AUTOMATIC IMPLEMENTATION AND PLATFORM ADAPTATION OF DISCRETE FILTERING AND WAVELET ALGORITHMS

A Dissertation
Presented to the Faculty of the Graduate School
of Carnegie Mellon University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Aca Gačić
December 2004

Acknowledgements

I owe my deepest gratitude to my advisors José M. F. Moura and Markus Püschel for their continuous support and their encouragement at critical stages during my Ph.D. studies. Their dedication and clear vision provided me with the constant source of inspiration and guidance throughout my research and helped me regain focus during challenging times.

The writing of a Ph.D. dissertation is often a lonely experience; however, I was incredibly fortunate to work with a team of unselfish, dedicated, and curious people involved in the SPIRAL project. My special thanks go to Franz Franchetti and Yevgen Voronenko for their tremendous help in making this thesis possible. Both Franz and Yevgen helped me stay focused and spent long nights, weekends, and even holidays doing everything they can to ensure that this thesis is improved with compelling experimental results. My sincere thanks are extended to my friends and officemates David Sepiashvili and Marek Telgarsky for their patience and kindness.

I would further like to thank the members of my committee: Jeremy Johnson, Jelena Kovačević, Tsuhan Chen, and my M.Sc. program advisor at the University of Pittsburgh Marwan A. Simaan for fruitful discussions and honest feedback.

I am proud to have studied at Carnegie Mellon University, in most part because of all the kind and helping people I have met here. At times, I felt a need for a second Ph.D. degree to resolve many administrative issues, but Carol Patterson, Lynn Philibin, and Elaine Lawrence made that other part of my academic life much easier. I am grateful to them and to all other CMU staff. I would also like to thank all of my CMU friends for making the days (and nights) spent in the office more fun, and my other friends in the area for making Pittsburgh my true home.

Over a great distance, I extend my love to my parents Danilo and Štefica whose patience and courage I can never forget. I wish I could have shared the whole experience with you day by day as always before. Saša, thank you for being tough on me when I needed it most and kind to me all the time. That's what big brothers are for.

Iva, I sincerely believe I could not have come this far without you on my side. I only wish I could have been there for you more often during this process. Thank you for holding on and being brave.

TABLE OF CONTENTS

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Automatic performance tuning systems . . . 2

1.1.2 Digital filtering and wavelet kernels . . . 4

1.1.3 Existing software packages for digital filtering and wavelet applications . . . 5

1.2 Problem Statement . . . 7

1.2.1 Challenges . . . 8

1.3 Organization of the Thesis . . . 10

2 SPIRAL Code Generator **12**

2.1 SPIRAL System Overview . . . 12

2.2 SPIRAL's Mathematical Framework . . . 15

2.2.1 Key concepts . . . 15

2.2.2 Algorithm space . . . 21

2.2.3 Formula optimization . . . 22

2.3 Translation Into Code . . . 23

2.4 Verification and Runtime Measurements . . . 26

2.5 Search . . . 27

2.5.1 SPIRAL's optimization techniques . . . 28

2.5.2 Practical considerations . . . 31

2.6 FIR Filters and the DWT in SPIRAL: Challenges . . . 31

3 Discrete FIR Filters and Wavelets and Their Algorithms **34**

3.1 FIR filters . . . 34

3.1.1 Signal representations . . . 34

3.1.2 Convolution . . . 35

3.1.3 Generalized convolutions as multiplications in polynomial algebras . . . 36

3.1.4 Embedding one convolution type into another . . . 39

3.1.5 Reducing long convolutions to multiple short convolutions . . . 40

3.1.6 Divide-and-conquer methods . . . 43

3.1.7 Short convolution algorithms . . . 46

3.1.8 Transform domain filtering algorithms . . . 48

3.2 Discrete Wavelet Transforms . . . 51

3.2.1 Multiresolution analysis . . . 52

3.2.2 Discrete wavelet transform . . . 54

3.2.3 Polyphase representation . . . 60

3.2.4 Lattice factorization . . . 62

3.2.5 Lifting scheme factorization . . . 64

3.3 Summary . . . 66

4 Mathematical Preliminaries **68**
4.1 Matrices . . . 68
4.1.1 Basic definitions . . . 68
4.1.2 Special matrices . . . 70
4.2 Constructs . . . 71
4.2.1 Basic matrix constructs . . . 71
4.2.2 Subspace decomposition constructs . . . 73
4.2.3 Overlapped constructs . . . 74
4.3 Operators . . . 77
4.3.1 Permutations . . . 77
4.3.2 Gather and scatter operators . . . 81
4.3.3 Extension and reduction operators . . . 83
4.4 Summary . . . 91

5 Filter Transforms and Rules **92**
5.1 FIR filter transforms . . . 92
5.1.1 Filter and convolution transforms . . . 92
5.1.2 Filters with signal extension . . . 95
5.1.3 Filter banks and polyphase matrices . . . 99
5.1.4 Composition of filters and convolutions . . . 101
5.2 Breakdown rules for filter and convolution transforms . . . 103
5.2.1 Basic identities and rules . . . 104
5.2.2 Block convolution rules . . . 106
5.2.3 Multidimensional rules . . . 106
5.2.4 Embedding rules . . . 108
5.2.5 Karatsuba rules . . . 109
5.2.6 Transform-domain rules . . . 111
5.2.7 Blocking rules . . . 111
5.3 Concluding remarks . . . 112

6 Discrete Wavelet Transforms and Rules **115**
6.1 Discrete Wavelet Transforms . . . 115
6.1.1 Nonperiodic DWT . . . 116
6.1.2 Inverse Nonperiodic DWT . . . 118
6.1.3 Finite DWT and IDWT of infinite and extended signals . . . 120
6.1.4 Periodic DWT and IDWT . . . 124
6.2 Breakdown Rules for DWT Algorithms . . . 126
6.2.1 Mallat recursive rules . . . 127
6.2.2 Polyphase rules . . . 129
6.2.3 Lattice factorization rules . . . 131
6.2.4 Lifting scheme rules . . . 133
6.3 Concluding Remarks . . . 134

7 Experimental Results **137**
7.1 Overview of Experiments and Platforms . . . 137
7.2 Runtime Performance Benchmarks . . . 141
7.3 FIR Filter Methods Across Multiple Platforms . . . 147
7.3.1 Setup of experiments . . . 148
7.3.2 Time-domain methods . . . 148
7.3.3 Karatsuba methods . . . 150
7.3.4 Transform-domain methods . . . 151
7.3.5 Comparison of all methods for FIR filters . . . 156
7.4 DWT Methods on Multiple Platforms . . . 157
7.4.1 Evaluation of DWT implementation methods . . . 157
7.5 Compiler Issues . . . 159
7.5.1 Compiler vectorization . . . 160
7.6 Concluding Remarks . . . 160

8 Conclusion **163**
8.1 Major Contributions . . . 164
8.2 Limitations of the Current Framework . . . 165
8.3 Future Work . . . 166

A Lifting scheme factorizations **168**
A.1 Euclidean Algorithm . . . 168
A.2 Division of Laurent polynomials . . . 169
A.3 Lifting scheme factorizations . . . 170

B Generalized Karatsuba Methods **173**
B.1 Standard Radix-n Karatsuba Algorithm . . . 173
B.2 Radix-n Karatsuba Cost Analysis . . . 175

C Results of FIR Filter Transform and DWT Experiments **178**

Bibliography **183**

LIST OF FIGURES

2.1 The architecture of SPIRAL 14
2.2 Rule trees = formulas = algorithms. 21
2.3 A DFT_{16} rule tree 22
2.4 Rule tree manipulation for evolutionary search 31

3.1 Filtering interpreted as polyphase filtering with two channels. 44
3.2 Radix-2 divide-and-conquer single step filter decomposition 45
3.3 Filter bank interpretation of the DWT. 57
3.4 Filter bank representation of the inverse DWT. 57
3.5 Wavelet packet trees: full (left); pruned (center); DWT (right). 58
3.6 Haar scaling function and the wavelet. 59
3.7 Noble identities (top); Efficient filtering using the Noble identity (bottom). 61
3.8 Polyphase representation of one filter bank stage. 61
3.9 Lattice decomposition of an orthogonal filter bank. 64
3.10 Lifting scheme for one stage of the forward DWT. 66

6.1 Computing channel filters using the Noble identities. 117
6.2 Computing synthesis channel filters using Noble identities. 120

7.1 Comparison of different filtering methods and IPP FIR function on Athlon-1.7 (lower is better). 143
7.2 Comparing the best found DWT implementation and IPP code on Athlon-1.73 (lower is better) 144
7.3 Comparing the best found filter code and IPP filter code on P4B-3.0-win 145
7.4 Comparing the best found DWT code and IPP DWT code on P4B-3.0-win for three wavelets: rational 5/3, Daubechies 9/7, and Daubechies 30 146
7.5 Performance of the best found FIR filter code in MFLOPS 147
7.6 Performance of the best found DWT code in MFLOPS (higher is better) 147
7.7 Comparison of the time-domain methods on Xeon-1.7 (lower is better) 149
7.8 Blocking methods for gcc-O6-3.2 and gcc-O1-3.2 on Xeon-1.7 150
7.9 Performance in MFLOPS for best found time-domain method. 151
7.10 Comparison of Karatsuba methods and the best blocking/nesting strategy. 152
7.11 Comparison of the best found time-domain and transform-domain methods. 154
7.12 Comparison of time-domain and transform-domain methods for circulant matrices. 155
7.13 Comparison of run times for time-domain and Karatsuba methods compiled by icc-8.0-lin and by gcc-O1-3.3 compilers on P4-1.6-lin platform 159
7.14 Effect of compiler options gcc-mac-O1 and gcc-mac-O3 on the run time of: (a) time-domain methods, and (b) Karatsuba methods for filters on Macintosh platform. . . 160
7.15 SSE3 vectorization speedup for FIR filters obtained by Intel-SSE3 compiler 161
7.16 Run times of randomly generated rule trees for: (a) FIR filter transform of size 128 and 17 filter taps on P4C-3.2, and (b) Daubechies 9-7 DWT of size 64 on Xeon-1.7. 162

C.1 Runtime comparison of all lifting scheme factorizations for Daubechies 9/7 wavelet transform 178
C.2 Comparison of Mallat, Lifting and Polyphase rules on P4B-3.0-win. 179
C.3 Comparison of Mallat, Lifting and Polyphase rules on Athlon-1.73. 180
C.4 Comparison of Mallat, Lifting and Polyphase rules on Macintosh. 181
C.5 Performance of DWT algorithms on different platforms. 182

LIST OF TABLES

2.1 Rule examples 19

4.1 Table of mathematical objects. 91

7.1 Computer platforms used for experiments. 140
7.2 Compilers and compiler options. 141
7.3 Two rule tree examples for Karatsuba method found by search 152
7.4 Best found methods for FIR filter transform of various lengths on different platforms. 156

Abstract

Automatic Implementation and Platform Adaptation of Discrete Filtering and Wavelet Algorithms

Aca Gačić
José M. F. Moura, Markus Püschel
Carnegie Mellon University 2004

Moore's law, with the doubling of the transistor count every 18 months, poses serious challenges to high-performance numerical software designers: how to stay close to the maximum achievable performance on ever-changing and ever-faster hardware technologies? Up-to-date numerical libraries are usually maintained by large teams of expert programmers who hand-tune the code to a specific class of computer platforms, sacrificing portability for performance. Every new generation of processors reopens the cycle of implementing, tuning, and debugging.

The SPIRAL system addresses this problem by automatically generating and implementing algorithms for DSP numerical kernels and searching for the best solution on the platform of interest. Using search, SPIRAL adapts code to take optimal advantage of the available platform features, such as the architecture of the memory hierarchy and register banks. As a result, SPIRAL generates high-performance implementations for DSP transforms that are competitive with the best hand-coded numerical libraries provided by hardware vendors.

In this thesis, we focus on automatic implementation and platform adaptation of filtering and wavelet kernels, which are at the core of many performance-critical DSP applications. We formulate many well-known algorithms for FIR filters and discrete wavelet transforms (DWT) using a concise and flexible symbolic mathematical language and integrate it in the SPIRAL system. This enables automatic generation and search over the comprehensive space of competitive algorithms, often leading to complex solutions that are hardly ever considered by a human programmer.

Experimental results show that our automatically generated and tuned code for FIR filters and DWTs is competitive and sometimes even outperforms hand-coded numerical libraries provided by hardware vendors. This implies that the richness and the extent of the automatically generated search space can match human ingenuity in achieving high performance. Our system generates high-quality code for digital filtering and wavelet kernels across most current and compatible future computer platforms and frees software developers from tedious and time-consuming coding at the machine level.

CHAPTER 1

INTRODUCTION

Since the very early days of numerical data processing using computers, high-performance computing has been the topic of research and steady improvement. For numerically intensive tasks, it is important to get the most out of the available hardware. Development of high-level programming languages, such as C and Fortran, and compiler technologies has made it possible to avoid coding at the machine level and still utilize many features of the computing platform. However, even today, for applications with the highest demand for speed, programmers hand tune code in assembly to unveil the full power of the target computer architecture. The implementations become ever more dependent on the specific features of the target platform, whose complexity increases with the advances in design and manufacturing technology. It is desirable to automate the most difficult and time-consuming coding tasks, first at the level independent of the underlying architecture, and then tuning to the specific platform.

One of the most important practical problems in high-performance computing is the design of efficient implementations of digital signal processing (DSP) operations. The goal of this thesis is to automate the design of implementations for ubiquitous digital filters and increasingly more popular wavelet processing systems across a spectrum of computer platforms. In this chapter, we discuss the context and the scope of this work and the possible contribution to the larger area of automatic performance tuning for DSP kernels.

1.1 Motivation

Numerical computing problems at hand have grown in size and complexity following the increased capability of computer hardware. However, with the growing complexity and diversity of computer platforms, it becomes increasingly more difficult to stay close to the achievable performance. Numerical software libraries need to be tuned to rapidly changing compiler technologies, processor designs, and machine architectures on one side, and to new numerical kernels on the other. To fully exploit available hardware resources, the designers of efficient implementations need a comprehensive knowledge of the capabilities of the specific microarchitecture with all of its intricacies, as well as a deep understanding of numerical algorithms and how to manipulate them to get the most out of the available architecture. Thus, optimization of implementations for any given computer platform is time and resource consuming. Furthermore, the performance of the tuned solution is strongly correlated with the hardware for which it is targeted and typically falls far below the peak

performance when ported.

Having both portability and high performance are often two conflicting goals. The tuning process has to be repeated for any new target platform and any new numerical problem. Moore's law, with the doubling of the transistor count every 18 months, thus, poses serious challenges to high-performance software designers since new and more complex architectures are arriving at a fast rate. Up-to-date numerical libraries are usually maintained by large teams of expert programmers who hand-tune the code to a specific class of computer platforms. Every new generation of processors reopens the cycle of implementing, tuning, and debugging.

To avoid hand coding and porting, the tuning of numerical libraries should be automated. This is addressed by research efforts commonly known as *automatic performance tuning* for numerical kernels. Ideally, general purpose compilers should provide a seamless transition between conceptual programming techniques and high performance. In reality, compiler technology has not been able to keep up with the complexity of the problem of tuning the software to computer architectures. There are several reasons for the underachievement of general purpose compilers for numerical software:

- The process of code optimization is not well understood and performs well only on very simple code segments;
- Most numerical algorithms exhibit highly complex data flow patterns that can dramatically affect the performance on multi-level memory hierarchies found on most modern computer architectures;
- To achieve the peak performance of numerical processors, implementations have to make use of specialized instruction sets, such as single instruction multiple data (SIMD) and fused multiply-add (FMA). Compilers can usually perform code optimizations only for very simple structures to make full use of the available instructions.
- Compilers have the disadvantage of operating at the code level where much of the structural information about the algorithms is destroyed when represented by a high-level programming language.

The goal is to overcome limitations of general purpose compilers by automatic tuning systems. These systems use domain-specific knowledge to generate a set of competitive algorithms, either at installation time or at compilation time. They sometimes probe the hardware system for information that can be used in the optimization and search for the best solution in the restricted set of candidates, either by measuring the actual run time or using performance models. Because of the complexity of the problem, most performance tuning systems implement only basic functions that are used as building blocks in more complex applications.

1.1.1 Automatic performance tuning systems

At the core of performance-critical applications are linear algebra and digital signal processing (DSP) kernels. There have been several projects designing performance tuning systems for specific computational kernels.

In the domain of linear algebra computations, ATLAS (Automatically Tuned Linear Algebra Software) and PHiPAC (Portable High Performance ANSI-C) are software packages that provide a library of platform-optimized linear algebra routines called BLAS (Basic Linear Algebra Subroutines) [1, 2, 3]. ATLAS focuses on optimizing dense matrix-vector and matrix-matrix multiplications. ATLAS uses flexible blocking and loop unrolling strategies, low-level code scheduling, and removal of unnecessary dependencies in blocks of code to achieve the desired performance. The blocking strategies are searched to improve cache and register locality in order to minimize cache misses and register spills. Besides blocking, ATLAS performs ordering of floating point operations to utilize the pipelined floating point units. The scheduling is performed in order to overcome the delays caused by latencies associated with floating point operations and prevent pipeline stalls by removing data and instruction dependencies. ATLAS focuses specifically on BLAS routines since they have simple structure and provide the numerical building blocks for the comprehensive linear algebra library of routines LAPACK [4]. Porting of LAPACK is achieved by regenerating the BLAS routines for the target platform.

Sparse matrix multiplications are used in numerous scientific computing applications. Two related projects Sparsity and BeBOP investigate automatic generation of algorithms tuned to memory hierarchies with multi-level cache memories [5, 6]. Sparsity focuses on sparse matrix-vector multiplication kernels and uses dynamic register and cache blocking methods. It profiles the target platform using a set of measurements on a dense matrix to establish the lower bound on performance. It then uses this information together with the approximation of the performance of a blocked sparse matrix with a similar structure of non-zero elements to determine the optimal blocking strategy. Automatic tuning for both dense and sparse matrices is summarized in [7].

In the area of DSP kernels, the discrete Fourier transform (DFT) tuning package FFTW has achieved considerable success [8, 9]. FFTW provides a portable library of C subroutines for the DFT, carefully designed to match most platforms and current compilers. FFTW uses a library of pre-generated, highly optimized "codelets" for smaller size transforms and a flexible plan to recursively compute larger size transforms from their smaller counterparts. This plan is flexible in order to adapt the implementation to the target platform through search. The newest release of FFTW also includes implementations of DCTs, DSTs, and the DHT [9]. Another package called UHFFT uses an approach similar to FFTW to generate, search, and optimize multidimensional FFTs with the added support for the prime factor and split-radix algorithms [10].

The SPIRAL system automatically tunes implementations for many DSP transforms to the chosen platform [11, 12, 13]. SPIRAL automatically generates fast algorithms for DSP transforms on the mathematical level using a computer algebra system. Algorithms are represented in an elegant mathematical language based on a small set of carefully designed rules. Rules are applied recursively to generate mathematical formulas that specify the data flow and the order of operations in the algorithm. SPIRAL provides a compiler of formulas into a high-level language (C or Fortran) to create actual implementations. At this level, the compiler provides additional implementation options such as the degree of loop unrolling. SPIRAL then searches the space of available algorithms and implementations using different optimization techniques to find the best solution. There is no code library prior to the optimization. SPIRAL generates code on the run, evaluates the current implementation, and uses the run time to navigate the search space. Besides the DFT, SPIRAL

includes a comprehensive set of linear transforms such the discrete sine and cosine transforms (DCTs and DSTs), the real DFT, the Walsh-Hadamard transform (WHT), the discrete Hartley transform (DHT), and many others.

1.1.2 Digital filtering and wavelet kernels

Digital filters are the backbone of most digital signal processing (DSP) systems. In the most general sense, a linear digital filter is any discrete linear system identified by its transfer function. Such systems are important as they can be realized using a small set of computational building blocks such as adders, multipliers, and delays (memory units). However, in practice, filters are regarded as linear systems that in one way or the other shape the spectral characteristics of a signal. Of special importance are so-called finite impulse response (FIR) filters that can be realized as discrete systems without feedback loops which makes them stable under any conditions. Another advantage of FIR filters is that they can be designed to have linear phase transfer characteristics using simple methods — a feature important in many phase sensitive DSP applications such as audio and speech processing. For these properties and the straightforwardness of their design and implementation, FIR filters have a widespread use in signal processing. Example applications include multi-path echo cancelation in wireless communications, speech synthesis, image enhancement, and biomedical signal processing, among many others.

In real-time systems, the design of FIR filter is limited to causal filters, i.e., to filters whose response cannot occur before the excitation; however, off-line software implementations do not have this restriction since the dimension of time becomes a simple shift in memory that can be performed in any direction. A notion of time, however, might need to be preserved to correctly handle boundary conditions when finite linear operators are applied to infinite signals.

Most DSP hardware platforms provide processing units that specialize in implementing multiply-accumulate operations that occur in digital filtering. The hardware is optimized to implement filters by using high-level parallelism and instruction pipelining and can process signals at very high rates. Even general purpose platforms sometimes include specialized instructions, such as fused multiply-add instructions, that allow efficient implementation of digital filters. However, implementing filters in software is a nontrivial problem because of the complex architecture of modern computers, multilevel memory hierarchy, and deficiencies of modern compilers. The latter is a serious problem for high-performance software developers since standard code optimization techniques are not well suited to take advantage of the regular structure of the filtering operation. One example where generalized optimization techniques can hurt filter implementation is the case when the filter coefficients are repeated as in the case of linear phase filters. Multiplication of the signal samples by the same coefficient then happens at different points in time. Compilers apply common subexpression elimination to store the intermediate results for later use and avoid excessive computations. However, on modern platforms, it is often more efficient to repeat the same operation than to store the result as a temporary variable and then retrieve it later; this can cause costly register spills and even cache misses. Filtering can also be performed in the frequency domain using transform-based methods. These methods typically reduce the computational cost from $\mathcal{O}\left(n^2\right)$ to $\mathcal{O}\left(n \log n\right)$ but destroys the regular structure of the computations so their advantage is size dependent.

Digital filters are also used for multirate signal processing, which started as a technique used in speech processing about three decades ago. Today, multirate filter banks find applications in image compression, audio signal processing, numerical solutions of differential equations, statistical signal processing, and many others. Furthermore, multirate filter banks are used to implement wavelet systems, which expand signals into subspaces providing different levels of detail in time and frequency. Wavelet-based signal processing has become an exciting problem-solving tool in many applications. In digital hardware, the discrete wavelet transform (DWT) implemented using multirate filter banks provides means to apply wavelet signal processing techniques to discrete signals [14, 15, 16]. Different DWTs have found application in image compression, detection, denoising, communication theory, statistical estimation, numerical solutions to partial differential equations, scientific computing, biomedical signal processing, and elsewhere.

Even though the definition of the DWT is not unique, since it can be constructed using different wavelet bases and signal extensions, the basic structure of DWT computations is similar across different definitions. The DWT is computed as a recursive bank of filters, so convolution is the basic computational block. However, different factorization techniques are available, such as the lifting scheme and lattice factorization that we discuss in Chapter 3, which reduce the computational cost at the expense of increased critical path. Using the polyphase representation of multirate filter banks, DWTs are converted into basic filtering blocks for which all above-mentioned filtering methods apply.

Two-dimensional filtering and 2-D DWTs are important for image processing applications. If 2-D filters are separable, they can be implemented using a Kronecker product of 1-D filters used for construction, i.e., applied separately to rows and columns of the image. Special computational kernels are needed for the non-separable case.

For performance-critical applications, filtering and DWT kernels often present one of the most computationally intensive blocks. An example is the image compression coding standard JPEG2000 where the DWT is used to localize details of the image prior to quantization [17]. Providing highly optimized portable implementations for a family of filtering and wavelet transforms has a significant impact on the overall efficiency of these applications.

1.1.3 Existing software packages for digital filtering and wavelet applications

Due to the widespread use of filtering techniques and wavelet tools in DSP applications, there are many available software libraries. Most of these libraries focus on providing a range of analysis and design tools in a GUI environment and are less dedicated to achieving high performance. The mathematical software system MATLAB provides a library of routines for wavelet system implementation and many wavelet processing and analysis tools [18]. MATLAB makes use of a library of optimized and platform tuned subroutines for the BLAS kernel and the DFT by mainly relying on ATLAS optimization tools and the FFTW package [7, 9]. However, it is not even clear whether MATLAB provides optimized implementations for filters, much less for the DWT. It is possible that, in MATLAB the transform domain methods utilize optimized FFTW libraries for real and complex DFTs; however, the transform-domain approach might not provide the most efficient filter implementations for all filter and input signal lengths. We will extensively explore the tradeoffs between

speed and computational cost in this thesis. In MATLAB, the DWT is implemented in independent stages (levels) where at each stage the input signal is filtered and then followed by downsampling operations. MATLAB uses a straightforward implementation of Mallat's algorithm [14] almost with practically no optimization stages to improve performance. Further, Mallat's algorithm is only one of many competing alternatives, such as the lifting steps (LS) implementation [19]. The lifting scheme reduces the arithmetic cost and provides in-place computations, which makes it a preferred choice for numerical computation of the DWT in many problems, e.g., in the image compression coding standard JPEG2000 [17]. The software package LIFTPACK provides C routines for fast computation of 1D and 2D biorthogonal wavelet transforms using the lifting scheme [20].

At this point, we mention several other popular software packages for wavelet-based signal processing:

- Wavelet Extension Pack from Math Soft;
- Mathematica Wavelet Explorer from Wolfram Research [21];
- WaveLab — a library of Matlab routines for wavelet analysis and wavelet packets from Stanford [22];
- LastWave — free (GNU) signal processing software in C [23];
- QccPack, an open-source package that provides routines for wavelet based signal processing, including the DWT, among many other compression, quantization, and coding methods [24]
- JasPer, a reference C-implementation of JPEG2000 that uses the two-dimensional DWT [25].

We emphasize that none of these software libraries provides direct solutions for automatic tuning of filtering and wavelet kernels to different platforms. The code, at best, includes subroutines that most likely have been hand coded, or, as in the case of MATLAB, use existing platform adaptation packages as building blocks for constructing implementation methods.

Breitzman [26], in his Ph.D. thesis, developed a software package that automatically generates and implements a large class of linear and circular convolution algorithms based on the Chinese remainder theorem (CRT) and multi-dimensional convolution techniques that nest smaller convolutions inside larger convolutions to reduce the operation count. MAPLE package is used to represent and manipulate algorithms on a symbolical mathematical level and generate a large space of alternative solutions. Further, a proprietary compiler is used to automatically translate the algorithms into code. Breitzman provides a detailed cost analysis for CRT-based and split-nesting algorithms for convolutions of sizes up to 1024. The focus of this work is to provide an automatic generator of a comprehensive space of CRT-based algorithms for smaller convolutions and analyze performance in terms of the computational cost with respect to FFT-based algorithms.

On the other hand, hardware vendors like Intel provide hand-optimized libraries for a set of filtering routines and wavelet transforms. Intel maintains these libraries through a large team of expert programmers who most probably hand-tune the code in assembly to adapt it to the family of Intel processors. Intel provides the Math Kernel Library (MKL) and Intel Performance Primitives (IPP), libraries of routines for a variety of DSP functions. The IPP includes support for FIR and IIR

filters, the DWT, and other multirate filter banks. Libraries are tuned for a set of Intel processors such as: Itanium (IA-64), Pentium 4 and Xeon (IA-32), XScale, and StrongARM.

We conclude that the existing software libraries for FIR filters and wavelet transforms are either:

- portable but not optimized and tuned to the host platform, or
- hand-optimized for a specific platform, but not portable.

To close this gap, we provide a system that will automatically generate platform-adapted code for FIR filters and the DWT. We expect our automatically generated code to be competitive with the best available hand-coded implementations including vendor libraries that are available for the target platform.

1.2 Problem Statement

Automatic performance tuning systems are designed to avoid the costly cycle of re-implementing and re-tuning implementations for numerical kernels. They try to address these two important problems:

1. Designing high-performance implementations of DSP algorithms is a complex and difficult task that involves tuning at both the algorithmic and the implementation level.
2. Rapid evolution of computing platforms makes hand-tuned code quickly obsolete and the tuning process has to be repeated.

Because of the growing number of performance-critical DSP applications, automatic tuning for DSP kernels, especially DSP transforms and filters, becomes increasingly more important for scientists. Despite the success achieved for the discrete Fourier transform and for several other trigonometric transforms (e.g., [13, 9]), digital filtering and wavelet kernels are still not automatically tuned, although a considerable effort has been made to investigate optimization techniques for a set of chosen platforms (see, e.g., [27, 28, 29]).

The SPIRAL system provides a set of functional blocks that enables efficient automatic generation and translation of algorithms into C code, as well as an intelligent search module for optimization of the implementations based on the measured run time. In addition, SPIRAL offers its own set of verification and debugging tools that facilitate testing.

What we are trying to achieve. There are several important objectives we hope to accomplish in this thesis. Using the current SPIRAL system and integrating filtering and wavelet kernels, we would like to be able to

- *Automatically generate* a comprehensive space of competitive algorithms for FIR filters and DWTs using a suitable and concise mathematical language;
- *Automatically generate* code from the algorithms represented as mathematical formulas;
- *Adapt* code to the target platform using available search techniques;
- *Outperform* or be competitive with the available hand-coded implementations and vendor libraries.

Using SPIRAL, we propose to automatically generate *high-performance* software implementations for FIR filters and a class of DWTs that will be competitive with the best available hand-coded libraries.

> The *goal of this thesis* is to design a new framework for representing and manipulating existing filtering and wavelet algorithms, and incorporate this framework in the SPIRAL system to automatically generate and tune software implementations for FIR filters and discrete wavelet transforms (DWTs) across a wide range of platforms. In other words, this thesis tries to *close the gap* between: 1) existing software implementations that provide portability; and 2) hand-tuned libraries that achieve high performance by specializing for specific architectures.

Problems we do not address. To further clarify our goals and avoid misleading implications, we list problems that we are not addressing in this thesis.

- We do not derive new filtering and wavelet algorithms. We borrow existing algorithms available in the literature and formulate them using our symbolic mathematical language in order to automatically represent, manipulate, and generate the space of possible alternative algorithms. To achieve this, we expand the signal processing language (SPL) in SPIRAL with additional constructs and primitives and express a comprehensive collection of existing filtering and wavelet algorithms in SPL — a non-trivial task as we will show in chapters 5 and 6.
- We are not trying to optimize algorithms to run on specific architectures, nor do we attempt to draw general conclusions on the efficiency of specific methods for FIR filter and DWT implementations.
- We do not design and implement specific filters and wavelet systems. We enable the SPIRAL system to automatically implement filtering and wavelet kernels as numerical building blocks, where high performance is achieved automatically through search over a comprehensive space of implementations.
- We do not provide a highly-efficient source code or any pre-compiled libraries for FIR filters and DWTS. Our system generates code "from scratch" on the platform of choice by installing the system and searching for the best adapted solution.

To meet these objectives, we need to solve several important problems and overcome several challenges to enable SPIRAL to generate high-performance filter and wavelet implementations. We address these challenges next.

1.2.1 Challenges

Our goal is to enable the automatic generation of high-performance implementations for FIR filters and an entire class of discrete wavelet transforms using SPIRAL. There are several stages in the

process that need to be addressed and solved.

1. Define different discrete operations for FIR filtering and wavelet signal processing and identify existing fast algorithms for their implementation;

2. Select a subset of algorithms that have properties suitable for implementation on modern computer platforms;

3. Develop a mathematical language that will enable: *i*) the efficient representation of the set of algorithms and methods; *ii*) easy manipulation of these representations to efficiently generate different algorithms; and *iii*) preserving enough structural information to allow the machine to interpret the mathematical formulas as programming structures;

4. Capture a comprehensive class of algorithms for FIR filters and DWTs using the developed mathematical language;

5. Define and implement new mathematical objects in the system. Implement all algorithms and methods using the defined objects. Verify the algorithms represented as mathematical formulas for correctness;

6. Design templates for translating the mathematical formulas into the programming language;

7. Verify if the generated code produces the correct outputs and whether the program structure (code schedule, loops, etc.) is as expected and designed by templates;

8. Use the existing search engine provided by SPIRAL and adjust the search techniques to suit the specific characteristics of the mathematical description for FIR filter and DWT algorithms;

9. Enable search over the space of algorithms and implementations and ensure that the search converges in reasonable time. If the search space is too large, determine the effective heuristics to restrict the space of possibilities.

The above steps should enable us to use SPIRAL to generate the space of implementations, evaluate their performance, and search for the optimized one on the target platform.

We first identify a class of discrete-time signal processing operations that represent FIR filtering, such as the linear and the circular convolutions and the filtering operation for infinite and infinitely extended signals on a finite number of outputs. We also define the discrete wavelet transforms in a similar manner by using different signal models. Next, we find the existing fast algorithms for these operations in the literature. The algorithms are usually represented in the form of mathematical formulas with summations or even only as verbal descriptions, which are all unsuitable for machine representation and automatic algorithm generation. The first challenge is to capture all algorithms as mathematical formulas using a small set of constructs and operators that exhibit the structure of the computations in a concise form, manageable from the perspective of automatic generation and manipulation by a computer.

The next challenge is to integrate the new framework into the existing SPIRAL's framework initially designed for generation of DSP trigonometric transforms, test it, and use it to automatically generate fast implementations. Since the filtering and wavelet algorithms have a considerably

different structure, many concepts and assumptions have to be revised to accommodate new transforms and algorithms. For example, since the convolution operation has a very regular structure, it is conceivable that it might sometimes be advantageous to do a more expensive straightforward implementation of the convolution than to implement an algorithm with a lower arithmetic cost but a more complex data flow pattern. For that reason, the assumption often used for trigonometric transforms that a lower cost algorithm usually yields faster run times can be severely violated for filters. Furthermore, code scheduling and the standard compiler optimization techniques can adversely affect the regular structure of computations as we mentioned in Section 1.1.2. For these reasons, the algorithms for FIR filters and the DWTs have to be carefully analyzed, properly represented using mathematical constructs, and optimized using the right implementation choices, such as compiler and loop unrolling options, in order to achieve high performance.

In the following chapters of this thesis, we develop all required concepts, implement the new framework, and demonstrate the effectiveness of the approach by conducting a set of carefully chosen experiments. We address many practical issues as they arise and discuss the approach to solve them. We also mention the limitations of the developed framework and the system as a whole.

1.3 Organization of the Thesis

In Chapter 3, we provide a brief overview of the SPIRAL system. The focus will be on SPIRAL's mathematical framework and the basic concepts on which it is built. This will provide the foundation for the framework we develop for FIR filters and DWTs in this thesis. We also review other important modules of the SPIRAL system and focus on properties and functions that will enable us to conduct our experiments in Chapter 7. In this part of the thesis, we introduce the reader to the concepts of automatic tuning of software implementations to computer platforms through efficient algorithm generation, translation, evaluation, and optimization.

In Chapter 3, we introduce the concepts of digital signal processing, digital filters, wavelet expansions, and discrete wavelet transforms. We review the most important classes of algorithms for efficient implementation of FIR filters and DWTs found in the literature. For most of the methods we present only the material required for understanding the basic principles of the covered algorithms, the notation, and the equations we will be using later to construct the computational kernels. We provide references for readers who wish to get into more details. When necessary, we elaborate on some of the algorithmic issues in the appendixes, mostly when the topic is not well covered in the literature.

Chapter 4 introduces basic mathematical definitions, properties, and identities that we use to develop the framework for FIR filters and the DWTs. For completeness, we cover some of the well known matrix operations and overview their properties. We introduce new matrices, matrix constructs, and operators that we need for developing the mathematical formalism for representing the algorithms.

We develop the mathematical framework for FIR filters in Chapter 5. First, we introduce the definitions of FIR filter transforms, the notation we use to represent them, and their properties. Examples are used to clarify these definitions. We introduce useful transforms derived from the basic definitions, such as circulant transforms and filter bank transforms. Using the mathematical

language we developed in Chapter 4, we capture all the reviewed algorithms from Chapter 3 in a set of formal rules that spans the algorithm space.

We follow the same pattern in Chapter 6 for the DWTs. Depending on the chosen signal model, we provide several definitions of the DWT, the exact matrix forms using the developed language, and the inverse DWT definitions that enable perfect reconstruction. For each of the DWTs, we capture a set of rules for their implementation, based on the algorithms covered in Chapter 3.

We set the experiments and present a selection of relevant results in Chapter 7. We choose a set of different platforms to demonstrate the effects of platform adaptation for filter and wavelet implementations. We compare a set of frequently used methods for speed and computational cost. Several benchmarks tests are performed on the best found filtering and DWT implementations both against the theoretical peak performance on the host platform, and against reference implementations.

Finally, in Chapter 8 we discuss our results and identify issues that demand further investigation. We suggest the direction of future research, extensions of the current approach, and provide pointers for improvement of the given solutions.

CHAPTER 2

SPIRAL CODE GENERATOR

Performance-critical software routines typically require careful tuning of code to take full advantage of specific features of the target computer platform. In the previous chapter, we discussed automatic performance tuning systems whose goal is to automatically tune software libraries to continuously changing computer architectures and compiler technologies. Their purpose is to avoid tedious and time-consuming hand coding that require architecture and algorithm experts and employ teams of programmers.

In Section 1.1.1, we mentioned several examples of automatic performance tuning systems that focus mainly on basic linear algebra kernels and a few signal processing transforms. These systems use domain-specific knowledge to search for the optimal blocking or recursion strategies mostly to tune the implementations to match the memory hierarchy of the target computer platform.

The SPIRAL system focuses on the domain of linear digital signal processing (DSP) algorithms [11, 12, 13]. Instead of searching over different coding solutions to adapt to the underlying architecture, SPIRAL formulates the problem as an optimization problem over the space of different algorithms and implementations for the chosen task. Tuning to the platform is achieved by automatic algorithm generation, implementation, and search over the space of efficient candidates. Many alternative solutions and optimization techniques are represented at a very high mathematical level, which allows SPIRAL to use the algorithmic knowledge, independent of the specific platform features. This approach ensures high portability across current and future computer platforms, and unifies the optimization techniques that appear common to many known algorithms in the DSP domain. Our goal in this thesis is to integrate the mathematical knowledge about FIR filters, discrete wavelet transforms, and applicable efficient algorithms into the SPIRAL framework and use SPIRAL's code generator and search engine to automatically find and create code optimized for the given computer platform. In this chapter, we introduce the SPIRAL system and cover basics of the major system components including the mathematical framework used for efficient algorithm representation and generation, compiler techniques for mapping the mathematics into executable code, and search strategies that produce optimized solutions given a reasonable amount of time.

2.1 SPIRAL System Overview

The SPIRAL system generates optimized platform-adapted implementations for a variety of important DSP transforms. SPIRAL takes a conceptually different approach from other performance

tuning systems. Different DSP algorithms are represented using concise mathematical formulas built recursively to enable efficient generation and manipulation. The formulas are further used to perform general optimizations at the structural level of the algorithms, preparing them for translation into code. SPIRAL then employs a domain-specific compiler to turn the mathematical formulas into target code. Many standard compiler optimization techniques tailored to utilize the special structure of DSP algorithms are performed at this level. The target code is then compiled using a generic compiler into an executable, which is measured for performance. The process of generating implementations is repeated for a large number of algorithms to minimize the run time. The generation of new algorithms and the choice of implementation parameters is controlled by SPIRAL's search engine, which efficiently navigates the space of algorithms and implementations in search for the optimized code. Thus, SPIRAL's approach to automatic performance tuning can be seen as an optimization problem in the space of algorithm and implementation alternatives, where the cost function is typically the run time at execution [13]. The optimization technique is determined by the choice of search strategy. We now briefly explain each step of the optimization process individually.

The architecture of SPIRAL is schematically shown in Figure 2.1. We discuss each block in the figure starting from the top. At the algorithm level, the *formula generator* uses a small set of mathematical constructs and primitives to symbolically represent algorithms for DSP transforms. The main idea behind this approach relies on the fact that many DSP algorithms exhibit highly structured data flow patterns and recursive computation. It turns out, as we will shortly see, that the algorithm structure can be captured in a form of decomposition rules that govern different data flow patterns. The rules can be combined and applied recursively to provide many different ways to decompose a transform, giving rise to numerous alternative algorithms. Due to the very regular structure of computations in DSP algorithms, most of the decomposition rules can be represented using a relatively small set of carefully chosen constructs. The rules are applied recursively as many times as desired or possible, which creates mathematical formulas that express the exact structure of the algorithm. This *rule formalism* enables efficient generation of the whole space of algorithms whose extent depends on the number of rules included in the system and the size of the problem. The algorithms that are generated and represented as mathematical formulas in the formula generator undergo additional optimizations using formula manipulation rules. The optimizations are performed strictly at the mathematical level to fully exploit the available structural information and prepare the formulas for translation. The adapted formulas already provide hints about their actual implementations, such as loop index mappings for composed permutations, but more specific choices, such as the size of the loop body, are left to lower decision levels. The formulas are expressed in a domain-specific language called SPL (signal processing language) and sent to the SPL compiler that takes over the algorithm at the implementation level.

The job of the *SPL compiler* is to translate an SPL formula into C or Fortran code. The compiler uses a library of translation templates for all mathematical constructs and formula atoms and matches them with the given formula. At this stage, the compiler employs specific implementation strategies, controlled by the search engine, such as the degree of loop unrolling on the global and local levels. Implementation level optimization is performed to further improve code including standard compiler optimization techniques such as common subexpression elimination, dead code elimination, constant folding, and others, and code scheduling strategies to localize computations

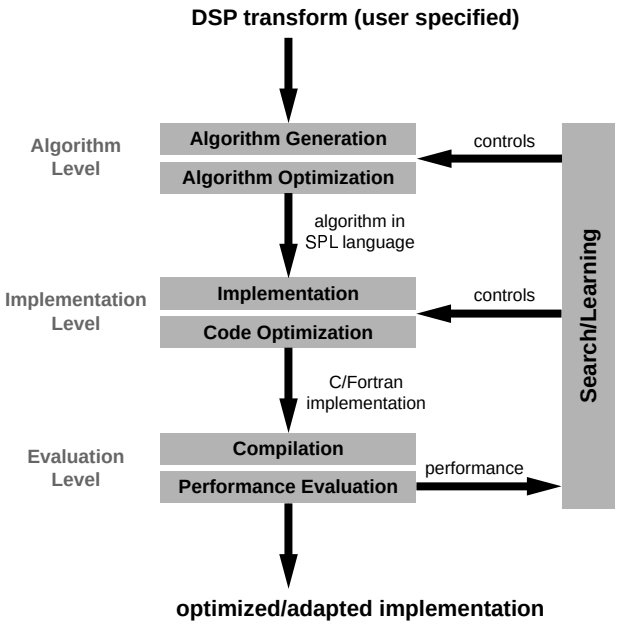


Figure 2.1: The architecture of SPIRAL

to match the memory hierarchy of the computing platform. Even though these optimizations are handled by standard C or Fortran compilers, additional preparation is required for automatically generated code that typically contains large portions of straightline code. To summarize, given a formula, the SPL compiler produces code based on additional implementation options.

