinds of ethics for three kinds of engineering. *IEEE Technology and Society Magazine*, 20(3):31 – 38, Fall 2001.

[101] Health and Human Services Commission on Research Integrity. Professional Misconduct Regarding Involving Research. *Professional Ethics Report*, VIII(3), Summer 1995.

[102] Xfig Drawing Program for the X Windows System. http://www.xfig.org.

[103] METIS - Family of Multilevel Partitioning Algorithms. http://glaros.dtc.umn.edu/gkhome/views/metis.

[104] METIS - A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/manual.pdf.

BIOGRAPHICAL SKETCH

Rafael Arce-Nazario started CISE doctoral studies in August 2002, after obtaining a study leave from UPR-Humacao, where he is an Auxiliary Professor with the Department of Physics and Electronics since 1997. Mr. Arce-Nazario holds a BS Computer Engineering from UPR-Mayagüez (May 1992) and a MS Electrical and Computer Engineering from the University of Wisconsin-Madison (December 1993). He has worked both in private industry as well as in academia.

During the past five years, Rafael worked under the supervision of Dr. Manuel Jiménez Cedeño, with whom he researched various themes related to the general area of electronic design automation. From 2002-03, he focused his research efforts on the representation and manipulation of logic functions, helping expand a method previously proposed by Dr. Jiménez's. A related paper, 'Integer Pair Representation for Multiple-Output Logic' by Arce-Nazario and Jiménez was published in the Proceedings of the IEEE-Midwest Symposium on Circuits and Systems 2003.

During July-October 2004, Rafael did a graduate internship with IBM, Rochester, working with the Chip Physical Design and CAD Tools group, where he was exposed to areas of the Physical Design automation of VLSI Circuits. Along with his mentor, Dr. Bob Lembach, he coauthored the paper 'A diagnostic method for detecting and assessing the impact of physical design optimizations on routing', which was published in the Proceedings of the ACM-International Symposium on Physical Design 2005.

The research conducted by Arce-Nazario during his Ph.D. studies generated the following publications and presentations:

**Journal Articles**

(a) R. Arce Nazario, M. Jiménez, D. Rodríguez. 'Mapping of Discrete Cosine Transforms onto Distributed Hardware Architectures'. Submitted to the Journal of VLSI Signal Processing. April 2007. Springer. Status: Under revision.

(b) R.. Arce Nazario, M. Jiménez, D. Rodriguez. 'Algorithmic-level Exploration of Discrete Signal Transforms for Partitioning to Distributed Hardware Architectures'. Accepted on May 2007 for publication in IET Computers & Digital Techniques.

**Papers in Conference Proceedings**

(a) R. Arce Nazario, M. Jiménez, D. Rodríguez. 'DMAGIC: A High-level Partitioning Methodology for Discrete Signal Transforms onto Distributed Hardware Architectures'. Submitted to the 11th Annual Workshop on High Performance Embedded Computing. MIT Lincoln Lab. September 2007

(b) R. Arce Nazario, M. Jiménez, D. Rodríguez. 'Partitioning Exploration for Automated Mapping of Discrete Cosine Transforms onto Distributed Hardware Architectures'. Accepted to the 50th IEEE Midwest Symposium on Circuits and Systems. August 2007. Montreal, Canada.

(c) R. Arce Nazario, M. Jiménez, D. Rodríguez. 'High-level Partitioning of Discrete Signal Transforms for Multi-FPGA Architectures'. 16th IEEE International Conference on Field Programmable Logic and Applications. August 2006. Madrid, Spain.

(d) R. Arce Nazario, M. Jiménez, D. Rodríguez. 'Functionally-aware Partitioning of Discrete Signal Transforms for Distributed Hardware Architectures'. 49th IEEE Midwest Symposium on Circuits and Systems. August 2006. San Juan, PR.

(e) R. Arce Nazario, M. Jiménez, D. Rodríguez. 'Effects of High-Level Discrete Signal Transform Formulations on Partitioning for Distributed Hardware Architectures'. IEEE on Symposium Field-Programmable Custom Computing Machines. April 2006. Napa, CA

(f) R. Arce Nazario, M. Jiménez, D. Rodríguez. 'An Assessment Of High-Level Partitioning Techniques For Implementing Discrete Signal Transforms On Distributed Hardware Architectures'. 48th IEEE Midwest Symposium on Circuits and Systems. August 2005. Cincinnati, Ohio.

(g) R. Lembach, R. Arce-Nazario, D. Eisenmenger, and C. Wood. 'A diagnostic method for detecting and assessing the impact of physical design optimizations on routing'. ACM International Symposium on Physical Design. April 2005. San Francisco, CA. April 2007.

(h) R. Arce Nazario, M. Jiménez, 'Integer Pair Representation for Multiple Output Logic', 47th IEEE Midwest Symposium on Circuits and Systems. Cairo, Egypt. December 2003.

**Others posters, presentations and papers**

(a) R. Arce-Nazario and Manuel Jiménez. 'High-Level Partitioning of Discrete Signal Transforms for Distributed Hardware Architectures' . Poster presentation: Puerto Rico Interdisciplinary Scientific Meeting. Bayamn, Puerto Rico. February 2007.

(b) R. Arce-Nazario and Manuel Jiménez. 'High-Level Partitioning of Discrete Signal Transforms for Distributed Hardware Architectures' . Poster presentation: Workshop on Grid Services, Automated Information Processing, and Wireless Sensor Networks. San Juan, Puerto Rico. February 2007.

(c) R. Arce-Nazario, Manuel Jiménez, and Domingo Rodriguez. 'High-level Partitioning of Discrete Signal Transforms for Multi-FPGA Architectures'. Poster presented at WALSAIP Project HP Labs research visit. Mayagez, Puerto Rico. October 2006.

(d) R. Arce-Nazario and Manuel Jiménez. 'High-Level Partitioning of Discrete Signal Transforms for Distributed Hardware Architectures' . Poster presentation: Puerto Rico Interdisciplinary Scientific Meeting. Cayey, Puerto Rico. March 2006.

(e) R. Arce Nazario, M. Jiménez, D. Rodríguez. 'High-Level Partitioning Techniques For Implementing Discrete Signal Transforms On Distributed Hardware Architectures'. Poster presentation in Annual EPSCoR conference. Rio Grande, PR. September 2005.

(f) R. Arce Nazario, M. Jiménez, 'High-Level Partitioning Of DSP Algorithms For Multi-FPGA Systems'. Poster presentation. GEM Consortium Future Faculty and Professionals Symposium. Las Vegas, NV, June 2004.

(g) R. Arce Nazario, M. Jiménez, 'High-Level Partitioning Of DSP Algorithms For Multi-FPGA Systems - A First Approach', Proceedings of the Computing Research Conference, Mayagez, PR, April 2004

(h) R. Arce Nazario, M. Jiménez, 'Integer Pair Representation for Multiple Output Logic', PRSGC Second Congress on Integrating NASA Research and Education Projects in Puerto Rico, San Juan, PR, November 2003

(i) R. Arce Nazario, M. Jiménez, 'Integer Pair Representation for Multiple Output Logic', Proceedings of the Computing Research Conference, Mayagez, PR, April 2003.