The source code is then compiled using, for example, a standard C compiler and the executable is run and timed for performance in the *evaluation module*. SPIRAL invokes an outside compiler using either a set of predefined platform-specific flags or searches for the optimal set of flags if desired. A special timing package is used to accurately measure the run time across a variety of platforms.

The execution run time is used by the *search engine* to guide the generation of the next algorithm based on some search strategy. The search engine also controls implementation options in the formula translator, which expands the search space to a product of the algorithm space and the implementation strategies. Thus, for a given transform, the search engine navigates through the space of possible algorithms and their possible implementations to find the code best matched to the target platform. Given that the search space is finite, exhaustive search could conceivably yield the optimal platform-adapted implementation; however, in most cases, exhaustive enumeration is not feasible in a reasonable amount of time. To overcome this limitation, SPIRAL employs various search techniques, including dynamic programming, random search, and evolutionary search [30]. SPIRAL allows the user to specify whether the search strategies should be interleaved and whether the search should be terminated after a period of time or let run until a terminating condition is reached.

To summarize, SPIRAL generates an extensive space of implementations where not only different algorithms are tested for best performance, but also different programming constructs and implementation strategies are used to find the ones that are best supported on the host platform.

The broad space of competitive implementations is efficiently searched by SPIRAL's search engine. The best found implementation usually contains intricate programming structures and large portions of unrolled code, unlikely to be chosen by a human programmer. Searching this space of fast alternatives ensures highly optimized, platform-adapted code for a given DSP transform.

We first introduce the fundamental concepts of SPIRAL's mathematical framework, explain the rule formalism and relation to the space of different algorithms, and set the ground for developing a framework to enable SPIRAL to produce highly-optimized code for filtering and wavelet kernels.

2.2 SPIRAL's Mathematical Framework

At the core of the SPIRAL system is the mathematical framework developed to allow efficient representation and manipulation of DSP algorithms. Since SPIRAL focuses on the domain of linear DSP operations, such as linear transforms and linear filtering, all objects are represented in matrix form. Algorithms for efficient computation of DSP transforms are symbolically represented by mathematical equations generated by a library of specific rules, much like the role of grammar rules used to build formal languages [31].

Most of the well-known and widely used fast DSP algorithms rely on structural decompositions of a DSP transform into smaller size transforms and mathematical primitives using the universal divide-and-conquer approach. Algorithms can be very complicated and typically have an inherent recursive structure that is difficult to express using standard mathematical tools. Fortunately, the structure of most DSP algorithms is remarkably regular and conveniently expressed using mathematical operators with nice properties and clear interpretations in terms of needed programming constructs.

To accomplish full automation of code generation and adaptation, the mathematical framework has to provide not only the precise and concise description of DSP algorithms, but also enable easy generation of the full spectrum of admissible algorithms and their easy manipulation and modification. Furthermore, the mathematical description of algorithms has to be clear and unambiguous so that a computer is able to effectively translate it into desired code. This is exactly what SPIRAL's framework provides and precisely what enables automated algorithm generation and search through the space of alternative solutions.

The next sections introduce and explain several key concepts of SPIRAL's framework that allow the generation of the space of algorithms from the core library of mathematical rules.

2.2.1 Key concepts

In SPIRAL, all linear DSP operators are treated as matrices represented either symbolically or explicitly by their elements. The goal in SPIRAL is the automatic implementation of important DSP operators such as DSP transforms and linear filters, all simply referred to as *transforms*. SPIRAL represents algorithms for fast implementation of transforms as matrix equations. So, the matrix formulation of a DSP transform is the starting point of our discussion.

Transform. A *transform* in SPIRAL is a class of parameterized and typically highly structured matrices. To *compute* the transform T or to *apply* the transform T to the signal $\mathbf{x} = (x_0, \ldots, x_{n-1})$ means to compute the matrix-vector product $\mathbf{y} = T\mathbf{x}$. The main goal of SPIRAL is to compute

a transform T as efficiently as possible. To differentiate transforms from other matrices, we write them in bold face $\mathbf{T}$. We provide a few important examples of DSP transforms.

The discrete Fourier transform (DFT) of size n is defined as

$$\mathbf{DFT}_n = \left[e^{-2\pi jk\ell/n}\right]_{k,\ell=0,\ldots,n-1} \qquad j = \sqrt{-1}. \tag{2.1}$$

We say that $\mathbf{y} = \mathbf{DFT}\,\mathbf{x}$ is the DFT transform of the input sequence $\mathbf{x}$.

The discrete cosine transform (DCT) of type 2 is given by

$$\mathbf{DCT\text{-}2}_n = \left[\cos\tfrac{\pi k(2\ell+1)}{2n}\right]_{0\le k,\ell<n}. \tag{2.2}$$

The discrete Hartley transform (DHT) is defined as

$$\mathbf{DHT}_n = \left[\cos\tfrac{2\pi k\ell}{n} + \sin\tfrac{2\pi k\ell}{n}\right]_{0\le k,\ell<n}. \tag{2.3}$$

The linear convolution is represented by the convolution transform

$$\mathbf{Conv}_n(\mathbf{h}) = \underbrace{\begin{bmatrix} h_{-r} & & & & & \\ \vdots & \ddots & & & & \\ h_l & \cdots & h_{-r} & & & \\ & \ddots & & \ddots & & \\ & & h_l & \cdots & h_{-r} & \\ & & & \ddots & \vdots & \\ & & & & h_l & \end{bmatrix}}_{n \text{ columns}}, \quad \mathbf{h} = \begin{bmatrix} h_{-r} \\ \vdots \\ h_l \end{bmatrix}. \tag{2.4}$$

Transforms can be defined also recursively. For example, the Walsh-Hadamard transform is defined as

$$\mathbf{WHT}_n = \begin{bmatrix} \mathbf{WHT}_{\frac{n}{2}} & \mathbf{WHT}_{\frac{n}{2}} \\ \mathbf{WHT}_{\frac{n}{2}} & -\mathbf{WHT}_{\frac{n}{2}} \end{bmatrix}, \quad \mathbf{WHT}_2 = \mathbf{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{2.5}$$

Constructs and Operators. There are several matrix constructs that are indispensable in representing DSP transforms and their algorithms. As an illustration, we provide a few examples in this section. We shall go into much more detail in Chapter 4 where we define all required mathematical objects.

The *Kronecker* or the *tensor* product is at the top of the list of useful constructs. Consider the $m \times n$ matrix $A = [a_{i,j}]$ and $r \times s$ matrix $B = [b_{i,j}]$. The tensor product is the $mr \times ns$ block matrix C of the following form

$$C = A \otimes B = \begin{bmatrix} a_{0,0}B & \ldots & a_{0,n}B \\ \vdots & \ddots & \vdots \\ a_{m,0}B & \ldots & a_{m,n}B \end{bmatrix}. \tag{2.6}$$

For example, if I_n is the $n \times n$ identity matrix then the tensor product $C = (\mathrm{I}_s \otimes A)$ is a block diagonal matrix with the matrix A repeated s times on the diagonal.

As we will shortly see, the tensor product has a pervasive presence in systematic analysis and representation of DSP algorithms; however, its importance reaches far beyond the realm of DSP transforms [32].

Another well-known and useful construct is the *direct sum* of matrices. For matrices A and B defined above, the direct sum is defined as

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}. \tag{2.7}$$

We also define new matrix constructs specialized for representation of filters and wavelets. The *column overlapped tensor product*, for example, is given by

$$\mathrm{I}_s \otimes^k A = \begin{bmatrix} \boxed{\mathrm{A}} & & & \\ & \boxed{\mathrm{A}} & & \\ & & \boxed{\mathrm{A}} & \\ & & & \ddots \\ & & & & \boxed{\mathrm{A}} \end{bmatrix}, \tag{2.8}$$

where s is the number of times matrix A is repeated, and k is the number of overlapping rows.

We provide just this single example at this point. More will be said in Chapters 4 and 5 when we introduce other constructs and transforms.

Besides matrix constructs, we define many matrix operators. Operators in SPIRAL are special parameterized matrices that do not represent transforms, i.e., the matrices that are not independently implemented, and are used as auxiliary elements in transform algorithms. The most often encountered examples are various permutation matrices whose purpose is to reorder the processed data. An example is the so-called *stride permutation* matrix that performs the following mapping of indices:

$$l^{rs}_s \ : \begin{cases} \{0,\ldots,n-1\} \mapsto \{0,\ldots,n-1\} & \\ i \mapsto i \cdot s \bmod n, & i = 0,\ldots,n-2 \\ i \mapsto n-1, & i = n-1 \end{cases}. \tag{2.9}$$

In matrix form, the stride permutation is, e.g.,

$$\mathrm{L}^4_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad (x_0, x_2, x_1, x_3)^T = \mathrm{L}^4_2 \, \mathbf{x}. \tag{2.10}$$

Rules. Decomposition or breakdown *rules* are mathematical identities that specify how to factor a transform into matrices that are either sparser, have lower complexity, or exhibit a structure that has some advantage for implementation. A typical rule exploits the redundancies of the transform computations and implicitly uses common subexpression elimination to reduce the computational cost. Fast algorithms rely on such "good" rules for transform decomposition where the term "fast" refers to the reduced arithmetic cost. However, sometimes, the purpose of breakdown rules is not to reduce the cost but to improve the data access for locality or to simply provide a bridge to other transforms and rules.

We illustrate the concept of rules with a familiar example. It can be easily verified that the DFT of size 4 can be decomposed using the following factorization:

$$\mathbf{DFT}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & j & -1 & -j \\ 1 & -1 & 1 & -1 \\ 1 & -j & -1 & j \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & j \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.11}$$

where $j = \sqrt{-1}$. If we regard multiplication with j and the sign change as no operations, the original $\mathbf{DFT}_4$ requires 12 complex additions whereas in the decomposed form (2.11) there is a need for only 8 additions. The original matrix is decomposed into a product of sparse matrices by taking advantage of repeated patterns in the DFT matrix. At this point we emphasize that, not only are the matrices sparse, but they are also highly structured. We observe that the first matrix from the right is precisely the permutation matrix L_2^4 in (2.10). The second matrix from the right is a block diagonal matrix with blocks being $\mathbf{DFT}_2$ defined in (2.5). Using the tensor product (2.6), we can write this matrix as $\mathrm{I}_2 \otimes \mathbf{DFT}_2$. Similarly, we represent the first matrix from the left as $\mathbf{DFT}_2 \otimes \mathrm{I}_2$. We can now write the decomposition in (2.11) as

$$\mathbf{DFT}_4 \rightarrow (\mathbf{DFT}_2 \otimes \mathrm{I}_2)\, \mathrm{T}_2^4\, (\mathrm{I}_2 \otimes \mathbf{DFT}_2)\, \mathrm{L}_2^4 \tag{2.12}$$

The diagonal matrix T_2^4 will be defined shortly. It turns out that the above decomposition is a special case of the famous divide-and-conquer approach for computing the DFT, first invented by Cooley and Tukey, known as the fast Fourier transform (FFT) algorithm [33]. The decomposition can be generalized to any DFT of composite size $n = p \cdot q$ and represented using the same notation [34].

Cooley-Tukey Rule

$$\mathbf{DFT}_n \rightarrow (\mathbf{DFT}_p \otimes \mathrm{I}_q)\, \mathrm{T}_q^n\, (\mathrm{I}_p \otimes \mathbf{DFT}_q)\, \mathrm{L}_p^n\,. \tag{2.13}$$

The twiddle matrix T_q^n is a diagonal matrix of complex roots of unity defined as

$$\mathrm{T}_q^n = \bigoplus_{i=0}^{p-1} \mathrm{diag}(1, w_n^{i\cdot 1}, \ldots w_n^{i\cdot(q-1)})$$

where $w_n^k = e^{2\pi j \frac{k}{n}}$ [35].

The identity (2.13) is a breakdown rule. Whenever we use rules, we denote them using the arrow sign instead of the equality sign. In the most general form, a rule is denoted as

$$\mathbf{T}_n \rightarrow \mathcal{R}_{type} \left\{ \mathbf{T}'_r, \mathbf{T}''_s, \ldots, \mathbf{T}_t^{(n)} \right\}, \tag{2.14}$$

which states that the rule $\mathcal{R}_{type}$ is applied to the transform $\mathbf{T}_n$ and decomposes it into either smaller transforms of the same type or other transforms of the same or smaller size. The Cooley-Tukey (CT) rule (2.13) can thus be seen as

$$\mathbf{DFT}_n \rightarrow \mathcal{R}_{\mathrm{CT}} \left\{ \mathbf{DFT}_p, \mathbf{DFT}_q \right\}.$$

It should be obvious that rules can be applied recursively. In the above example, if the sizes p and q are composite $p = r \cdot s$, $q = u \cdot v$, then we have

$$\mathbf{DFT}_n \rightarrow \mathcal{R}_{\mathrm{CT}} \left\{ \mathcal{R}_{\mathrm{CT}} \left\{ \mathbf{DFT}_r, \mathbf{DFT}_s \right\}, \mathcal{R}_{\mathrm{CT}} \left\{ \mathbf{DFT}_t, \mathbf{DFT}_u \right\} \right\}. \tag{2.15}$$

The recursion can be continued as many times as possible, leading, in this example, to all possible mixed-radix Cooley-Tukey FFT algorithms, as we will discuss shortly.

We now compare the rule (2.13) with the standard double-sum notation of the same decomposition usually found in the literature

$$y_{k_1+qk_2} = \sum_{n_2=0}^{p-1} \sum_{n_1=0}^{q-1} \left(x_{pn_1+n_2} w_q^{k_1 n_1} \right) w_n^{k_1 n_2} w_p^{k_2 n_2}. \tag{2.16}$$

Some of the major advantages of the rule notation that arise from the above comparison can be summarized in the following list.

1. The representation is very concise and the computational stages are clearly separated and ordered going from right to left;

2. The recursive structure is obvious since the rule can be applied again to the transforms on the right side of the identity;

3. The rule captures important structural information essential for designing the actual implementation. For example, the construct $\mathrm{I}_p \otimes \mathbf{DFT}_q$ suggests that the DFT of size q has to be applied in parallel p times after reordering of the input data using the stride permutation L^n_p. After scaling the result with the twiddle matrix, one need to apply the DFTs of size p to the data reordered at stride q as suggested by $\mathbf{DFT}_p \otimes \mathrm{I}_q$.

Every transform has a set of associated decomposition rules that can be used freely as long as the conditions, such as the size of the transform, are satisfied. In that case, we say that the rule is *applicable* and by *applying* the rule we mean substituting the transform with (2.14). As an illustration, Table 2.1 lists some of the rules for the transforms we defined in this section. The operators P and Q are permutations and S are summation matrices. For more details, the reader is referred to [13, 35, 36, 37].

Table 2.1: Rule examples

Trigonometric Rule	$\mathbf{DFT}_n$	$\rightarrow$	$\mathbf{CosDFT}_n + \sqrt{-1} \cdot \mathbf{SinDFT}_n$
Good-Thomas Rule	$\mathbf{DFT}_n$	$\rightarrow$	$P_n \left(\mathbf{DFT}_p \otimes \mathbf{DFT}_q\right) Q_n, \quad n = p \cdot q, \ \gcd(p,q) = 1$
Cosine-DCT Rule	$\mathbf{CosDFT}_n$	$\rightarrow$	$S_n \cdot \left(\mathbf{CosDFT}_{n/2} \oplus \mathbf{DCT\text{-}2}_{n/4}\right) S'_n \cdot \mathrm{L}^n_2$
Sine-DCT Rule	$\mathbf{SinDFT}_n$	$\rightarrow$	$S_n \cdot \left(\mathbf{SinDFT}_{n/2} \oplus \mathbf{DCT\text{-}2}_{n/4}\right) S'_n \cdot \mathrm{L}^n_2$
DCT 2-4 Rule	$\mathbf{DCT\text{-}2}_n$	$\rightarrow$	$P_n \cdot (\mathbf{DCT\text{-}2}_{n/2} \oplus (\mathbf{DCT\text{-}4}_{n/2})^{P'_n}) \cdot (\mathrm{I}_{n/2} \otimes F_2)^{P''_n}$
Overlap-Add Rule	$\mathbf{Conv}_n(h(z))$	$\rightarrow$	$\mathrm{I}_s \otimes^{l+r} \mathbf{Conv}_{n/s}(h(z)), \quad s \,\vert\, n$
Iterative WHT Rule	$\mathbf{WHT}_{2^k}$	$\rightarrow$	$\prod_{i=1}^{s} \left(\mathrm{I}_{2^{k_1+\cdots+k_{i-1}}} \otimes \mathbf{WHT}_{2^{k_i}} \otimes \mathrm{I}_{2^{k_{i+1}+\cdots+k_s}}\right)$

Rule trees and formulas. In the example involving the Cooley-Tukey rule (2.13), we have seen how to recursively apply rule (2.15). It is clear that rules can be applied recursively as long as there exists a transform on the right side of (2.14), and as long as there is at least one rule in the library for that particular transform. To make sure a transform has at least one applicable rule, we define the *terminal rules* or the *base case rules* which take the form

$$\mathbf{T}_n \rightarrow \mathcal{R}_{type}\,\{\}\,. \tag{2.17}$$

The terminal rules apply only if specific conditions are satisfied and cannot be followed by any other rules. After a terminal rule is applied, the recursion stops. For example, the base case rule for the DFT is simply

$$\mathbf{DFT}_2 \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \tag{2.18}$$

which means that the recursion can be terminated only after the size 2 DFT is reached.

The recursive application of rules can be naturally represented using tree structures we refer to as *rule trees*. Rule trees are determined by the root transform to be decomposed and by the type and the order of applied rules. We explain this concept using an example.

Consider again the Cooley-Tukey rule (2.13) and apply it to the $\mathbf{DFT}_8$. Since we can factor 8 as either $8 = 4 \cdot 2$ or $8 = 2 \cdot 4$, there are two ways to apply the rule. For example,

$$\mathbf{DFT}_8 \rightarrow (\mathbf{DFT}_2 \otimes \mathrm{I}_4)\, \mathrm{T}^8_4\, (\mathrm{I}_2 \otimes \mathbf{DFT}_4)\, \mathrm{L}^8_2\,. \tag{2.19}$$

After we apply the same rule again on $\mathbf{DFT}_4$, we obtain

$$\mathbf{DFT}_8 \rightarrow (\mathbf{DFT}_2 \otimes \mathrm{I}_4)\, \mathrm{T}^8_4\, \big(\mathrm{I}_2 \otimes (\mathbf{DFT}_2 \otimes \mathrm{I}_2)\, \mathrm{T}^4_2\, (\mathrm{I}_2 \otimes \mathbf{DFT}_2)\, \mathrm{L}^4_2\big)\, \mathrm{L}^8_2\,. \tag{2.20}$$

As the final step, we apply the terminal rule (2.18) to the $\mathbf{DFT}_2$.

$$\mathbf{DFT}_8 \rightarrow \left(\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \mathrm{I}_4\right) \mathrm{T}^8_4 \left(\mathrm{I}_2 \otimes \left(\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \mathrm{I}_2\right) \mathrm{T}^4_2 \left(\mathrm{I}_2 \otimes \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\right) \mathrm{L}^4_2\right) \mathrm{L}^8_2\,. \tag{2.21}$$

The recursion terminates at this point since there are no more transforms on the right side of the identity. The formula we obtained represents one of the possible algorithms based on the Cooley-Tukey approach.

When a formula is fully expanded, such as the formula above, it uniquely represents an *algorithm*. In the rule tree representation, this means that the leaves are terminals, i.e., there are no transforms in the leaf nodes. We call such a tree a *fully expanded rule tree*, which graphically represents an algorithm for computing the transform at the root node, whereas the corresponding formula is a mathematical description of the same algorithm. This description is obtained using similar principles used to construct formal languages [31]. For that reason, the whole mathematical framework for representation of the DSP algorithms is called the signal processing language (SPL).

Let us go back to our example. The recursive application of the Cooley-Tukey rule led to formula (2.21). The recursion can be represented by the rule tree shown to the left in Figure 2.2. The root node is the $\mathbf{DFT}_8$ transform that is decomposed using the Cooley-Tukey rule into the $\mathbf{DFT}_2$ and $\mathbf{DFT}_4$, where the latter is further decomposed using the same rule into two $\mathbf{DFT}_2$ transforms. At that point, all $\mathbf{DFT}_2$ transforms are terminated using rule (2.18) resulting in the fully expanded rule tree.

A rule tree is determined by the order of rules that are applied to transforms that represent the nodes of the tree. The types of rules applied at each node are denoted on each level of the rule tree. In this example, the only applied rule was the Cooley-Tukey rule. The terminal rules applied to each leaf node are omitted from the tree graph to save space.

As we show in Figure 2.2, there is one more way to expand the rule tree from the $\mathbf{DFT}_8$ using just the Cooley-Tukey rule, by decomposing it into the $\mathbf{DFT}_4$ to the left and $\mathbf{DFT}_2$ to the right. These

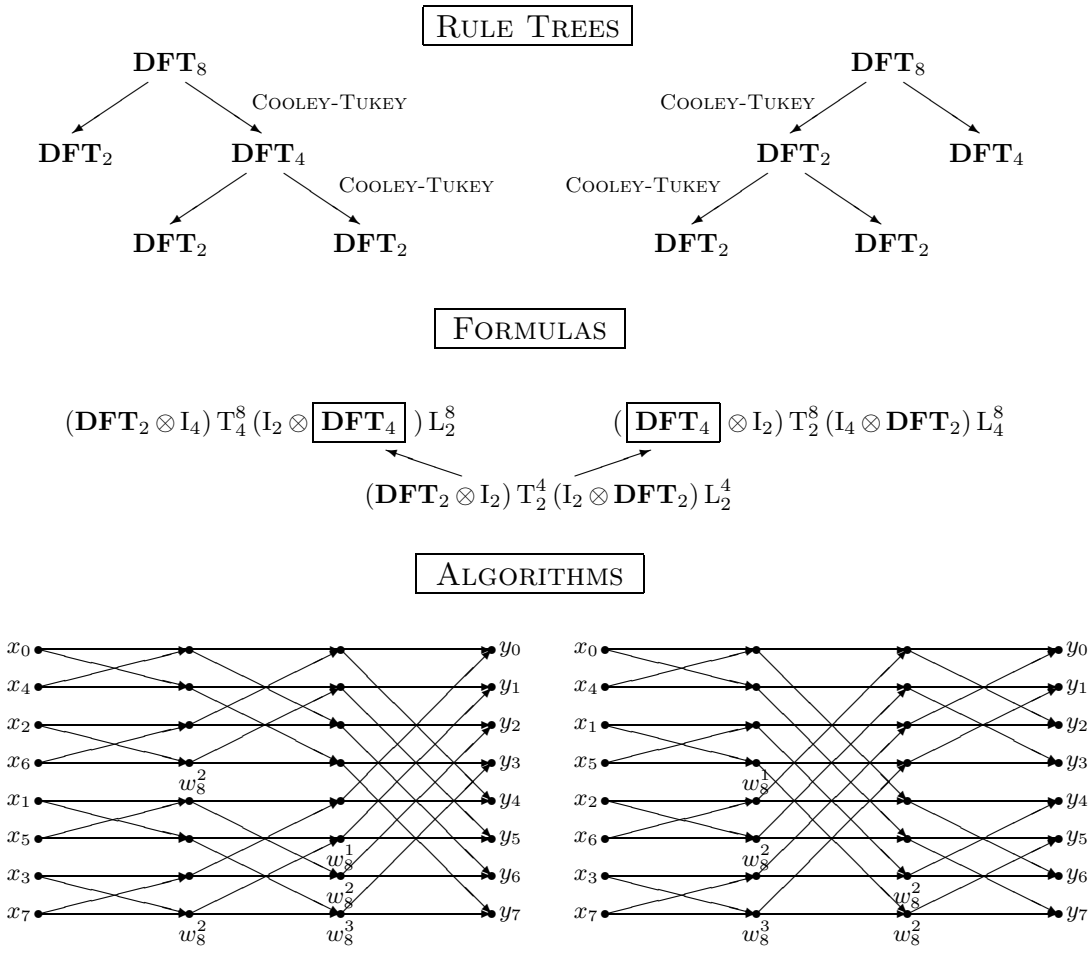


Figure 2.2: Rule trees = formulas = algorithms.

two trees are very similar but their formulas, shown below the trees, are different, proving that they lead to different algorithms. To illustrate the difference, we also include the data flow diagram for each of the two algorithms at the bottom of the figure. The diagrams show that the computational cost of both algorithms is exactly the same but the data flow patterns differ considerably.

2.2.2 Algorithm space

The rule trees, formulas, and algorithms are therefore tightly related. For each algorithm, there is exactly one rule tree and the corresponding mathematical formula that is obtained by substituting transforms with the specified rules going from the root node all the way down the rule tree. This way, even the most complicated algorithms can be represented by a simple tree data structure where only the applied rules and the nodes are needed to fully specify the algorithm. Furthermore, the modifications and variations of similar algorithms can be easily obtained by changing the order of the applied rules and substituting and swapping the subtrees. This is a remarkable advantage of the rule tree representation because it allows efficient generation of a large number of algorithms by properly choosing the right set of rules. As we have seen in our example, different instantiations

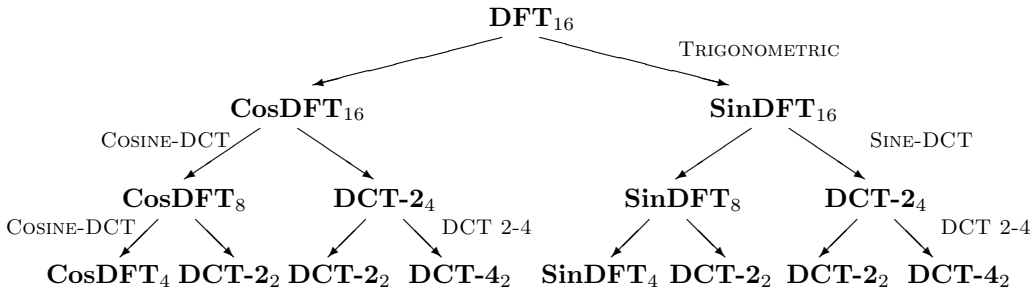


Figure 2.3: A DFT_{16} rule tree

of the Cooley-Tukey rule (2.13) can be applied in different combinations, all of which generate rule trees that represent all possible mixed-radix FFT algorithms.

The number of rule trees and the corresponding algorithms generated using the rule mechanism depends on the transform size and the number of applicable rules at each decomposition stage. Even though there are typically only a few rules per transform, the number of algorithms is still excessively large even for moderate size transforms because rules often cross over to other transforms so that many more rules for other transforms participate in the expansion. For the DFT, there are many other rules besides the Cooley-Tukey rule, e.g., the Good-Thomas and the trigonometric rule shown in Table 2.1. As an example, by combining the rules from the table, we can derive a very large number of rule trees. One such rule tree is shown in Figure 2.3. In general, the number of different algorithms grows exponentially with the size of the transform and the number of applicable rules. SPIRAL, for example, reports more than 10^{19} different algorithms for the $\mathbf{DCT\text{-}2}_{64}$.

As a final note, we mention that in some cases the transforms are related to one another by a simple transposition. In that case, the rules that apply to one transform apply also to another by simply transposing them. For example, since the DFT transform is symmetric then $\mathrm{DFT} = \mathrm{DFT}^T$ and every DFT rule that we transpose will also apply to the DFT. Thus, the transposed Cooley-Tukey (CT) rule

$$\mathbf{DFT}_n \rightarrow \mathrm{L}^n_q \left(\mathrm{I}_p \otimes \mathbf{DFT}_q\right) \mathrm{T}^n_q \left(\mathbf{DFT}_p \otimes \mathrm{I}_q\right), \tag{2.22}$$

will also be valid. For our $\mathbf{DFT}_8$ example, this means that we can have additional rule trees by invoking the transpose $\mathcal{R}^T_{CT}$ instead of the regular $\mathcal{R}_{CT}$. It is interesting to also note that the left rule tree in Figure 2.2 is, in fact, the decimation-in-time (DIT) radix-2 FFT algorithm for the $\mathbf{DFT}_8$, whereas the right rule tree, with all CT rules substituted by the transpose version (2.22) represents the decimation-in-frequency (DIF) radix-2 FFT algorithm [38].

2.2.3 Formula optimization

The generation of algorithms is facilitated by using the rule formalism and the concept of rule trees. The recursive substitution of rules in rule trees generates formulas that precisely define the data flow and the order of computations, i.e., algorithms. The formulas capture important structural information about the algorithms. The permutation matrices, for example, suggest how the data

should be ordered or readdressed before actual arithmetic operations take place. The tensor product suggests the use of loops in the program. In our DFT example, the tensor product of the form $\mathrm{I}_4 \otimes \mathbf{DFT}_2$ suggests that the DFT of size 2 is implemented four times, possibly in a loop. It is up to the formula translator, the SPL compiler, to determine how to actually implement these constructs.

Before the compiler is invoked, however, it is advantageous to use the high-level structural information contained in a formula to perform additional optimizations symbolically rather than on the code level. This is achieved by using *formula manipulation rules* as well as by intelligently combining the dependent elements of the formula.

As an example, consider an $n \times n$ matrix A and the following identity

$$A \otimes \mathrm{I}_s = \mathrm{L}^{sn}_n \, (\mathrm{I}_s \otimes A) \, \mathrm{L}^{sn}_s \, . \tag{2.23}$$

The identity is one of the manipulation rules that suggests that the tensor product $\mathbf{DFT}_4 \otimes \mathrm{I}_2$ can be computed as $\mathrm{L}^8_4 \, (\mathrm{I}_2 \otimes \mathbf{DFT}_4) \, \mathrm{L}^8_2$, i.e., as two DFTs of size 4 in a loop with the inputs readdressed as suggested by the stride permutation L^8_2. Further, stride permutations can be fused together using manipulation rules such as

$$\mathrm{L}^{rst}_r \, \mathrm{L}^{rst}_s = \mathrm{L}^{rst}_{rs} \, . \tag{2.24}$$

Many other manipulation rules for tensor products and stride permutations can be found in [32, 34].

Since the explicit reordering of data is considered very expensive on most architectures because it often violates the locality constraints, one of the most effective optimization techniques is the fusion of successive permutations, such as $\left(\mathrm{I}_2 \otimes \mathrm{L}^4_2\right) \mathrm{L}^8_2$ in the $\mathbf{DFT}_8$ example (2.20), and also the fusion of permutations and other formula elements such as the tensor product of matrices, e.g., $(\mathrm{I}_2 \otimes \mathbf{DFT}_2) \, \mathrm{L}^4_2$. This is achieved by representing constructs and permutations using the sums of gather and scatter operators that solve the readdressing problem by composing simple index mapping functions according to a set of separate function composition rules. Much more on such optimization techniques can be found in [39].

All high-level symbolic computations and representations of data structures such as rule trees are implemented in SPIRAL using the GAP computer algebra system [40]. The GAP system already provides the framework for symbolic mathematical computations for matrices and matrix constructs and can perform exact arithmetic for rational numbers and cyclotomic irrationals, such as square roots of rational numbers and twiddle factors. Another advantage of GAP is that it is open source and can be extended to fit the needs of SPIRAL. Since the search engine needs to control the generation of rule trees and since most implementation options can be specified using tags in formulas, the search engine is also implemented in GAP to take full use of the efficient algorithm generation framework.

2.3 Translation Into Code

After the algorithm is generated as a formula and several optimization steps are performed at the formula level, the second level of the SPIRAL, the implementation level (see Figure 2.1), translates the formula into code. This level is also called the *SPL compiler* since it translates the algorithms

written in the SPL language into the target code. The SPL compiler can be divided into two separate sub-levels: implementation level and optimization level.

Degree of loop unrolling. Before the actual compilation, every formula written in the SPL language is *tagged* with parameters that specify certain implementation choices. One example is the choice of the degree of loop unrolling that determines the maximum size of the block of loop code that is converted into a block of straightline code. The tags can be placed locally on elements and parts of formulas to specify exactly which portions of the algorithm need to be loop unrolled. Conversely, the size of blocks for unrolling B can be also set globally in which case every matrix or sub-formula of size smaller than B will be unrolled at the compilation. It turns out that in most cases the local tagging does not improve considerably over the global setting [12].

Templates and intermediate code generation. Elements of a formula, such as constructs and parameterized matrices, are matched with a library of code *templates*. A template consists of a head that is written in the SPL language and represents an elementary part of a formula, and the body that contains a code fragment written in C-like language that implements that formula element. The head of the template can be a mathematical construct, such as the tensor product, a matrix operator such as the stride permutation, or a full sub-formula. The purpose of a template is to substitute an element of a formula with the code fragment of the matching template. Templates also include conditions, such as the size of the element, under which the substitution can occur. A template for the stride permutation L^n_s is given here as an example

$\mathrm{L}^n_s[n \geq 1 \wedge s \geq 1 \wedge s \bmod n = 0]$

```
blk = n / s
do i0 = 0..str-1
  do i1 = 0..blk-1
    y[i1 + i0*blk] = x[i0*str + i1]
  end
end
```

The template is matched with the symbol L^n_s every time the condition $[n \geq 1 \wedge s \geq 1 \wedge s \bmod n = 0]$ is satisfied and substituted for the given code fragment. Let L^6_3 be the given stride permutation. The substitution yields

```
do i0 = 0..2
  do i1 = 0..1
    y[i1 + i0*2] = x[i0*3 + i1]
  end
end
```

If the size n is lower than the global unrolling size B, then the code is unrolled to

```
y[0] = x[0];
y[1] = x[3];
y[2] = x[1];
y[3] = x[4];
```

```
y[4] = x[2];
y[5] = x[5];
```

The SPL compiler builds an abstract syntax tree from the formula and recursively substitutes the constructs and matrices with the code fragments going from the top of the tree down to its leaves. The local variables in the substituted code are given global names to link the code fragments into the whole code and a common name space. Code from the templates is the intermediate code that is further subject to several code optimization techniques.

Compiler optimization. The intermediate code for the translated formula is further optimized using the standard compiler optimization techniques. The intermediate code is represented using the single static assignment, which means that each variable is assigned a value only once in the entire code segment. This simplifies the optimization methods that include constant and copy propagation, dead code elimination, algebraic simplifications, common subexpression elimination, and array scalarization [13]. The optimizations are interdependent. For example, constant propagation can allow new algebraic simplifications that can, in turn, create new constants that need to be propagated further. Hence, several cycles of the optimization steps are needed to converge to the desirable solution. The iterations are halted when the entire optimization pass does not change the code.

The optimizations performed at the intermediate code level might seem redundant since the target code is compiled using the standard C or Fortran compiler, which have all of the above optimization steps already built in. However, the standard general purpose compilers cannot optimize well code generated by a machine since such code typically involves long portions of straightline code [2, 41, 42, 43]. The SPL compiler, thus, prepares the code in a way to make the job of the standard target code compiler easier.

We also note that, in addition to standard code optimization techniques, SPIRAL also enables various code scheduling methods to achieve better locality. The goal is to optimize register allocation and usage, and minimize the number of register spills.

Target code generation. After optimization of the intermediate code, the last step in the code generation process is to translate the intermediate code into one of the target codes. SPIRAL currently provides backends for generation of C and Fortran target codes including the generation of code supporting special instruction sets such as fused multiply-add (FMA) and single instruction multiple data (SIMD), also known as vector instruction sets, for different standards.

The target code backend inserts initialization functions and constant tables, provides proper function declarations, and makes sure that all programming constructs are assigned the right syntax. If, in addition, the FMA backend is enabled, it reorders the code using the directed acyclic graph (DAG) representation and substitutes successive additions and multiplications with the FMA instructions [44].

SPIRAL further provides the SIMD backend that generates vector code for a variety of proprietary SIMD instruction sets, such as SSE2 on Pentium 4 based platforms and AltiVec for Motorola platforms. Utilizing SIMD instruction sets can lead to a dramatic boost in performance. Theoretically, 4-way SIMD instructions, such as AltiVec, could lead to four times speedup. However, most standard C/Fortran compilers cannot efficiently vectorize code except for algorithms with very

simple structure, which is not the case for most DSP algorithms. SPIRAL handles the problem at the formula level. Special formula manipulation rules are applied to extract the constructs that can be computed using vector instructions. For example, given a formula F, a construct that occurs very often in DSP formulas is

$$F \otimes \mathrm{I}_4 \,. \tag{2.25}$$

Scalar code for this formula implements $y = (F \otimes \mathrm{I}_4)x$. However, using a set of 4-way vector instructions, the code can be implemented as $y = Fx$ where each scalar instruction can be replaced by a corresponding vector instruction, e.g., scalar additions with 4-way additions, and so on. To preserve the portability and avoid compiler limitations, all algorithm manipulations are done at the high formula level, and the special instructions are implemented as a set of C macros on top of current vector extensions. For much more detail, we refer the reader to [45, 46, 47].

2.4 Verification and Runtime Measurements

The process of code generation proceeds through several steps and the overall correctness of the produced code depends on the correctness of each stage. If the generated algorithm implementation does not produce the expected output for a given input, tracking the error in such complex system could prove to be very involving. To facilitate debugging, SPIRAL provides verification at two major levels: 1) verification of rule trees/formulas checks the correctness of rules and generated rule trees; 2) verification of the code tests whether the produced code outputs the right result for fixed input data.

Rule and formula verification. At the formula generator level, the first step in verification is to check the correctness of the breakdown rules for each transform. We mentioned earlier that each transform is a parameterized matrix. A rule is verified by first applying it to the transform, converting the resulting formula to a matrix, and, finally, comparing the matrix to the transform definition. If the transform coefficients are rational numbers or cyclotomic irrationals, the element-wise comparison is exact and the verification either returns true or false. In the case of filters and wavelet transforms, the coefficients can be real numbers represented in finite precision, as we shall see later in this thesis. The verification returns an error that is compared to a preset threshold to determine whether the rule is correct or not.

A rule tree is verified similarly, by recursively converting the sub-formulas to matrices starting from the leaf nodes and going up the tree. The resolution of formulas into matrices is performed symbolically since all objects and constructs of the SPL language are symbols in GAP. Each symbol has a matrix definition or is defined as a matrix operation. The GAP framework is then used to perform matrix calculations.

It is worth mentioning again that the whole verification is performed solely at the mathematical level without generating code. If an error occurs at this level, it is certain that either the rules are incorrect or the matrix definitions of involved formula elements need to be checked.

Code verification. SPIRAL's verifier can also check the correctness of the generated code. The principle is similar to the verification on the formula level: the comparison is made between the code obtained by the direct implementation of the transform $\mathbf{T}$ as a matrix and the code produced

for the tested formula F — the actual algorithm for the given transform. In other words, SPIRAL generates code for the tested algorithm represented by the formula $\tilde{\mathbf{y}} = F\mathbf{x}$ and also code for the transform implemented as a straightforward matrix-vector multiplication $\mathbf{y} = \mathbf{T}\,\mathbf{x}$. Both programs are executed using the same input $\mathbf{x}$. The error $\|\tilde{\mathbf{y}} - \mathbf{y}\|$, based on a chosen norm, is compared to a threshold determining whether the code is correct or not.

The choice of the input test vector $\mathbf{x}$ can affect the result. SPIRAL offers two choices: the test input is randomly chosen, or the code is tested against all vectors from the standard basis

$$\mathbf{e}_i = (0, \ldots 0, \underbrace{1}_{i-\text{th}}, 0, \ldots, 0), \quad i = 1, \ldots, n$$

where n is the size of the input. The standard basis verification is complete because SPIRAL implements only linear transforms for which the standard basis spans all possible cases. On the other hand, in the case of the random vector verification, it can happen that the incorrect code passes the test. This, however, happens in very rare cases. Another alternative is to perform a similar comparison between two different formulas.

SPIRAL can perform extensive tests on the whole space of transforms and algorithms by generating random rule trees for random transforms and verifying the code. The tests can be run periodically for a limited time to check for bugs in the system.

Run time measurement. Algorithm and implementation optimization in SPIRAL requires a well defined and stable performance measure. The concepts of algorithm generation and optimization on which the SPIRAL system is based do not depend on the chosen cost measure. SPIRAL allows the user to specify the cost measure suited to the needs of the application. For example, the cost to be optimized can include accuracy of the implementation, instruction count, or arithmetic cost. In this thesis, however, we exclusively use run time as the cost measure.

To obtain a stable run time, SPIRAL measures the duration of the code execution multiple times and then averages the measurements. This is especially important for small transforms since the relative error due to compulsory cache misses and the current state of the system can be significant for short code. For the same reason, the first measurement is usually discarded as an outlier and the remaining measurements are averaged. To achieve portability, the C-library timing routine `clock()` is used to measure the run time. However, more precise built-in cycle counters are used whenever available (e.g., time stamp counters on Pentium platforms). If the timing routine is more precise, the time spent in the measurement is shorter since fewer measurements are needed to achieve the same accuracy.

2.5 Search

So far, we have explained the complete process of automatic algorithm generation and implementation in SPIRAL, from its conception using a small set of abstract rules to the actual optimized code. If we go back to Figure 2.1 that shows the architecture of the SPIRAL system, we see that this process is executed linearly stage by stage from the transform to fast code. Even though"good" breakdown rules guarantee a low cost algorithm and even though the formula and code optimization steps ensure that the implementation is efficient, different algorithm and implementation choices can

have a very significant impact on performance. As we shall see in Chapter 7, algorithms with the same operations count can have run times that differ significantly, sometimes even by a whole order of magnitude. This is due in part to the significant impact on the run time of the movement of data up and down the memory hierarchy on modern computer platforms, as well as the intricacies of general purpose compilers. Only some combinations of algorithms and implementation choices can yield really fast run times, and the choice heavily depends on the underlying computer platform.

The purpose of the search engine is to navigate the space of numerous alternatives in an efficient way, evaluate the choices, and choose the implementation that runs the fastest on the target platform. If the search space is broad enough, we can claim that such implementation is well adapted to the platform. The search process loops through the system trying to fine tune the algorithms and their implementations to achieve the best possible run time.

The search engine has to be able to control the formula generation and implementation choices and modify them according to the run time measurements obtained from the evaluation module. A straightforward method for searching for the tuned implementation generates all possible algorithms based on SPIRAL's finite set of rules, and all possible implementations using a set of parameters, such as the degree of loop unrolling, and evaluates all of them to find the best solution. For example, given a DFT transform and the Cooley-Tukey rule (2.13) only, SPIRAL can generate all mixed-radix CT algorithms with arbitrary degree of loop unrolling. For the DFT of size 64 and larger, even when we restrict the search to one rule, this exhaustive search is unfeasible. Clearly, more efficient optimization techniques are required.

2.5.1 SPIRAL's optimization techniques

SPIRAL provides several optimization techniques that traverse either the entire search space or the restricted space to find a sub-optimal solution:

1. *Exhaustive search* generates and evaluates all possible rule trees and implementation options for a given transform. A rule tree with the current best run time is saved for comparison. When the whole space is exhausted, the best found code is precisely the desired optimal implementation.

2. *Random search* generates a preset number of random rule trees and measures the run time of the implementations generated by SPIRAL to determine the best one.

3. *Dynamic programming* is suitable for this optimization problem due to the tree structure of the algorithm representation. The optimization is first performed on the leaf nodes and the best found implementations are stored. The search is then repeated at the next higher node in the rule tree where the best found sub-tree is used for the lower nodes.

4. *Evolutionary search* performs genetic algorithm search for the best code. A set of rule trees is first randomly generated and then evolved using cross-breeding and mutation techniques on the best candidates to create a next, better performing generation.

We discuss in more detail dynamic programming, which is the most frequently used and fastest optimization method, as well as the evolutionary search, which often provides better performing

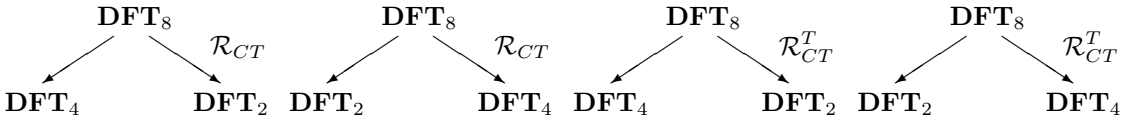
implementations.

Dynamic programming. Dynamic programming (DP) was initially developed to solve optimization problems for nonlinear time-varying systems [48]. The technique is based on the assumption that the system satisfies the *principle of optimality*, which states that if the optimal dynamic path is found then the optimality holds starting from any point on that path. For a discrete problem, the principle of optimality can be translated in the following way. If the optimal path is found that minimizes the cost starting from stage 0 and ending at stage N, then the same path is optimal starting from any point $n \in \{1, \ldots, N-1\}$. The impact is the following: the optimization can be simplified by starting from the last stage $N-1$, finding the optimal cost-to-go, and embedding the solution into the larger problem starting from stage $N-2$. By alternating the optimality principle and embedding technique, the optimization can continue recursively to encompass the entire interval $[0, N]$. Dynamic programming typically reduces the cost of the optimization problem from exponential to polynomial in the number of stages.

It is easy to see how the DP approach applies to the rule tree representation of algorithms. Given a rule tree R, the optimization starts at the leaf nodes. Since the size of the leaf nodes is usually very small or the number of available rules is small, the exhaustive search to find the optimal solution is feasible. The best found implementation for each node is stored efficiently into a hash table by recording the node (the transform), the recursive rule tree structure for that algorithm, and the implementation options. The DP then performs exhaustive search for the node that is one level higher but at this point applies the principle of optimality and substitutes the best found subtree, retrieved from the hash table, for every node below the current root node. When the optimal solution is found it is again stored in a hash table and the DP moves further one level up.

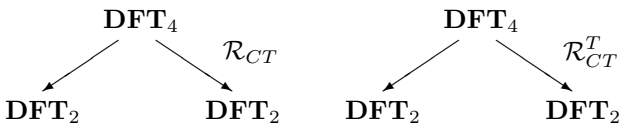
Consider a few simple rule tree examples for the DFT we presented in Figure 2.2 that used only the Cooley-Tukey (CT) rule. To make the example more meaningful for explaining the DP search, in addition to the CT rule $\mathcal{R}_{CT}$ (2.13), we apply also the transposed CT rule $\mathcal{R}_{CT}^T$ (2.22). The DP starts by first expanding rule trees top down. In this case the $\mathbf{DFT}_8$ is first looked up in the hash table to check if the best rule tree has already been found. If the hash table entry does not exist, the transform is expanded using $\mathcal{R}_{CT}$ and $\mathcal{R}_{CT}^T$, with a total of 4 different decompositions.

Expansion 1

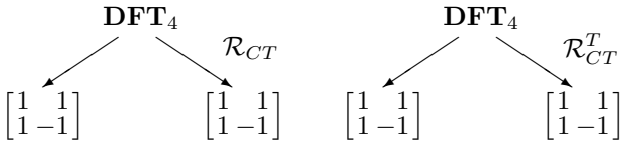


The only transforms at the lower level are $\mathbf{DFT}_2$ and $\mathbf{DFT}_4$. DP then continues expanding the lower level transforms, assuming they are not in the hash table, starting with the $\mathbf{DFT}_4$. Two expansions are possible in this case.

Expansion 2



The next transform in Expansion 1 is the $\mathbf{DFT}_2$. Since there is only one way to expand it, namely using the terminal rule (2.18), that yields also the best found rule tree and is stored in the hash table. DP further expands transforms in Expansion 2. Since there is only the $\mathbf{DFT}_2$, and since the best rule tree is already found, it is retrieved from the hash table and substituted into Expansion 2 trees.



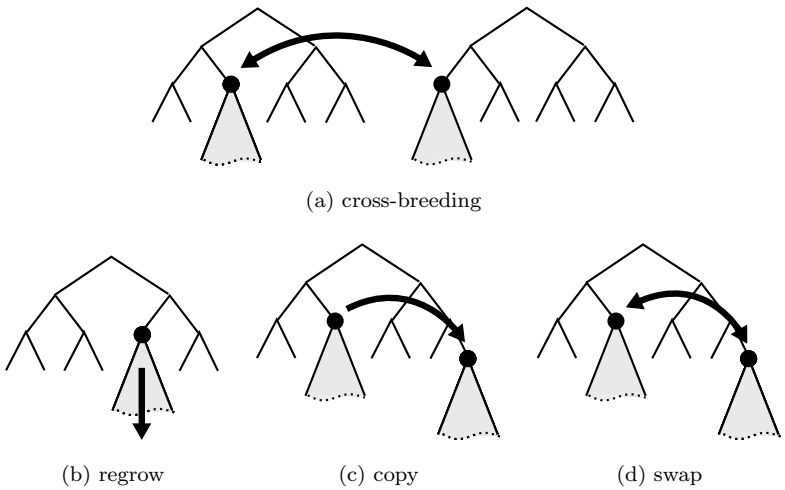
Both trees are evaluated and the best one is stored in the hash table. Assume that the right tree is the best. The DP climbs one level up, substitutes the best found tree for the $\mathbf{DFT}_4$ into all trees in Expansion 1, and measures all four to find the best one for the $\mathbf{DFT}_8$.

This simple example shows the DP technique applied to the rule tree framework. The DP first expands rule trees all the way down to the leaf node level and then recursively searches for the best implementation and substitutes it at the next higher level.

Even though the principle of optimality holds when optimizing algorithms for arithmetic cost, it does not generally hold for optimizing run times. The reason is that the data flow can be adversely affected by plugging in the algorithm for a smaller transform into larger transform rule trees. Slower running sub-trees can be a better match for the data flow of the whole algorithm and decrease the overall run time. Still, the DP approach gives satisfying results in most cases faster than other optimization methods. For that reason, most of the results we present in Chapter 7 are obtained using DP.

Evolutionary search. SPIRAL also provides an alternative to DP for a large number of algorithms. This optimization method uses principles of genetic algorithms to traverse the space of algorithms by evolving a population of rule trees using manipulations that mimic the theory of evolution by selection, mutation, and cross-breeding [49].

The evolutionary search called STEER first selects a population of n random rule trees. The population is then grown to a larger size by cross-breeding and by mutation of randomly selected rule trees. *Cross-breeding* is implemented by swapping a sub-tree from a selected node (transform) with the same node from a different tree in the population, as shown in Figure 2.4. In the same figure, we can graphically see that the mutation is performed using different methods. A sub-tree for a selected node can be discarded and then *regrown* by expanding the node into a different rule tree, or it can be *copied* to the same transform at a different location in the rule tree. Similar to cross-breeding, sub-trees can be *swapped* between the nodes inside the same rule tree. After the population is grown to a number of rule trees $N > n$, they are evaluated and only n best trees are selected for the next generation. The process is repeated through as many generations as necessary to reach the state where new generations do not improve the best individual rule tree. More information on this technique and results for the WHT transform can be found in [50].



(a) cross-breeding

(b) regrow (c) copy (d) swap

Figure 2.4: Rule tree manipulation for the evolutionary search: (a) cross-breeding; (b)–(d) three types of mutations: regrow, copy, and swap.[†]

2.5.2 Practical considerations

STEER, the evolutionary search method in SPIRAL converges to the solution much slower than dynamic programming, especially for larger transform sizes. However, the genetic algorithm approach produces better results when compared to DP in the situation when the principle of optimality is severely violated. For that reason, evolutionary search is usually used in conjunction with DP to efficiently find good code by employing it in the DP scheme on lower levels of the rule trees (smaller size nodes). The sub-optimality of DP for generating fast code can also be alleviated by modifying the approach to save k best rule trees instead of just one for all intermediate nodes. The hope is that one of those k best rule trees fits better the whole algorithm than the best found one [51, 52]. However, increasing the number of saved rule trees dramatically increases the cost of the search. SPIRAL provides DP search for up to four best rule trees.

We also mention that the search space increases with the range of possible degrees of loop unrolling. The sizes of nodes for which the straightline code is generated are typically between 2^2 and up to a maximum of 2^6 since most compilers cannot handle more than $2^6 \cdot 2^6 = 2^{12}$ lines of straightline code. In addition to searching over all possible rule trees, DP, for example, evaluates all rule trees for all unrolling settings at each level of the recursion. If, for example, the size of unrolled nodes is a two power set between 2^3 and 2^5 then, in addition to Expansion 2 on page 29, which includes only two rule trees, each rule tree is evaluated for all unrolling options — eight evaluations in total. For larger transforms this can considerably affect the search time.

2.6 FIR Filters and the DWT in SPIRAL: Challenges

The SPIRAL system provides an efficient and extensible framework for automatic generation of the space of algorithms as concise mathematical formulas. It further provides a full-fledged compiler for

[†] *source*: Püschel et al. [13]

translation of formulas into C or Fortran code. The compiler includes code optimization techniques such as loop unrolling, code simplification, and scheduling. Two main modules can automatically produce code for any algorithm generated by a set of rules included in the rule library and for any of the available implementation options. This two-module system is repeatedly invoked by the optimization module to generate and evaluate algorithms and search for the optimized implementation on the target platform. The SPIRAL system, thus, finds a high performance implementation for a given DSP transform by searching a comprehensive space of fast alternatives. Since the evaluation is run on the platform of interest, the implementation is optimized for that platform only. The same code usually performs below par on other platforms and the search process has to be repeated. However, search is performed precisely once per platform and per transform.

Our goal in this thesis is to use the current SPIRAL code generator to enable automatic generation, implementation, and tuning of fast algorithms for FIR filtering and the discrete wavelet transforms (DWTs). Since SPIRAL specializes in linear DSP transforms, we formulate FIR filters and the DWT as transforms in SPIRAL, develop a new framework to formulate existing algorithms using the rule formalism, and implement the framework to perform the optimizations using SPIRAL's modules.

Rules and the generated rule trees for FIR filters and the DWT have a very different structure from the ones that SPIRAL can currently produce. New rules for these transforms require new constructs that lead to code with a specific structure uncommon to trigonometric transforms. The new constructs and primitives also require new templates in the formula translator to map the formulas into code. Furthermore, a different approach to build rule trees implies different search and hashing strategies that have to be incorporated in the search engine. Hence, to integrate the new framework, the structure of SPIRAL has to be modified at almost all levels. This is a challenging task; however, the reward is a flexible framework that enables SPIRAL to automatically generate numerous implementations for FIR filters and the DWTs on a variety of platforms in search for the fastest code.

The first task is to identify in the literature the algorithms and the methods that are used for efficient computation of FIR filters and DWTs. Since wavelet transforms are essentially recursive filter banks, the core operation in all algorithms is the convolution of discrete sequences. Convolutions are typically represented either in terms of convolution sums or as polynomial multiplication.

In the literature, algorithms are introduced using these representations; however, they are not the most suitable models for representation on the computer, especially from the perspective of automatic generation. To be able to automatically and efficiently generate fast algorithms from a comprehensive space of good candidates, the algorithms have to be represented using a concise and structured mathematical language that enables parameterization, easy manipulation, and interleaving of a range of different methods. Such language should enable machine generation of a large number of options for testing and evaluation without any input from humans. The machine optimized implementations are based on algorithms hardly ever tried by human programmers because of their unintuitive and highly involved structure.

One of our goals is to capture the core structure of many available algorithms using a constructive mathematical language based on rules. For this purpose we achieve the following:

- Develop a new theoretical framework using a set of carefully defined constructs, operators, and definitions;
- Use the developed mathematical language to define FIR filters and DWTs as finite transforms and provide practical definitions of filtering and wavelet operators;
- Capture all major filtering and wavelet algorithms in a set of rules that fit SPIRAL's mathematical framework. The rules have to be concise and have to construct all required algorithms and their variations by simple recursive application leading to rule trees;
- Implement the rules in SPIRAL and verify them for correctness;
- Ensure that all rules and involved constructs translate into, not only correct code, but also the desired efficient programming structures.
- Enable the SPIRAL's search engine to efficiently store the best found algorithms and optimize the search taking into consideration special properties of the algorithms.

We start by defining FIR filters and discrete wavelet transforms and reviewing selected algorithms for fast filtering and wavelet expansion techniques. In Chapter 4 we formally introduce all the required mathematical definitions and properties required to develop the framework for FIR filters and DWTs in Chapters 5 and 6, respectively.

CHAPTER 3

DISCRETE FIR FILTERS AND WAVELETS AND THEIR ALGORITHMS

In this chapter, we review discrete FIR filters and discrete wavelet transforms. We also overview well-known algorithms for their computation. In the standard literature, these algorithms are usually provided in the form of summations, accompanied by supporting verbal descriptions, illustrations, graphs, and diagrams. These representations are adopted and widely used in the signal processing community. However, the representations will need to be adapted to capture the recursive structure of most algorithms, the order of operations, and data flow patterns to enable automated algorithm and code generation. We start with basic definitions and proceed with providing the necessary background for understanding the concepts of fast algorithms for digital filtering and wavelet transform expansions.

3.1 FIR filters

The theory of discrete-time signals and systems provides the mathematical foundation for the implementation of physical systems in hardware. Discrete-time signals are mathematically represented as sequences of numbers, which we denote as $\{x_k\}_{k\in\mathbb{Z}}$. These sequences are generally of infinite duration. However, in many practical cases they have a finite support length with $x_k = 0$ for $k < a$ and $k > b$, which we then denote as $\{x_k\}_a^b$. *Discrete linear filters* are frequently used to process discrete-time signals. The linearity of the filtering process allows for the unique description of filters by their impulse response, i.e., the response of the filter to the unit impulse excitation $\delta_k = \{1\}_k^k$. The impulse response of a filter is, therefore, also a sequence $\{h_k\}$ that can be of either finite or infinite support. When the impulse response is finite, a filter is appropriately called a *finite impulse response* (FIR) filter. Otherwise, it is called an *infinite impulse response* (IIR) filter.

3.1.1 Signal representations

Sequences of numbers are used to mathematically describe discrete-time signals. However, it is often useful to represent and analyze discrete-time signals and discrete linear filters in the z-transform

domain. The z-transform of a discrete-time signal $\{x_k\}_a^b$ is defined as the polynomial

$$x(z) = \sum_{i=a}^{b} x_i z^{-i}. \tag{3.1}$$

Since the limits a and b of the summation can be both positive and negative, the z-transform is a mapping from the sequences of numbers $\{x_k\}_a^b$ to the ring of Laurent polynomials. The degree $\deg(x)$ of a Laurent polynomial $x(z)$ is given by

$$\deg(x) = |b - a|. \tag{3.2}$$

The z-transform exists for every finite length sequence and, hence, every such sequence $\{x_k\}$ can be uniquely represented by the polynomial $x(z)$ given by (3.1). Sequences of finite length p can also be seen as vectors in $\mathbb{C}^{p\times 1}$. These different representations of discrete time signals are important tools in representing computational algorithms mathematically. We now summarize these definitions.

For a discrete-time signal $\{x_k\}$ of finite support, where the index variable k belongs to the interval $k \in [-r, l]$, the following representations are equivalent:

$$\begin{aligned} \textbf{sequence} &: \{x_k\}_{-r}^{l} \\ \textbf{polynomial} &: x(z) = x_l z^{-l} + \cdots + x_0 + \cdots + x_{-r} z^{r} \\ \textbf{column vector} &: \mathbf{x} = (x_{-r}, \ldots, x_0, \ldots, x_l)^T. \end{aligned} \tag{3.3}$$

We will use all three representations interchangeably. We choose the appropriate format based on conciseness, clarity, and easiness of manipulation.

3.1.2 Convolution

Linear filtering is inextricably tied to the convolution operation. The discrete filtering of a sequence $\{x_k\}$ with a filter described by its impulse response $\{h_k\}$ is the discrete linear convolution of these two sequences. The *linear convolution*, or simply, convolution of two sequences is denoted as

$$\{y_k\} = \{h_k\} * \{x_k\} \tag{3.4}$$

and is commonly described by the *convolution sum*. The result of a convolution is a sequence $\{y_k\}$ with elements computed as

$$y_k = \sum_{i=a}^{b} h_i \; x_{k-i} = \sum_{i=n-a}^{k-b} h_{k-i} \; x_k. \tag{3.5}$$

We refer to the length of the filter impulse response support $p = a - b + 1$ simply as the *filter length*. In the case where the input support length n and filter length l are finite, so is the support of the output with the length equal to $n + p - 1$.

Using the polynomial representation of signals, the convolution operation becomes simple polynomial multiplication in $\mathbb{C}[z]$.

$$\{y_n\} = \{h_n\} * \{x_n\} \xrightarrow{z-\text{transform}} y(z) = h(z) \cdot x(z) \tag{3.6}$$

This identity arises from the fact that the z-transform is an isomorphism of the algebra of finite discrete-time sequences and the algebra of Laurent polynomials, with the convolution as the multiplication operation in the first algebra.

We will often use this relation to work with filters, filter banks, and wavelets. The main advantages of the polynomial notation are its conciseness and its mathematical clarity, since many computational algorithms are derived using properties of polynomial algebras.

When working with finite length signals, it is useful to define other types of convolutions that arise from different signal extensions at the boundaries [53]. The assumption on the nature of signals implied by the signal extension translate to residue class (quotient) Laurent polynomial algebras $\mathbb{C}[z]/p(z)$ using the z-transform as the isomorphism. We will discuss some important examples in the next section.

Before we continue, we briefly mention the third way to represent filters and convolutions. Since filters are discrete linear operators, they can be represented using matrices. The matrix notation will be particularly useful for generating computer implementations in later chapters, since it best exhibits structural information important for designing efficient implementations.

Let the sequence $\{x_k\}_{-r}^{l}$ be represented by the column vector $\mathbf{x}$. We observe that (3.5) can be seen as as a matrix-vector product

$$\mathbf{y} = \begin{bmatrix} h_{-r} & & & & & \\ \vdots & \ddots & & & & \\ h_l & \cdots & h_{-r} & & & \\ & \ddots & & \ddots & & \\ & & h_l & \cdots & h_{-r} & \\ & & & \ddots & \vdots & \\ & & & & h_l & \end{bmatrix} \cdot \mathbf{x} \tag{3.7}$$

If the sequences are finite, with the input length n and the filter length p, then the dimensions of the convolution matrix are $n + p - 1 \times n$. For infinite sequences, the above matrix is also infinite. In the finite case, matrices are not square and the multiplication is not commutative. However, if we define convolutions as polynomial multiplications in $\mathbb{C}[z]/p(z)$, then the corresponding matrices are square and the isomorphism of algebras is provided using the *regular representations* [54].

3.1.3 Generalized convolutions as multiplications in polynomial algebras

Correspondence between convolutions and polynomial algebras is important. It allows us to generalize the concept of convolution and derive more efficient algorithms by decomposing longer convolutions into generalized shorter convolutions using the Chinese remainder theorem. Residual class Laurent polynomial algebras $\mathbb{C}[z]/p(z)$ are especially important. If $p(z)$ has degree N, and we choose the standard basis $b = \{1, z, \ldots, z^{N-1}\}$, then the z-transform provides an isomorphism from $\mathbb{C}[z]/p(z)$ to discrete-time signal algebras with respect to some useful convolutions such as circular convolution. In general, given a polynomial $p(z)$ of degree N, the polynomial product in $\mathbb{C}[z]/p(z)$ corresponds to the linear convolution of discrete signals of length N that are extended outside of the boundary using a corresponding linear extension. This linear extension is inherently determined by the choice of the generator polynomial $p(z)$ and the basis b of $\mathbb{C}[z]/p(z)$. For different choices of bases b, one must find the appropriate isomorphism, not necessarily provided by the z-transform. For more details the reader is referred to [53] and [55]. Important examples include circular convolutions,

symmetric convolutions, and short convolutions modulo cyclotomic polynomials.

Polynomial algebras $\mathbb{C}[z]/p(z)$ can be represented using the matrix notation since the polynomial product is a linear operation. Formally, the isomorphism of algebras is provided by the regular representation $\phi : \mathbb{C}[z]/p(z) \mapsto \mathbb{C}^{n\times n}$ w.r.t. basis b. If we choose the standard basis $b = \{1, z, \ldots, z^{N-1}\}$, then the mapping ϕ becomes

$$h(z) \mapsto M(\mathbf{h}) = \sum_{i=a}^{b-1} h_i * \mathcal{C}_f^i \tag{3.8}$$

where $h(z) = \sum_{n=a}^{b} h_n z^{-n}$, and $\mathcal{C}_f$ is the companion matrix of $p(z)$.

Circular convolution. Consider polynomial multiplication modulo $(1 - z^{-N})$, where $N \in \mathbb{N}$, i.e.,

$$y(z) = (h(z) \cdot x(z)) \bmod (1 - z^{-N}),$$

i.e., the multiplication in $\mathbb{C}[z]/(1 - z^{-N})$. If we choose the standard basis $b = \{1, z, \ldots, z^{N-1}\}$ then the z-transform is an isomorphism of the algebra and the corresponding algebra of discrete-time sequences with respect to the *circular convolution*, also referred to as the *cyclic convolution*.

It follows directly that the circular convolution on N points is defined as

$$y_n = h_n \circledast x_n = \sum_{m=0}^{N-1} h_m \, x_{(n-m) \bmod N}. \tag{3.9}$$

To differentiate between the two, the convolution sum in (3.5) is also called the *linear convolution*.

To obtain the matrix representation of the circular convolution, we use (3.8). The companion matrix $\mathcal{C}_s$ for $s(z) = 1 - z^{-N}$

$$\mathcal{C}_s = S_N = \begin{bmatrix} 0 & \ldots & 0 & 0 & 1 \\ 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & 1 & 0 \end{bmatrix} \tag{3.10}$$

is the circular shift matrix and generates the algebra of circulants since

$$C(\mathbf{h}) = \sum_{i=a}^{b} h_i \, S_N^i, \tag{3.11}$$

and since shift matrices commute with respect to multiplication.

The matrix representation for the circular convolution is, therefore, the matrix-vector product

$$\mathbf{y} = C(\mathbf{h}) \cdot \mathbf{x} = \begin{bmatrix} h_0 & h_{n-1} & h_{n-2} & \ldots & h_1 \\ h_1 & h_0 & h_{n-1} & \ldots & h_2 \\ h_2 & h_1 & h_0 & \ldots & h_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{n-1} & h_{n-2} & h_{n-3} & \ldots & h_0 \end{bmatrix} \cdot \mathbf{x} \tag{3.12}$$

Hence, the algebra of circulant matrices $C(\mathbf{h})$ is isomorphic to $\mathbb{C}[z]/(1-z^{-N})$ w.r.t. the standard basis $b = \{1, z, \ldots, z^{N-1}\}$ and, in turn, to the algebra of discrete-time signals with respect to circular convolution.

Symmetric convolutions. Symmetric FIR filters, i.e., filters with symmetric impulse response, are frequently used in digital signal processing because they have linear phase and, therefore, do not introduce phase distortions. An interesting case arises when the filtered signal is also a symmetric sequence, or more often when the signal is symmetrically extended at the boundaries. In such a case, it turns out that one can formally perform efficient *symmetric convolution* similar to circular convolution of cyclically extended signals.

There are sixteen different types of symmetries: whole point or half point, symmetric or anti-symmetric extensions at each boundary of the fundamental sequence. The symmetries are grouped into four classes where the convolution can be performed only within a certain class. This gives us 64 types of convolutions; only 40 are truly different [53].

Püschel and Moura showed in [55] that the 16 different signal symmetries arise from polynomial products in $\mathbb{C}[z]/p(z)$, where $p(z)$ and the basis b are chosen to be one of four types of Chebyshev polynomials $C \in \{T, U, V, W\}$. Depending on the choice of $p(z)$ and the basis b, there are 16 different options, each determining the type of extension at right and left boundaries, respectively. For example, symmetric convolution of signals with half-point symmetry at both ends is the polynomial multiplication in $\mathbb{C}[z]/2(z-1)U_{n-1}$ with $b = \{1, 2z-1, V_2, \ldots, V_n\}$ as the basis.

As before, matrix representations of symmetric convolutions are obtained as a linear combination of matrices that are images of basis polynomials $b = \{C_0, C_1, \ldots, C_{n-1}\}$. In the case of standard basis this resulted in companion matrices. Here, the action of basis elements has to be computed using properties of Chebyshev polynomials.

To illustrate the concept, we consider convolution of an odd symmetric (whole-point symmetric) FIR filter $\{h_n\}$ and the input sequence $\{x_n\}$ of length N that is half-point symmetrically extended on the left and half-point anti-symmetric on the right boundary. Using the results in [55], we pick the T basis for $\{h_n\}$ and the V basis for $\{x_n\}$. We then have

$$\mathbf{h} = \sum_{i=0}^{L-1} h_i T_i, \quad \mathbf{x} = \sum_{i=0}^{N-1} x_i V_i. \tag{3.13}$$

Furthermore, the result of this type of convolution is a half-point symmetric and half-point anti-symmetric sequence just as the input. So it is also represented using the V basis $\mathbf{y} = \sum_{i=0}^{N-1} y_i V_i$. Using the properties $T_0 = 1, T_1 = z, \text{and} C_k = 2zC_{k-1} - C_{k-2}$, we can compute images of all basis elements T_i with respect to the V basis.

$$\begin{aligned} T_0 \cdot V_i &= V_i \\ T_1 \cdot V_i &= x \cdot V_i = \frac{1}{2}(V_{i-1} + V_{i+1}) \\ &\vdots \\ T_k \cdot V_i &= \frac{1}{2}(V_{i-k} + V_{i+k}) \end{aligned} \tag{3.14}$$

The representing matrix $M(\mathbf{h})$ for the product $h(T)x(V)$ is symmetric. The upper left and the lower right $(L-1) \times (L-1)$ corners are determined by the boundary conditions, in turn determined

by the choice of the basis and the algebra generating polynomial. The upper triangular matrix is determined by the property $V_{-1} = V_0$ inherent to the V basis, whereas the lower triangular part is governed by the condition $V_n = -V_{n-1}$ obtained from the relation between T_n and the V basis.

$$T_n = T_n \cdot V_0 = \frac{1}{2}(V_{n-1} + V_n) = 0.$$

For example, if $L = 2$, the representing matrix is

$$M(\mathbf{h}) = \frac{1}{2}\begin{bmatrix} 2h_0 + h_1 & h_1 + h_2 & h_2 & & & & \\ h_1 + h_2 & 2h_0 & h_1 & \ddots & & & \\ & h_2 & h_1 & \ddots & & \ddots & \\ & & \ddots & & \ddots & h_1 & h_2 \\ & & & \ddots & h_1 & 2h_0 & h_1 - h_2 \\ & & & & h_2 & h_1 - h_2 & 2h_0 - h_1 \end{bmatrix}. \tag{3.15}$$

3.1.4 Embedding one convolution type into another

We have seen that many types of convolutions can be computed as products in polynomial algebras. Choosing the appropriate convolution will depend on the properties of convolved signals and their extensions. To obtain efficient computational algorithms, it is often useful to compute one type of convolution by means of another.

Given the result of a linear convolution of sequences $\{h_n\}$ and $\{x_n\}$, which we know is the product of polynomials $h(z)$ and $x(z)$, we can always obtain other convolutions arising from residual class polynomial algebras $\mathbb{C}[z]/p(z)$ by simply reducing the result modulo $p(z)$, i.e,

$$c(z) = y(z) \bmod p(z), \quad y(z) = h(z) \cdot x(z). \tag{3.16}$$

Conversely, it is possible to compute the linear convolution of $\{h\}$ and $\{x\}$ through other convolution types that arise as polynomial products in $\mathbb{C}[z]/p(z)$ if and only if the following condition is satisfied

$$\deg(h) + \deg(x) < \deg(p) \tag{3.17}$$

since in that case $(h(z) \cdot x(z)) \bmod p(z) = h(z) \cdot x(z)$. The proof is very easy and we omit it here.

Finally, we observe that, by combining the above conversions, we can perform any convolution type through any other convolution. Assume we want to compute the product $h(z) \cdot x(z)$ in $\mathbb{C}[z]/m_1(z)$ by using the product in $\mathbb{C}[z]/m_2(z)$. If $\deg(h) + \deg(x) < \deg(m_2)$ then we have

$$(h(z) \cdot x(z)) \bmod m_1(z) = ((h(z) \cdot x(z)) \bmod m_2(z)) \bmod m_1(z). \tag{3.18}$$

This means we can compute the convolution in the first algebra and then perform the reduction modulo $m_2(z)$ to obtain the desired product.

As an illustration, the linear convolution is equal to the circular convolution if the input signal is finite with length N and is extended (padded) by $L - 1$ zeros, where L is the filter length.

$$h_n * x_n = h_n \circledast \hat{x}_n, \qquad \hat{x} = \begin{cases} x_n, & 0 \le n < N \\ 0, & N \le n < N + L - 1 \end{cases} \tag{3.19}$$

which is the same as saying that the product $h(z) \cdot x(z)$ is equal to the product $(h(z) \cdot x(z)) \bmod (1 - z^{-M})$ when $M = N + L - 1$.

3.1.5 Reducing long convolutions to multiple short convolutions

Reducing long convolutions or long polynomial products into a large number of shorter convolutions can sometimes have computational advantages. We break down these cases into two classes:

1. The length of the input sequence is often much larger than the filter length $N \gg L$. In such cases the representing matrix is sparse with only a narrow strip of non-zero elements along the diagonal and near diagonal locations. It is advantageous to break down this matrix into smaller, more dense matrices to benefit from transform domain methods or from the locality of the computations. The long convolution is computed in segments, which are then appropriately combined to obtain the correct result. These methods are referred to as the *block convolution* methods, which we discuss next.

2. If the length of the input and the filter are similar, for example, by performing block convolution methods, then it is still possible to reduce the convolution to shorter convolutions by mapping the convolutions to higher dimensions. The mapping can be done for linear and cyclic convolutions where the latter is known as the Agarwal-Cooley algorithm [56]. The advantage of these methods stem from many efficient short convolution algorithms based on polynomial multiplications modulo short cyclotomic polynomials. However, the mapping to higher dimensions is the basis for the Karatsuba algorithm [57] that reduces the arithmetic cost by exploiting redundancies of the representing Toeplitz matrix.

We discuss these two classes of algorithms in the following subsections.

Block convolutions

If the input sequence is very long when compared to the filter length ($N \gg k$), it is not efficient to use transform-based methods since the size of the transform is equal to $N + k - 1$, which is very large. The solution is to block the input signal into segments, compute the convolution separately on each segment, and then to combine the results to obtain the final output. There are two different blocking strategies known as overlap-add and overlap-save.

Overlap-add. One way to perform long convolution using a number of shorter ones is to segment the input into independent blocks of length B,

$$\{x_n\}_0^N = \sum_{s=0}^{\lceil N/B \rceil} \{x_{n+sB}\}_0^B, \tag{3.20}$$

and compute the final output as the overlapped sum of the convolutions of the filter with each segment

$$\{y_n\}_0^{N+L-1} = \sum_{s=0}^{\lceil N/b \rceil} \{y_{n+sB}\}_0^{B+L-1}, \quad , \{y_{n+sB}\}_0^{B+L-1} = \{x_{n+sB}\}_0^B * \{h_n\}_0^L. \tag{3.21}$$

The overlap between the neighboring sequences at the output is $L - 1$ points long and they need to be added together to obtain the correct result. This blocking procedure is referred to as the

overlap-add algorithm [38].

Overlap-save. Another alternative is the *overlap-save* algorithm. In this case, the input segments of length B are extended to length $B+L-1$ and overlapped on $L-1$ points

$$\{x_n + sB\}_0^{B+L-1}, \quad 0 \le s \le \lceil N/b \rceil. \tag{3.22}$$

We can perform circular convolution for each segment as

$$\{y_n + sB\}_0^{B+L-1} = \{x_n + sB\}_0^{B+L-1} \circledast \{h_n\}_0^L \tag{3.23}$$

and retain only the last B values since they represent the correct output values because of the relation (3.19). What is left is the exact segment of the output result. These segments are concatenated to obtain the final output. We could also perform linear convolutions on segments

$$\{y_n + sB\}_0^{B+2L-2} = \{x_n + sB\}_0^{B+L-1} * \{h_n\}_0^L \tag{3.24}$$

and discard the first $L-1$ and the last $L-1$ values which would lead to the same result.

The difference from the overlap-add method is that the results of the convolutions on segments do not have to be overlapped and added at the expense of computing the convolutions on $L-1$ more points in each input segment. For more details see, for example, [38].

We mention here that the overlap-add and overlap-save methods are inextricably tied to the circular convolution. The circular convolution is typically performed on the segments because the fast transform methods to compute it benefit from the reduced size obtained by blocking. However, we shall see that it is important to consider block convolution methods independently of the short convolution methods because of their important structure. This is the reason why we offer two interpretations of these two methods that use the linear convolution and the circular convolution found in the literature.

Nesting of convolutions using multidimensional polynomial products

Reduction of long convolutions to shorter convolutions can be obtained by converting one-dimensional polynomial products into multi-dimensional products of shorter polynomials. We start with the nesting technique for linear convolutions.

Let $h(z)$ and $x(z)$ be polynomials in $\mathbb{C}[z]$, and let the product $y(z) = h(z) \cdot x(z)$ represent the linear convolution of sequences $\{x_k\}$ and $\{h_k\}$ both of length N. If N is composite $N = N_1 \cdot N_2$, then we can rewrite $h(z)$ and $x(z)$ as

$$\begin{aligned} x(z) = x(z_1, z_2) &= \sum_{n_1=0}^{N_1-1} x_{n_1}(z_2) z_1^{n_1} \\ h(z) = h(z_1, z_2) &= \sum_{n_1=0}^{N_1-1} h_{n_1}(z_2) z_1^{n_1} \end{aligned}, \tag{3.25}$$

where $z_1 = z^{N_2}, z_2 = z$, and

$$\begin{aligned} x_{n_1}(z_2) &= \sum_{n_2=0}^{N_2-1} x_{N_2 n_1 + n_2} z_2^{n_2} \\ h_{n_1}(z_2) &= \sum_{n_2=0}^{N_2-1} h_{N_2 n_1 + n_2} z_2^{n_2} \end{aligned}. \tag{3.26}$$

In other words, the sub-polynomials of degree N_1 are coefficients of a polynomial of degree N_2 which results in a two-dimensional polynomial. The result of the multiplication is a polynomial of degree $2N_1 - 1$ with coefficients being polynomials of degree $2N_2 - 1$. In other words

$$y(z_1, z_2) = h(z_1, z_2) \cdot x(z_1, z_2) = \sum_{n=0}^{2N_2-2} \sum_{m=0}^{N_1-1} h_m(z_2) \cdot x_{n-m}(z_2) z_1^n. \tag{3.27}$$

The goal of this procedure is to make use of the short convolution algorithms and apply them to products of N_1 and N_2 degree polynomials [58]. However, the structure of these algorithms is weaker than the similar structure of algorithms for the circular convolution that we discuss later. Further, the deficiencies of the short convolution algorithms are that they favor fewer multiplications at the expense of larger number of additions, which is not suitable for implementation on modern computer platforms. Therefore, the above described nesting procedure is not widely used, so we either rely on techniques that nest circular convolutions because they provide a connection between multi-dimensional filtering methods and important multi-dimensional FFT algorithms, or we revert to simpler nesting techniques for linear convolutions, discussed next, since they lead to divide-and-conquer algorithms for fast polynomial multiplication.

We slightly change the mapping of indices in (3.26). Again we require $N = N_1 \cdot N_2$ and map indices $n \to (n_1, n_2)$ as $n = N_2 n_1 + n_2$. We obtain 2-D polynomials

$$\begin{aligned} x(z) = x(z_1, z_2) &= \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x_{N_2 n_1 + n_2} z_1^{n_1} z_2^{n_2} \\ h(z) = h(z_1, z_2) &= \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} h_{N_2 n_1 + n_2} z_1^{n_1} z_2^{n_2} \end{aligned}, \tag{3.28}$$

where $z_1 = z^{N_2}, z_2 = z$. The result can be obtained as

$$y(z_1, z_2) = h(z_1, z_2) \cdot x(z_1, z_2) = \sum_{n=0}^{2N_2-2} \sum_{m=0}^{N_1-1} h_m(z_1) \cdot x_{n-m}(z_1) z_2^n. \tag{3.29}$$

This procedure describes polynomial multiplication performed by the polyphase components and can be seen as filtering in the polyphase channels. For example, if N is factored as $N = N/2 \cdot 2$ then we have a 2-channel case. The $x(z)$ and $h(z)$ polynomials are downsampled into their even and odd components. The polynomial multiplication is performed as the following polynomial matrix-vector product:

$$\begin{bmatrix} y_e(z) \\ y_o(z) \end{bmatrix} = \begin{bmatrix} h_e(z) & z \cdot h_o(z) \\ h_o(z) & h_e(z) \end{bmatrix} \cdot \begin{bmatrix} x_e(z) \\ x_o(z) \end{bmatrix} \tag{3.30}$$

Here the index denotes either even (e) or odd (o) component. Of course, the mapping to two-dimensional convolutions can be recursively generalized to multi-dimensional convolutions as needed.

Similar nesting procedures can be derived for circular convolutions. The most important case arises when the size of the sequences can be decomposed into a product of relatively prime factors $N = N_1 N_2, (N_1, N_2) = 1$. This method was developed and analyzed by Agarwal and Cooley [56]. Since the factors are relatively prime, we can write $an_1 + bn_2 = 1$. Using the Chinese remainder theorem, we can map the indices as

$$n = e_1 n_1 + e_2 n_2 \bmod N \tag{3.31}$$

where

$$n_1 \equiv n \bmod N_1 \qquad e_1 \equiv a \bmod N \tag{3.32}$$

$$n_2 \equiv n \bmod N_2 \qquad e_2 \equiv b \bmod N \tag{3.33}$$

Therefore, we can split the polynomials $h(z)$ and $x(z)$ as

$$h_{n_1}(z) = \sum_{n_2=0}^{N_2-1} h_{e_1 n_1 + e_2 n_2 \bmod N} z^{n_2} \tag{3.34}$$

$$x_{n_1}(z) = \sum_{n_2=0}^{N_2-1} x_{e_1 n_1 + e_2 n_2 \bmod N} z^{n_2}. \tag{3.35}$$

The circular convolution $h(z)x(z) \bmod (z^N - 1)$ is then performed as

$$\hat{y}(z) = \sum_{n=0}^{N_1-1} \sum_{m=0}^{N_2-1} \left(h_m(z) \cdot x_{n-m}(z) \bmod (1 - z^{N_2}) \right) z^{N_1 n} \bmod (1 - z^{N_1}), \tag{3.36}$$

where

$$\hat{y}(z) = \sum_{n=0}^{N_1-1} \sum_{m=0}^{N_2-1} y_{e_1 n_1 + e_2 n_2 \bmod N} z^{nm}. \tag{3.37}$$

This is essentially a two dimensional circular convolution, or one-dimensional circular convolution of size N_1 with the coefficients being circular convolutions of size N_2 [56, 59, 60]. Going from 1-D to 2-D circular convolution allows application of 2-D transform-domain algorithms.

3.1.6 Divide-and-conquer methods

The multi-dimensional mapping algorithms rely on efficient computation of short convolutions, either by taking advantage of short convolution algorithms that reduce the number of multiplications at the expense of increased number of additions, or by using the multi-dimensional transform-based methods.

In this section, we discuss another class of algorithms usually referred to as divide-and-conquer algorithms. They utilize the redundancies associated with nested filter computations to reduce both the required multiplications and the additions. The divide-and-conquer methods rely on the principle of fast polynomial multiplication, which is better known as the Karatsuba method for polynomials in the scientific computing community. The Karatsuba algorithm was initially developed for the multiplication of large integers [61]. Therefore, we shall refer to divide-and-conquer methods also as Karatsuba methods.

In its original form, the Karatsuba method divides polynomials into two parts by separating the even and the odd coefficients. We are interested in computing the product $y(z) = h(z)x(z)$, where $\deg(h) = L$, $\deg(x) = N$ and $2 \,|\, L$, $2 \,|\, N$. The first step is to split all polynomials into their even and odd factors as

$$\begin{aligned} h_e(z^2) &= \frac{1}{2}\left(h(z) + h(-z)\right) & x_e(z^2) &= \frac{1}{2}\left(x(z) + x(-z)\right) \\ h_o(z^2) &= \frac{1}{2z}\left(h(z) - h(-z)\right) & x_o(z^2) &= \frac{1}{2z}\left(x(z) - x(-z)\right) \end{aligned} . \tag{3.38}$$

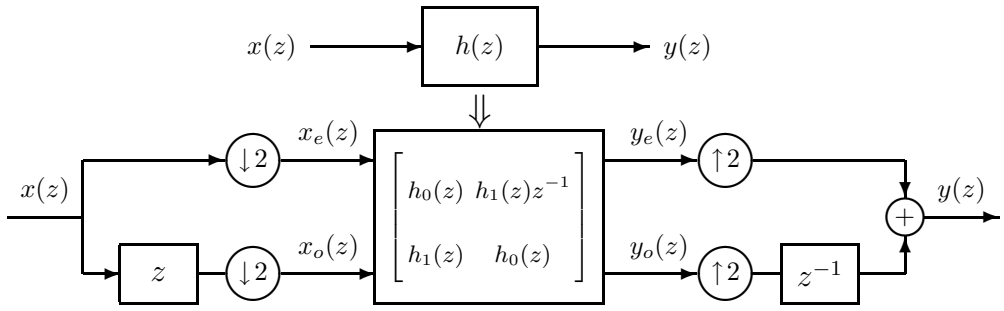


Figure 3.1: Filtering interpreted as polyphase filtering with two channels.

If we perform the multiplication as

$$y_e(z^2) + y_o(z^2)z^{-1} = (h_e(z^2) + h_o(z^2)z^{-1})(x_e(z^2) + x_o(z^2)z^{-1})$$

and group the matching terms, we obtain

$$\begin{aligned} y_e(z) &= h_e(z)x_e(z) + h_o(z)x_o(z)z^{-1} \\ y_o(z) &= h_e(z)x_o(z) + h_o(z)x_e(z) \end{aligned}, \tag{3.39}$$

where $y(z) = y_e(z^2) + y_o(z^2)z^{-1}$. This implementation requires four multiplications of polynomials of degree $N/2$ and two additions. The decomposition can be schematically seen as what is called a polyphase decomposition of the original filter, shown in Figure 3.1.

The conquer step is obtained by observing that the cross product $y_o(z) = h_e(z)x_o(z)+h_o(z)x_e(z)$ can be written as

$$y_o(z) = (h_e(z) + h_o(z))\,(x_e(z) + x_o(z)) - h_e(z)x_e(z) - h_o(z)x_o(z). \tag{3.40}$$

Since both $h_e(z)x_e(z)$ and $h_o(z)x_o(z)$ are already computed for $y_e(z)$ in (3.39) and since $h_e(z)+h_o(z)$ can also be precomputed, there is only one additional multiplication, a total of three as opposed to four multiplications (convolutions) required in (3.39). The diagram of this decomposition is shown in Figure 3.2.

Careful analysis of the method reveals that the procedure performs filtering through a multi-rate filter bank with three channels [62]. The new algorithm requires only three multiplications of polynomials of degree $L/2$ with some extra additions. However, the total cost is reduced for both multiplications and additions. This is easy to see since the original problem requires L multiplications and $L-1$ additions per output point, whereas the above decomposition needs only $\frac{3}{4}L$ multiplications and $\frac{3}{4}L + \frac{1}{2}$ additions (see Appendix B for more details).

Of course, the decomposition can be continued on downsampled filters recursively as long as $2 \mid L/2$. If $L = 2^n$, the radix-2 divide-and-conquer algorithm can proceed n times until the length of the filters is reduced to only one tap. If k successive decompositions are performed, the cost is reduced to

$$\begin{aligned} C_{adds}(k, L) &= \left(\frac{3}{2}\right)^k \left[\frac{L}{2^k} - 1\right] + 4\left[\left(\frac{3}{2}\right)^k - 1\right] \\ C_{mult}(k, L) &= \left(\frac{3}{2}\right)^k \frac{L}{2^k} \end{aligned}. \tag{3.41}$$

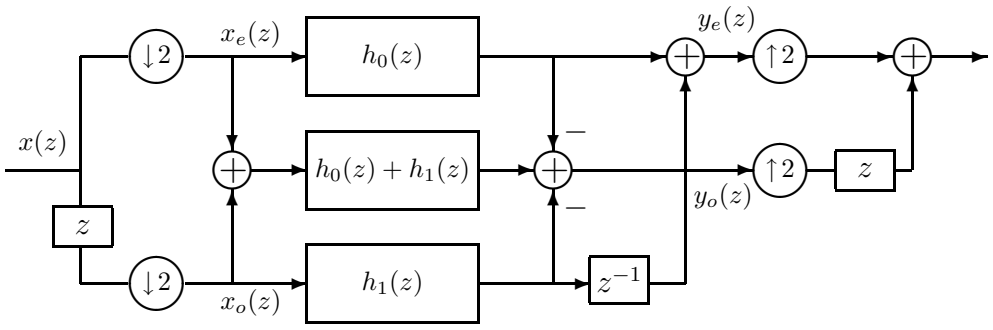


Figure 3.2: Radix-2 divide-and-conquer single step filter decomposition

Detailed cost analysis is presented in Appendix B.

This method has been well studied in the literature, with different authors referring to it by different names [57, 62]. In this thesis, we refer to it as the radix-2 Karatsuba algorithm for linear convolution. However, the decomposition we presented is not the only possible radix-2 algorithm. A similar algorithm with the same cost but different order of computations can be obtained by simply transposing the diagram in Figure 3.2. Other variants can also be readily derived [57].

Of course, the divide-and-conquer approach is not limited to the radix-2 algorithm. Decompositions involving higher radices can be readily derived, as well as mixed-radix solutions where the input and the output sequences are downsampled at different rates. We provide a general approach for deriving a class of divide-and-conquer algorithms in Appendix B. We also analyze and compare the costs of algorithms with higher radices.

As an example, we demonstrate a radix-3 Karatsuba decomposition. We require that $3 \,|\, L$ and $3 \,|\, N$, although this condition can be relaxed by zero-padding. Given a polynomial $p(z)$, we denote the polyphase components as $p_k(z^3) = \sum_i p_{3i+k} z^{3i+k}$. Then, the polynomial product $y(z) = h(z)x(z)$ can be decomposed into

$$\begin{bmatrix} y_0(z) \\ y_1(z) \\ y_2(z) \end{bmatrix} = \begin{bmatrix} h_0(z) & h_2(z)z^{-1} & h_1(z)z^{-1} \\ h_1(z) & h_0(z) & h_2(z)z^{-1} \\ h_2(z) & h_1(z) & h_0(z) \end{bmatrix} \begin{bmatrix} x_0(z) \\ x_1(z) \\ x_2(z) \end{bmatrix}. \tag{3.42}$$

By grouping the terms and representing cross products as

$$x_i h_j + h_i x_j = (x_i + x_j)(h_i + h_j) - h_i x_i - h_j x_j$$

we obtain the following decomposition of (3.42):

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & w & w & 0 & 0 & 1 \\ 1 & 1 & w & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix} \operatorname{diag}\{h_0, h_1, h_2, h_0{+}h_1, h_0{+}h_2, h_1{+}h_2\} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}, \tag{3.43}$$

where $w = z^{-1}$. We omit the polynomial argument z to save space. Even though the number of multiplications per output point is $\frac{2}{3}L$ and the number of additions is $\frac{2}{3}L+2$, which is lower than the

radix-2 decomposition, the radix-2 algorithm is asymptotically cheaper than the radix-3 algorithm when recursive decompositions are performed. The higher radix algorithms could still be preferred with the non two-power filter lengths. The data flow patterns are also considerably different, which might be beneficial on some computer platforms. The efficiency of different divide-and-conquer algorithms remains to be seen, and we shall investigate their performance in Chapter 7.

3.1.7 Short convolution algorithms

In Section 3.1.5 we presented methods for reducing long convolutions to a computation involving a combination of shorter convolutions. The reason for this transformation is the existence of numerous algorithms that efficiently compute short convolutions. We presented two approaches: the nesting of convolutions is used when the input and the filter lengths are of approximately the same size, whereas the overlap-add/save methods are used when the input sequence length is considerably larger. In the latter case, the advantage is apparent when the convolutions are computed in the transform domain, which we discuss in the next section. However, nesting techniques rely on either multi-dimensional transforms with the convolution property, or on efficient short convolution algorithms [35, 59].

Short convolution algorithms that minimize the number of required multiplications at the expense of an increased number of additions have been a focus of algorithm developers in the the early days of digital signal processing (DSP). The fundamental idea behind short convolution algorithms is the use of polynomial algebras and the Chinese remainder theorem (CRT) to perform the convolution in factor sub-algebras that require a smaller number of multiplications.

The simplest idea uses the Cook-Toom algorithm for polynomial multiplication in quotient algebras $y(z) = h(z)x(z) \in \mathbb{C}[z]/p(z)$, where p(z) has n distinct complex roots:

$$p(z) = (z - a_0)(z - a_1) \cdots (z - a_{n-1}). \tag{3.44}$$

The CRT can be applied for this factorization since $p_i(z) = z - a_i$ are trivially relatively prime. The CRT implies the isomorphism

$$\mathbb{C}[z]/p(z) \cong \bigoplus_i \mathbb{C}[z]/(z - a_i).$$

This means that there exists an isomorphism of algebras $\phi : \mathbb{C}[z]/p(z) \to \bigoplus_i \mathbb{C}[z]/(z - a_i)$ such that the multiplication in $\mathbb{C}[z]/p(z)$ can be performed independently in each $\mathbb{C}[z]/(z - a_i)$ *. The mapping ϕ is essentially the reduction of $h(z)x(z)$ modulo each $(z - a_i)$, which is just a polynomial evaluation on the a_i points. In matrix form, the representation of ϕ is a Vandermonde matrix

$$F = \begin{bmatrix} 1 & a_0 & \cdots & a_0^{n-1} \\ 1 & a_1 & \cdots & a_1^{n-1} \\ \vdots & \vdots & & \vdots \\ 1 & a_{n-1} & \cdots & a_{n-1}^{n-1} \end{bmatrix}, \quad F\mathbf{x} = \begin{bmatrix} x(a_0) \\ x(a_1) \\ \vdots \\ x(a_{n-1}) \end{bmatrix} \tag{3.45}$$

The Vandermonde matrix is, of course, invertible since it represents an isomorphic mapping.

The multiplication in sub-algebras is a simple scalar multiplication of polynomials $h(z)$ and $x(z)$ evaluated on a_i points. Hence, $y(a_i) = h(a_i)x(a_i) \in \mathbb{C}/(z - a_i)$. The polynomial $y(z)$ is then

*In fact, it is more appropriate to define it as an isomorphism of modules rather than algebras, since, in general, $\{h_k\}$ and $\{x_k\}$ do not lie in the same space [54].

reconstructed (interpolated) from these points by ϕ^{-1}. In matrix form, the Cook-Toom algorithm is now

$$\mathbf{y} = F^{-1}\left((F\mathbf{h}) \odot (F\mathbf{x})\right), \tag{3.46}$$

where $\odot$ is the element-wise product [35].

If one of the factors $h(z)$ is fixed, then the product $F\mathbf{h}$ can be precomputed. In that case (3.47) becomes

$$\mathbf{y} = F^{-1}\,\mathrm{diag}(\hat{h}_0, \hat{h}_1, \ldots, \hat{h}_{n-1})F\mathbf{x}, \quad \hat{\mathbf{h}} = F\mathbf{h}. \tag{3.47}$$

Probably the most famous Cook-Toom algorithm is obtained for the circular convolution on n points $y(z) = h(z)x(z) \bmod z^n - 1$. The roots of the polynomial $z^n - 1$ are complex roots of unity $w^k = e^{j2\pi k/n}$, $\quad k = 0, \ldots, n-1$, where $j = \sqrt{-1}$. The Vandermonde matrix F in (3.45) becomes the discrete Fourier transform (DFT), and the Cook-Toom algorithm is the well-known convolution theorem for the DFT:

$$\mathbf{y} = \mathrm{DFT}^{-1}\,\mathrm{diag}(\hat{h}_0, \hat{h}_1, \ldots, \hat{h}_{n-1})\,\mathrm{DFT}\,\mathbf{x}, \quad \hat{\mathbf{h}} = \mathrm{DFT}\,\mathbf{h}. \tag{3.48}$$

The problem with (3.48) is that the sampling points are complex, inducing a complex matrix (DFT) and complex multiplications (diagonal matrix). The fast Fourier transform (FFT) algorithms improve the cost of the algorithm dramatically; however, for short circular convolutions and real input signals, the advantage is lost due to complex multiplications.

It may be advantageous to design an algorithm as a multiplication in $\mathbb{C}[z]/p(z)$ where $p(z)$ has simple roots, e.g., integers 0, 1, and -1. Since we know from Section 3.1.4 that the linear convolution can be embedded into any polynomial algebra multiplication as long as condition (3.17) is satisfied, we can compute $h(z)x(z)$ as $h(z)x(z) \bmod p(z)$.

EXAMPLE 3.1. *Consider a linear convolution of two sequences* $\mathbf{h}$ *and* $\mathbf{x}$ *of length 2. To embed it into a quotient polynomial algebra we need a polynomial* $p(z)$ *of degree at least 3. Let us choose* $p(z) = z(z-1)(z+1)$ *with simple roots. The Cook-Toom algorithm is*

$$\mathbf{y} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{2} & -\frac{1}{2} \\ -1 & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \left(\begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \end{bmatrix} \mathbf{h} \odot \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \end{bmatrix} \mathbf{x} \right).$$

The algorithm requires 5 additions and 3 multiplication in case $\mathbf{h}$ *is fixed. An improvement to only 3 additions can be achieved by selecting* ∞ *as a sampling point [35, 63]. This is in contrast to 4 multiplications and 1 addition for the standard implementation*

Unfortunately, for longer convolutions, this type of algorithms minimizes the number of multiplications by using excessive number of additions. Winograd has proposed a modified version of the Cook-Toom algorithm where the polynomial $p(z)$ can have factors of degrees larger than one [64]. This increases the number of multiplications slightly but considerably reduces the number of required additions when compared to the Cook-Toom algorithm. For example, the 3-point circular convolution can be computed by factoring

$$p(z) = z^3 - 1 = (z-1)(z^2 + z + 1)$$

and computing multiplication modulo the factor polynomials. The resulting algorithm requires 4 multiplications and 11 additions as opposed to 9 multiplications and 6 additions for the standard implementation. A table of some of the best short convolution algorithms can be found in [59] in Section 3.7. A much more detailed overview and cost analysis of short convolution algorithms and all algorithms derived from the Chinese remainder theorem can be found in [26].

The algorithms discussed so far aim at reducing the number of multiplications since, traditionally, multipliers required more processor power that reflects into slower execution time in software and more expensive architecture in hardware. However, modern computer platforms and specialized DSP processors implement multiplications much more efficiently; hence, it remains to be seen if and when short convolution algorithms outperform other algorithms that are more suitable for computer implementation.

3.1.8 Transform domain filtering algorithms

We already introduced the DFT as an efficient method for computing the circular convolution of two sequences. As a direct consequence of the Cook-Toom algorithm, the convolution property of the DFT says that we can compute the circular convolution in the transform domain by element-wise multiplication:

$$\mathcal{DFT}\left\{h_n \circledast x_n\right\} = \mathcal{DFT}\{h_n\} \odot \mathcal{DFT}\{x_n\}. \tag{3.49}$$

In matrix form, the circular convolution is represented by a circulant matrix defined in (3.12). We have seen in (3.48) that the circular convolution can be performed through the DFT; hence, we can write

$$C(\mathbf{h}) = \mathrm{DFT}^{-1} \cdot D \cdot \mathrm{DFT}, \quad D = \mathrm{diag}(\hat{h}_0, \hat{h}_1, \ldots, \hat{h}_{n-1}), \quad \hat{\mathbf{h}} = \mathrm{DFT}\,\mathbf{h}. \tag{3.50}$$

Thus, the Cook-Toom algorithm provides an indirect proof that all circulant matrices are diagonalized by the DFT transform. This is called the *convolution property* of the DFT. The original problem of computing circular convolution requires generally $\mathcal{O}\left(N^2\right)$ operations. Since the DFT of size N can be computed using $\mathcal{O}(N \log N)$ operations, this method can save on the number of operations, especially for large filter lengths k. However, the DFT is not suitable for real-valued sequences since it represents them on the complex unit circle. One of the properties of the DFT says that the DFT of a real sequence is a conjugate symmetric sequence. This obvious redundancy can be overcome by designing real transforms that preserve the convolution property [65].

Real transforms related to the DFT can be designed by separating and combining real and imaginary parts of the transform so that the mirrored computations are eliminated. Popular examples include the real discrete Fourier transform (RDFT) and the discrete Hartley transform (DHT).

The RDFT is designed to separate the real and the imaginary part of the DFT, and to treat them as real coefficients [66].

$$\mathrm{RDFT}_N = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \cos\left(\frac{2\pi}{N}\right) & \cos\left(\frac{2\cdot 2\pi}{N}\right) & \dots & \cos\left(\frac{(N-1)\cdot 2\pi}{N}\right) \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \cos\left(\frac{(N/2-1)2\pi}{N}\right) & \cos\left(\frac{2(N/2-1)2\cdot 2\pi}{N}\right) & \dots & \cos\left(\frac{(N-1)(N/2-1)\cdot 2\pi}{N}\right) \\ 1 & -1 & 1 & \dots & 1 \\ 1 & \sin\left(\frac{(N/2-1)2\pi}{N}\right) & \sin\left(\frac{2(N/2-1)2\cdot 2\pi}{N}\right) & \dots & \sin\left(\frac{(N-1)(N/2-1)\cdot 2\pi}{N}\right) \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \sin\left(\frac{2\pi}{N}\right) & \sin\left(\frac{2\cdot 2\pi}{N}\right) & \dots & \sin\left(\frac{(N-1)\cdot 2\pi}{N}\right) \end{bmatrix} \tag{3.51}$$

From the definition of the RDFT we observe that it computes one half of the DFT of real data with real and imaginary parts of the result split into two vectors mirrored around the mid-point $N/2$ ($\omega = \pi$). This relation to the DFT can be represented in matrix form as

$$\mathrm{RDFT}_N = A_R \cdot \mathrm{DFT}_N, \qquad A_R = \frac{1}{2}\begin{bmatrix} 2 & & & \\ & I_{N/2-1} & & J_{N/2-1} \\ & & 2 & \\ & -i\cdot J_{N/2-1} & & i\cdot I_{N/2-1} \end{bmatrix}. \tag{3.52}$$

A similar relationship between other trigonometric transforms and generalized DFT transforms can be found in [67]. The convolution property for the RDFT can be now easily derived from (3.52) and (3.50) to obtain the RDFT transform domain method for computing the circular convolution.

$$C(\mathbf{h}) = \mathrm{RDFT}_N^{-1} \cdot X(\mathbf{h}) \cdot \mathrm{RDFT}_N, \tag{3.53}$$

where

$$X(\mathbf{h}) = \begin{bmatrix} c_0 & & & & & & & & \\ & c_1 & & & & & & -d_1 & \\ & & \ddots & & & & \iddots & & \\ & & & c_{\frac{N}{2}-1} & & -d_{\frac{N}{2}-1} & & & \\ & & & & c_{\frac{N}{2}} & & & & \\ & & & d_{\frac{N}{2}-1} & & c_{\frac{N}{2}-1} & & & \\ & & \iddots & & & & \ddots & & \\ & d_1 & & & & & & c_1 & \\ & & & & & & & & c_0 \end{bmatrix}, \quad \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N/2} \\ d_{N/2-1} \\ \vdots \\ d_1 \end{bmatrix} = \mathrm{RDFT}_N \cdot \mathbf{h}. \tag{3.54}$$

The inverse RDFT can be obtained as $\mathrm{RDFT}_N^{-1} = \mathrm{RDFT}^{\mathrm{T}} \cdot \frac{1}{N} D$ where $D = \mathrm{diag}\{1, 2, \dots, 2, 1, 2, \dots, 2\}$.

The discrete Hartley transform is another real transform alternative to the DFT [68].

$$\mathrm{DHT}_N = \left[\cos(2\pi ij/N) + \sin(2\pi ij/N)\right]_{N\times N} \tag{3.55}$$

It can be also derived from the DFT in a fashion similar to (3.52). Britanak and Rao [67] provide relations between a number of trigonometric transforms in their generalized form. The relation between the DHT and the RDFT is given as

$$\mathrm{DHT}_N = A_H \cdot \mathrm{RDFT}_N, \qquad A_H = \frac{1}{2}\begin{bmatrix} 2 & & & \\ & I_{N/2-1} & & J_{N/2-1} \\ & & 2 & \\ & J_{N/2-1} & & -I_{N/2-1} \end{bmatrix}.^{\dagger} \tag{3.56}$$

It is now straightforward to obtain the convolution property for the DHT from (3.53) and (3.56).

$$C(\mathbf{h}) = \mathrm{DHT}_N^{-1} \cdot X'(\mathbf{h}) \cdot \mathrm{DHT}_N, \tag{3.57}$$

where

$$X'(\mathbf{h}) = \begin{bmatrix} 2d_0 & & & & \\ & d_1 + d_{N-1} & & & d_1 - d_{N-1} \\ & & \ddots & \iddots & \\ & & & 2d_{\frac{N}{2}} & \\ & & \iddots & & \ddots \\ & d_{N-1} - d_1 & & & d_{N-1} + d_1 \end{bmatrix}, \qquad \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{N-1} \end{bmatrix} = \mathrm{DHT}_N \cdot \mathbf{h} \tag{3.58}$$

The above discussion is meant to show that the important convolution property can be transferred to real transforms in order to use them for more efficient computation of the circular convolution by using fast transform algorithms. The real transforms require only half of the arithmetic operations of the complex ones, and we expect that to transfer to their fast algorithms [69, 66, 70].

It can also be shown that, under certain symmetry conditions on the filters and input sequences, the discrete cosine (DCT) and the discrete sine transforms (DST) also have convolution properties and can be used to implement filtering of symmetric sequences [53, 55]. This type of filtering relies on the diagonalization properties of the DCTs and DSTs with respect to the generalized concept of convolution based on residue class polynomial algebras. We discussed these concepts in Section 3.1.3. More details can be found in [55].

Linear phase FIR filters are very important for signal processing. Since they have symmetric coefficients, linear or circular convolutions can be embedded into a special symmetric convolution for which there is a trigonometric transform that has the convolution property similar to (3.50). We return to the example given at the end of Section 3.1.3, where we assumed that the filter is odd and whole-point symmetric, and the input sequence is half-point symmetric and anti-symmetric at the boundaries. In this case the symmetric convolution matrix looks like (3.15), and has the following convolution property:

$$M(\mathbf{h}) = \text{DCT-4}^{-1} \cdot D \cdot \text{DCT-4}, \quad D = \mathrm{diag}(\hat{h}_0, \hat{h}_1, \ldots, \hat{h}_{n-1}), \quad \hat{\mathbf{h}} = \text{DCT-3} \cdot \mathbf{h} \tag{3.59}$$

where DCT-3 and DCT-4 are type 3 and type 4 discrete cosine transforms. So, this symmetric convolution can be performed in the transform domain as pointwise multiplication, The procedure

[†]The relation is given in Britanak [67] on page 140; however, there is a slight confusion with what f^I represents. The correct interpretation is that f^I is the generalized RDFT and not the GDFT of the input sequence

can, in turn, be used for filtering with odd symmetric filters by embedding linear into symmetric convolution as explained in Section 3.1.4.

This concludes our review of the most important filtering and convolution methods. In Chapter 5 we represent filtering and convolution operators as matrices to be implemented on a computer. We capture many of the algorithms discussed in this chapter in a set of breakdown rules that include a small set of mathematical constructs that can be translated into efficient code. Our goal is to provide a comprehensive methodology for generating and implementing the whole range of algorithms and utilize numerous degrees of freedom to search for the optimized solution on a target computer platform.

3.2 Discrete Wavelet Transforms

Wavelet theory addresses the problem of signal expansions in scaled subspaces that provide different levels of detail. The problem of multi-scale spaces has been known to mathematicians for quite some time but it caught considerable attention of scientists and engineers only in the mid eighties, especially after the work of Ingrid Daubechies [71] and Stephane Mallat [72]. The theory turned practical when Daubechies established a link between continuous-time wavelets and multirate filter banks, which had already been well studied and understood. Such tree-structured filter banks had been heavily used even before the arrival of wavelet theory, for example, in speech processing [73]. Daubechies, further, provided systematic methods for deriving orthonormal wavelet bases with compact support that led to efficient FIR filter bank structures and enormous activity in the signal processing communities [71]. Mallat, on the other hand, described wavelets using multiresolution analysis, which provided important clues on possible applications and interpretation of wavelet analysis.

The areas of research using, either directly or indirectly, wavelet analysis have since grown rapidly and the new theoretical results are spurred by many successful applications. Wavelets had a significant impact on data compression [16, 74, 75], denoising [76], detection, numerical solutions to partial differential equations (e.g., [77, 78]), and many other research areas.

In practical terms, wavelets owe their success to the discrete version of the continuous wavelet transform. The discrete wavelet transform (DWT), sometimes also referred to as the discrete-time wavelet transform (DTWT), is necessary for implementation in both hardware and software. It is one of very few useful transforms that have extremely low $\mathcal{O}(N)$ computational cost when compared to most other fast transforms, such as trigonomatric transforms, which have $\mathcal{O}(N \log N)$ cost. The DWT is inextricably tied to recursive multirate filter banks and is used for multiresolution expansion of discrete-time signals in a flexible way. The DWT is a linear transform and as such can be seen as a matrix. However, at this point we note that, unlike the DFT and other linear transforms, the DWT does not have a fixed definition. The DWT matrix will depend on the choice of the wavelet basis, the extension of the signals at the boundaries, and the number of levels of expansion. Hence, "the" in front of the DWT is slightly misleading so we will use it cautiously and point to differences in definitions as needed.

In the rest of the chapter, we will:

- Provide a very brief overview of the wavelet theory;
- Explain the origins of the DWT;
- Briefly touch upon some of the frequently used wavelets;
- Review standard algorithms for the computation and the required theoretical results;
- Provide motivation for introducing some of the algorithms and methods from the perspective of efficient computer platform implementation.

There are many ways to introduce wavelets. For any theoretical and practical purpose, *multiresolution analysis* (MRA) is a good point to start. It provides basic theoretical understanding of the principles of wavelet bases expansion and draws a connection to multirate filter banks without extensive mathematical preparation for a signal processing expert.

3.2.1 Multiresolution analysis

The concept of *multiresolution* arises from the idea of scaling and embedding function spaces [79]. Define a set of functions $S = \{\varphi(t-k)\},\ k \in \mathbb{Z}$, as integer translates of the square-integrable function $\varphi(t) \in (L_2(\mathbb{R}))$. Let the subspace $\mathcal{V}_0 \subset L_2(\mathbb{R})$ be spanned by the functions in S *. An arbitrary function $f(t) \in \mathcal{V}_0$ can be now represented as

$$f(t) = \sum_k c_k \varphi(t-k) \tag{3.60}$$

We can naturally "scale" the subspace $\mathcal{V}_0$ by dilating the set of spanning functions and form a subspace $\mathcal{V}_1$ spanned by $\{\varphi(2t-k)\}$. Since the spanning functions are contracted by the factor of two, any function can be represented in finer detail in the new subspace. By continuing to contract the spanning set, the scaling of subspaces $\mathcal{V}_j = \mathrm{Span}_k\{\varphi(2^j-k)\}$ adds more and more detail to each subsequent subspace $\mathcal{V}_j$. Spaces $\mathcal{V}_j$ created this way are called *scaling spaces*, whereas the spanning functions $\varphi(2^j-k)$ are called the *scaling functions*.

In addition to scaling the subspaces, to construct the multiresulution analysis (MRA), we require that the subspaces are nested so that any $\mathcal{V}_k$ at a lower scale is a subspace of $\mathcal{V}_{k+n}$ at a higher scale. In other words

$$\mathcal{V}_0 \subset \mathcal{V}_1 \subset \mathcal{V}_2 \subset \cdots \tag{3.61}$$

so that the information at a coarser level is contained in finer detail spaces. We also require the scaled subspaces to complete the Hilbert space $\mathcal{V}_\infty \equiv L_2(\mathbb{R})$. The spaces also have to satisfy the scaling condition

$$f(t) \in \mathcal{V}_j \quad \Rightarrow \quad f(2^m t) \in \mathcal{V}_{j+m}$$

We shall use a short notation for the functions in the spanning set of $\mathcal{V}_j$

$$\varphi_{j,k}(t) = \varphi(2^j t - k). \tag{3.62}$$

The spanning set $\{\varphi_{j,k}(t)\}$ forms either a basis, or an overcomplete spanning set treated by the theory of frames [16]. In this thesis, we consider only cases where each spanning is a basis.

*More precisely, $\mathcal{V}_0$ is the closure of Span S

Because of the telescoping of the spaces in (3.61), the basis functions at a lower scale $\mathcal{V}_j$ can be expanded by the basis at the next scaling level $\mathcal{V}_{j+1}$

$$\varphi_{j,0}(t) = \sum_k \sqrt{2}\, h_k\, \varphi_{j+1,k}(t), \qquad k \in \mathbb{Z} \tag{3.63}$$

The relation is called the *scaling equation* or the refinement equation, and the weights h_k are called *scaling coefficients* [14].

MRA creates a foundation for defining the wavelet expansion sets in the following way. Since $\mathcal{V}_j \subset \mathcal{V}_{j+1}$, it is possible to find a subspace $\mathcal{W}_j$ such that it represents a complement of $\mathcal{V}_j$ to $\mathcal{V}_{j+1}$:

$$\mathcal{W}_j \oplus \mathcal{V}_j = \mathcal{V}_{j+1}, \quad \mathcal{W}_j \subset \mathcal{V}_{j+1} \tag{3.64}$$

Such subspaces are called wavelet spaces of different scaling levels. It is clear now how wavelet spaces can be used to span the entire $L_2(\mathbb{R})$

$$L_2(\mathbb{R}) = \mathcal{V}_0 \oplus \mathcal{W}_0 \oplus \mathcal{W}_1 \oplus \cdots \tag{3.65}$$

The basis functions for the wavelet spaces $\mathcal{W}_j$ are called *wavelets* where the basic function at the lowest level of resolution $\mathcal{W}_0$ is called the *mother wavelet* $\psi_0(t)$. All other wavelet basis functions are then created by integer translates and octave dilates of the mother wavelet $\psi_{j,k} = \psi(2^j - k) \in \mathcal{W}_j$.

Since the wavelet space at one level is a subspace of the scaling space at the next level, as established in (3.64), we can also express these wavelets as a weighted sum of refined scaling functions, similar to (3.63):

$$\psi_{j,0}(t) = \sum_k \sqrt{2}\, g_k\, \psi_{j+1,k}(t), \qquad k \in \mathbb{Z} \tag{3.66}$$

The above relation is called the *wavelet equation* and the weights g_k are referred to as the *wavelet coefficients*.

Wavelet series. Consider a square-integrable function $f(t) \in L_2(\mathbb{R})$. From the decomposition of $L_2(\mathbb{R})$ in (3.65), we know that $f(t)$ can be represented as a weighted sum of basis functions of the lowest level scaling space and bases of all wavelet spaces

$$\mathcal{B} = \{\varphi_{0,k}(t), \psi_{0,k}(t), \psi_{1,k}(t), \psi_{2,k}(t), \ \ldots\}, \qquad k \in \mathbb{Z} \tag{3.67}$$

and, hence, we can perform a series expansion of $f(t)$ using the basis set $\mathcal{B}$:

$$f(t) = \sum_k 2^{1/2} c_{0,k} \varphi_{0,k}(t) + \sum_k \sum_j d_{j,k} 2^{j/2} \psi_{j,k}(t), \quad k, j \in \mathbb{Z},\ j > 0, \tag{3.68}$$

At this point, we note that the above expansion is usually referred to as the discrete wavelet transform (DWT) of the function $f(t)$. However, the "discrete" in the name does not imply a transform of discrete sequences. The terminology is slightly misleading for a signal processing expert familiar with the standard terminology for the Fourier signal analysis, where a trigonometric series expansion of an $L_2[-\pi, \pi)$ signal is called the Fourier series and the discrete Fourier transform (DFT) is a trigonometric series expansion of a periodic discrete-time signal. To avoid confusion, we refer to (3.68) as the *wavelet series* expansion. We defer the definition of the "true" discrete version of the transform till the next section.

The coefficients of the wavelet series (3.68) $c_{j,k}$ and $d_{j,k}$ are called the scaling and wavelet expansion coefficients, respectively. It should be clear from the above discussion that the coarsest scaling level is chosen to be 0 by convention. It can be any integer number j_0 from which the MRA could be constructed as previously described. We note that for $j_0 = -\infty$, the coarsest space is empty and the wavelet series expansion in (3.68) becomes

$$f(t) = \sum_k \sum_j d_{j,k} 2^{j/2} \psi_{j,k}(t), \quad k, j \in \mathbb{Z}$$

For practical purposes, we are interested in obtaining a discrete version of (3.68). Consider again a function $f(t) \in L_2(\mathbb{R})$ that we want to expand into wavelet series. Let us assume that there exist a detail level J at which the best approximation $\hat{f}(t)$ to $f(t)$ from $\mathcal{V}_J$ defined as

$$\|f(t) - \hat{f}(t)\| = \inf\{\|f(t) - g(t)\| : g(t) \in \mathcal{V}_j\}$$

has acceptably small error ϵ. Then we can represent $f(t)$ as

$$f(t) = \hat{f}(t) + \epsilon = \sum_k c_{J,k}\, \varphi_{J,k}(t) + \epsilon \tag{3.69}$$

The last equation can also be seen as a sampling of $f(t)$ with the coefficients $c_{J,k}$ representing the discrete-time approximation signal. We are now ready to introduce the discrete version of the wavelet transform.

3.2.2 Discrete wavelet transform

Let us assume that the coefficients $c_{J,k}$ are available and treated either as expansion coefficients of a continuous-time signal $f(t) \in \mathcal{V}_J$ or as a discrete-time sequence $\{x_k\}$. Using the MRA, we realize that every signal in $\mathcal{V}_J$ can be also represented in $\mathcal{V}_{J-1} \oplus \mathcal{W}_{J-1}$ as

$$f(t) = \sum_k c_{J-1,k}\, \varphi_{J-1,k}(t) + \sum_k d_{J-1,k}\, \psi_{J-1,k}(t) \tag{3.70}$$

To establish the relationship between coefficients at different scaling levels, we substitute the scaling (3.63) and the wavelet equations (3.66) into the inner product computation of coefficients

$$c_{J,k} = \langle f(t), \varphi_{J,k}(t) \rangle.$$

After the substitution, we obtain key equations that allow us to recursively compute scaling and wavelet coefficients at all coarser levels:

$$\begin{aligned} c_{j-1,n} &= \sum_m h_{m-2n}\, c_{j,\,m} \\ d_{j-1,n} &= \sum_m g_{m-2n}\, c_{j,\,m} \end{aligned} \tag{3.71}$$

Starting from the finest-detail level J, we could proceed one level down to compute all expansion coefficients by always recursing on the scaling coefficients $\{c_{j,k}\}$ and bookkeeping the wavelet coefficients $\{d_{j,k}\}$ at each subsequent stage.

DWT definition. Let the starting coefficients at the finest level J represent the input sequence, i.e., $\{x_k\} = \{c_{J,k}\}$. If we proceed with the recursion (3.71) exactly J times, we obtain the following

set of coefficients

$$\{y_n\} = \{\{c_{0,k}\}, \{d_{0,k}\}, \{d_{1,k}\} \ldots, \{d_{J-1,k}\}\} \tag{3.72}$$

The sequence $\{y_n\}$ is the J-level *discrete wavelet transform* (DWT) of the sequence $\{x_k\}$. If the number of recursions proceeds without bound, then $\{y_n\} = \{\{d_{J-1,k}\}, \{d_{J-2,k}\}, \ldots, \{d_{-\infty,k}\}\}$ and we call it simply the DWT.

DWT matrix. The DWT is clearly a linear transform and can be represented by a matrix. In the case of signals of infinite duration the matrix is infinite in size. At this point, we restrict our focus to an important class of wavelet systems built with the compact support wavelet functions. If the mother wavelet $\varphi(t)$ has compact support on the interval $t \in [0, N]$, and if the wavelet space $\mathcal{W}_0$ is an *orthogonal* complement of the scaling space $\mathcal{V}_0$, then the father scaling function has also compact support. It can be proven rather easily that, in that case, the scaling and the wavelet coefficients $\{h_k\}$ and $\{g_k\}$ also have limited support in the interval $[0, N]$ (see, e.g., [14]). The recursive equations (3.71) for one level of recursion are represented by the following infinite matrix

$$W = \begin{bmatrix} \ddots & \vdots & \vdots & \vdots & \vdots & & & & \\ & h_0 & h_1 & h_2 & \cdots & h_{N-1} & & & \\ & g_0 & g_1 & g_2 & \cdots & g_{N-1} & & & \\ & & & h_0 & h_1 & h_2 & \cdots & h_{N-1} & \\ & & & g_0 & g_1 & g_2 & \cdots & g_{N-1} & \\ & & & & & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{3.73}$$

If the sequence $\{x_k\}$ is finite, however, the above matrix is banded. Assume that the length of the input signal is N. The equations (3.71) suggest that, when the length of the scaling and wavelet coefficients is greater than two, some additional values at the boundaries of $\{x_k\}$ are needed to compute all coefficients at the next lower scale. Hence, the sequence $\{x_k\}$ has to be extended by exactly $L - 2$ points at the boundaries, where L is the length of either the scaling $\{h_k\}$ or the wavelet $\{g_k\}$ coefficients, whichever is greater. The type of extension depends on the signal model used [80, 81, 15]. For example, if $N = L = 4$, a one-level DWT matrix will look like

$$\mathbf{y} = W_4(\mathbf{h}, \mathbf{g})\mathbf{x} = \begin{bmatrix} h_0 & h_1 & h_2 & h_3 & & \\ & & h_0 & h_1 & h_2 & h_3 \\ g_0 & g_1 & g_2 & g_3 & & \\ & & g_0 & g_1 & g_2 & g_3 \end{bmatrix} E\,\mathbf{x} \tag{3.74}$$

where E is the signal extension operator that takes care of the boundary conditions. Notice that we slightly reordered the output from (3.73). A two-level DWT can be obtained by recursing only on the upper half of the output:

$$\mathbf{y} = \begin{bmatrix} W_2(\mathbf{h}, \mathbf{g})E & \\ & \mathrm{I}_2 \end{bmatrix} W_4(\mathbf{h}, \mathbf{g})\; E\,\mathbf{x} \tag{3.75}$$

Since the recursion (3.71) reduces the number of coefficients by two after descending down each level, there can be at most $\lfloor \log_2 N \rfloor$ levels. Without loss of generality, assume $N = 2^J$ and that the starting level is $J = \log_2 N$. The DWT algorithm terminates at level $j = 0$, since $\{c_{0,n}\}$ and

$\{d_{0,n}\}$ contain only one sample, which generates the fully recursed DWT. We have already seen one example of the fully recursed 2-level DWT in (3.75).

Multirate filter bank representation. The implications of the equations (3.71) is multifold. They explain how to efficiently obtain coefficients at different resolution levels of the expansion of the original signal, which immediately provides a fast algorithm for implementing the DWT. We refer to this algorithm as Mallat's algorithm. If the length of the scaling and the wavelet coefficients is much smaller then the signal length N, then the complexity of Mallat's algorithm is only $\mathcal{O}(N)$ as compared to $\mathcal{O}\left(N^2\right)$ if the DWT was implemented simply as a linear transformation.

Equations (3.71) also provide an important insight into the nature of the DWT. If we compare the equations with the convolution sum in (3.5), we observe that they represent convolutions with filters whose impulse responses are $\{h_{-k}\}$ and $\{g_{-k}\}$ where the resulting sequence is downsampled by two [16, 14, 15]. In the polynomial representation, we can write equations (3.71) as

$$\begin{aligned} c_{j-1}(z^2) &= \frac{1}{2}\left(h(z^{-1})c_j(z) + h(-z^{-1})c_j(-z)\right) \\ d_{j-1}(z^2) &= \frac{1}{2}\left(g(z^{-1})c_j(z) + g(-z^{-1})c_j(-z)\right) \end{aligned} \tag{3.76}$$

The scaling and wavelet coefficients are taps of the time-reversed filters $h(z^{-1})$ and $g(z^{-1})$, and one level of the recursion is performed by filtering the input signal with these two filters followed by the downsampling by two. It turns out that the filter $h(z^{-1})$ consisting of the scaling coefficients is a lowpass filter, whereas the wavelet filter $g(z^{-1})$ is highpass. The recursion can be continued by filtering and downsampling the lowpass portion of the output until the signal is reduced to only one sample.

The whole procedure is schematically presented in Figure 3.3. The input signal is lowpass and highpass filtered and then downsampled to obtain expansion coefficients at the next lower scaling level. The filter bank is recursed in the lowpass branch until a one sample output is obtained. The DWT of $\{x_k\}$ is the output sequence $\{y_k\}$. To establish a connection with the MRA, we indicated the scaling and the wavelet subspaces after each stage of the filter bank.

The above description of the DWT is closely related to the multirate signal processing that had been researched and applied well before the ground-breaking work of Daubechies [73]. The multirate filter bank representation provided better understanding of the wavelet analysis for a signal processing expert with the traditional knowledge of Fourier analysis and spawned a tremendous amount of activity in the area of wavelet signal processing. An excellent introduction to this topic can be found in several texts [16, 15, 82].

The DWT filter bank is considered as the analysis filter bank since the output represents the coefficients of the wavelet expansion. It is often necessary to process the signal in the transform domain and then reconstruct the result using the inverse DWT or the synthesis filter bank. The synthesis filter bank can be obtained by transposing the graph in Figure 3.3. Transposition changes all nodes into adders, downsamplers into upsamplers, and analysis filters $(h(z), g(z))$ into synthesis filters $(\tilde{h}(z), \tilde{g}(z))$. The synthesis filter bank tree is shown in Figure 3.4. For orthogonal wavelet bases, the inverse DWT matrix is simply the transpose of the forward DWT matrix. The transposition of the filter matrices equals the time-reversal in the z-domain representation of the filters, so the synthesis filters in this case are simply $\tilde{h}(z) = h(z^{-1})$, $\tilde{g}(z) = g(z^{-1})$. The advantage of orthogonal

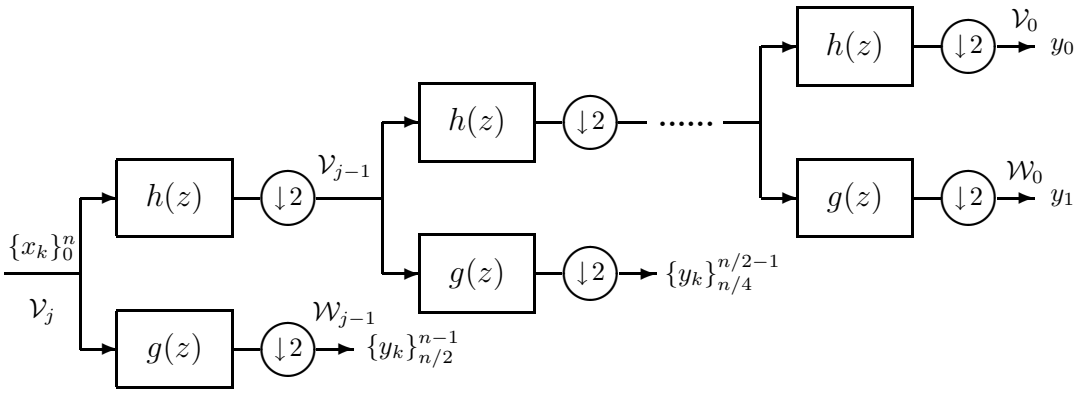


Figure 3.3: Filter bank interpretation of the DWT.

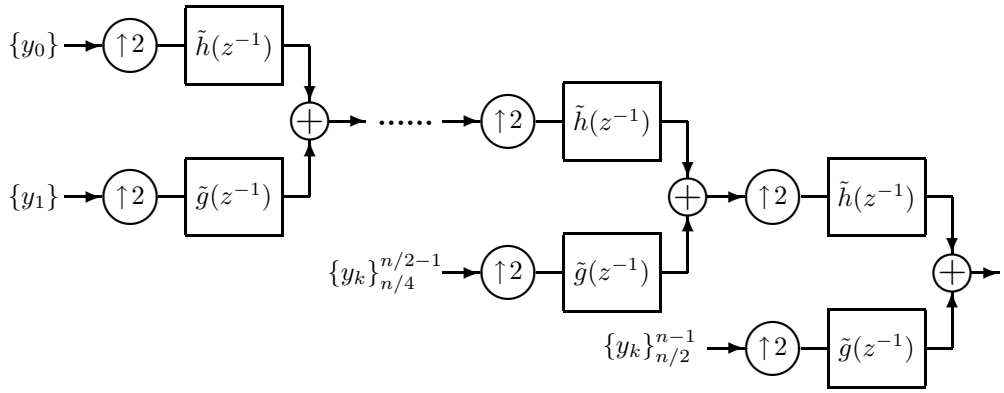


Figure 3.4: Filter bank representation of the inverse DWT.

systems is that they are simple to construct and that Parseval's theorem holds, i.e., that the energy of the signal is preserved in the transform domain. However, imposing the orthogonality as the condition removes many other degrees of freedom so that, for example, it is no longer possible to design filters with the linear phase: a major drawback of orthogonal wavelet systems. The *biorthogonal* systems are a generalization of orthogonal wavelets and allow a more flexible design. We next discuss this and other generalizations of wavelet analysis and synthesis tools.

Generalizations of the DWT. Wavelets are an extremely versatile tool for modeling and analyzing signals. We have seen that the wavelet expansion is designed to extract properties of signals at different resolutions in both time and frequency. This expansion strategy is very useful when the signal does not have fast changing components at low frequencies but has high-frequency components (e.g., noise) that need to be localized in time (e.g., for denoising) [16, 14]. We expand some of the definitions and review some generalizations that further empower wavelet tools.

In Section 3.2.1, we developed multiresolution analysis for the case when the scaling is done with a factor of two which is often referred to as the *dyadic expansion*. The same concepts can be developed for any integer dilation factor M, which leads to *M-channel filter banks*, and accordingly, to the *M-band DWT*. Further, we mentioned that if the wavelets are compactly supported, the lowpass and the highpass filters are FIR. The filter bank implementation of the DWT in Figure 3.3 does not

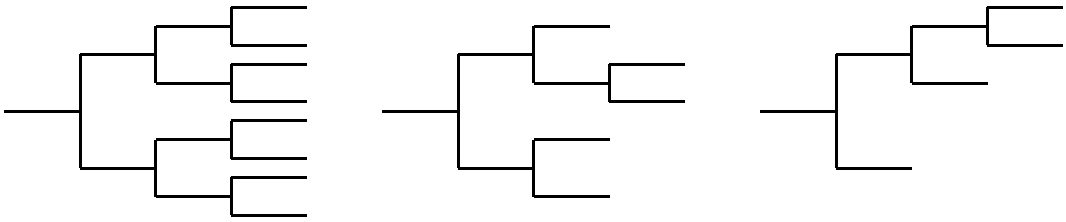


Figure 3.5: Wavelet packet trees: full (left); pruned (center); DWT (right).

require this condition. The filters can also be running IIR filters for real-time implementations in both hardware and software. However, FIR filters have the advantage of being stable and having simple linear phase design. Moreover, when the filters are FIR, the DWT can be represented as a banded matrix, which allows treatment of the DWT as other block transforms such as the DFT.

The standard dyadic DWT always splits the scaling spaces into lower resolution wavelet and scaling subspaces. This creates a one-sided filter bank tree shown in Figure 3.3. However, it is possible to expand any branch of the tree, or equivalently, to expand the subspaces of the wavelet spaces as needed. For example, starting from $\mathcal{V}_J = \mathcal{V}_{J-1} \oplus \mathcal{W}_{J-1}$ we could expand both $\mathcal{V}_{J-1} = \mathcal{V}_{J-2} \oplus \mathcal{W}_{J-2}$ and $\mathcal{W}_{J-1} = \mathcal{W}_{J-1,0} \oplus \mathcal{W}_{J-1,1}$ or either one of them to obtain different expansions adjusted to the properties of the expanded signal. This is illustrated in Figure 3.5 for three different constellations. The expansions obtained this way are referred to as *wavelet packets*. The fully expanded wavelet packet tree provides tiling of the time-frequency plane similar to the short-time Fourier transform and thus requires $\mathcal{O}(N \log N)$ operations. However, the full tree is usually adaptively pruned to find the best wavelet basis that matches the characteristics of the signal and achieves specific objectives [83, 84, 85, 86].

Wavelet examples. The design of the wavelet systems in the discrete case boils down to the design of filters in the wavelet filter bank. After satisfying basic properties and requirements for wavelet systems, such as perfect reconstruction, completeness, etc., there are many remaining degrees of freedom that can be used to design wavelets that satisfy additional properties such as orthogonality, smoothness, regularity, symmetry, compactness, and many more. Based on these choices, numerous classes of wavelet systems have been designed that have good properties for specific applications. We briefly mention only some of them next.

The simplest meaningful example is the Haar basis, the shortest among all wavelet bases. The Haar wavelet and the scaling functions are box functions shown in Figure 3.6. In the case of the Haar wavelet it is very easy and instructive to determine the wavelet and the scaling coefficients. It is clear from equations (3.63) and (3.66), and the shape of the box functions in Figure 3.6 that the scaling coefficients are $\mathbf{h} = \frac{1}{\sqrt{2}}(1,1)^T$ and the wavelet coefficients $\mathbf{g} = \frac{1}{\sqrt{2}}(1,-1)^T$. The corresponding filters are a simple moving average and a moving difference filter, respectively, a simple filter bank of lowpass and highpass blocks. The wavelet and the scaling functions are of compact support and so are the filter impulse responses. It is interesting to note that this is the only case when the DWT matrix is natively square, i.e., no extension operator is needed.

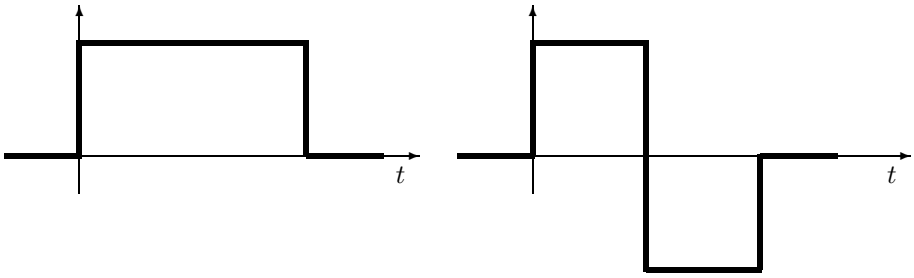


Figure 3.6: Haar scaling function and the wavelet.

EXAMPLE 3.2. *According to* (3.75) *we have*

$$\mathrm{DWT}_4^{Haar} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

The Haar DWT is also known as the rational Haar transform (RHT) since it was discovered long before wavelet theory came to life. It is also interesting to notice that, if we use the Haar basis and the fully expanded wavelet packet tree, we obtain the Walsh-Hadamard transform [15].

The next best known is the *sinc* wavelet: a direct consequence of the Haar basis and Shannon's sampling theorem. The lowpass filter impulse response in this case is the discrete sinc function $h_k = \mathrm{sinc}\left(\frac{\pi}{2}k\right)$ and the highpass is derived from the relation $\psi(t) = 2\varphi(2t) - \varphi(t)$.

Daubechies constructed orthogonal wavelets where the number of zero moments of the wavelet function are maximized, which translates into the lowpass and the highpass filters $h(z)$ and $g(z)$ having maximum number of zeros at frequencies 0 and π, or equivalently, being maximally flat at those frequencies; hence, they are called the *Daubechies maxflat* wavelets [87, 88]. The Daubechies wavelets can be of arbitrary even length L; the details of the design can be found in [71, 87].

For $L = 2$ we have the Haar basis, for $L = 4$ we get the Daubechies D_4 wavelets with the lowpass filter

$$\mathbf{h}_{D_4} = \frac{1}{4\sqrt{2}} \left(1 + \sqrt{3},\ 3 + \sqrt{3},\ 3 - \sqrt{3},\ 1 - \sqrt{3}\right)^T$$

and the highpass filter is simply given as $g(z) = h(-z^{-1})$, which is always the case for orthogonal wavelets. For larger lengths, the coefficients have an increasingly complex closed form [89]. It can be shown that the orthogonal wavelets can be parameterized and represented very nicely using the so-called *lattice decomposition* [82]. The choice of parameters is used to design optimal system for a specific problem [90].

Spline functions have also been used for wavelet design for their smoothness and symmetry properties. The most popular are the cubic splines for which the scaling coefficients are

$$\mathbf{h}_{\mathrm{cspline}} = \frac{1}{\sqrt{2}} \left(\frac{1}{16},\ \frac{1}{4},\ \frac{3}{8},\ \frac{1}{4},\ \frac{1}{16}\right)^T$$

However, splines are not orthogonal to integer translations, but they can be orthogonalized to generate the *Battle-Lemarié* wavelets [87, 15]. The problem with these is that they have infinite

support and other unsatisfactory properties. If splines are used as biorthogonal wavelets, then the compact support of splines translates into FIR filters where the synthesis filters are solved as a set of linear equations [91]. This class of wavelets are called *Cohen-Daubechies-Feauveau biorthogonal* family of biorthogonal wavelets and are widely used, e.g., in FBI fingerprint and the JPEG2000 image compression standards [17, 92].

Daubechies designed wavelets systems named *coiflets* that have both zero wavelet and zero scaling function moments, based on work by Coifman [87, 93]. Furthermore, the number of zero moments can be traded between the scaling function and the wavelet to better suit the application (e.g., [94]).

In addition, there are many classes of complex wavelet systems based on the Gaussian function and complex exponentials, such as the Morlet wavelets and the Gabor wavelets. In this thesis, however, we consider only real wavelet systems.

3.2.3 Polyphase representation

The multirate filter bank formulation of the wavelet expansion was very important, not only because it made wavelet theory accessible to the signal processing experts, but also because it allowed direct application of many already developed techniques for filter bank implementations. One very useful technique, both in terms of efficiency and clarity, for representing single and multirate filter banks is called the *polyphase representation* developed in the mid seventies [95].

Consider again the multirate filter bank tree in Figure 3.3. We observe that the downsampling is performed after each filtering stage, which effectively removes every other sample of the filter output. This is clearly not very efficient since every second cycle the system is performing redundant operations. For an M-channel filter bank this inefficiency is even more pronounced. The polyphase representation makes use of the *Noble* identities that explain how to commute downsamplers and filters. Figure 3.7 shows one Noble identity and the application of the identity to achieve more efficient filtering with the downsampled filters $h_e(z)$ and $h_o(z)$ we defined in (3.38). Instead of filtering the input signal with a bank of filters and then throwing away half of the results, the identities show us that the downsampling can be done first, followed by filtering usign filters with downsampled coefficients. This speeds up the computation by the downsampling factor; in this case by two.

Using the decomposition from Figure 3.7, we can apply the identities to each stage of the wavelet filter bank tree. If the identity is applied to both the lowpass filter $h(z)$ and the highpass filter $g(z)$ and the polyphase channels carrying subsampled input signal are merged together, then we obtain the identity shown in Figure 3.8. The input signal is split into even and odd samples by shifting and downsampling, which creates two separate sequences in two channels. Each polyphase sequence is filtered with the corresponding downsampled filters having only half the coefficients: even coefficient filters for the even channel and odd coefficient filters for the odd channel. The results are then added as suggested in Figure 3.7 and the whole filtering operation can be represented as the filter matrix operation shown in Figure 3.8. The exact derivation can be found in [15, 82].

The filter matrix

$$P(z) = \begin{bmatrix} h_e(z) & h_o(z) \\ g_e(z) & g_o(z) \end{bmatrix} \tag{3.77}$$

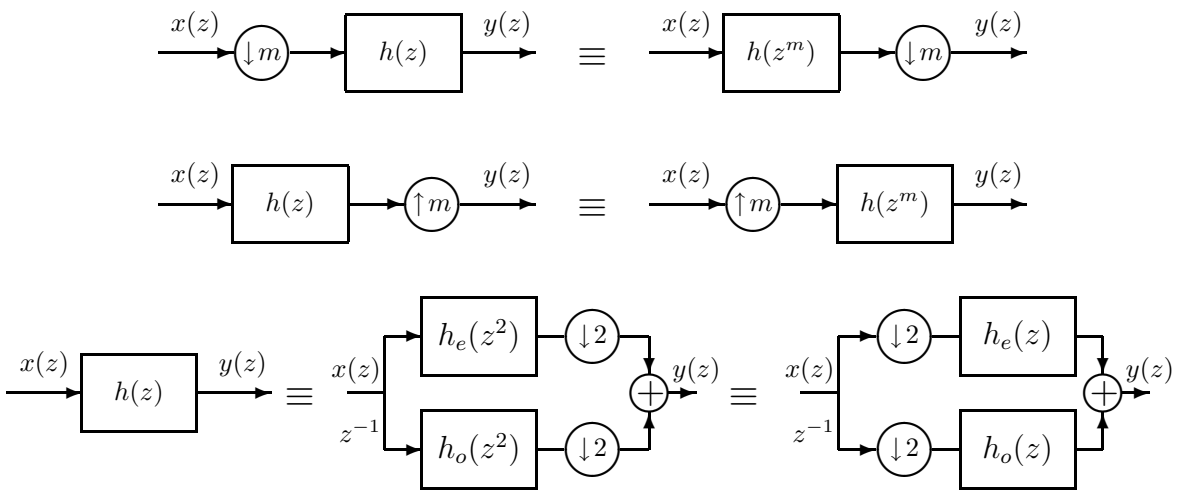


Figure 3.7: Noble identities (top); Efficient filtering using the Noble identity (bottom).

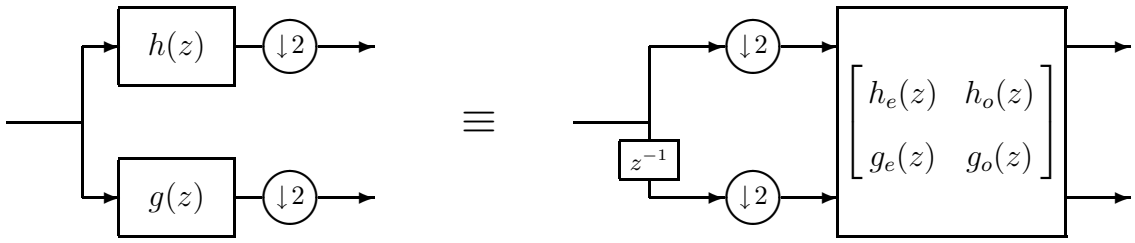


Figure 3.8: Polyphase representation of one filter bank stage.

is called the *polyphase matrix*. The elements of the matrix are polynomials describing downsampled filters defined in (3.38). A similar polyphase representation exists for the synthesis filter bank that yields the synthesis polyphase matrix

$$\tilde{P}(z) = \begin{bmatrix} \tilde{h}_e(z) & \tilde{g}_e(z) \\ \tilde{h}_o(z) & \tilde{g}_o(z) \end{bmatrix}. \tag{3.78}$$

The matrix $P(z)$ plays an important role in designing wavelet systems since it captures all properties and conditions that are either required or desired for the chosen wavelet basis. For example, the perfect reconstruction (PR) defined in (3.82) establishes a relationship between analysis and synthesis filters

$$\begin{aligned} h(z) &= z\tilde{g}(-z^{-1}) \qquad & g(z) &= -z\tilde{h}(-z^{-1}) \\ \tilde{g}(z) &= -zh(-z^{-1}) \qquad & \tilde{h}(z) &= zg(-z^{-1}). \end{aligned} \tag{3.79}$$

Given the analysis filter bank we can compute the synthesis filters and vice versa.

If the system is orthogonal then

$$\tilde{P}(z) = P(z)^T, \tag{3.80}$$

in which case

$$h(z) = \tilde{h}(z), \quad g(z) = \tilde{g}(z), \quad h(z) = zg(-z^{-1}), \quad g(z) = -zh(z^{-1}) \tag{3.81}$$

Many other necessary and sufficient conditions are easier to derive using this formulation [16]. Furthermore, the polyphase representation is used to exploit the relationship between filters, such as the ones in (3.81), and derive even more efficient implementations. We explore these possibilities next.

3.2.4 Lattice factorization

We have seen in the previous section that the polyphase matrices play an important role in filter design for wavelet filter banks, and that they capture many important properties of wavelet systems. Polyphase matrices reveal important structural information about the system, which enables a more efficient implementation.

In some important special cases, the polyphase matrix takes an especially regular form. One such case is the orthogonal 2-channel filter bank, or equivalently, a 2-band DWT with an orthogonal basis. From the perfect reconstruction condition (3.82), the orthogonality condition is obtained as $\tilde{P}(z) = P(z)^T$ and the filters have to satisfy (3.80). Such polyphase matrix has the property of being *paraunitary*, i.e., it is unitary on the unit circle $|z| = 1$ [82, 96], with the condition

$$P(z^{-1})^T P(z) = \mathrm{I}\,. \tag{3.82}$$

This condition gives a very special structure to the polyphase matrix that is exploited in factorizations.

The *lattice factorization* is based on factoring the polyphase matrix with the paraunitary property into elementary paraunitary matrices. Without loss of generality, we assume that the lowpass and

highpass filters are FIR and causal, i.e., $h(z) = \sum_{i=0}^{L} h_k z^{-k},\ g(z) = \sum_{i=0}^{L} g_k z^{-k}$. Next, we design two elementary paraunitary matrices

$$\mathrm{R}_\theta = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \tag{3.83}$$

representing Givens rotations, and the delay operator

$$\Delta(z) = \begin{bmatrix} 1 & 0 \\ 0 & z^{-1} \end{bmatrix}. \tag{3.84}$$

It is easy to see that a cascade of these two matrices

$$H(z) = \mathrm{R}_{\theta_p}\, \Delta(z)\, \mathrm{R}_{\theta_{p-1}} \cdots \Delta(z)\, \mathrm{R}_{\theta_0} \tag{3.85}$$

will again be paraunitary. It turns out that the above cascade of simple matrices is sufficient to implement any real paraunitary system [82]. Thus, when the polyphase matrix $P(z)$ is real, we can factorize it into the following cascade

$$P(z) = \alpha\, \mathrm{R}_{\theta_l}\, \Delta(z)\, \mathrm{R}_{\theta_{l-1}} \cdots \Delta(z)\, \mathrm{R}_{\theta_0} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \tag{3.86}$$

where l is the filter length of the downsampled filters in $P(z)$, and α is a scalar factor. The structure of the factorized system is shown in Figure 3.9. Further improvement can be obtained by factoring out $\cos\theta_m$ from each of the Givens rotations R_i and gathering them all into the constant β. The rotations become the following butterfly computations

$$\begin{bmatrix} \cos\theta_i & \sin\theta_i \\ -\sin\theta_i & \cos\theta_i \end{bmatrix} = \cos\theta_i \begin{bmatrix} 1 & \tan\theta_i \\ -\tan\theta_i & 1 \end{bmatrix} = \cos\theta_i \begin{bmatrix} 1 & \alpha_i \\ -\alpha_i & 1 \end{bmatrix}. \tag{3.87}$$

The constant β can be written as a function of all α's as

$$\beta = \frac{\alpha}{\prod_k \sqrt{1+\alpha_k^2}}.$$

The synthesis polyphase matrix $P(z^{-1})^T$ factorization can be obtained easily by transposing and time-reversing (3.86)

$$P(z^{-1})^T = \alpha \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \mathrm{R}_{\theta_0}\, \Delta(z^{-1})\, \mathrm{R}_{\theta_1} \cdots \Delta(z^{-1})\, \mathrm{R}_{\theta_l}\,. \tag{3.88}$$

Assume that the lowpass and highpass filters $h(z)$ and $g(z)$ are real and satisfying the orthogonality condition (3.80). The lattice decomposition shown in Figure 3.9 can be obtained recursively by solving the system of equations

$$\begin{aligned} (1+\alpha_j^2) h^{(j-1)}(z) &= h^{(j)}(z) - \alpha_j g^{(j)}(z) \\ (1+\alpha_j^2) z^{-2} g^{(j-1)}(z) &= \alpha_j h^{(j)}(z) + g^{(j)}(z) \end{aligned}, \tag{3.89}$$

by initializing $h^{(j)}(z) = h(z)$ and $g^{(j)}(z) = g(z)$, and then solving for α_j so that the highest power of $h^{(j)}(z)$ is canceled.

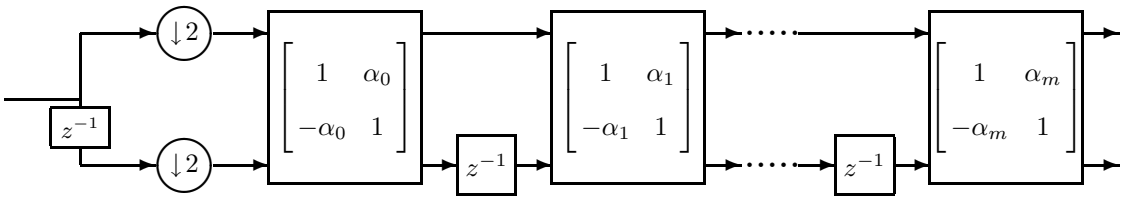


Figure 3.9: Lattice decomposition of an orthogonal filter bank.

The lattice factorization with rotations calculated as shown in Figure 3.9 requires $2(l+1)+2$ multiplications and $2(l+1)$ additions as compared to $4l$ multiplications and $4l-2$ additions per output point, an approximate two times speedup for the lattice structure. Further, the lattice factorization alleviates the effects of the quantization on finite precision arithmetics since the orthogonality of the transform is preserved and, more importantly, the perfect reconstruction condition holds. The lattice factorization can be used as the (only) design tool for multirate filter banks to optimize the coefficients so as to minimize the stopband energy [97].

3.2.5 Lifting scheme factorization

The lattice factorization from the previous section reduces the arithmetic cost of computations for one stage of the wavelet filter bank by decomposing the paraunitary polyphase into elementary stages of rotations and delays. The disadvantage of this approach is that it does not apply to biorthogonal systems, for which the polyphase matrix is not paraunitary. Based on the work of Lounsbery et al. [98], Sweldens and Daubechies designed a new technique for factorizing any polyphase matrix that satisfies property (3.82) into elementary matrices called the *lifting steps* [19]. The algorithm is referred to as the *lifting scheme* factorization and is used both as an efficient implementation of the DWT and as a versatile tool for designing biorthogonal wavelet systems. The original goal was to develop techniques for the design of second generation wavelets; however, the lifting scheme has found a wide use in efficient implementations of first generation wavelet systems (e.g., [17]).

The lifting scheme (LS) is based on the Euclidean algorithm for polynomials that is used for constructions factoring filter banks into ladder structures [99, 100]. Consider a polyphase matrix $P(z)$ obtained for the filter pair $\{h(z), g(z)\}$ and defined in (3.77). The perfect reconstruction condition (3.82) implies that the determinant of $P(z)$ is a monomial

$$\det P(z) = cz^p. \tag{3.90}$$

It follows that the matrix $P(z)$ can be factored into polynomial matrices whose determinant is also a monomial, or 1 as a special case. The matrices

$$S_i(z) = \begin{bmatrix} 1 & s_i(z) \\ 0 & 1 \end{bmatrix}, \text{and } T_i(z) = \begin{bmatrix} 1 & 0 \\ t_i(z) & 1 \end{bmatrix} \tag{3.91}$$

clearly satisfy this condition. We call them *primal* and *dual lifting matrices*.

Without any loss of generality, assume that the degree of the polynomial $h_e(z)$, as defined in (3.2), is larger than the degree of $h_o(z)$. By dividing $h_e(z)$ with $h_o(z)$ we obtain

$$h_e(z) = h_o(z)t_0(z) + r_e(z) \tag{3.92}$$

where $s(z)$ is the quotient, and $r_e(z)$ is the remainder of the division. Because of the requirement $\det P(z) = 1$, the same equation must hold for the highpass filter

$$g_e(z) = g_o(z)t_0(z) + r_o(z). \tag{3.93}$$

If we rename $h_e^1(z) = r_e(z)$, and $h_o^1(z) = r_o(z)$ and use the matrix form, these two equations amount to

$$\begin{bmatrix} h_e(z) & h_o(z) \\ g_e(z) & g_o(z) \end{bmatrix} = \begin{bmatrix} h_e^1(z) & h_o(z) \\ g_e^1(z) & g_o(z) \end{bmatrix} \begin{bmatrix} 1 & 0 \\ t_0(z) & 1 \end{bmatrix} \tag{3.94}$$

The original polyphase matrix is factored into the new polyphase matrix $P^1(z)$ with the determinant one and the lifting step matrix. Since $\deg(h_e^1) < \deg(h_o)$, the division can proceed as

$$\begin{aligned} h_o(z) &= h_e^1(z)s_0(z) + h_o^2(z) \\ g_o(z) &= g_e^1(z)s_0(z) + g_o^2(z) \end{aligned} \tag{3.95}$$

and, hence, we obtain

$$\begin{bmatrix} h_e(z) & h_o(z) \\ g_e(z) & g_o(z) \end{bmatrix} = \begin{bmatrix} h_e^1(z) & h_o^2(z) \\ g_e^1(z) & g_o^2(z) \end{bmatrix} \begin{bmatrix} 1 & s_0(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ t_0(z) & 1 \end{bmatrix} \tag{3.96}$$

The division can be continued alternating between even and odd filters that produce alternating primal and dual lifting steps. The procedure is equivalent to the Euclidean algorithm for polynomials that terminates with the greatest common divisor (GCD) for the polynomials $h_e(z)$ and $h_o(z)$. Since these polynomials are relatively prime because of the condition (3.82), the $\gcd(h_e, h_o) = 1$. This ensures that the algorithm terminates with the diagonal matrix of constant coefficients with the determinant one. Thus, the polyphase matrix is decomposed into

$$P(z) = \begin{bmatrix} \alpha & 0 \\ 0 & 1/\alpha \end{bmatrix} \cdot \prod_{i=0}^{m} \begin{bmatrix} 1 & s_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ t_i(z) & 1 \end{bmatrix} \tag{3.97}$$

The described factorization method is not unique due to the fact that the division with remainder of Laurent polynomials is not unique. The consequence of this is that there may be many lifting schemes for the same polyphase matrix. A detailed analysis of the degrees of freedom in choosing the LS is given in Appendix A.

The lifting scheme reduces the arithmetic cost for computing the polyphase matrix asymptotically by one half; this is similar to the savings obtained for the lattice factorization of orthogonal filter banks. Furthermore, it allows for in-place implementation of each stage of the DWT, which is easy to see in Figure 3.10 that schematically describes the lifting scheme factorization. Each of the two channels requires allocation of one memory block that is updated every other lifting step by the filtered sequence from the other channel.

Other advantages of the LS we do not consider in this thesis are the possibility of adaptive wavelet transforms and, as was the case with lattice structures, robustness to quantization errors.

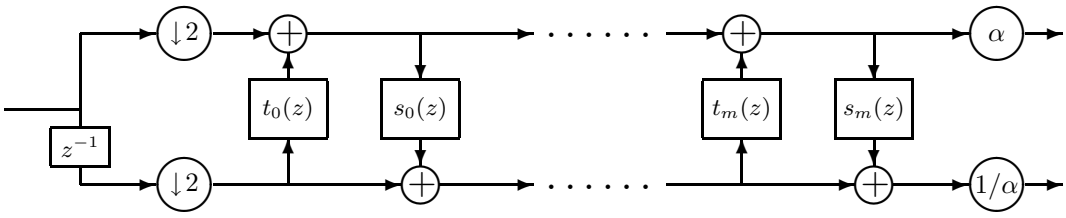


Figure 3.10: Lifting scheme for one stage of the forward DWT.

Additionally, with LS it is rather simple to design nonlinear wavelet transforms, such as integer-to-integer DWTs [101, 102].

The LS is widely used, for example, in the JPEG2000 standard [17]. However, (3.97) shows that the LS algorithm increases the length of the critical path. As a consequence, it may actually increase runtime even though it reduces the arithmetic cost. We will perform experiments to investigate this behavior in Section 7.

3.3 Summary

This chapter provides background information on digital FIR filters and discrete wavelet signal processing required to understand the concepts of representing and generating fast algorithms for their computation. We introduced the definitions of linear, circular, and generalized convolutions, and discussed some of the most frequently used algorithms for their efficient implementation. The presented algorithms can be broadly classified into algorithms implemented fully in the time domain and algorithms implementing filters in transform domain. We decide to present a more interesting classification from the implementation perspective:

- Algorithms that do not reduce the computational cost but provide better locality and better data flow patterns: overlap-add, overlap-save, nesting, and other blocking techniques;
- Algorithms that reduce the arithmetic cost of the direct filtering implementation, but still have $\mathcal{O}\left(n^2\right)$ cost: divide-and-conquer methods;
- Methods that reduce the cost to $\mathcal{O}\left(n \log n\right)$: transform-domain methods based on the convolution property of trigonometric transforms such as the DFT and the DHT;
- Simple techniques that serve as a gateway to other efficient methods (Agarwal-Cooley algorithm for multi-dimensional convolution, embedding methods).

We explore the advantages and disadvantages of each class of algorithms in Chapter 7, where we design experiments to compare different approaches.

In the second part of this chapter, we introduced wavelets multi-resolution analysis and drew a connection between critically-sampled filter banks and the discrete wavelet transform (DWT). Several generalizations of the DWT were discussed, including wavelet packets and multi-channel filter banks. We reviewed most of the commonly used efficient methods for implementing the DWTs.

Since the DWT can be computed using Mallat's equations (3.71), the definition of the DWT already lends itself to a very fast implementation method of only $\mathcal{O}(n)$ cost, where n is the length of the input sequence. Because of the special properties of the wavelet systems, the arithmetic cost can be further reduced by techniques such as the lifting scheme and the lattice factorization. However, the reduced cost comes at the price of increased critical path of the computation and the increased memory bandwidth. We compare these methods and report our results in Chapter 7.

CHAPTER 4

MATHEMATICAL PRELIMINARIES

In the previous chapter we have reviewed known algorithms for efficient computation of FIR filtering and wavelet transform operations. We want to formulate these algorithms using the rule formalism to be able to automatically generate implementations using SPIRAL and search for those that are best matched to the target platform. In this chapter we provide necessary mathematical definitions, concepts, and tools we use to derive and represent implementations of important filtering and wavelet algorithms.

4.1 Matrices

Matrix representations of linear operators are the foundation of our mathematical framework. We start by introducing basic definitions and conventions that will be used throughout this thesis.

4.1.1 Basic definitions

Linear transformation. A linear transformation from $\mathbb{C}^n$ to $\mathbb{C}^m$ is a function $T : \mathbb{C}^n \mapsto \mathbb{C}^m$ such that

$$\mathbf{y} = T(c\mathbf{v} + \mathbf{w}) = c(T\mathbf{v}) + T\mathbf{w} \tag{4.1}$$

where $\mathbf{v}, \mathbf{w} \in \mathbb{C}^n, \mathbf{y} \in \mathbb{C}^m$ and $c \in \mathbb{C}$. Given an ordered basis $\{\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}\}$ for $\mathbb{C}^n$, the transformation T is uniquely defined by its action on the basis vectors $\mathbf{y}_i = T\mathbf{x}_i,\ i = 0, \ldots, n-1$ [103]. If we choose the standard basis $\{\mathbf{e}_0, \ldots, \mathbf{e}_{n-1}\}$ for $\mathbb{C}^n$, defined by

$$\begin{aligned} \mathbf{e}_0 &= (1, 0, 0, \ldots, 0)^T \\ \mathbf{e}_1 &= (0, 1, 0, \ldots, 0)^T \\ &\cdots\cdots \\ \mathbf{e}_{n-1} &= (0, 0, 0, \ldots, 1)^T, \end{aligned} \tag{4.2}$$

represented as column vectors, then $\mathbf{t}_i = T\mathbf{e}_i,\ i = 0, \ldots, n-1$ is also unique.

Matrix. A matrix A over a field $\mathbb{F}$ is a rectangular array of elements $a_{i,j} \in \mathbb{F}$ with m rows and n columns

$$A = \begin{bmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} \end{bmatrix}. \tag{4.3}$$

In other words, A is an element of $\mathbb{F}^{m\times n}$.

In writing and referring to matrices we introduce a few conventions for the clarity and the brevity of the notation.

1. If necessary, we explicitly denote the dimensions of the matrix in the subscript $A_{m\times n}$.

2. If most of the elements of the matrix are zero, we either completely omit showing them, or we replace zeros with dots, e.g.,

$$\begin{bmatrix} 4 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 4 & & \\ 2 & & \\ & 1 & 1 \end{bmatrix} = \begin{bmatrix} 4 & \cdot & \cdot \\ 2 & \cdot & \cdot \\ \cdot & 1 & 1 \end{bmatrix}. \tag{4.4}$$

3. If the matrix A has dimensions $1 \times n$ or $n \times 1$, then it is a row or a column *vector*, respectively. By convention, we use small letters in boldface to refer to column vectors, e.g., $\mathbf{x} = (x_0, \ldots, x_{n-1})^T$. Row vectors are denoted as $\mathbf{x}^T$.

4. The analogy between sequences and vectors should be obvious. Given a sequence $\{x_k\}_0^{n-1}$, the corresponding vector is $\mathbf{x} = (x_0, \ldots, x_{n-1})^T$

Matrix-vector product. Given a matrix $A = [a_{i,j}] \in \mathbb{C}^{m\times n}$ and a column vector $\mathbf{x} \in \mathbb{C}^{n\times 1}$, the matrix-vector product is defined by

$$\mathbf{y} = (y_0, \ldots, y_{m-1})^T = A\mathbf{x}, \quad y_i = \sum_{j=0}^{n-1} a_{i,j}x_j \tag{4.5}$$

Matrix representation of linear transformations. We are now ready to draw the connection between linear transformations and matrices. Given a linear transformation $T : \mathbb{C}^n \mapsto \mathbb{C}^m$ and the standard bases (4.2) for both $\mathbb{C}^n$ and $\mathbb{C}m$, we know that

$$\mathbf{t}_i = T\mathbf{e}_i, \quad i = 0, \ldots, n-1 \tag{4.6}$$

is unique. Then, for any $\mathbf{x} \in \mathbb{C}^n$, we can write

$$\mathbf{y} = T\mathbf{x} = x_0\mathbf{t}_0 + x_1\mathbf{t}_1 + \cdots + x_{n-1}\mathbf{t}_{n-1}, \tag{4.7}$$

where

$$\mathbf{x} = x_0\mathbf{e}_0 + x_1\mathbf{e}_1 + \cdots + x_{n-1}\mathbf{e}_{n-1}. \tag{4.8}$$

The proof is elementary and follows directly from the linearity of the transform (4.1). The representation of the vector $\mathbf{x}$ with respect to the standard basis is $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})^T$ where $x_i, \quad i = 0, \ldots, n$ are the coordinates. Equivalently, we can represent images of the standard basis $\mathbf{t}_j, \quad j = 0, \ldots n$ as vectors $\mathbf{t}_j = (t_{0,j}, \ldots, t_{m,j})^T$. By convention, the representations are column rather then row vectors. We can then represent (4.7) by the matrix vector product

$$\mathbf{y} = T\mathbf{x} = \begin{bmatrix} t_{0,0} & \cdots & t_{0,n-1} \\ \vdots & \ddots & \vdots \\ t_{m-1,0} & \cdots & t_{m-1,n-1} \end{bmatrix} \mathbf{x}. \tag{4.9}$$

where $\mathbf{y} = (y_0, y_1, \ldots, y_m - 1)^T$ and the columns of the matrix T are given by (4.6) w.r.t the standard basis of $\mathbb{C}^m$.

Matrix generating function. An indirect way to define matrices is by using matrix generating functions. A matrix generating function $f(i,j)$ is the mapping of the form

$$a : \begin{cases} \{0, \ldots, m-1\} \times \{0, \ldots, n-1\} \to \mathbb{C} \\ (i,j) \mapsto a(i,j) \end{cases} . \tag{4.10}$$

Given a generating function for the matrix A, the elements of the matrix are images of the matrix generating function $a_{i,j} = a(i,j)$ where $i = 0, \ldots, m-1$ and $j = 0, \ldots, n-1$. We use the same notation for the element $a_{i,j}$ and the image $a(i,j)$ hoping that it will not confuse the reader. We often use a shorthand notation for matrices $A = [a(i,j)]_{m \times n}$, or simply $A = [a_{i,j}]_{m \times n}$. This notation is especially useful when the generating function is given analytically.

Matrix transpose. Given a matrix $A = [a_{i,j}]_{m \times n}$, the *transposed* matrix is defined as $A^T = [a_{j,i}]_{n \times m}$.

4.1.2 Special matrices

We define a few special matrices that have simple structure.

Diagonal matrix. A diagonal matrix $\mathrm{diag}(\mathbf{a})$ is a square matrix of size $n \times n$ where the n diagonal elements are specified by the vector $\mathbf{a} = (a_0, \ldots, a_{n-1})^T \in \mathbb{C}^{n \times 1}$ and the rest of the matrix is zero:

$$\mathrm{diag}(\mathbf{a}) = \begin{bmatrix} a_0 & & \\ & \ddots & \\ & & a_{n-1} \end{bmatrix} \tag{4.11}$$

Given a generating function

$$f : \begin{cases} \{0, \ldots, n-1\} \to \mathbb{C} \\ i \mapsto f(i) \end{cases} .$$

for the diagonal elements $a_i = f(i)$, we write $\mathrm{diag}(\mathbf{a}) = \mathrm{diag}(f)_n$.

Zero matrix. The zero matrix $0_{m \times n}$ is an $m \times n$ matrix of zeros,

$$0_{m \times n} = [0]_{m \times n} \tag{4.12}$$

Identity matrix. The identity matrix I_n is the diagonal matrix with all n diagonal elements one and all off-diagonal elements zero, i.e.,

$$\mathrm{I}_n = \mathrm{diag}(\mathbf{1}), \quad \mathbf{1} = (1, \ldots, 1). \tag{4.13}$$

Opposite identity matrix. The opposite identity matrix, or the flip matrix, is the square matrix of ones on the opposite diagonal, i.e.,

$$J_n = \begin{bmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{bmatrix} \tag{4.14}$$

4.2 Constructs

We define several matrix operations that are instrumental in expressing DSP algorithms including discrete filtering and wavelet transform algorithms. Since these operations mathematically describe the structure of the algorithms and define basic implementation strategies, we call them mathematical *constructs*. They provide the binding glue between basic building parts of algorithms.

4.2.1 Basic matrix constructs

The following is a list of well-known matrix constructs that we provide here for completeness. We also establish several notational conventions that we use throughout this document.

Scalar product. Given an $m \times n$ matrix $A = [a_{i,j}]_{m\times n}$ over a field $\mathbb{F}$ and the element $c \in \mathbb{F}$, the scalar product $c \cdot A$ is defined as $c \cdot A = [c \cdot a_{i,j}]_{m\times n}$

Matrix of matrices. Let the matrices $A_{i,j} \in \mathbb{C}^{m_{i,j}\times n_{i,j}}$, where $i = 0, \ldots, r-1$, and $j = 0, \ldots, s-1$, be concatenated into a larger rectangular array C of $r \times s$ blocks. We assume that the dimensions of matrices $A_{i,j}$ agree, i.e., that

$$\begin{aligned} m_i &= m_{i,0} = m_{i,1} = \cdots = m_{i,s}, \quad \forall i \in \{0, \ldots, r-1\} \\ n_j &= n_{0,j} = n_{1,j} = \cdots = n_{r,j}, \quad \forall j \in \{0, \ldots, s-1\} \end{aligned}$$

Then the matrix C of dimensions $(\sum_{k=0}^{r-1} m_k) \times (\sum_{l=0}^{s-1} n_l)$ is a *matrix of matrices* or a *block matrix*.

$$C = \begin{bmatrix} A_{0,0} & \cdots & A_{0,s-1} \\ \vdots & \ddots & \vdots \\ A_{r-1,0} & \cdots & A_{r-1,s-1} \end{bmatrix} = [A_{i,j}]_{r\times s} \tag{4.15}$$

If C is a block matrix with $1 \times s$ or $r \times 1$ blocks then we call it the *horizontal stacking* or the *vertical stacking* of matrices, respectively. We sometimes denote the stacking of matrices as

$$C = \left[\, A \,|\, B \,\right], \quad \text{and} \quad C = \left[\begin{array}{c} A \\ \hline B \end{array}\right]. \tag{4.16}$$

We also define the iterative stacking using the associativity of the stacking operations.

$$\left[\begin{array}{c} \\ \hline \\ \hline \\ \end{array}\right]_{i=0}^{r-1} A_i = \begin{bmatrix} A_0 \\ \vdots \\ A_{r-1} \end{bmatrix} = [A_i]_{r\times 1} \tag{4.17a}$$

$$\left[\begin{array}{c|c|c} & & \end{array}\right]_{i=0}^{s-1} A_i = \begin{bmatrix} A_0 & \cdots & A_{s-1} \end{bmatrix} = [A_i]_{1\times s} \tag{4.17b}$$

Sums of matrices. The matrix sum of $m \times n$ matrices $A = [a_{i,j}]_{m\times n}$ and $B = [b_{i,j}]_{m\times n}$ over a field $\mathbb{F}$ is performed as the element-wise sum in $\mathbb{F}$

$$C = A + B = [a_{i,j} + b_{i,j}]_{m\times n} = [c_{i,j}]_{m\times n}. \tag{4.18}$$

Thus, all the properties of the addition in $\mathbb{F}$ translate directly to matrix sums:

1. $A + (B + C) = (A + B) + C$ (*Associativity*) (4.19a)
2. $A + B = B + A$ (*Commutativity*) (4.19b)
3. $A + 0_{n\times m} = A, \quad A + (-1) \cdot A = 0_{n\times m}$ (*Zero and negative matrix*) (4.19c)

Using the associativity property, we define the *iterative matrix sum* simply as

$$\sum_{i=0}^{N-1} A_i = A_0 + A_1 + \cdots + A_{N-1} \tag{4.20}$$

Matrix composition. Let $A = [a_{i,j}]_{m\times l}$ be an $m \times l$ matrix and $B = [b_{i,j}]_{l\times n}$ be an $l \times n$ matrix. Then the *composition* or the *product* of matrices A and B is an $m \times n$ matrix C given by

$$C = A \cdot B = \left[\sum_{k=0}^{l-1} a_{i,k} \cdot b_{k,j}\right] = [c_{i,j}]_{m\times n} \tag{4.21}$$

Basic properties of the matrix product are

1. $A \cdot (B \cdot C) = (A \cdot B) \cdot C$ (*Associativity*) (4.22a)
2. $A \cdot \mathrm{I}_n = A = \mathrm{I}_m \cdot A, \quad A \in \mathbb{C}^{m\times n}$ (*Identity matrix*) (4.22b)
3. $A \cdot A^{-1} = A^{-1} \cdot A = \mathrm{I}_n, \quad A \in \mathbb{C}^{n\times n}$ (*Inverse matrix*) (4.22c)

If it exists, A^{-1} is the inverse of the square matrix A by respect to multiplication. A square matrix A is said to be *orthogonal* if $A^T = A^{-1}$.

Conjugation of matrices. The *conjugation* of a square matrix A_n with another square matrix B_n is the similarity transformation

$$C_n = A^B = B^{-1} \cdot A \cdot B \tag{4.23}$$

We list some of the useful properties of the conjugation

1. $C = A^B \Rightarrow C^{B^{-1}} = A$ (*Symmetry*) (4.24a)
2. $\left(\prod_{i=0}^{N-1} A_i\right)^B = \prod_{i=0}^{N-1} A_i^B$ (*Distributivity*) (4.24b)
   $\left(\sum_{i=0}^{N-1} A_i\right)^B = \sum_{i=0}^{N-1} A_i^B$
3. $\left(\left(A^{B_1}\right)^{B_2}\right)^{\cdots B_{N-1}} = A^{B_1 \cdot B_2 \cdots B_{N-1}}$ (*Iterative conjugation*) (4.24c)
4. $Ax = \lambda x \Rightarrow A^B y = \lambda y$, where $y = B^{-1}x$ (*Conservation of the spectrum*) (4.24d)

The last property shows that the matrix spectrum is preserved by the conjugation and, hence, also is the matrix determinant $\det A = \det A^B$.

Conjugation can be seen as a change of basis specified by B of the linear transformation represented by A. In the case B is a permutation, conjugation is a reordering of the basis.

4.2.2 Subspace decomposition constructs

We introduce constructs that decompose linear operators on vector spaces into multiple operators on subspaces. They are instrumental in expressing many important structured matrices and serve as a tool for decomposing large structured matrices leading to fast algorithms.

Direct sum. The *direct sum* of matrices $A_{m_1 \times n_1}$ and $B_{m_2 \times n_2}$ is the $(m_1 + m_2) \times (n_1 + n_2)$ block diagonal matrix

$$C = A \oplus B = \left[\begin{array}{c|c} A & 0 \\ \hline 0 & B \end{array}\right] \tag{4.25}$$

Through the associaticity property we define the iterative direct sum as

$$\bigoplus_{i=0}^{N-1} A_i = \begin{bmatrix} A_0 & & & \\ & A_1 & & \\ & & \ddots & \\ & & & A_{N-1} \end{bmatrix} \tag{4.26}$$

If $A_i, B_i, i = 0, \ldots, N-1$ are all $n \times n$ matrices then the following holds true

$$1. \quad \left(\bigoplus_{i=0}^{N-1} A_i\right)\left(\bigoplus_{i=0}^{N-1} B_i\right) = \bigoplus_{i=0}^{N-1} (A_i \cdot B_i) \tag{4.27a}$$

$$2. \quad \left(\bigoplus_{i=0}^{N-1} A_i\right)^{\left(\bigoplus_{i=0}^{N-1} B_i\right)} = \bigoplus_{i=0}^{N-1} A_i^{B_i} \tag{4.27b}$$

Tensor product. The *tensor* or *Kronecker* product of matrices $A = [a_{i,j}]_{m_1 \times n_1}$ and $B = [b_{i,j}]_{m_2 \times n_2}$ is the $m_1 m_2 \times n_1 n_2$ block matrix C of the following form

$$C = A \otimes B = \begin{bmatrix} a_{0,0}B & \ldots & a_{0,n}B \\ \vdots & \ddots & \vdots \\ a_{m,0}B & \ldots & a_{m,n}B \end{bmatrix} \tag{4.28}$$

We list some of the most important properties of the tensor product. The proof can be found in [32].

$$A \otimes \mathrm{I}_1 = \mathrm{I}_1 \otimes A = A \qquad (Identity) \tag{4.29a}$$

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C \qquad (Associativity) \tag{4.29b}$$

$$(A + B) \otimes C = (A \otimes C)(B \otimes C) \qquad (Bilinearity) \tag{4.29c}$$
$$A \otimes (B + C) = (A \otimes B)(A \otimes C)$$

$$(A \otimes B)(C \otimes D) = AC \otimes BD \qquad (Resolution\ of\ the\ product) \tag{4.29d}$$

$$\left(\bigoplus_{i=0}^{n-1} A_i\right) \otimes B = \bigoplus_{i=0}^{n-1} (A_i \otimes B) \qquad (Distributivity\ over\ direct\ sum) \tag{4.29e}$$

The proof can be found in [32].
Using the above properties, we can derive several useful identities

1. $\mathrm{I}_m \otimes \mathrm{I}_n = \mathrm{I}_{mn}$ (4.30a)
2. $(A \otimes B)^T = A^T \otimes B^T$ (4.30b)
3. $A^{-1} \otimes B^{-1} = (A \otimes B)^{-1}$ (4.30c)
4. $A_{m1\times n_1} \otimes B_{m_2\times n_2} = (A \otimes \mathrm{I}_{m_2})(\mathrm{I}_{n_1} \otimes B) = (\mathrm{I}_{m_1} \otimes A)(B \otimes \mathrm{I}_{n_2})$ (4.30d)
5. $\left(\bigotimes_{i=0}^{n-1} A_i\right)\left(\bigotimes_{i=0}^{n-1} B_i\right) = \bigotimes_{i=0}^{n-1}(A_i \cdot B_i)$ (4.30e)
6. $\prod_{i=0}^{n-1} A_i \otimes B_i = \prod_{i=0}^{n-1} A_i \otimes \prod_{i=0}^{n-1} B_i$ (4.30f)

The tensor product is frequently encountered in expressing DSP algorithms as it captures the structure of many algorithms in a concise and unified way. Closely related to the tensor product is a family of permutations with which it shares many important properties. We defer the discussion of these properties until later in this chapter.

4.2.3 Overlapped constructs

We introduce two overlapped constructs that are required for expressing filtering and wavelet algorithms. Overlapped matrices arise in convolution operators with Toeplitz structure, as well as in block transforms. The overlapped constructs represent a family of binary operators that are parameterized by the integral parameter k.

Overlapped direct sum. The *row overlapped direct sum* and the *column overlapped direct sum* of matrices $A = [a_{i,j}]_{m_1\times n_1}$ and $B = [b_{i,j}]_{m_2\times n_2}$ are matrices $C = [c_{i,j}]_{(m_1+m_2)\times(n_1+|n_2-k|)}$ and $D = [d_{i,j}]_{(m_1+|m_2-k|)\times(n_1+n_2)}$ defined, respectively, as

$$C = A \oplus_k B = \begin{bmatrix} \boxed{\text{A}} & \\ & \boxed{\text{B}} \end{bmatrix}, \quad D = A \oplus^k B = \begin{bmatrix} \boxed{\text{A}} & \\ & \boxed{\text{B}} \end{bmatrix}, \tag{4.31}$$

where the parameter k specifies the number of overlapping columns or rows, respectively.

In general, the overlapped direct sum can be defined for any $k \in \mathbb{Z}$. A negative overlap k simply means that there are $|k|$ zero columns (rows) between matrices A and B. This is illustrated in (4.32).

$$A \oplus_k B = \begin{bmatrix} \boxed{\text{A}} & \\ & \boxed{\text{B}} \end{bmatrix}, k < 0, \qquad A \oplus^k B = \begin{bmatrix} \boxed{\text{A}} & \\ & \boxed{\text{B}} \end{bmatrix}, k < 0, \tag{4.32}$$

If, on the other hand, $k > n_2$ for the row overlapped direct sum,or $k > m_2$ for the column overlapped direct sum, then the size of the resulting matrix increases with k, which we can see from

the dimensions of matrices $C_{(m_1+m_2)\times(n_1+|n_2-k|)}$ and $D_{(m_1+|m_2-k|)\times(n_1+n_2)}$. This case is shown in (4.33).

$$A \oplus_k B = \left[\begin{array}{c} \boxed{\text{A}} \\ \boxed{\text{B}} \end{array}\right], k > n_2 \qquad A \oplus^k B = \left[\begin{array}{cc} & \boxed{\text{B}} \\ \boxed{\text{A}} & \end{array}\right], k > m_2 \tag{4.33}$$

By definition, the reference point of the overlap is the bottom right corner of the first matrix in the overlapped direct sum. From (4.33) we observe that the reference point of the resulting matrix does not correspond to the reference point of the last matrix in the summation. It implies that many of the properties, such as associativity of the overlapped direct sum, do not hold in this case and require special treatment. This irregularity is inherent to all overlapping constructs regardless of the definition or the choice of the reference point.

The set of all parameterized operators is bipartitioned into two subsets. We, therefore, consider two separate cases: the "regular" case with $k \leq n_2$ or $k \leq m_2$, and the "irregular" case with $k > n_2$ or $k > m_2$.

The following basic properties hold for both cases:

1. $A \oplus_0 B = A \oplus^0 B = A \oplus B$ (4.34a)
2. $(A \oplus_k B)^T = A^T \oplus^k B^T$ (4.34b)
   $(A \oplus^k B)^T = A^T \oplus_k B^T$
3. $A \oplus_k B = (A \oplus B)(I_{n_1} \oplus_k I_{n_2})$ (4.34c)
   $A \oplus^k B = (I_{m_1} \oplus^k I_{m_2})(A \oplus B)$

The following lemmas provide conditions under which the associativity property holds.

Lemma 4.1 (Associativity of the overlapped direct sum). *In the regular case* (4.31)*, the associativity property holds for both the row and the column overlapped direct sum.*

$$A \oplus^{k_1} B \oplus^{k_2} C = A \oplus^{k_1} (B \oplus^{k_2} C) = (A \oplus^{k_1} B) \oplus^{k_2} C \qquad (\textit{Column-column}) \tag{4.35a}$$
$$A \oplus_{k_1} B \oplus_{k_2} C = A \oplus_{k_1} (B \oplus_{k_2} C) = (A \oplus_{k_1} B) \oplus_{k_2} C \qquad (\textit{Row-row}) \tag{4.35b}$$
$$A \oplus_{k_1} B \oplus^{k_2} C = A \oplus_{k_1} (B \oplus^{k_2} C) = (A \oplus_{k_1} B) \oplus^{k_2} C \qquad (\textit{Row-column}) \tag{4.35c}$$
$$A \oplus^{k_1} B \oplus_{k_2} C = A \oplus^{k_1} (B \oplus_{k_2} C) = (A \oplus^{k_1} B) \oplus_{k_2} C \qquad (\textit{Column-row}) \tag{4.35d}$$

The proof is intuitive from the illustration (4.31) and we omit it here.

In the irregular case (4.33), only the left associativity holds. The right associativity can still be obtained by appropriate mappings of the overlap parameters. This represents a generalization in the sense that it incorporates the regular associativity as a special case.

Lemma 4.2 (Left associativity in the irregular case). *For matrices $A_{m_1\times n_1}$, $B_{m_2\times n_2}$, and $C_{m_3\times n_3}$ we have*

$$A \oplus^{k_1} B \oplus^{k_2} C = (A \oplus^{k_1} B) \oplus^{k_2} C \quad = A \oplus^{k_1+\langle k_2-m_2-\langle k_1-m_2\rangle\rangle} (B \oplus^{k_2-\langle k_1-m_2\rangle} C) \tag{4.36a}$$
$$A \oplus_{k_1} B \oplus_{k_2} C = (A \oplus_{k_1} B) \oplus_{k_2} C \quad = A \oplus_{k_1+\langle k_2-n_2-\langle k_1-m_2\rangle\rangle} (B \oplus_{k_2-\langle k_1-n_2\rangle} C) \tag{4.36b}$$

where

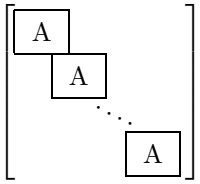
$$\langle k \rangle = \begin{cases} k, & k \geq 0 \\ 0, & k < 0 \end{cases} . \tag{4.37}$$

Proof. We start with the row overlapped case. The overlapped direct sum is left associative by definition. The right parenthesizing requires the change of the overlap parameter. We first notice that the matrix C should be overlapped with the matrix B by k_2 subtracted by the shift in the reference (bottom-right) point going from $A \oplus_{k_1} B$ to B, which is equal to $\langle k_1 - m_2 \rangle$. Second, the overlap between matrices A and B need to be readjusted by adding the shift in the reference (upper-left) point when going from B to $B \oplus^{k_2 - \langle k_1 - m_2 \rangle} C$, which is equal to $\langle (k_2 - \langle k_1 - m_2 \rangle) - m_2 \rangle$. For the column overlap case the proof is similar. ■

Overlapped tensor product. We use the overlapped direct sum to define the *row overlapped tensor product* and the *column overlapped tensor product*,respectively, as

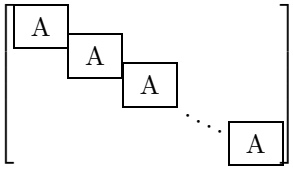
Row overlapped tensor

$$\mathrm{I}_s \otimes_k A = \bigoplus_{i=0}^{s-1}{}_k A$$



Column overlapped tensor

$$\mathrm{I}_s \otimes^k A = \bigoplus_{i=0}^{s-1}{}^k A$$



(4.38)

Most of the properties of the overlapped tensor product follow directly from the properties of the overlapped direct sum so we list them without proof. Let A be an $m \times n$ matrix. Then

$$(\mathrm{I}_s \otimes_k A)^T = \mathrm{I}_s \otimes^k A^T \tag{4.39a}$$

$$\mathrm{I}_s \otimes_0 A = \mathrm{I}_s \otimes^0 A = \mathrm{I}_s \otimes A \tag{4.39b}$$

$$\begin{aligned} \mathrm{I}_s \otimes_k A &= (\mathrm{I}_s \otimes A)(\mathrm{I}_s \otimes_k \mathrm{I}_n) \\ \mathrm{I}_s \otimes^k A &= (\mathrm{I}_s \otimes^k \mathrm{I}_m)(\mathrm{I}_s \otimes A). \end{aligned} \tag{4.39c}$$

Lemma 4.3 (Composition of overlapped tensors in the regular case). *Let us assume that $k \leq n$ for $\mathrm{I}_s \otimes_k A$ and that $k \leq m$ for $\mathrm{I}_s \otimes^k A$. Then we have a regular row and column overlapped tensor product and the following properties hold*

$$\begin{aligned} \mathrm{I}_s \otimes_k (\mathrm{I}_t \otimes_k A) &= \mathrm{I}_{st} \otimes_k A \\ \mathrm{I}_s \otimes_k (\mathrm{I}_t \otimes^k A) &= \mathrm{I}_{st} \otimes^k A \end{aligned} \tag{4.40a}$$

$$\begin{aligned} \mathrm{I}_s \otimes_{k_1} (\mathrm{I}_t \otimes_{k_2} A) &= (\mathrm{I}_{st} \otimes A)(\mathrm{I}_s \otimes_{k_1} (\mathrm{I}_t \otimes_{k_2} \mathrm{I}_n)) \\ \mathrm{I}_s \otimes^{k_1} (\mathrm{I}_t \otimes^{k_2} A) &= (\mathrm{I}_s \otimes^{k_1} (\mathrm{I}_t \otimes^{k_2} \mathrm{I}_m))(\mathrm{I}_{st} \otimes A) \end{aligned} \tag{4.40b}$$

Proof. The first two identities follow directly from the associativity property of the overlapped direct

sum (4.35). The proof of (4.40b) goes as follows:

$$\begin{aligned}
\mathrm{I}_s \otimes_{k_1} (\mathrm{I}_t \otimes_{k_2} A) &\overset{(p.1)}{=} (\mathrm{I}_s \otimes (\mathrm{I}_t \otimes_{k_2} A))(\mathrm{I}_s \otimes_{k_1} \mathrm{I}_{t(n-k_2)+k_2}) \\
&\overset{(p.1)}{=} (\mathrm{I}_s \otimes (\mathrm{I}_t \otimes A)(\mathrm{I}_t \otimes_{k_2} \mathrm{I}_n))(\mathrm{I}_s \otimes_{k_1} \mathrm{I}_{t(n-k_2)+k_2}) \\
&\overset{(p.2)}{=} (\mathrm{I}_{st} \otimes A)(\mathrm{I}_s \otimes (\mathrm{I}_t \otimes_{k_2} \mathrm{I}_n))(\mathrm{I}_s \otimes_{k_1} \mathrm{I}_{t(n-k_2)+k_2}) \\
&\overset{(p.1)}{=} (\mathrm{I}_{st} \otimes A)(\mathrm{I}_s \otimes_{k_1} (\mathrm{I}_t \otimes_{k_2} \mathrm{I}_n))
\end{aligned}$$

The labels over the equality signs refer to the identity used in the derivation. Label (p.1) denotes property (4.39c) and label (p.2) denotes property (4.29d). The proof for the column overlapped tensor product is similar. ■

4.3 Operators

We introduce a few classes of parameterized *operators*. Even though transforms are also linear operators, in our framework we treat them separately to indicate that they are suitable for decomposition. Unlike transforms, operators typically perform simple reordering, copying, and localized operations on data, using relatively small number of arithmetic operations. Because of their simple structure, operators are normally not decomposed to obtain fast algorithms.

4.3.1 Permutations

We start by defining operators that perform permutations of sequences.

General permutation. Given a finite set A, a permutation is a one-one mapping of the set A onto itself.

$$\pi \ : \ A \mapsto A$$

Two sets A and B are equivalent if there is a one to one correspondence between them, i.e., if there exists a bijective mapping from A to B. Since every finite set A of size n is equivalent to the set $S = \{0, 1, \ldots, n-1\}$, we can define any permutation on A uniquely as a permutation on S. This can be formally achieved by ordering and indexing the elements of A to obtain sequences or vectors of elements $(a_0, \ldots, a_{n-1})$. The permutation π is then defined by the 1-1 mapping

$$\pi \ : \begin{cases} \{0, \ldots, n-1\} \mapsto \{0, \ldots, n-1\} \\ i \mapsto \pi(i) \end{cases} \tag{4.41}$$

We will use the above definition throughout the rest of the thesis. It implies that a permutation represents a mapping of *indices* of elements of A, not the elements themselves. To clarify what this exactly means, we use the matrix representation of permutations. As before, we assume that the linear transformations of sequences are represented by matrices that operate from the left on column vectors. The permutation matrix is then defined using the following generating function

$$\hat{\pi} \ : \begin{cases} \{0, \ldots, n-1\} \times \{0, \ldots, n-1\} \mapsto \{0, 1\} \\ (i, j) \mapsto \hat{\pi}(i, j) \end{cases}, \tag{4.42}$$

where

$$\hat{\pi}(i,j) \;= \begin{cases} 1, & j = \pi(i) \\ 0, & \text{else} \end{cases} \tag{4.43}$$

The matrix generating function $\hat{\pi}$ generates the permutation matrix $[\hat{\pi}_{i,j}]$. To avoid reference to the generating function to obtain the permutation matrix, we simply write $\mathrm{mat}(\pi) = [\hat{\pi}_{i,j}]$.

The permutation matrix can be readily obtained as

$$P = \mathrm{mat}(\pi) = \begin{bmatrix} \mathbf{e}_{\pi(0)}^T \\ \mathbf{e}_{\pi(1)}^T \\ \vdots \\ \mathbf{e}_{\pi(n-1)}^T \end{bmatrix} \tag{4.44}$$

We refer to a permutation matrix simply as permutation.

EXAMPLE 4.1. *Let π be a permutation on $S_4 = \{0, \ldots, 3\}$ defined as*

$$\pi : \Big\{ \; 0 \mapsto 0, \quad 1 \mapsto 3, \quad 2 \mapsto 1, \quad 3 \mapsto 2 \quad ,$$

Using (4.44)*, we obtain the matrix representation of π as*

$$P = \mathrm{mat}(\pi) = \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \end{bmatrix}.$$

The reordered sequence is simply $(x_0, x_3, x_1, x_2)^T = P \cdot \mathbf{x}$.

Permutation matrices are orthogonal with exactly one non-zero element in each row and column.

$$P = [\hat{\pi}_{i,j}] \;\Rightarrow\; P^{-1} = [\hat{\pi}_{i,j}^{-1}] = P^T, \tag{4.45}$$

where

$$\hat{\pi}^{-1}(i,j) \;= \begin{cases} 1, & j = \pi^{-1}(i) \\ 0, & \text{else} \end{cases} \quad .$$

Stride permutation. The *stride permutation* is frequently encountered in representations of DSP algorithms. It is determined by the parameter s called the *stride* and the size of the permuted set $n = r \cdot s$.

$$l_s^{rs} \;: \begin{cases} \{0, \ldots, n-1\} \mapsto \{0, \ldots, n-1\} \\ i \mapsto i \cdot s \bmod n, & i = 0, \ldots, n-2 \\ i \mapsto n-1, & i = n-1 \end{cases} \quad , \quad \mathrm{L}_s^{rs} = \mathrm{mat}(l_s^{rs}) \tag{4.46}$$

The stride permutation L_s^{rs} reorders a sequence of length n by collecting the elements at the stride s.

EXAMPLE 4.2.

$$\mathrm{L}^6_2 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix}, \quad (x_0, x_2, x_4, x_1, x_3, x_5)^T = \mathrm{L}^6_2\, \mathbf{x}$$

The most fundamental property of the stride permutation is the blocking property. Let A be an $m \times n$ matrix $A = [a_{i,j}]_{m \times n}$, and let r and s be stride parameters such that $r|m$ and $s|n$. Then A is blocked by two stride permutations in the following way.

$$A^{[r,s]} = \mathrm{L}^m_r\, A\, \mathrm{L}^n_{n/s} = \begin{bmatrix} A_{0,0} & \cdots & A_{0,s-1} \\ \vdots & \ddots & \vdots \\ A_{r-1,0} & \cdots & A_{r-1,s-1} \end{bmatrix} = [A_{i,j}]_{r \times s} \tag{4.47}$$

where

$$A_{k,l} = [a_{ri+k,sj+l}]_{\frac{m}{r} \times \frac{n}{s}}. \tag{4.48}$$

For a square matrix $A = [a_{i,j}]_{n \times n}$, it immediately follows that

1. $(L^{rs}_s)^T = L^{rs}_r$ (4.49a)

2. $A^{[s,s]} = A^{\mathrm{L}^{rs}_r}, \quad n = rs$ (4.49b)

Several properties of the stride permutation, in conjunction with the tensor product and the overlapped tensor product, are essential for deriving fast algorithms for many DSP transforms, including filtering and wavelet transforms.

The stride permutation is inextricably tied to the tensor product due to the following property

$$(\mathrm{I}_s \otimes A)^{\mathrm{L}^{ns}_n} = \mathrm{L}^{ns}_n (\mathrm{I}_s \otimes A)\, \mathrm{L}^{ns}_s = A \otimes \mathrm{I}_s, \quad A = [a_{i,j}]_{n \times n} \tag{4.50a}$$

The formal proof can be found in [35]. This property can be easily generalized for rectangular matrices to

$$(\mathrm{I}_s \otimes A)^{[s,s]} = \mathrm{L}^{ms}_m (\mathrm{I}_s \otimes A)\, \mathrm{L}^{ns}_s = A \otimes \mathrm{I}_s, \quad A = [a_{i,j}]_{m \times n} \tag{4.50b}$$

This generalizes (4.50a), which follows when $m = n$. Further generalization is the blocking property of the stride permutation shown in (4.47), which we denote as $A^{[r,s]}$ similar to the conjugation operation.

The stride permutation does not go hand in hand with the overlapped tensor products as well as with the tensor product. Still, we can derive a useful property that decomposes the overlapped tensor products into blocks of smaller overlapped tensor products.

Let A be an $m \times n$ matrix, and let k be the overlap. If the two stride parameters r and s satisfy $r|m$, $s|n$, and $s|k$, then

$$\mathrm{L}^{tm}_r (\mathrm{I}_t \otimes_k A)\, \mathrm{L}^{t(n-k)+k}_{n-k+\frac{k}{s}} = \begin{bmatrix} \mathrm{I}_t \otimes_{k/s} A_{0,0} & \cdots & \mathrm{I}_t \otimes_{k/s} A_{0,s-1} \\ \vdots & \ddots & \vdots \\ \mathrm{I}_t \otimes_{k/s} A_{r-1,0} & \cdots & \mathrm{I}_t \otimes_{k/s} A_{r-1,s-1} \end{bmatrix} \tag{4.51a}$$

where $A_{i,j}$ are defined in (4.48). To make the relation more clear, we can write the stride permutation $\mathrm{L}^{t(n-k)+k}_{n-k+k/s}$ as $(\mathrm{L}^{s(n-k)+k}_{s})^T$. The relation for the column overlapped tensor product is obtained by simple transposition of (4.51a). Let $B = A^T$ be an $n \times m$ matrix. Then,

$$\mathrm{L}^{t(n-k)+k}_{s}(\mathrm{I}_t \otimes^k A)\,\mathrm{L}^{tm}_{\frac{tm}{r}} = \left[\mathrm{I}_t \otimes_{k/s} B_{i,j}\right]_{r\times s} \tag{4.51b}$$

A few special cases of the properties (4.51) require special attention. We discuss those cases in the next chapter when we provide rules for FIR filtering algorithms.

Linear permutation. The *linear permutation* is very similar to the stride permutation Given the linearity parameter a and the size of the permuted set n, where $a \nmid n$, we define linear permutation by

$$\bar{l}^n_a : \begin{cases} \{0,\ldots,n-1\} \mapsto \{0,\ldots,n-1\} \\ i \mapsto i \cdot a \bmod n, \quad i = 0,\ldots,n-1 \end{cases}, \quad \bar{\mathrm{L}}^n_a = \mathrm{mat}(\bar{l}^n_a) \tag{4.52}$$

The parameter a is the stride parameter.

The difference between the linear and the stride permutation comes from the fact that there is no need to treat the $n-1$ element separately. When there is no danger of confusing the reader, we shall sometimes use the linear permutation notation to represent a stride permutation for the sake of brevity. So, by slightly abusing the notation, we shall occasionally write

$$\bar{\mathrm{L}}^n_a = \mathrm{L}^n_a,$$

when $a \mid n$.

Example 4.3.

$$\bar{\mathrm{L}}^5_3 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & 1 & \cdot & \cdot \end{bmatrix}, \quad (x_0, x_3, x_1, x_4, x_2)^T = \bar{\mathrm{L}}^5_3\,\mathbf{x}$$

Example 4.4.

$$\bar{\mathrm{L}}^n_1 = \mathrm{I}_n,$$

so, the identity matrix I_n represents the linear permutation $\iota : i \mapsto i$ on the set $\{0,\ldots,n-1\}$.

Affine permutation. The *affine permutation* is similar to the linear permutation with one additional parameter, the offset b.

Given the linearity parameter a, the offset b and the size of the permuted set n, where $a \nmid n$, we define the affine permutation by

$$\bar{l}^n_{a,b} : \begin{cases} \{0,\ldots,n-1\} \mapsto \{0,\ldots,n-1\} \\ i \mapsto (i \cdot a + b) \bmod n, \quad i = 0,\ldots,n-1 \end{cases}, \quad \bar{\mathrm{L}}^n_{a,b} = \mathrm{mat}(\bar{l}^n_{a,b}) \tag{4.53}$$

Example 4.5.

$$\bar{\mathrm{L}}^3_{2,2} = \mathrm{J}_3, \quad (x_2, x_1, x_0)^T = \mathrm{J}_3\,\mathbf{x}$$

The example is an instantiation of the following identity.

$$\bar{\mathrm{L}}^n_{n-1,n-1} = \mathrm{J}_n \tag{4.54}$$

The opposite identity matrix represents an affine permutation that reverses the order of the elements. For this reason, we sometimes call it the flip matrix.

Chinese remainder theorem permutation. The *Chinese remainder theorem permutation* arises from the decomposition of $\mathbb{Z}/n$ into $\mathbb{Z}/n_1 \times \mathbb{Z}/n_2$ using the Chinese remainder theorem (CRT), where $n = n_1 n_2$ are relatively prime factors of n. The CRT states that any number $a \in \mathbb{Z}/n$ can be uniquely represented by $a_1 \in \mathbb{Z}/n_1$ and $a_2 \in \mathbb{Z}/n_2$ using the isomorphism

$$f : \begin{cases} \{0, \ldots, n-1\} \mapsto \{0, \ldots, n_1 - 1\} \times \{0, \ldots, n_2 - 1\} \\ a \mapsto (a \bmod n_1,\ a \bmod n_2) \end{cases}$$

with the inverse given by

$$f^{-1} : \begin{cases} \{0, \ldots, n_1 - 1\} \times \{0, \ldots, n_2 - 1\} \mapsto \{0, \ldots, n-1\} \\ (a_1, a_2) \mapsto a_1 e_1 + a_2 e_2 \bmod n \end{cases}$$

where e_1 and e_2 are the idempotents of $\mathbb{Z}/n_1 \times \mathbb{Z}/n_2$. Since n_1 and n_2 are relatively prime, then by the Euclidean algorithm $n_1 f_1 + n_2 f_2 = 1$. The idempotents can be obtained as $e_1 = n_2 f_2 \bmod n$ and $e_2 = n_1 f_1 \bmod n$. The details are provided in [35]. If we order the pairs (a_1, a_2) from the product set $\{0, \ldots, n_1 - 1\} \times \{0, \ldots, n_2 - 1\}$ in the "row-wise" order $((0,0), (0,1), \ldots, (n_1 - 1, n_2 - 1))$, then the isomorphism f is the Chinese remainder permutation σ_{n_1,n_2} given by

$$\sigma_{n_1,n_2} : \begin{cases} \{0, \ldots, n-1\} \mapsto \{0, \ldots, n-1\} \\ i \mapsto e_2 \cdot i \bmod n + e_1 \cdot \left\lfloor \frac{i}{n_2} \right\rfloor \end{cases} , n = n_1 n_2, \tag{4.55}$$

We shall denote the representing matrix as

$$\mathrm{CRT}_{n_1,n_2} = \mathrm{mat}(\sigma_{n_1,n_2}) \tag{4.56}$$

4.3.2 Gather and scatter operators

We require operators that collect a subset of vector elements using a predefined rule. We also need the inverse of this operation, the operator that stores the vector elements into a larger vector at predefined positions.

Gather. The gather operator is similar to a permutation except that it collects elements of a set A to form a subset $B \subset A$. It is a mapping of A onto B.

$$f \ : \ A \mapsto B \subset A$$

By ordering and indexing the n elements of A and m elements of B, we can define the gather operator as the following mapping of indices.

$$f \ : \begin{cases} \{0, \ldots, m-1\} \mapsto \{0, \ldots, n-1\} \\ i \mapsto f(i) \end{cases} \tag{4.57}$$

where $m \leq n$. For the ordered sets A and B, gather operator maps $a_{f(i)} \mapsto b_i$. For $m = n$, the gather function becomes a permutation. Equivalent to permutations, we can obtain the matrix

representations of the gather operator either by the matrix generating function defined in (4.42) or directly as in (4.44):

$$\mathrm{G}^f_{m,n} = \begin{bmatrix} \mathbf{e}^T_{f(0)} \\ \mathbf{e}^T_{f(1)} \\ \vdots \\ \mathbf{e}^T_{f(m-1)} \end{bmatrix} \tag{4.58}$$

where $\mathbf{e}_i \in \mathbb{C}^n$ is the standard basis (4.2). Obviously, $\mathrm{G}^f_{m,n}$ is an $m \times n$ matrix with no more than one unit element per row and column.

Scatter. Scatter is a mapping

$$f \;:\; A \supset B \mapsto A$$

For the ordered sets A and B, gather is a mapping of indices defined in (4.57). The scatter operator maps the elements of ordered sets as $b_i \mapsto a_{f(i)}$. In matrix form, the scatter matrix is simply the transpose of the gather matrix.

$$\mathrm{S}^f_{m,n} = \left(\mathrm{G}^f_{m,n}\right)^T \tag{4.59}$$

The scatter matrix is also the left inverse of the gather matrix.

$$\mathrm{G}^f_{m,n}\,\mathrm{S}^f_{m,n} = \mathrm{I}_m$$

Example 4.6.

$$f \;:\; \begin{cases} \{0,1,2\} \mapsto \{0,1,2,3,4\} \\ i \mapsto i - 2 \bmod 5 \end{cases}$$

$$\mathrm{G}^f_{3,5} = \begin{bmatrix} \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 \\ 1 & \cdot & \cdot & \cdot & \cdot \end{bmatrix} = \left(\mathrm{S}^f_{3,5}{}^T\right) \tag{4.60}$$

Downsampling and upsampling. Let f be an affine mapping of indices.

$$f \;:\; \begin{cases} \{0,\dots,m-1\} \mapsto \{0,\dots,n-1\} \\ i \mapsto i \cdot s + t \end{cases} \quad , \quad m = \left\lfloor \frac{n-t-1}{s} + 1 \right\rfloor \tag{4.61}$$

Then the associated gather operator is the *downsampling* operator or the *downsampler* by s with the offset t.

$$\mathrm{G}^{i\mapsto is}_{m,n} = (\downarrow s)^t_n \tag{4.62}$$

The associated scatter operator is the *upsampling* operator or the *upsampler* by s with the offset t.

$$\mathrm{S}^{i\mapsto is}_{m,n} = (\uparrow s)^t_n \tag{4.63}$$

We omit the offest parameter from the notation when it is equal to zero.

Example 4.7.

$$(\downarrow 2)^1_6 = \begin{bmatrix} \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix}$$

The following properties are satisfied under the condition that $s \mid n$ and $k \mid n$:

$$\begin{aligned}(\downarrow s)_n^t &= \mathrm{I}_k \otimes (\downarrow s)_{\frac{n}{k}}^t \\ (\downarrow s)_n^t &= \mathrm{I}_k \otimes (\downarrow s)_{\frac{n}{k}}^t\end{aligned} \tag{4.64}$$

There is an obvious relationship between the downsampling/upsampling operators and the stride permutation

$$\mathrm{L}_s^{rs} = \begin{bmatrix} (\downarrow s)_{rs}^0 \\ \vdots \\ (\downarrow s)_{rs}^{s-1} \end{bmatrix}, \qquad \mathrm{L}_r^{rs} = \begin{bmatrix} (\uparrow s)_{rs}^0 & \cdots & (\uparrow s)_{rs}^{s-1} \end{bmatrix} \tag{4.65}$$

4.3.3 Extension and reduction operators

The *extension* is, in general, a mapping $E(\mathbf{x}) : \mathbb{C}^n \mapsto \mathbb{C}^{l+n+r}$, where l and r are the number of left and right extension points, respectively. More precisely, we define the extension operator as the mapping

$$\mathrm{E}_{n,l,r}^{f_l,f_r} : \begin{cases} \mathbb{C}^n \mapsto \mathbb{C}^l \oplus \mathbb{C}^n \oplus \mathbb{C}^r \\ \mathbf{x} \mapsto \begin{bmatrix} f_l(\mathbf{x}) \\ \mathbf{x} \\ f_r(\mathbf{x}) \end{bmatrix} \end{cases}, \tag{4.66}$$

where f_l and f_r are extension mappings at the left and the right boundary, respectively. These mappings are in general non-linear functions that extrapolate the signal at the boundaries.

$$f_l = \begin{cases} \mathbb{C}^n \mapsto \mathbb{C}^l \\ \mathbf{x} \mapsto f_l(\mathbf{x}) \end{cases}, \quad f_r = \begin{cases} \mathbb{C}^n \mapsto \mathbb{C}^r \\ \mathbf{x} \mapsto f_r(\mathbf{x}) \end{cases}, \tag{4.67}$$

In the case the extension is a linear transformation, it can be represented in matrix form using (4.7).

$$\mathrm{E}_{n,r,l}^{f_l,f_r} = \begin{bmatrix} \mathrm{E}_l \\ I_n \\ \mathrm{E}_r \end{bmatrix} \tag{4.68}$$

Matrices E_l and E_r are generated by the following matrix generating functions induced by the extension functions f_l and f_r

$$\hat{f}_l = \begin{cases} \{0,\ldots,l-1\} \times \{0,\ldots,n-1\} \mapsto \mathbb{C} \\ (i,j) \mapsto f_l(\mathbf{e}_j)_i \end{cases}, \quad \hat{f}_r = \begin{cases} \{0,\ldots,r-1\} \times \{0,\ldots,n-1\} \mapsto \mathbb{C} \\ (i,j) \mapsto f_r(\mathbf{e}_j)_i \end{cases},$$

where $\mathbf{e}_j$ are standard basis vectors in $\mathbb{C}^n$. In other words the columns of the matrices E_l and E_r are images of the standard basis as was discussed in (4.7).

In many cases, the domain of the extension functions is $\mathbb{C}^{n'} \subset \mathbb{C}^n$. It is then convenient to represent (4.68) by separating the collection of the input elements using gather operator and actual computations into two stages. Let the domain of f_l be $\mathbb{C}^{l'}$ and the domain of f_r be $\mathbb{C}^{r'}$.

$$\mathrm{E}_{n,r,l}^{f_l,f_r} = (C_l \oplus \mathrm{I}_n \oplus C_r) \begin{bmatrix} \mathrm{G}_{l',n}^{f_l} \\ \mathrm{I}_n \\ \mathrm{G}_{r',n}^{f_r} \end{bmatrix}, \tag{4.69}$$

where C_l and C_r represent $l \times l'$ and $r \times r'$ matrices that perform necessary computations on subspaces, and f_l' and f_r' are appropriate mappings of indices.

We also define the *reduction* operator as the transpose of the extension operator. Using the representation (4.69), we can write

$$\mathrm{R}_{n,l,r}^{f_l,f_r} = \left(\mathrm{E}_{n,l,r}^{f_l,f_r}\right)^T = \left[\,\mathrm{S}_{l,n}^{f_l} \,|\, \mathrm{I}_n \,|\, \mathrm{S}_{r,n}^{f_r}\,\right] \left(C_l^T \oplus \mathrm{I}_n \oplus C_r^T\right) \tag{4.70}$$

Therefore, the reduction inherits the same properties and special cases from the extension operators.

In many important cases of linear extensions the matrices C_l nd C_r are very simple, and the extension matrix is little more than the stacking of identity and opposite identity matrices. We define several important extension operators that fit this description.

Zero-padding extension (*zero*). Zero-padding simply adds l zeros on the left side and r zeros on the right side of the input sequence. In the matrix form the zero-padding extension is

$$\mathrm{E}_{n,r,l}^{\mathrm{zero,zero}} = \mathrm{E}_{n,r,l}^{\mathrm{zero}} = \begin{bmatrix} 0_{l\times n} \\ \\ \mathrm{I}_n \\ \\ 0_{r\times n} \end{bmatrix} \tag{4.71}$$

We omit writing both extension types in the superscript if they are the same. The reduction operator $\mathrm{R}_{n,r,l}^{\mathrm{zero}}$ obtained by transposing (4.71) is reducing the sequence by cutting l points from the left and r points from the right.

We use the zero extension and reduction operators to construct matrices stacked with zero matrices horizontally and vertically. Let A be an $m \times n$ matrix. Then,

$$\mathrm{E}_{m,r,l}^{\mathrm{zero}}\, A = \begin{bmatrix} 0_{l\times n} \\ \\ A \\ \\ 0_{r\times n} \end{bmatrix}, \qquad A\,\mathrm{R}_{n,r,l}^{\mathrm{zero}} = \begin{bmatrix} 0_{l\times m} & A & 0_{r\times m} \end{bmatrix}. \tag{4.72}$$

We introduce an important property of the zero-padding that allows swapping of the extension operator and the stride or the linear permutation.

Let $2\,|\,n$ and let $\bar{\mathrm{L}}_2^k$ represent linear permutation if $2 \nmid k$ and stride permutation if $2\,|\,k$. Then,

$$\begin{aligned} \bar{\mathrm{L}}_2^{n+l+r}\, \mathrm{E}_{n,l,r}^{\mathrm{zero}} &= \left(\mathrm{E}_{\lceil \frac{n}{2} \rceil, \lceil \frac{l}{2} \rceil, \lceil \frac{r-n \bmod 2}{2} \rceil}^{\mathrm{zero}} \oplus \mathrm{E}_{\lfloor \frac{n}{2} \rfloor, \lfloor \frac{l}{2} \rfloor, \lfloor \frac{r+n \bmod 2}{2} \rfloor}^{\mathrm{zero}}\right) \bar{\mathrm{L}}_2^n \qquad 2\,|\,l \\ \bar{\mathrm{L}}_{2,1}^{n+l+r}\, \mathrm{E}_{n,l,r}^{\mathrm{per}} &= \left(\mathrm{E}_{\lfloor \frac{n}{2} \rfloor, \lfloor \frac{l}{2} \rfloor, \lceil \frac{r-n \bmod 2}{2} \rceil}^{\mathrm{zero}} \oplus \mathrm{E}_{\lceil \frac{n}{2} \rceil, \lceil \frac{l}{2} \rceil, \lfloor \frac{r+n \bmod 2}{2} \rfloor}^{\mathrm{zero}}\right) \bar{\mathrm{L}}_2^n \qquad 2 \nmid l \end{aligned} \tag{4.73}$$

In the special case when $2\,|\,n$, $2\,|\,l$, and $2\,|\,r$, we have

$$\mathrm{L}_2^{n+l+r}\, \mathrm{E}_{n,l,r}^{\mathrm{zero}} = \left(\mathrm{I}_2 \otimes \mathrm{E}_{\frac{n}{2}, \frac{l}{2}, \frac{r}{2}}^{\mathrm{zero}}\right) \mathrm{L}_2^n \tag{4.74}$$

Periodic extension (per). As the name implies, the *periodic extension* extends the input sequence periodically at the boundaries.

Let $l = q_1 n + l'$ be the left and $r = q_2 n + r'$ be the right extension length. The periodic extension matrix is

$$\mathrm{E}^{\mathrm{per}}_{n,r,l} = \begin{bmatrix} \mathrm{R}^{\mathrm{zero}}_{l',(n-l'),0} \\ \left[\begin{matrix} \\ - \\ \\ \end{matrix} \right]_{i=0}^{q_1+q_2} \mathrm{I}_n \\ \mathrm{R}^{\mathrm{zero}}_{r',0,(n-r')} \end{bmatrix} \tag{4.75}$$

The periodic extension is thus a block matrix of identity and zero matrices.

Example 4.8.

$$\mathrm{E}^{per}_{3,2,1} = \begin{bmatrix} \cdot & 1 & \cdot \\ \cdot & \cdot & 1 \\ 1 & \cdot & \cdot \\ \cdot & 1 & \cdot \\ \cdot & \cdot & 1 \\ 1 & \cdot & \cdot \end{bmatrix}, \qquad \mathrm{R}^{per}_{3,1,4} = \begin{bmatrix} \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 \\ \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot \\ 1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot \end{bmatrix}$$

So, for example, $(x_1, x_2, x_0, x_1, x_2, x_0)^T = \mathrm{E}^{per}_{3,2,1}\, \mathbf{x}$.

One of very important properties of the periodic extension that we will extensively use for FIR filter and DWT rules is that it can be conjugated by stride permutation and preserve the periodic structure.

Let $2 \mid n$ and let $\bar{\mathrm{L}}^n_2$ represent linear permutation if $2 \nmid (n + l + r)$ and stride permutation if $2 \mid (n + l + r)$. Then,

$$\begin{aligned} \bar{\mathrm{L}}_2^{n+l+r}\, \mathrm{E}^{\mathrm{per}}_{n,l,r} &= \left(\mathrm{E}^{\mathrm{per}}_{\frac{n}{2},\lceil \frac{l}{2} \rceil,\lceil \frac{r}{2} \rceil} \oplus \mathrm{E}^{\mathrm{per}}_{\frac{n}{2},\lfloor \frac{l}{2} \rfloor,\lfloor \frac{r}{2} \rfloor} \right) \mathrm{L}^n_2 \qquad 2 \mid l \\ \bar{\mathrm{L}}_{2,1}^{n+l+r}\, \mathrm{E}^{\mathrm{per}}_{n,l,r} &= \left(\mathrm{E}^{\mathrm{per}}_{\frac{n}{2},\lfloor \frac{l}{2} \rfloor,\lceil \frac{r}{2} \rceil} \oplus \mathrm{E}^{\mathrm{per}}_{\frac{n}{2},\lceil \frac{l}{2} \rceil,\lfloor \frac{r}{2} \rfloor} \right) \mathrm{L}^n_2 \qquad 2 \nmid l \end{aligned} \tag{4.76}$$

In the special case when $2 \mid n$, $2 \mid l$, and $2 \mid r$, we have

$$\mathrm{L}_2^{n+l+r}\, \mathrm{E}^{\mathrm{per}}_{n,l,r} = \left(\mathrm{I}_2 \otimes \mathrm{E}^{\mathrm{per}}_{\frac{n}{2},\frac{l}{2},\frac{r}{2}} \right) \mathrm{L}^n_2 \tag{4.77}$$

Symmetric extensions. There are four different types of symmetric extension operators. Based on the type of symmetry, we have symmetric and anti-symmetric extension, and based on the point of symmetry we have whole-point symmetry and half-point symmetry. We define each of them next.

Whole-point symmetric extension. The whole-point symmetric extension extends the sequence symmetrically at the boundaries, where the point of symmetry is the boundary element of the sequence. The sequence is extended by mirroring the elements over the boundary element. If the lengths of the left and the right extension l and r, respectively, are smaller than the length of the sequence n then the extension matrix is simply

$$\mathrm{E}^{\mathrm{ws}}_{n,r,l} = \begin{bmatrix} \mathrm{J}_l\, \mathrm{R}^{\mathrm{zero}}_{l,1,(n-l-1)} \\ \\ \mathrm{I}_n \\ \\ \mathrm{J}_r\, \mathrm{R}^{\mathrm{zero}}_{r,(n-r-1),1} \end{bmatrix} \tag{4.78}$$

The boundary extension matrices are of the form $\mathrm{J}_k\, \mathrm{R}^{\mathrm{zero}}_{k,a,b}$ which represents opposite identity matrices (4.14) horizontally stacked with appropriate zero matrices as shown in (4.72).

In the more general case, let $l = q_1 n + l'$ and $r = q_2 n + r'$. We define the whole-point symmetric extension matrix by

$$\mathrm{E}^{\mathrm{ws}}_{n,r,l} = \begin{bmatrix} \left[\begin{matrix} q_1 \\ — \\ i=0 \end{matrix}\right] E_{l,i} \\ \\ \mathrm{I}_n \\ \\ \left[\begin{matrix} q_2 \\ — \\ i=0 \end{matrix}\right] E_{r,i} \end{bmatrix} \tag{4.79}$$

$$E_{l,i} = \begin{cases} \mathrm{J}_{n-1}\, \mathrm{R}^{\mathrm{zero}}_{n-1,1,0} & i = 2k \\ \mathrm{I}_{n-1}\, \mathrm{R}^{\mathrm{zero}}_{n-1,0,1} & i = 2k+1 \\ \mathrm{J}_{l'}\, \mathrm{R}^{\mathrm{zero}}_{l',1,n-l'-1} & i = q_1,\ q_1 = 2k \\ \mathrm{I}_{l'}\, \mathrm{R}^{\mathrm{zero}}_{l',n-l'-1,1} & i = q_1,\ q_1 = 2k+1 \end{cases}, \quad k = 0, 1, \ldots$$

$$E_{r,i} = \begin{cases} \mathrm{J}_{n-1}\, \mathrm{R}^{\mathrm{zero}}_{n-1,0,1} & i = 2k \\ \mathrm{I}_{n-1}\, \mathrm{R}^{\mathrm{zero}}_{n-1,1,0} & i = 2k+1 \\ \mathrm{J}_{r'}\, \mathrm{R}^{\mathrm{zero}}_{r',1,n-r'-1} & i = q_1,\ q_1 = 2k \\ \mathrm{I}_{r'}\, \mathrm{R}^{\mathrm{zero}}_{r',n-r'-1,1} & i = q_1,\ q_1 = 2k+1 \end{cases} \quad k = 0, 1, \ldots \tag{4.80}$$

We provide a few examples to illustrate the definition.

EXAMPLE 4.9.

$$\mathrm{E}^{ws}_{5,2,1} = \begin{bmatrix} . & . & 1 & . & . \\ . & 1 & . & . & . \\ 1 & . & . & . & . \\ . & 1 & . & . & . \\ . & . & 1 & . & . \\ . & . & . & 1 & . \\ . & . & . & . & 1 \\ . & . & . & 1 & . \end{bmatrix}, \qquad \mathrm{R}^{ws}_{3,0,5} = \begin{bmatrix} 1 & . & . & . & 1 & . & . & . \\ . & 1 & . & 1 & . & 1 & . & 1 \\ . & . & 1 & . & . & . & 1 & . \end{bmatrix}$$

In the first example we have $(x_2, x_1, x_0, x_1, x_2, x_3, x_4, x_3)^T = \mathrm{E}^{ws}_{5,2,1}\, \mathbf{x}$.

Half-point symmetric extension. The *half-point symmetric* extension extends the sequence symmetrically at the boundaries. Unlike the whole-point extensions, the point of symmetry is in

between the boundary element of the sequence and the next outside element. The boundary element in this case is mirrored together with all the other elements to create the extension.

If the lengths of the left and the right extension l and r, respectively, are smaller than the length of the sequence n, then the extension matrix is similar to (4.78)

$$\mathrm{E}^{\mathrm{hs}}_{n,r,l} = \begin{bmatrix} \mathrm{J}_l \, \mathrm{R}^{\mathrm{zero}}_{l,0,(n-l)} \\ \\ \mathrm{I}_n \\ \\ \mathrm{J}_r \, \mathrm{R}^{\mathrm{zero}}_{r,(n-r),0} \end{bmatrix} \tag{4.81}$$

Again, let $l = q_1 n + l'$ and $r = q_2 n + r'$. We define the half-point symmetric extension matrix by

$$\mathrm{E}^{\mathrm{hs}}_{n,r,l} = \begin{bmatrix} \left[\begin{matrix} q_1 \\ \hline \\ i=0 \end{matrix}\right] E_{l,i} \\ \\ \mathrm{I}_n \\ \\ \left[\begin{matrix} q_2 \\ \hline \\ i=0 \end{matrix}\right] E_{r,i} \end{bmatrix} \tag{4.82}$$

$$E_{l,i} = \begin{cases} \mathrm{J}_n & i = 2k \\ \mathrm{I}_n & i = 2k+1 \\ \mathrm{J}_{l'} \, \mathrm{R}^{\mathrm{zero}}_{l',0,n-l'} & i = q_1,\ q_1 = 2k \\ \mathrm{I}_{l'} \, \mathrm{R}^{\mathrm{zero}}_{l',n-l',0} & i = q_1,\ q_1 = 2k+1 \end{cases} \qquad k = 0, 1, \ldots$$

$$E_{r,i} = \begin{cases} \mathrm{J}_n & i = 2k \\ \mathrm{I}_n & i = 2k+1 \\ \mathrm{J}_{r'} \, \mathrm{R}^{\mathrm{zero}}_{r',0,n-r'} & i = q_1,\ q_1 = 2k \\ \mathrm{I}_{r'} \, \mathrm{R}^{\mathrm{zero}}_{r',n-r',0} & i = q_1,\ q_1 = 2k+1 \end{cases} \qquad k = 0, 1, \ldots \tag{4.83}$$

Half-point anti-symmetric extension. The *half-point anti-symmetric* extension mirrors the sequence anti-symmetrically over a half point between the boundary element of the sequence and the next outside element. The anti-symmetry is achieved by inverting the sign of the mirrored extension.

First, let $l, r < n$. Then the extension matrix is given by

$$\mathrm{E}^{\mathrm{ha}}_{n,r,l} = \begin{bmatrix} -\mathrm{J}_l \, \mathrm{R}^{\mathrm{zero}}_{l,0,(n-l)} \\ \\ \mathrm{I}_n \\ \\ -\mathrm{J}_r \, \mathrm{R}^{\mathrm{zero}}_{r,(n-r),0} \end{bmatrix} \tag{4.84}$$

If we now let $l = q_1 n + l'$ and $r = q_2 n + r'$, then we define the half-point anti-symmetric extension

matrix by

$$\mathrm{E}^{\mathrm{ha}}_{n,r,l} = \begin{bmatrix} \left[\begin{matrix} q_1 \\ — \\ i=0 \end{matrix}\right] E_{l,i} \\ \mathrm{I}_n \\ \left[\begin{matrix} q_2 \\ — \\ i=0 \end{matrix}\right] E_{r,i} \end{bmatrix} \tag{4.85}$$

$$E_{l,i} = \begin{cases} -\mathrm{J}_n & i = 2k \\ \mathrm{I}_n & i = 2k+1 \\ -\mathrm{J}_{l'}\,\mathrm{R}^{\mathrm{zero}}_{l',0,n-l'} & i = q_1,\ q_1 = 2k \\ \mathrm{I}_{l'}\,\mathrm{R}^{\mathrm{zero}}_{l',n-l',0} & i = q_1,\ q_1 = 2k+1 \end{cases} \qquad k = 0, 1, \ldots \tag{4.86}$$

$$E_{r,i} = \begin{cases} -\mathrm{J}_n & i = 2k \\ \mathrm{I}_n & i = 2k+1 \\ -\mathrm{J}_{r'}\,\mathrm{R}^{\mathrm{zero}}_{r',0,n-r'} & i = q_1,\ q_1 = 2k \\ \mathrm{I}_{r'}\,\mathrm{R}^{\mathrm{zero}}_{r',n-r',0} & i = q_1,\ q_1 = 2k+1 \end{cases} \qquad k = 0, 1, \ldots$$

EXAMPLE 4.10.

$$\mathrm{E}^{hs}_{4,2,1} = \begin{bmatrix} . & 1 & . & . \\ 1 & . & . & . \\ 1 & . & . & . \\ . & 1 & . & . \\ . & . & 1 & . \\ . & . & . & 1 \\ . & . & 1 & . \end{bmatrix}, \qquad \mathrm{E}^{ha}_{3,0,5} = \begin{bmatrix} 1 & . & . \\ . & 1 & . \\ . & . & 1 \\ . & . & -1 \\ . & -1 & . \\ -1 & . & . \\ 1 & . & . \\ . & 1 & . \end{bmatrix}$$

For example, $(x_0, x_1, x_2, -x_2, -x_1, -x_0, x_0, x_1)^T = \mathrm{E}^{ha}_{3,0,5}\,\mathbf{x}$.

Whole-point anti-symmetric extension. The *whole-point anti-symmetric* extension operator extends the sequence by mirroring the elements over a boundary element and inverting the sign. Since no element can be antisymmetric to itself except the zero element, the sequence is first extended by one zero at the boundary. The sequence is then extended by mirroring and sign inversion over that zero element.

In the case when $l, r < n$, the extension matrix is given by

$$\mathrm{E}^{\mathrm{wa}}_{n,r,l} = \begin{bmatrix} -\mathrm{J}_l\,\mathrm{R}^{\mathrm{zero}}_{l,0,(n-l)} \\ 0_{1\times n} \\ \mathrm{I}_n \\ 0_{1\times n} \\ -\mathrm{J}_r\,\mathrm{R}^{\mathrm{zero}}_{r,(n-r),0} \end{bmatrix} \tag{4.87}$$

If we now let $l = q_1 n + l'$ and $r = q_2 n + r'$, then we define the half-point anti-symmetric extension matrix by

$$\mathrm{E}^{\mathrm{wa}}_{n,r,l} = \begin{bmatrix} \left[\begin{matrix} q_1 \\ \Big[\!\!-\!\!\Big] \\ i=0 \end{matrix}\right] \begin{bmatrix} E_{l,i} \\ 0_{1\times n} \end{bmatrix} \\ \mathrm{I}_n \\ \left[\begin{matrix} q_2 \\ \Big[\!\!-\!\!\Big] \\ i=0 \end{matrix}\right] \begin{bmatrix} 0_{1\times n} \\ E_{r,i} \end{bmatrix} \end{bmatrix} \tag{4.88}$$

$$E_{l,i} = \begin{cases} -\mathrm{J}_n & i = 2k \\ \mathrm{I}_n & i = 2k+1 \\ -\mathrm{J}_{l'}\,\mathrm{R}^{\mathrm{zero}}_{l',0,n-l'} & i = q_1,\ q_1 = 2k \\ \mathrm{I}_{l'}\,\mathrm{R}^{\mathrm{zero}}_{l',n-l',0} & i = q_1,\ q_1 = 2k+1 \end{cases} \qquad k = 0, 1, \ldots$$

$$E_{r,i} = \begin{cases} -\mathrm{J}_n & i = 2k \\ \mathrm{I}_n & i = 2k+1 \\ -\mathrm{J}_{r'}\,\mathrm{R}^{\mathrm{zero}}_{r',0,n-r'} & i = q_1,\ q_1 = 2k \\ \mathrm{I}_{r'}\,\mathrm{R}^{\mathrm{zero}}_{r',n-r',0} & i = q_1,\ q_1 = 2k+1 \end{cases} \qquad k = 0, 1, \ldots \tag{4.89}$$

EXAMPLE 4.11.

$$\mathrm{E}^{wa}_{5,0,2} = \begin{bmatrix} 1 & . & . & . & . \\ . & 1 & . & . & . \\ . & . & 1 & . & . \\ . & . & . & 1 & . \\ . & . & . & . & 1 \\ . & . & . & . & . \\ . & . & . & . & -1 \end{bmatrix}, \qquad \mathrm{R}^{ws}_{3,0,5} = \begin{bmatrix} 1 & . & . & . & . & . & -1 & . \\ . & 1 & . & . & . & -1 & . & . \\ . & . & 1 & . & -1 & . & . & . \end{bmatrix}$$

For example, $(x_0, x_1, x_2, 0, -x_2, -x_1)^T = \mathrm{E}^{wa}_{3,0,3}\,\mathbf{x}$.

Combination of extensions. The left and the right extension methods are, in general, not of the same type and all of the above methods as well as others can be combined freely.

EXAMPLE 4.12.

$$\mathrm{E}^{ws,ha}_{4,2,2} = \begin{bmatrix} . & . & 1 & . \\ . & 1 & . & . \\ 1 & . & . & . \\ . & 1 & . & . \\ . & . & 1 & . \\ . & . & . & 1 \\ . & . & . & -1 \\ . & . & -1 & . \end{bmatrix} \qquad \mathrm{E}^{per,ws}_{3,1,3} = \begin{bmatrix} . & . & 1 \\ 1 & . & . \\ . & 1 & . \\ . & . & 1 \\ . & 1 & . \\ 1 & . & . \\ . & 1 & . \end{bmatrix} \tag{4.90}$$

Reduction modulo polynomials. From definitions (4.71) – (4.87) we can readily obtain the reduction matrices by transposition. They are all of the form

$$\mathrm{R}_{n,l,r}^{f_l,f_r} = \left(\mathrm{E}_{n,l,r}^{f_l,f_r}\right)^T \tag{4.91}$$

Reduction operators are also encountered in polynomial multiplication based algorithms. Since all sequences can be viewed as polynomials using the z-transform analogy discussed in Chapter 5, the reduction matrices can be viewed as the matrix representation of the polynomial reduction of $x(z)$ modulo some $p(z)$.

Let $x(z) = x_{-l}z^l + \cdots + x_{n+r}z^{-(n+r)}$ and $y(z)$ be the reduction of $x(z)$ modulo a monic polynomial of the form $a(z) = z^{-n} - a_{n-1}z^{-(n-1)} - \cdots - a_1 z^{-1} - 1$. The operation modulo $a(z)$ implies that $z^{-n} = a_{n-1}z^{-(n-1)} + \cdots + a_1 z^{-1} + 1$, and that $z = z^{-(n-1)} + a_{n-1}z^{-(n-2)} + \cdots + a_2 z^{-1} + a_1$. Without loss of generality, we assume $a_n = 1$ and $a_0 = 1$ to avoid writing the scaling factor. Then the representation of the reduction operation is the matrix-vector product

$$\mathbf{y} = \mathrm{R}_{n,l,r}^{a(z)}\,\mathbf{x} = \left[\, R_l \,|\, \mathrm{I}_n \,|\, R_r \,\right] \tag{4.92}$$

where the matrices R_r and R_l are given by the matrix generating functions

$$\hat{r}_r : \begin{cases} \{0,\ldots,n-1\} \times \{0,\ldots,r-1\} \to \mathbb{C} & \\ (i,j) \mapsto a_i & : \; j = 0 \\ (i,j) \mapsto a_0 \cdot \hat{r}_r(m-1, j-1) & : \; i = 0, j > 0 \\ (i,j) \mapsto \hat{r}_r(i-1,j-1) + a_i \cdot \hat{r}_r(m-1,j-1) & : \; \text{else} \end{cases} \tag{4.93a}$$

$$\hat{r}_l : \begin{cases} \{0,\ldots,n-1\} \times \{0,\ldots,r-1\} \to \mathbb{C} & \\ (i,j) \mapsto a_{i+1} & : \; j = l-1 \\ (i,j) \mapsto \hat{r}_l(0, j+1) & : \; i = n-1, j < l-1 \\ (i,j) \mapsto \hat{r}_l(i+1,j+1) + a_{i+1} \cdot \hat{r}_l(0,j+1) & : \; \text{else} \end{cases} \tag{4.93b}$$

We also define extension operators modulo $a(z)$ as

$$\mathrm{E}_{n,l,r}^{a(z)} = \left(\mathrm{R}_{n,l,r}^{a(z)}\right)^T \tag{4.94}$$

EXAMPLE 4.13. *Let $p(z) = z^{-4} - 1$ and let $x(z) = x_{-1}z + \cdots + x_6 z^{-6}$. Using (4.93) we generate the $\mathrm{R}_{4,1,2}^{p(z)}$ matrix.*

$$\mathrm{R}_{4,1,2}^{p(z)} = \begin{bmatrix} . & 1 & . & . & . & 1 & . \\ . & . & 1 & . & . & . & 1 \\ . & . & . & 1 & . & . & . \\ 1 & . & . & . & 1 & . & . \end{bmatrix} = \mathrm{R}_{4,1,2}^{per} = \left(\mathrm{E}_{4,1,2}^{per}\right)^T$$

In general,

$$\mathrm{R}_{n,l,r}^{1-z^{-n}} = \mathrm{R}_{n,l,r}^{per} \tag{4.95}$$

When the reduction or extension operators are induced by residue class polynomial algebras we call them *polynomial reduction* and *polynomial extension* operators, respectively.

Table 4.1: Table of mathematical objects.

Special matrices		**Constructs**		**Operators**	
0_n	Zero	$A \cdot B$	Composition	$\mathrm{G}^f_{m,n}, \mathrm{S}^f_{m,n}$	Gather and scatter
I_n	Identity	A^B	Conjugation	$(\uparrow s)^t_n, (\downarrow s)^t_n$	Up/Downsampler
J_n	Opposite identity	$[A_{i,j}]_{m\times n}$	Matrix of matrices	L^{rs}_s	Stride permutation
$\mathrm{diag}(\mathbf{a})$	Diagonal	$A \oplus B$	Direct sum	$\bar{\mathrm{L}}^{rs}_{s,t}$	Affine permutation
		$A \otimes B$	Tensor product	$\mathrm{E}^{f_l,f_r}_{n,l,r}$	Extensions
		$A \otimes_k B$	Overlapped tensor		
		$A \oplus_k B$	Overlapped direct sum		

4.4 Summary

In this chapter, we developed the mathematical tools required for designing the framework for representing FIR filters, discrete wavelet transforms, and fast algorithms toward the goal of automatically generating their implementations. In the next two chapters, we define the filtering and wavelet operations as SPIRAL transforms and build a library of breakdown rules that span the space of available efficient algorithms. All required mathematical constructs, operators, matrices, and their properties have been defined in this chapter, and we will refer to them as needed.

Before we start developing the required framework, we recall that the mathematical constructs such as the tensor product, overlapped constructs, and direct sums are used to capture the inherent structure of the computed objects. We shall use these constructs, as well as the set of matrices and operators to write the breakdown rules for FIR filters and DWTs. In Table 4.1 we summarize the mathematical objects we covered in this chapter.

CHAPTER 5

FILTER TRANSFORMS AND RULES

5.1 FIR filter transforms

The basic building block in filtering and wavelet algorithms is the convolution operation (3.5). Because of its linearity, it can be represented as the matrix-vector multiplication where the matrix is an infinite Toeplitz matrix with diagonals as the values of the $\{h_n\}$ sequence, and where the vector represents the values of the $\{x_n\}$ sequence:

$$\begin{bmatrix} \vdots \\ y_{-2} \\ y_{-1} \\ y_0 \\ y_1 \\ y_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} \ddots & \vdots & \vdots & \vdots & & & \\ \cdots & h_0 & h_{-1} & h_{-2} & \cdots & & \\ \cdots & h_1 & h_0 & h_{-1} & h_{-2} & \cdots & \\ \cdots & h_2 & h_1 & h_0 & h_{-1} & h_{-2} & \cdots \\ & \cdots & h_2 & h_1 & h_0 & h_{-1} & \cdots \\ & & \cdots & h_2 & h_1 & h_0 & \cdots \\ & & & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \cdot \begin{bmatrix} \vdots \\ x_{-2} \\ x_{-1} \\ x_0 \\ x_1 \\ x_2 \\ \vdots \end{bmatrix} \tag{5.1}$$

In order to restrict the computation of the convolution to finite sequences, we assume that the filter length is finite, i.e., we consider only FIR filters. From this point on, there are two approaches to band the matrix (5.1). We can either consider that the input sequence $\{x_n\}$ is infinite and that we compute the output sequence on an n-point interval $\{y_k\}_0^{n-1}$. This will give us the FIR filter transform. The second approach is to assume that the input sequence is finite $\{x_k\}_0^{n-1}$ and that we compute the output on all non-zero points. This will be the convolution transform.

5.1.1 Filter and convolution transforms

Filter transform.. Given the impulse response $\{h_n\}_{-r}^{l}$ of an FIR filter and its z-transform $h(z) = h_l z^{-l} + \cdots + h_0 z + \cdots + h_{-r} z^r$, where $l \geq 0$ and $r \geq 0$, we define the *filter transform* computed on n output points as the $n \times n + l + r$ matrix

$$\mathbf{Filt}_n(h(z)) = \begin{bmatrix} h_l & \cdots & h_{-r} & & \\ & h_l & \cdots & h_{-r} & \\ & & \ddots & & \ddots \\ & & & h_l & \cdots & h_{-r} \end{bmatrix} \tag{5.2}$$

We always assume that the output sequence is computed for the interval $[0, n-1]$. In other words

$$(y_0, \ldots, y_{n-1})^T = \mathbf{Filt}_n(h(z))\,(x_{-l}, \ldots, x_{n+r})^T. \tag{5.3}$$

If $r < 0$ or $l < 0$ then we assume $r = 0$ and $l = 0$, respectively. In those cases we explicitly write the zero coefficients of the polynomial following z^0. For example, if the polynomial $h(z) = z^2$ then, $l = 0, r = 2$ and we write $h(z) = 0 + 0z + z^2$.

EXAMPLE 5.1.

$$h(z) = z^{-2} = z^{-2} + 0z^{-1} + 0$$

$$\mathbf{Filt}_4\big(z^{-2}\big) = \begin{bmatrix} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & 1 & . & . \end{bmatrix}$$

So, we write $\mathbf{Filt}_4\big(z^{-2}\big)$ *when we actually mean* $\mathbf{Filt}_4\big(z^{-2} + 0z^{-1} + 0z^0\big)$.

This might seem a bit odd at first glance, but we shall see later that it is absolutely necessary to preserve the time shift information contained in filtered sequences. If the filtering transforms were invariant to shifts in time then the implementations would require special treatment and case distinctions in numerous situations where this information is crucial.

The following identities are obtained straight from the definition

$$\mathbf{Filt}_n(1) = \mathrm{I}_n \qquad (\textit{Identity}) \tag{5.4a}$$

$$\mathbf{Filt}_n\big(z^{-l}\big) = \mathrm{I}_n\,\mathrm{R}^{\mathrm{zero}}_{n,l,0} \qquad (\textit{Delay}) \tag{5.4b}$$

$$\mathbf{Filt}_n(z^r) = \mathrm{I}_n\,\mathrm{R}^{\mathrm{zero}}_{n,0,r} \qquad (\textit{Advance}) \tag{5.4c}$$

Convolution transform. If the input signal has a finite support of length n and the filter is FIR with length k, we define the *convolution transform* as the following $n + l + r \times n$ matrix.

$$\mathbf{Conv}_n(h(z)) = \begin{bmatrix} h_{-r} & & & & & \\ \vdots & \ddots & & & & \\ h_l & \cdots & h_{-r} & & & \\ & \ddots & & \ddots & & \\ & & h_l & \cdots & h_{-r} \\ & & & \ddots & \vdots \\ & & & & h_l \end{bmatrix} \tag{5.5}$$

Here, we always assume that the input sequence is limited to the interval $[0, n-1]$.

$$(y_{-r}, \dots, y_{n+l})^T = \mathbf{Filt}_n(h(z))\,(x_0, \dots, x_{n-1})^T. \tag{5.6}$$

EXAMPLE 5.2.

$$\mathbf{Conv}_4\big(z^{-1}\big) = \begin{bmatrix} . & . & . & . \\ 1 & . & . & . \\ . & 1 & . & . \\ . & . & 1 & . \\ . & . & . & 1 \end{bmatrix}$$

Following the discussion for the filter transform, we write $\mathbf{Conv}_n\big(z^{-1}\big)$ when we mean $\mathbf{Conv}_n\big(z^{-1} + 0z^0\big)$

Similar to (5.4a) we obtain the following basic identities

$$\mathbf{Conv}_n(1) = \mathrm{I}_n \qquad (\textit{Identity}) \tag{5.7a}$$

$$\mathbf{Conv}_n\big(z^{-l}\big) = \mathrm{E}^{\mathrm{zero}}_{n,l,0}\,\mathrm{I}_n \qquad (\textit{Delay}) \tag{5.7b}$$

$$\mathbf{Filt}_n(z^r) = \mathrm{E}^{\mathrm{zero}}_{n,0,r}\,\mathrm{I}_n \qquad (\textit{Advance}) \tag{5.7c}$$

The relationship between the filter and the convolution transforms is given by the transposition combined with time reversal

$$\mathbf{Conv}_n(h(z)) = \mathbf{Filt}_n\big(h(z^{-1})\big)^T. \tag{5.8}$$

Properties of filter and convolution transforms. The basic property of the filter and convolution transforms is their linearity, which we give without the proof.

Lemma 5.1. *Given the two polynomials* $h(z) = h_{l_1}z^{-l_1} + \cdots + h_0 + \cdots + h_{-r_1}z^{r_1}$ *and* $g(z) = g_{l_2}z^{-l_2} + \cdots + g_0 + \cdots + g_{-r_2}z^{r_2}$ *and the scalars* a *and* b *it follows that*

$$\mathbf{Filt}_n(ah(z) + bg(z)) = a\,\mathbf{Filt}_n(h(z))\,\mathrm{R}^{zero}_{m_1,L_1,R_1} + b\,\mathbf{Filt}_n(g(z))\,\mathrm{R}^{zero}_{m_2,L_2,R_2} \tag{5.9a}$$

$$\mathbf{Conv}_n(ap(z) + bq(z)) = a\,\mathrm{E}^{zero}_{m_1,L_1,R_1}\,\mathbf{Conv}_n(p(z)) + b\,\mathrm{E}^{zero}_{m_2,L_2,R_2}\,\mathbf{Conv}_n(q(z)), \tag{5.9b}$$

where $m_i = n_i + l_i + r_i$, *and*

$$\begin{aligned} L_i &= l_{max} - \langle l_i \rangle, \quad l_{max} = \max\{l_1, l_2\} \\ R_i &= r_{max} - \langle r_i \rangle, \quad r_{max} = \max\{r_1, r_2\} \end{aligned}, \qquad \langle x \rangle = \begin{cases} 0, & x \le 0 \\ x, & x > 0 \end{cases} \tag{5.10}$$

The lemma states that the transforms are linear in the polynomial parameter as long as the dimensions of the matrices are adjusted by stacking zero matrices to match the dimensions.

Corollary 5.2. *Given the polynomial* $h(z) = h_l z^{-l} + \cdots + h_0 + \cdots + h_{-r}z^r$ *the filter transform can be decomposed into*

$$\mathbf{Filt}_n(h(z)) = \sum_{i=-r}^{0} h_i\,\mathbf{Filt}_n\big(z^{-i}\big)\,\mathrm{R}^{zero}_{n-i,l,r+i} + \sum_{i=1}^{l} h_i\,\mathbf{Filt}_n\big(z^{-i}\big)\,\mathrm{R}^{zero}_{n+i,l-i,r} \tag{5.11}$$

The same identity holds for the convolution transform.

Corollary 5.3. *Given the polynomial* $h(z) = h_l z^{-l} + \cdots + h_0 + \cdots + h_{-r}z^r$ *with* $l, r > 0$. *The filter transform can be decomposed into*

$$\mathbf{Filt}_n(h(z)) = \mathbf{Filt}_n(h_l(z))\,\mathrm{R}^{zero}_{n+l,0,r} + \mathbf{Filt}_n(h_r(z))\,\mathrm{R}^{zero}_{n+r,l,0} \tag{5.12}$$

where $h_l(z) = h_l z^{-l} + \cdots + h_0$ *and* $h_r(z) = h_{-1}z + \cdots + h_{-r}z^r$.

5.1.2 Filters with signal extension

We saw in the previous section the straightforward ways to band an infinite filtering matrix. It is sometimes required that the number of input points equal the number of output points. An immediate solution is to limit the input sequence to, for example, n points as in (5.5) and to compute the output on some n out of $n+k-1$ total non-zero points. This, first, requires an assumption that the input sequence is zero outside of the interval of interest. Second, it requires a way to specify which n output points need to be computed. The first assumption is based on the approximation of the signal by its truncated version with zeros outside of the interval. Depending on the nature of the signal and the application of the filtering operation, this approximation may not be particularly well fit. The properties of the signal can be better modeled by extending the signal outside of the interval of interest by some method. This procedure and the selection of the output points can be formalized by defining the *extended filter transform*.

We introduce the *extended filter transform* as an $n \times n$ matrix

$$\mathbf{Filt}_n^{f_l,f_r}(h(z)) = \mathbf{Filt}_m(h(z)) \cdot \mathrm{E}_{n,l,p}^{f_l,f_r} \tag{5.13}$$

Thus, the extended filter transform represents the filter transform with the input sequence extended to the appropriate length using the method specified by the extension operator $\mathrm{E}_{n,l,r}^{f_l,f_r}$. The length of the extension of the extended filter transform need not be specified since it is implicitly given in $h(z) = h_l z^{-l} + \cdots + h_0 z + \cdots + h_{-r} z^r$.

$$\begin{aligned}
\mathbf{Filt}_n^{f_l,f_r}(h(z)) &= \left[\begin{array}{c|c|c} \mathrm{E}_{l,0,n-l}^{\mathrm{zero}} F_l & \mathbf{Filt}_n^{\mathrm{zero}}(h(z)) & \mathrm{E}_{r,n-r,0}^{\mathrm{zero}} F_r \end{array}\right] \cdot \mathrm{E}_{n,l,r}^{f_l,f_r} \\
&= \left[\begin{array}{ccc|ccccc|cc}
h_l & \cdots & h_1 & h_0 & \cdots & h_{-r} & & & & & \\
 & \ddots & & & \ddots & & \ddots & & & & \\
 & & h_l & & & h_0 & & \ddots & & & \\
 & & & \ddots & & & \ddots & & \ddots & & \\
 & & & & h_l & \cdots & \cdots & h_0 & \cdots & h_{-r}
\end{array}\right] \cdot \left[\begin{array}{c} \mathrm{E}_l \\ \hline \mathrm{I}_n \\ \hline \mathrm{E}_r \end{array}\right]
\end{aligned} \tag{5.14}$$

When extension is a linear transformation, the extended filter matrix can be represented as

$$\mathbf{Filt}_n^{f_l,f_r}(h(z)) = \left[\begin{array}{ccccccc}
a_{1,0} & a_{1,2} & \cdots & \cdots & \cdots & \cdots & a_{1,n} \\
\vdots & \vdots & & & & & \vdots \\
a_{l,0} & a_{l,2} & \cdots & \cdots & \cdots & \cdots & a_{l,n} \\
h_l & \cdots & h_0 & \cdots & h_{-r} & & \\
 & \ddots & & \ddots & & \ddots & \\
 & & h_l & \cdots & h_0 & \cdots & h_{-r} \\
b_{1,0} & b_{1,2} & \cdots & \cdots & \cdots & \cdots & b_{1,n} \\
\vdots & \vdots & & & & & \vdots \\
b_{p,0} & b_{p,2} & \cdots & \cdots & \cdots & \cdots & b_{p,n}
\end{array}\right] = \left[\begin{array}{c} A_l \\ \hline \mathbf{Filt}_{n-l-r}(h(z)) \\ \hline B_r \end{array}\right] \tag{5.15}$$

where, from (5.14), $A_l = F_l \cdot E_l + \mathrm{R}_{l,0,n-l}^{\mathrm{zero}} \mathbf{Filt}_n^{\mathrm{zero}}(h(z))$ and $B_r = F_r \cdot E_r + \mathrm{R}_{r,n-r,0}^{\mathrm{zero}} \mathbf{Filt}_n^{\mathrm{zero}}(h(z))$ are left and right boundary filters, respectively.

If the extension operator is linear, then the linearity of the extended filter transform in the

polynomial $h(z)$ follows directly from the linearity of the filter transform (5.9a).

$$\begin{aligned}\mathbf{Filt}_n^{f_l,f_r}(ap(z)+bq(z)) &= \mathbf{Filt}_n(ap(z)+bq(z))\cdot \mathrm{E}_{n,l,r}^{f_l,f_r}\\ &= (a\,\mathbf{Filt}_n(p(z)) + b\,\mathbf{Filt}_n(q(z)))\cdot \mathrm{E}_{n,l,r}^{f_l,f_r}\\ &= a\,\mathbf{Filt}_n^{f_l,f_r}(p(z)) + b\,\mathbf{Filt}_n^{f_l,f_r}(q(z))\end{aligned} \tag{5.16}$$

EXAMPLE 5.3.

$$\mathbf{Filt}_4^{ws,hs}\big(z+z^{-1}\big) = \begin{bmatrix} 1 & . & 1 & . & . & . \\ . & 1 & . & 1 & . & . \\ . & . & 1 & . & 1 & . \\ . & . & . & 1 & . & 1 \end{bmatrix} \mathrm{E}_{4,1,1}^{ws,hs} = \begin{bmatrix} . & 2 & . & . \\ 1 & . & 1 & . \\ . & 1 & . & 1 \\ . & . & 1 & 1 \end{bmatrix}$$

Lemma 5.4. *Given the polynomial $h(z) = h_l z^{-l} + \cdots + h_0 + \cdots + h_{-r} z^r$ with $l, r > 0$. The extended filter transform can be decomposed into*

$$\mathbf{Filt}_n^{f_l,f_r}(h(z)) = \mathbf{Filt}_n^{f_l}(h_l(z)) + \mathbf{Filt}_n^{f_r}(h_r(z)) \tag{5.17}$$

where $h_l(z) = h_l z^{-l} + \cdots + h_0$ and $h_r(z) = h_{-1} z + \cdots + h_{-r} z^r$.

Proof.

$$\begin{aligned}\mathbf{Filt}_n^{f_l,f_r}(h(z)) &\overset{(p.1)}{=} \mathbf{Filt}_n(h(z))\,\mathrm{E}_{n,l,r}^{f_l,f_r}\\ &\overset{(p.2)}{=} \big(\mathbf{Filt}_n(h_l(z))\,\mathrm{R}_{n+l,0,r}^{\mathrm{zero}} + \mathbf{Filt}_n(h_r(z))\,\mathrm{R}_{n+r,l,0}^{\mathrm{zero}}\big)\,\mathrm{E}_{n,l,r}^{f_l,f_r}\\ &= \mathbf{Filt}_n^{f_l}(h_l(z)) + \mathbf{Filt}_n^{f_r}(h_r(z))\end{aligned}$$

where $(p.1)$ refers to identity (5.13) and $(p.2)$ to (5.12). The last equality is obtained by observing that

$$\mathrm{R}_{n+l,0,r}^{\mathrm{zero}}\,\mathrm{E}_{n,l,r}^{f_l,f_r} = \Big[\,\mathrm{I}_{n+l}\,|\,0_{n+l\times r}\,\Big]\begin{bmatrix} E_l \\ \mathrm{I}_n \\ E_r \end{bmatrix} = \mathrm{E}_{n,l,0}^{f_l}$$

$$\mathrm{R}_{n+r,l,0}^{\mathrm{zero}}\,\mathrm{E}_{n,l,r}^{f_l,f_r} = \Big[\,0_{n+r\times l}\,|\,\mathrm{I}_{n+r}\,\Big]\begin{bmatrix} E_l \\ \mathrm{I}_n \\ E_r \end{bmatrix} = \mathrm{E}_{n,0,r}^{f_r}$$

■

From the definition of the extended filter transform we obtain a few special cases that are frequently encountered in algorithm representations. They are the Toeplitz and the circulant transform.

Toeplitz. The *Toeplitz transform* represents a Toeplitz matrix whose elements are given by a polynomial $b(z)$.

$$\mathbf{T}_n(b(z)) = [b_{i-j}]_{n\times n}\,, \qquad b(z) = b_{n-1}\,z^{-(n-1)} + \cdots + b_0\,z + \cdots + b_{-(n-1)}\,z^{n-1}$$

It can be easily seen that the Toeplitz transform is a special case of the extended filter transform

$$\mathbf{T}_n(b(z)) = \mathbf{Filt}_n^{\mathrm{zero}}(b(z))\,. \tag{5.18}$$

Using the linearity of the extended filter transform, we can represent the Toeplitz transform by

$$\mathbf{T}_n(h(z)) = h_l\,\mathbf{T}_n\big(z^{-l}\big) + \cdots + h_0\,\mathrm{I}_n + \cdots h_r\,\mathbf{T}_n(z^r)$$

Toeplitz transforms of the form $\mathbf{T}_n\big(z^{-k}\big)$ and $\mathbf{T}_n\big(z^{k}\big)$ are the left and the right shift operators. In hardware, these operators are implemented by shift registers.

Example 5.4.

$$\mathbf{T}_5\big(z - 1 + z^{-1}\big) = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 \\ 0 & 1 & -1 & 1 & 0 \\ 0 & 0 & 1 & -1 & 1 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

An important role of the Toeplitz transform is to represent the independent blocks of the filter transform. We investigate the possible representations for several cases of filter transforms.

Given a polynomial $h(z) = h_l z^{-l} + \cdots + h_0 z + \cdots + h_{-r} z^r$, let the polynomials $h_l(z)$ and $h_r(z)$ be defined as

$$\begin{aligned} h_l(z) &= h_l z^{-l} + \cdots + h_1 z^{-1} \\ h_r(z) &= h_{-r} z^r + \cdots + h_{-1} z \end{aligned} \tag{5.19}$$

Let $n > l, r$. The filter transform $\mathbf{Filt}_n(h(z))$ can be represented using the Toeplitz transforms as

$$\mathbf{Filt}_n(h(z)) = \Big[\, \mathrm{E}^{\mathrm{zero}}_{l,0,n-l}\,\mathbf{T}_l\big(z^l h_l(z)\big) \,|\, \mathbf{T}_n(h(z)) \,|\, \mathrm{E}^{\mathrm{zero}}_{r,n-r,0}\,\mathbf{T}_r(z^{-r} h_r(z)) \,\Big] \tag{5.20}$$

In the case $n < l, r$, the extension operators become the reduction operators but the form of (5.20) stays the same.

$$\mathbf{Filt}_n(h(z)) = \Big[\, \mathrm{R}^{\mathrm{zero}}_{n,l-n,0}\,\mathbf{T}_l\big(z^l h_l(z)\big) \,|\, \mathbf{T}_n(h(z)) \,|\, \mathrm{R}^{\mathrm{zero}}_{n,0,r-n}\,\mathbf{T}_r(z^{-r} h_r(z)) \,\Big] \tag{5.21}$$

From (5.14) we can now represent the extended filter transform as

$$\begin{aligned} \mathbf{Filt}^{f_l,f_r}_n(h(z)) &= \Big[\, \mathrm{E}^{\mathrm{zero}}_{l,0,n-l}\,\mathbf{T}_l\big(z^l h_l(z)\big) \,|\, \mathbf{T}_n(h(z)) \,|\, \mathrm{E}^{\mathrm{zero}}_{r,n-r,0}\,\mathbf{T}_r(z^{-r} h_r(z)) \,\Big] \left[\begin{array}{c} E_l \\ \hline \mathrm{I}_n \\ \hline E_r \end{array}\right] \\ &= \begin{bmatrix} \mathbf{T}_l\big(z^l h_l(z)\big) \cdot E_l + \mathrm{R}^{\mathrm{zero}}_{l,0,n-l}\,\mathbf{T}_n(h(z)) \\ \mathbf{Filt}_{n-l-r}(h(z)) \\ \mathbf{T}_r(z^{-r} h_r(z)) \cdot E_r + \mathrm{R}^{\mathrm{zero}}_{r,n-r,0}\,\mathbf{T}_n(h(z)) \end{bmatrix} \end{aligned} \tag{5.22}$$

which provides the exact form of the matrices A_l and B_r in (5.15). The case when $n < l, r$ is obtained directly from (5.21). The last equality holds only when $n < l + r$, in which case the matrix $\mathrm{R}^{\mathrm{zero}}_{l,0,n-l}\,\mathbf{T}_n(h(z))$ represents the first l rows, and the matrix $\mathrm{R}^{\mathrm{zero}}_{r,n-r,0}\,\mathbf{T}_n(h(z))$ represents the last r rows of the Toeplitz transform $\mathbf{T}_n(h(z))$. In other cases, the matrix has to be obtained by the full multiplication in (5.22). This formula shows how to fuse the extension operator into the filter transform.

Circulant. The *Circulant transform* represents a circulant matrix whose elements are determined by the coefficients of a polynomial $a(z)$. The circulant transform is an $n \times n$ matrix defined

by

$$\mathbf{C}_n(a(z)) = [a_{i-j \bmod n}]_{n\times n}, \qquad a(z) = a_{n-1}z^{-(n-1)} + \cdots + a_0 \tag{5.23}$$

This definition can be generalized by noting that the circulant transform is a special case of the extended filter transform

$$\mathbf{C}_n(a(z)) = \mathbf{Filt}_n^{\text{per}}(a(z)) \tag{5.24}$$

With this definition, the degree of the polynomial $a(z)$ need not be $n-1$ as noted in (5.23). In general, a polynomial of degree $n' > n$ produces a circulant transform using the definition (5.24), although the elements of the circulant matrix are then not equal to the coefficients of the polynomial. If the polynomial $a(z)$ represents a z-transform of a causal filter as in (5.23), i.e., if it does not have positive degrees, then the coefficients form the first column of the circulant transform. Other columns are formed by the circular shift of the first one. We provide a few examples to clarify the definition.

EXAMPLE 5.5.

$$\mathbf{C}_n\Big(a_{n-1}z^{-(n-1)} + \cdots + a_0\Big) = \begin{bmatrix} a_0 & a_{n-1} & \cdots & a_1 \\ a_1 & a_0 & & a_2 \\ \vdots & & \ddots & \vdots \\ a_{n-1} & \cdots & \cdots & a_0 \end{bmatrix}$$

EXAMPLE 5.6.

$$\mathbf{C}_4\big(z^{-1} + 2 + 4z^1\big) = \begin{bmatrix} 2 & 4 & 0 & 1 \\ 1 & 2 & 4 & 0 \\ 0 & 1 & 2 & 4 \\ 4 & 0 & 1 & 2 \end{bmatrix}$$

EXAMPLE 5.7.

$$\mathbf{C}_2\big(2z^{-1} + 1 + z + 5z^2\big) = \begin{bmatrix} 2 & 1 & 1 & 5 & 0 \\ 0 & 2 & 1 & 1 & 5 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 6 & 3 \\ 3 & 6 \end{bmatrix}$$

An important property of the circulant transform is the multiplication of the circular shifts.

$$\mathbf{C}_n\big(z^2\big) = \mathbf{C}_n(z) \cdot \mathbf{C}_n(z) = \mathbf{C}_n(z)^2 \tag{5.25}$$

The circulant $\mathbf{C}_n\big(z^{-1}\big) = S_n$ is the circular shift matrix defined in (3.10).

Using properties (5.16) and (5.25), we can decompose any circulant transform into a linear combination of the circular shift matrices and their powers.

$$\begin{aligned} \mathbf{C}_n(h(z)) &= h_l\, \mathbf{C}_n\big(z^{-l}\big) + \cdots + h_0\, \mathrm{I}_n + \cdots h_r\, \mathbf{C}_n(z^r) \\ &= h_l S_n^l + \cdots + h_0\, \mathrm{I}_n + \cdots + h_r S_n^{-r \bmod n} \end{aligned} \tag{5.26}$$

This is exactly the decomposition described in Chapter 3 in equation (3.11). Hence, the circulant transform $\mathbf{C}_n(h(z))$ is the representation of the circular convolution with the filter $h(z)$ on n points.

5.1.3 Filter banks and polyphase matrices

Filter banks are encountered in multi-rate signal processing where the input signal is filtered by multiple filters (filter bank) at the same time. The operation of processing a signal through a filter bank can be viewed in the z-domain as a multiplication of an $s \times 1$ matrix of polynomials $h_0(z), \ldots, h_{s-1}(z)$ with the input signal polynomial $x(z)$ [82]. In the matrix representation of filter banks, the same issues of banding the infinite Toeplitz matrices arise as in the case of a single filter. We address these issues by defining the filter bank transform as a generalization of the filter transform in the following way.

Filter bank.. Let $[h_i(z)]_{s\times 1}]$ be an $s \times 1$ matrix of polynomials representing a filter bank of s filters with the transfer functions $h_i(z) = h_{l_i} z^{-l_i} + \cdots + h_{-r_i} z^{r_i}$. We define the filter bank transform as the filter transform given by

$$\mathbf{Filt}_n \left(\begin{bmatrix} h_0(z) \\ \vdots \\ h_{s-1}(z) \end{bmatrix} \right) = \begin{bmatrix} \mathbf{Filt}_n(h_0(z))\, \mathrm{R}^{\mathrm{zero}}_{n,\tilde{l}_0,\tilde{r}_0} \\ \mathbf{Filt}_n(h_1(z))\, \mathrm{R}^{\mathrm{zero}}_{n,\tilde{l}_1,\tilde{r}_1} \\ \vdots \\ \mathbf{Filt}_n(h_{s-1}(z))\, \mathrm{R}^{\mathrm{zero}}_{n,\tilde{l}_{s-1},\tilde{r}_{s-1}} \end{bmatrix}. \tag{5.27}$$

where

$$\begin{aligned} \tilde{l}_i &= l_{\max} - l_i, & l_{\max} &= \max_i l_i, \\ \tilde{r}_i &= r_{\max} - r_i, & r_{\max} &= \max_i r_i, \end{aligned} \tag{5.28}$$

We can also define the extended filter banks using the above definition

$$\mathbf{Filt}_n^{f_l,f_r} \left(\begin{bmatrix} h_0(z) \\ \vdots \\ h_{s-1}(z) \end{bmatrix} \right) = \mathbf{Filt}_n \left(\begin{bmatrix} h_0(z) \\ \vdots \\ h_{s-1}(z) \end{bmatrix} \right) \cdot \mathrm{E}^{f_l,f_r}_{n,l_{\max},r_{\max}} \tag{5.29}$$

Matrix of filter transforms.. Furthermore, multi-rate filter banks can be represented in the z-domain as matrices of polynomials using the so-called polyphase representation [15]. We extend the definition (5.27) to include $r \times s$ matrices of filters. Let $[h_{i,j}(z)]_{r\times s}$ be an $r \times s$ matrix of polynomials. The $r \times s$ filters have the transfer functions $h_{i,j}(z) = h_{l_{i,j}} z^{-l_{i,j}} + \cdots + h_{-r_{i,j}} z^{r_{i,j}}$. The filter transform of the matrix of polynomials is defined by

$$\mathbf{Filt}_n([h_{i,j}(z)]_{r\times s}) = \left[\mathbf{Filt}_n(h_{i,j}(z))\, \mathrm{R}^{\mathrm{zero}}_{n,\tilde{l}_{i,j},\tilde{r}_{i,j}} \right]_{r\times s}, \tag{5.30}$$

where

$$\begin{aligned} \tilde{l}_{i,j} &= l_{j\max} - l_{i,j}, & l_{j\max} &= \max_i l_{i,j}, \\ \tilde{r}_{i,j} &= r_{j\max} - r_{i,j}, & r_{j\max} &= \max_i r_{i,j}, \end{aligned} \tag{5.31}$$

Generalized matrix of filter transforms.. We also define a slightly more general form of the definition (5.30) where each filter transform in the matrix of filters has an explicitly specified left and

right extensions $L = [l_{i,j}]_{r\times s}$ and $R = [r_{i,j}]_{r\times s}$, rather then determined from the given polynomials. Also, the sizes of filter transforms are given by the matrix $N = [n_{i,j}]_{r\times s}$. In that case we have

$$\mathbf{Filt}_{N,L,R}([h_{i,j}(z)]_{r\times s}) = \left[\, \mathbf{Filt}_{n_{i,j}}(h_{i,j}(z))\, \mathrm{R}^{\mathrm{zero}}_{n_{i,j},\tilde{l}_{i,j},\tilde{r}_{i,j}} \,\right]_{r\times s}. \tag{5.32}$$

The parameters $\tilde{l}_{i,j}$ and $\tilde{r}_{i,j}$ are again specified by (5.31), where $l_{i,j}$ and $r_{i,j}$ are elements of matrices L and R, respectively. This definition will be useful for defining different types of wavelet transforms in Chapter 6.

Matrix of extended filter transforms. The extended filter transform of matrices of polynomials is defined by

$$\mathbf{Filt}^{f_l,f_r}_n([h_{i,j}(z)]_{r\times s}) = \mathbf{Filt}_n([h_{i,j}(z)]_{r\times s}) \cdot \bigoplus_{i=0}^{s-1} \mathrm{E}^{f_l,f_r}_{n,l_{j\max},r_{j\max}} \tag{5.33}$$

From this definition, it directly follows that

$$\mathbf{C}_n([h_{i,j}(z)]_{r\times s}) = [\mathbf{C}_n(h_{i,j}(z))]_{r\times s} \tag{5.34}$$

$$\mathbf{T}_n([h_{i,j}(z)]_{r\times s}) = [\mathbf{T}_n(h_{i,j}(z))]_{r\times s}\,. \tag{5.35}$$

We note that the definitions (5.27), (5.30), and (5.32) can be extended to convolution transforms by simple transposition. In that case we have

$$\mathbf{Conv}_n([h_{i,j}(z)]_{r\times s}) = \left[\, \mathrm{E}^{\mathrm{zero}}_{n,L_{i,j},R_{i,j}}\, \mathbf{Conv}_n(h_{i,j}(z)) \,\right]_{r\times s}, \tag{5.36}$$

and

$$\mathbf{Conv}_{N,L,R}([h_{i,j}(z)]_{r\times s}) = \left[\, \mathrm{E}^{\mathrm{zero}}_{n_{i,j},L_{i,j},R_{i,j}}\, \mathbf{Conv}_{n_{i,j}}(h_{i,j}(z)) \,\right]_{r\times s}. \tag{5.37}$$

We illustrate the definitions with a few examples.

Example 5.8.

$$\mathbf{Filt}_2\left(\begin{bmatrix} z^{-2}+2+z \\ 3+2z^2 \end{bmatrix}\right) = \left[\begin{array}{cccccc} 1 & . & 2 & 1 & . & . \\ . & 1 & . & 2 & 1 & . \\ \hline . & . & 3 & . & 2 & . \\ . & . & . & 3 & . & 2 \end{array}\right]$$

Example 5.9.

$$\mathbf{Filt}^{hs}_3\left(\begin{bmatrix} z^{-2} & z \\ 3+2z^2 & 1 \end{bmatrix}\right) = \left[\begin{array}{ccccccc|ccc} 1 & . & . & . & . & . & . & . & 1 & . & . \\ . & 1 & . & . & . & . & . & . & . & 1 & . \\ . & . & 1 & . & . & . & . & . & . & . & 1 \\ \hline . & . & 3 & . & 2 & . & . & 1 & . & . & . \\ . & . & . & 3 & . & 2 & . & . & 1 & . & . \\ . & . & . & . & 3 & . & 2 & . & . & 1 & . \end{array}\right] \begin{bmatrix} \mathrm{E}^{hs}_{3,2,2} & \\ & \mathrm{E}^{hs}_{3,0,1} \end{bmatrix}$$

$$= \left[\begin{array}{ccc|ccc} . & 1 & . & . & 1 & . \\ 1 & . & . & . & . & 1 \\ 1 & . & . & . & . & 1 \\ \hline 3 & . & 2 & 1 & . & . \\ . & 3 & 2 & . & 1 & . \\ . & 2 & 3 & . & . & 1 \end{array}\right]$$

5.1.4 Composition of filters and convolutions

Discrete-time signals have three equivalent representations described in (3.3). We discussed that the filtering of the sequence $\{x_k\}$ with an FIR filter $\{h_k\}$ can be seen as either polynomial multiplication $x(z)h(z)$ in (3.6), or as the matrix-vector product, where the type of matrix is either the convolution transform $\mathbf{Conv}_n(h(z))$, the filter transform$\mathbf{Filt}_n(h(z))$, or the extended filter transform $\mathbf{Filt}_n^{f_l,f_r}(h(z))$, depending on the method of reducing the infinite signal convolutions to finite cases.

We now explore the case when the input signal is processed by a cascade of two FIR filters given by their transfer functions $h(z)$ and $g(z)$. In the infinite case the result of this operation can be seen as the polynomial multiplication $y(z) = g(z)h(z)x(z)$. In the case of finite signals, the matrix representation of the multiplication has to be carefully defined. We start with the filter transform representation of the convolution with a sequence of two filters.

Lemma 5.5 (Composition of the convolution transforms). *Let the two polynomials* $h(z) = h_{l_1}z^{-l_1} + \cdots + h_0 + \cdots + h_{-r_1}z^{r_1}$ *and* $g(z) = g_{l_2}z^{-l_2} + \cdots + g_0 + \cdots + g_{-r_2}z^{r_2}$*, with* $r_i, l_i \geq 0$*, represent the transfer functions of two filters, and let their convolution transform representations on* n *input points be* $\mathbf{Conv}_n(h(z))$ *and* $\mathbf{Conv}_n(g(z))$*. Then the convolution transforms are composed as*

$$\begin{aligned} \mathbf{Conv}_n(h(z)\cdot g(z)) &= \mathbf{Conv}_{n+l_2+r_2}(h(z))\cdot\mathbf{Conv}_n(g(z)) \\ &= \mathbf{Conv}_{n+l_1+r_1}(g(z))\cdot\mathbf{Conv}_n(h(z)) \end{aligned} \tag{5.38}$$

Proof. Let $w(z) = g(z)x(z)$. Since the degree of $x(z)$ is n, the polynomial multiplication is represented as $\mathbf{w} = \mathbf{Conv}_n(g(z))\,\mathbf{x}$. We know from definition (3.5) that the degree of $w(z)$ is $n + l_2 + r_2$, so the multiplication $y(z) = h(z)w(z)$ can be written as $\mathbf{y} = \mathbf{Conv}_{n+l_2+r_2}(h(z))\,\mathbf{w}$. Therefore,

$$\mathbf{Conv}_n(h(z)\cdot g(z)) = \mathbf{Conv}_{n+l_2+r_2}(h(z))\cdot\mathbf{Conv}_n(g(z))$$

The second identity is obtained the same way. ■

Corollary 5.6 (Composition of filter transforms). *The filter transforms* $\mathbf{Filt}_n(h(z))$ *and* $\mathbf{Filt}_n(g(z))$*, with polynomials* $h(z)$ *and* $g(z)$ *given in lemma 5.5 are composed as*

$$\begin{aligned} \mathbf{Filt}_n(h(z)\cdot g(z)) &= \mathbf{Filt}_n(h(z))\cdot\mathbf{Filt}_{n+l_1+r_1}(g(z)) \\ &= \mathbf{Filt}_n(g(z))\cdot\mathbf{Filt}_{n+l_2+r_2}(h(z)) \end{aligned} \tag{5.39}$$

Proof. The identity is obtained by transposing and time-reversing the equations (5.38) and using the relation (5.8). ■

At this point, we remind the reader that it is wrong to jump to the conclusion that $\mathbf{Filt}_n\big(z^{-1}\cdot z\big) = \mathbf{Filt}_n(1)$. The reason is that z^{-1} is just a short notation for $z^{-1}+0z^0$ because of the condition $l, r \geq 0$. If, for example, $h(z) = z^t$ and $g(z) = z^{-s}$, then $l_1 = 0,\ r_1 = t,\ l_2 = s,\ r_2 = 0$, and identity (5.39) can be applied as defined.

Example 5.10.

$$\begin{aligned} \mathbf{Filt}_n\big(z^{-1}\cdot z\big) &= \mathbf{Filt}_n\big((z^{-1}+0z^0)\cdot(z+0z^0)\big) \\ &= \mathbf{Filt}_n\big(0z^{-1}+1+0z\big) \\ &= \mathbf{Filt}_n\big(z^{-1}\big)\cdot\mathbf{Filt}_{n+1}(z) \end{aligned}$$

Of course, both $\mathbf{Filt}_n\big(z^{-1}\cdot z\big)$ and $\mathbf{Filt}_n(1)$ produce the same result

$$(y_0\dots y_{n-1})^T=\mathbf{Filt}_n(1)\,(x_0\dots x_{n-1})=\mathbf{Filt}_{n+2}\big(z^{-1}\cdot z\big)\,(x_{-1}\dots x_n)$$

The only difference is the number of input points considered. In the second case, there are two more input points that do not contribute to the final result.

To avoid the additional zero points outside of the main interval that may appear as a consequence of filter compositions, the following manipulation of the formula can be applied. We give this result as a corollary and omit the proof.

Corollary 5.7. *Suppose that the polynomials $h(z)$ and $g(z)$ are of the form $h(z)=h'(z)z^{-s}$ and $g(z)=g'(z)z^t$, where $h'(z)=h'_0+\dots+h'_{p-1}z^{1-p}$ and $g'(z)=g'_0+\dots+g'_{q-1}z^{q-1}$. Then $f(z)=h(z)g(z)=z^{t-s}h'(z)g'(z)$ with $t\geq s$ and*

$$\begin{aligned}\mathbf{Filt}_n(f(z))&=\mathbf{Filt}_n(h(z)g(z))\\&=\mathbf{Filt}_n\big(z^{t-s}h'(z)g'(z)\big)\\&=\mathbf{Filt}_n\big(h(z)'z^{t-s}\big)\cdot\mathbf{Filt}_{n+p-1+t-s}(g'(z))\,.\end{aligned}$$

When $s>t$, the roles of $h(z)$ and $g(z)$ can be reversed.

Example 5.11. *Let $h(z)=z^s, g(z)=z^{-t}, s>t$.*

$$\begin{aligned}\mathbf{Filt}_n(h(z)g(z))&=\mathbf{Filt}_n\big(z^{s-t}\cdot 1\cdot 1\big)\\&=\mathbf{Filt}_n\big(z^{s-t}\big)\,\mathbf{Filt}_{n+s-t}(1)\\&=\mathbf{Filt}_n\big(z^{s-t}\big)\end{aligned}$$

Example 5.12.

$$\begin{aligned}\mathbf{Filt}_3\big((z^2+z)(2z^{-2}+z^{-3})\big)&=\mathbf{Filt}_3\big((z+1)z^{-1}\big)\,\mathbf{Filt}_4\big(2+z^{-1}\big)\\&=\begin{bmatrix}1&1&.&.\\.&1&1&.\\.&.&1&1\end{bmatrix}\begin{bmatrix}1&2&.&.&.\\.&1&2&.&.\\.&.&1&2&.\\.&.&.&1&2\end{bmatrix}\\&=\begin{bmatrix}1&3&2&.&.\\.&1&3&2&.\\.&.&1&3&2\end{bmatrix}\\&=\mathbf{Filt}_3\big(z^{-2}+3z^{-1}+2\big)\end{aligned}$$

Corollary 5.8 (Composition of extended filters). *The extended filter transform of the product of the polynomials $h(z)$ and $g(z)$ given in lemma 5.5 is decomposed as*

$$\begin{aligned}\mathbf{Filt}_n^{f_l,f_r}(h(z)\cdot g(z))&=\mathbf{Filt}_n(h(z))\cdot\mathbf{Filt}_{n+l_1+r_1}^{f_l,f_r}(g(z))\\&=\mathbf{Filt}_n(g(z))\cdot\mathbf{Filt}_{n+l_2+r_2}^{f_l,f_r}(h(z))\end{aligned}\tag{5.40}$$

Proof. The proof follows directly from the definition (5.13) and corollary 5.6. ∎

The identities (5.40) are always valid. However, we are interested in cases when the extended filter transforms can be decomposed into shorter extended filters. In other words, we look for cases when the extension can be distributed among all matrices in the composition. It turns out that this is possible if signal extensions arise from polynomial algebras $\mathbb{C}[z]/p(z)$ as we discussed in Section 4.3.3, in which case all matrices in the product are square.

Theorem 5.9 (Distribution of the extensions). *The extended filter transform of the product of the polynomials $h(z)g(z)$ is the product of extended filter transforms of $h(z)$ and $g(z)$ if the extension arises from polynomial algebras $\mathbb{C}[z]/p(z)$. The periodic* (4.75) *or any of the symmetric types* (4.78)–(4.87) *or their combinations belong to this class.*

$$\mathbf{Filt}_n^{f_l,f_r}(h(z)g(z)) = \mathbf{Filt}_n^{f_l,f_r}(h(z))\,\mathbf{Filt}_n^{f_l,f_r}(g(z))\,, \quad f_l, f_r \in \{\text{per,ws,hs,wa,ha}\} \tag{5.41}$$

Proof. Suppose $h(z), g(z) \in \mathbb{C}[z]/p(z)$ are two polynomials in the polynomial algebra modulo $p(z)$. Then

$$h(z)g(z) \bmod p(z) = h(z) \cdot (g(z) \bmod p(z)) \bmod p(z). \tag{5.42}$$

Suppose we define the polynomials $x(z) = x_0 + \cdots + x_{n-1}z^{-(n-1)}$, $h(z) = h_{l_1}z^{-l_1} + \cdots + h_0 + \cdots + h_{-r_1}z^{r_1}$ and $g(z) = g_{l_2}z^{-l_2} + \cdots + g_0 + \cdots + g_{-r_2}z^{r_2}$. We know that the matrix representation of $h(z)x(z)$ is $\mathbf{Conv}_n(h(z))\,\mathbf{x}$. Without loss of generality, suppose now that $p(z) = p_0 + \cdots + p_{n-1}z^{-(n-1)}$ is of the same degree n as $x(z)$. The matrix representation of the reduction modulo $p(z)$ is $\mathrm{R}_{n,l,r}^{p(z)}$ defined in (4.92), where $n + l + r$ is the length of the input sequence. So, the representation of $h(z) \cdot x(z) \bmod p(z)$ is

$$\mathrm{R}_{n,l_1,r_1}^{p(z)}\,\mathbf{Conv}_n(h(z))\,\mathbf{x}.$$

Then the representation of (5.42) is

$$\mathrm{R}_{n,l_1+l_2,r_1+r_2}^{p(z)}\,\mathbf{Conv}_n(h(z)g(z)) = \left(\mathrm{R}_{n,l_1,r_1}^{p(z)}\,\mathbf{Conv}_n(h(z))\right)\left(\mathrm{R}_{n,l_2,r_2}^{p(z)}\,\mathbf{Conv}_n(g(z))\right) \tag{5.43}$$

If we transpose this identity and use $z_1 = z^{-1}$ we obtain

$$\mathbf{Filt}_n(h(z_1)g(z_1))\,\mathrm{E}_{n,l_1+l_2,r_1+r_2}^{\mathrm{per}} = \left(\mathbf{Filt}_n(h(z_1))\,\mathrm{E}_{n,l_1,r_1}^{\mathrm{per}}\right)\left(\mathbf{Filt}_n(g(z_1))\,\mathrm{E}_{n,l_2,r_2}^{\mathrm{per}}\right) \tag{5.44}$$

which completes the proof.

As discussed in Section 3.1.3, the periodic extension arises from the polynomial algebra $\mathbb{C}[z]/(1-z^{-n})$, whereas the family of symmetric extensions arises from the polynomial algebras modulo Chebyshev polynomials. Thus, the theorem applies to periodic and all types of symmetric extensions. ■

5.2 Breakdown rules for filter and convolution transforms

In Chapter 3 we reviewed several methods for obtaining fast computational algorithms for convolutions. There are two main approaches to fast linear filtering: 1) reducing the arithmetic cost of computations by using the relationship to the discrete Fourier transform (DFT), or by exploiting the redundancies of the Toeplitz matrix structure; and 2) breaking down a large filter into multiple shorter filters to either localize the computations so that they can be scheduled better for the underlying platform and minimize data movement, or to make other algorithms, such as DFT based convolution, more efficient.

We define rules for filter, extended filter and convolution transforms. Rules are denoted as mappings of a transform into other transforms of the same or different type. The transforms are given in boldface. The right arrow operator denotes the decomposition operation. As discussed in Chapter 2, the rules provide the necessary formalism for the construction of rule trees that represent algorithms. The rules are applied to transforms, or non-terminals, as many times as possible until a base-case rule is reached. At that moment the rule tree is fully expanded and uniquely represents an algorithm.

We first define a few conversion rules, basic manipulation rules and the base-case rule before capturing rules for the set of well-known algorithms.

5.2.1 Basic identities and rules

We give a few important identities and basic rules for construction of filter transforms. We also revisit the identities and the properties obtained earlier in this chapter, and recast them as breakdown or transform conversion rules.

For a filter with transfer function $h(z) = h_l z^{-l} + \cdots + h_0 z + \cdots + h_{-r} z^r$ and length of the input sequence n, we define the following rules.

FILTER-CONVOLUTION RULE

$$\begin{aligned} \mathbf{Conv}_n(h(z)) &\rightarrow \mathbf{Filt}_n\big(h(z^{-1})\big)^T \\ \mathbf{Filt}_n(h(z)) &\rightarrow \mathbf{Conv}_n\big(h(z^{-1})\big)^T \end{aligned} \tag{5.45}$$

The filter-convolution rule provides a connection between filter and convolution transform rules. Using this rule, we can recast every filter transform rule into the related convolution transform rule and vice-versa. To avoid repetitive definitions we shall define rules only for the filter transform.

EXTENDED FILTER RULE

$$\mathbf{Filt}_n^{f_l,f_r}(h(z)) \rightarrow \mathbf{Filt}_m(h(z)) \cdot \mathrm{E}_{n,l,p}^{f_l,f_r} \tag{5.46}$$

We note that, by applying rule (5.45) to the extended filter rule, we obtain the representation of the convolution operation followed by the reduction operation.

$$\Big(\mathbf{Filt}_n^{f_l,f_r}(h(z))\Big)^T \rightarrow \mathrm{R}_{n,l,p}^{f_l,f_r} \cdot \mathbf{Conv}_m(h(z)) \tag{5.47}$$

If the reduction arises from the quotient polynomial algebra $\mathbb{C}[x]/p(z)$, as discussed in theorem 5.9, then rule (5.47) affords the representation of the polynomial multiplication in $\mathbb{C}[x]/p(z)$.

However, it is often more efficient to fuse the extension matrices inside filters to obtain closed forms for extended filters, such as circulants and Toeplitz transforms. The fusion for any linear extension matrix is obtained using the following rule

FUSED EXTENSION FILTER RULE

$$\mathbf{Filt}_n^{f_l,f_r}(h(z)) \rightarrow \begin{bmatrix} \mathbf{T}_l\big(z^l h_l(z)\big) \cdot E_l + \mathrm{R}_{l,0,n-l}^{\mathrm{zero}}\, \mathbf{T}_n(h(z)) \\ \mathbf{Filt}_{n-l-r}(h(z)) \\ \mathbf{T}_r(z^{-r} h_r(z)) \cdot E_r + \mathrm{R}_{r,n-r,0}^{\mathrm{zero}}\, \mathbf{T}_n(h(z)) \end{bmatrix} \tag{5.48}$$

where the matrices E_l and E_r are left and right extension matrices defined in (5.14) and $h_l(z)$ and $h_r(z)$ are polynomials defined in (5.19) on page 97.

BASE-CASE RULE

$$\mathbf{Filt}_n(h(z)) \to \mathrm{I}_n \otimes_{r+l} (h_l \dots h_{-r}) \tag{5.49}$$

The base case rule terminates the decomposition of the filter transform. Note that there are no transforms on the right side of the rule. The base case rules for the convolution and the extended filter transforms are obtained by applying (5.45) and (5.46).

TOEPLITZ IDENTITY

$$\mathbf{Filt}_n^{\mathrm{zero}}(h(z)) = \mathbf{T}_n(h(z)) \tag{5.50}$$

CIRCULANT IDENTITY

$$\mathbf{Filt}_n^{\mathrm{per}}(h(z)) = \mathbf{C}_n(h(z)) \tag{5.51}$$

FILTER BANK RULE

$$\mathbf{Filt}_n([h_{i,j}(z)]_{r\times s}) \to \left[\, \mathbf{Filt}_n(h_{i,j}(z))\, \mathrm{R}^{\mathrm{zero}}_{n,L_{i,j},R_{i,j}} \,\right]_{r\times s}, \tag{5.52}$$

The filter bank rule shows how a filter transform of a matrix of $r \times s$ polynomials $h_{i,j}$ defined in (5.30) is constructed as a matrix of single filter transforms. The parameters $L_{i,j}$ and $R_{i,j}$ are given in (5.31).

So, the filter transform of a matrix of polynomials is not the matrix of filter transforms, but rather a matrix of filters extended by a sufficient number of zero columns to match the dimensions of matrices and synchronize the filters in time.

EXTENDED FILTER BANK RULE

$$\mathbf{Filt}_n^{f_l,f_r}([h_{i,j}(z)]_{r\times s}) = \mathbf{Filt}_n([h_{i,j}(z)]_{r\times s}) \cdot \bigoplus_{i=0}^{s-1} \mathrm{E}^{f_l,f_r}_{n,l_{j\max},r_{j\max}} \tag{5.53}$$

Similar to the extended filter rule, the extended filter bank can be converted to the regular filter bank with extensions defined for each column of filters. The parameters $l_{j\max}$ and $r_{j\max}$ are given in (5.31).

FILTER COMPOSITION IDENTITY

$$\mathbf{Filt}_n(h(z)\cdot g(z)) = \mathbf{Filt}_n(h(z)) \cdot \mathbf{Filt}_{n+l_1+r_1}(g(z)) \tag{5.54}$$

The composition rules for convolution and extended filter transforms are obtained by applying (5.45) and (5.46).

EXTENSION DISTRIBUTION IDENTITY

$$\mathbf{Filt}_n^{f_l,f_r}(h(z)g(z)) = \mathbf{Filt}_n^{f_l,f_r}(h(z))\, \mathbf{Filt}_n^{f_l,f_r}(g(z)), \quad f_l, f_r \in \{\text{per,ws,hs,wa,ha}\} \tag{5.55}$$

The distribution rule applies to extended filter transforms with periodic and all symmetric extensions.

5.2.2 Block convolution rules

It is frequently the case that the length of the filtered signal is much larger then the filter length, as we discussed in section 3.1.5. When this is the case, the filter transform matrices become very sparse, with only a narrow strip of non-zero values on and around the diagonal. As we discussed in Section 3.1.5, the benefit of reducing this matrix into multiple smaller but denser matrices is becoming obvious in the transform domain, where the size of the transform depends only on the size of the input. The advantage of computing the convolution using the transform based methods is diminished if the filter transform has many zero elements, since in that case it might be faster to perform the filtering directly from the definition. To utilize the full power of transform-domain methods, we introduce the block convolution rules. Block convolution methods are well known and we reviewed them in detail in Section 3.1.5. However, against the traditional view, we discuss block convolutions independent of transform-based algorithms since they present basic methods for localizing the computations and input/output data access patterns.

Overlap-Save Rule

$$\mathbf{Filt}_n(h(z)) \to \mathrm{I}_s \otimes_{l+r} \mathbf{Filt}_{n/s}(h(z)), \quad s \mid n \tag{5.56}$$

The overlap save method was described in Chapter 3 as segmenting and overlapping the input sequence at enough points (3.22) to compute the exact output for n/s output points. The method is usually described in the literature as we presented it in Section 3.1.5. However, it can be concisely represented using the overlap save rule. By choosing larger blocking parameter s, the filter transform is reduced to smaller filter transforms computed on a smaller number of output points. Therefore, the overlap-save rule exhibits the output locality of the performed computations. If $s \nmid n$, then we have an obvious generalization of (5.56). Suppose $n = sq + t$ then

$$\mathbf{Filt}_n(h(z)) \to \left(\mathrm{I}_s \otimes_{l+r} \mathbf{Filt}_{n/s}(h(z))\right) \oplus_{l+r} \mathbf{Filt}_t(h(z)), \quad s \nmid n$$

Overlap-Add Rule

$$\mathbf{Filt}_n(h(z)) \to \mathrm{R}^{\mathrm{zero}}_{n,l+r,l+r} \left(\mathrm{I}_s \otimes^{l+r} \mathbf{Conv}_{(n+l+r)/s}\left(h(z^{-1})\right)\right), \quad s|(n+l+r) \tag{5.57}$$

The overlap-add method was given in Section 3.1.5 with equation (3.21). After capturing the algorithm as the rule (5.57), it becomes clear that the overlap-add method is simply the transposed operation of the overlap-save method. The only adjustment is the addition of a reduction operator that removes excess boundary rows that arise from the rectangular shape of the filter transform. The rule is valid whenever the block size s divides the number of columns $n + l + r$. If that is not the case, the simple adjustment of the rule is

$$\mathbf{Filt}_n(h(z)) \to \mathrm{R}^{\mathrm{zero}}_{n,l+r,l+r} \left(\mathrm{I}_s \otimes^{l+r} \mathbf{Conv}_{(n+l+r)/s}\left(h(z^{-1})\right) \oplus^{l+r} \mathbf{Conv}_t(h(z))\right), \quad s \nmid (n+l+r)$$

5.2.3 Multidimensional rules

We briefly reviewed the topic of multidimensional techniques in Section 3.1.5. Unlike block convolution methods, the assumption here is that the filter length is of the same order as the input sequence

length. In that case, the filter transform is already a dense matrix and the transform domain methods can be applied directly without the preprocessing using block convolution methods. However, it might be advantageous to split the large matrix into a block matrix of smaller filters, where they can be computed using short convolution or transform-domain algorithms. We define rules that achieve this objective, and refer to them as the nesting rules.

Let $h(z) = h_l z^{-l} + \cdots + h_0 z + \cdots + h_{-r} z^r$ be a polynomial representing a filter impulse response and define downsampled filters $h_0(z)$ and $h_1(z)$ as

$$h_0(z^2) = \frac{h(z) + (-1)^k h(-z)}{2z^{-k}}, \quad h_1(z^2) = \frac{h(z) + (-1)^{k-1} h(-z)}{2z^{k-1}}, \quad k = l \bmod 2 \tag{5.58}$$

Then the convolution transform can be decomposed into a matrix of convolution transforms with downsampled filters by the following rules

Convolution nesting rules

$$\begin{aligned} \mathbf{Conv}_n(h(z)) &\to \bar{\mathrm{L}}^{n+l+r}_{\frac{n}{2}+\lceil\frac{r+l}{2}\rceil}\, \mathbf{Conv}_{n/2}\left(\begin{bmatrix} h_0(z) & h_1(z)z^{-1} \\ h_1(z) & h_0(z) \end{bmatrix}\right) \mathrm{L}^n_2, \quad (r+l) \nmid 2 \\ \mathbf{Conv}_n(h(z)) &\to \mathrm{L}^{n+l+r}_{\frac{n+r+l}{2}}\, \mathbf{Conv}_{n/2}\left(\begin{bmatrix} h_0(z) & h_1(z)z^{-1} \\ h_1(z) & h_0(z) \end{bmatrix}\right) \mathrm{L}^n_2, \quad (r+l) \mid 2. \end{aligned} \tag{5.59a}$$

By transposing the above rules we obtain

Filter nesting rules

$$\begin{aligned} \mathbf{Filt}_n(h(z)) &\to \mathrm{L}^n_{n/2}\, \mathbf{Filt}_{n/2}\left(\begin{bmatrix} h_0(z) & h_1(z) \\ h_1(z)z & h_0(z) \end{bmatrix}\right) \bar{\mathrm{L}}^{n+r+l}_2, \quad (r+l) \nmid 2 \\ \mathbf{Filt}_n(h(z)) &\to \mathrm{L}^n_{n/2}\, \mathbf{Filt}_{n/2}\left(\begin{bmatrix} h_0(z) & h_1(z) \\ h_1(z)z & h_0(z) \end{bmatrix}\right) \mathrm{L}^{n+r+l}_2, \quad (r+l) \mid 2 \end{aligned} \tag{5.59b}$$

It is straightforward to obtain the nesting rule for the extended filter transforms from (5.59b) by applying (5.46). For example,

$$\mathbf{Filt}_n(h(z)) \to \mathrm{L}^n_{n/2}\, \mathbf{Filt}_{n/2}\left(\begin{bmatrix} h_0(z) & h_1(z) \\ h_1(z)z & h_0(z) \end{bmatrix}\right) \mathrm{L}^{n+l+r}_2 . \tag{5.60}$$

However, in a few special cases, the extension operator E can be commuted with the stride permutation to break down the extension operation into two parallel extensions of the same type. We discussed in Section 4.3.3 that this can be done in the case of zero extension, periodic extension, or a combination of the two. In other words, the condition is $f_l, f_r \in \{\mathrm{zero}, \mathrm{per}\}$. We capture this special case in the form of the following rule:

Extended filter nesting rule

$$\mathbf{Filt}^{f_l,f_r}_n(h(z)) \to \quad \mathbf{Filt}^{f_l,f_r}_{n/2}\left(\begin{bmatrix} h_0(z) & h_1(z) \\ h_1(z)z & h_0(z) \end{bmatrix}\right)^{\mathrm{L}^n_2}, \quad f_l, f_r \in \{\mathrm{zero}, \mathrm{per}\} \tag{5.61}$$

In other words, in the case of zero padded input or periodic extension (i.e., circulant matrix), an extended filter can be represented by a block matrix of downsampled filters with the same extension by conjugating the transform with a stride permutation. In the case the extension is periodic, we can write

CIRCULANT NESTING RULE

$$\mathbf{C}_n(h(z)) \rightarrow \quad \mathbf{C}_{n/2}\left(\begin{bmatrix} h_0(z) & h_1(z) \\ h_1(z)z & h_0(z) \end{bmatrix}\right)^{\mathrm{L}_2^n}. \tag{5.62}$$

So, the cyclic convolution can be computed by combining (nesting) smaller size cyclic convolutions, which can in turn be computed using short convolution algorithms. The circulant transform is represented by a block matrix where blocks are circulants.

As discussed in Chapter 3, it is possible to block a circulant matrix into a block circulant matrix where each block is also a circulant. This is achieved by mapping the indices into two sets of indices using the Chinese remainder theorem (CRT). This effectively maps a one-dimensional into two-dimensional cyclic convolution that is shorter in each dimension, allowing application of more efficient algorithms for shorter convolutions.

To use the CRT mapping, the size of the transform n has to be a composite number with relatively prime factors $n = n_1\, n_2,\ (n_1, n_2) = 1$. This is similar to the Good-Thomas prime factor algorithm for the DFT [35]. In this case the following rule applies.

AGARWAL-COOLEY RULE

$$\mathbf{C}_n(h(z)) \rightarrow \mathbf{C}_{n_2}\Big([h_{i-j \bmod n}(z)]_{n_1 \times n_1}\Big)^{\mathrm{CRT}^T_{n_1,n_2}}, \quad n = n_1 n_2,\ (n_1, n_2) = 1 \tag{5.63}$$

where,

$$h_i(z) = \sum_{j=0}^{n_2} \hat{h}_{in_1+j}\ z^j$$

$$(\hat{h}_0, \ldots, \hat{h}_{n-1})^T = \mathrm{CRT}_{n_1,n_2} \cdot (h'_0, \ldots, h'_{n-1})^T,$$

$$h'(z) = h(z) \bmod (z^n - 1)$$

The rule says that the circulant transform conjugated by the CRT permutation defined in (4.55) takes the form of a block circulant matrix with circulant transforms as blocks. This block transform represents a 2-D circular convolution, which allows not only application of short convolution algorithms for smaller circulants (i.e., circular convolutions) as was the case with the nesting rule (5.62), but also the same efficient algorithms can be applied along the other dimension [35].

5.2.4 Embedding rules

We mentioned in Chapter 3 that every convolution type can be embedded into other types by noticing that the linear convolution seen as polynomial product $h(z)x(z)$ is the same as other types of convolutions represented by polynomial products in $\mathbb{C}[z]/m_1(z)$ as long as $\deg(h) + \deg(x) < \deg(m_1)$. Then the linear convolution can be converted to other types by reducing the product modulo some other polynomial $m_2(z)$. We discussed this in Section 3.1.4.

From the representation (5.15) we immediately obtain the following important rule

FILTER EXTENDED RULE

$$\mathbf{Filt}_n(h(z)) \to \mathrm{R}^{\mathrm{zero}}_{n,l,r} \cdot \mathbf{Filt}^{f_l,f_r}_{n+l+r}(h(z))\,. \tag{5.64}$$

As a special case we have

FILTER CIRCULANT RULE

$$\mathbf{Filt}_n(h(z)) \to \mathrm{R}^{\mathrm{zero}}_{n,l,r} \cdot \mathbf{C}_{n+l+r}(h(z)) \tag{5.65}$$

If we combine this with the extended filter rule (5.46) then we obtain

FILTER EXTENSION EMBEDDING RULE

$$\mathbf{Filt}^{f'_l,f'_r}_n(h(z)) \to \mathrm{R}^{\mathrm{zero}}_{n,l,r} \cdot \mathbf{Filt}^{f_l,f_r}_{n+l+r}(h(z)) \cdot \mathrm{E}^{f'_l,f'_r}_{n,l',r'} \tag{5.66}$$

This rule is most general in that it provides means for expressing any extended filter through any other extended filter, thus embedding different convolution types into one another.

5.2.5 Karatsuba rules

We discussed divide-and-conquer methods for nested filters in Section 3.1.6. We refer to them as Karatsuba methods since they arise from the Karatsuba algorithm for large integer multiplication well known in the scientific computing community.

To capture rules for the Karatsuba methods, we note that the polyphase signal decomposition discussed in Section 3.1.6 is essentially splitting the input signal into polyphase channels by downsampling. Each channel corresponds to a downsampled signal with different offsets. Using the relation between the downsampling operator and the stride permutation given in (4.65), we can see the polyphase decomposition as a nesting of filters by conjugating the filter transform with a stride permutation. In fact, the two channel polyphase decomposition is captured by the nesting rules in (5.59) that form matrices of filters. The Karatsuba methods exploit the redundancies in the resulting filter matrices.

We start with the radix-2 Karatsuba rule for the convolution transform. Let $h(z) = h_l z^{-l} + \cdots + h_0 z + \cdots + h_{-r} z^r$ and let $h_0(z)$ and $h_1(z)$ be the downsampled filters defined in (5.58). For simplicity, we assume that $2 \mid (r + l + 1)$. The radix-2 convolution Karatsuba rule decomposes a convolution matrix obtained by the rule (5.59a) into a product of convolution matrices:

RADIX-2 KARATSUBA CONVOLUTION RULE

$$\begin{aligned}\mathbf{Conv}_n(h(z)) \to \bar{\mathrm{L}}^{n+l+r}_{\frac{n}{2}+\lceil\frac{r+l}{2}\rceil} \cdot \mathbf{Conv}_{\frac{n}{2}+\frac{r+l-1}{2}} \left(\begin{bmatrix} 1 & z^{-1} & 0 \\ -1 & -1 & 1 \end{bmatrix} \right) \cdot \\ \mathbf{Conv}_{\frac{n}{2}} \left(\begin{bmatrix} h_0(z) & & \\ & h_1(z) & \\ & & h_0(z)+h_1(z) \end{bmatrix} \right) \cdot \mathbf{Conv}_{\frac{n}{2}} \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \right) \cdot \mathrm{L}^n_2\end{aligned} \tag{5.67}$$

By transposition we obtain

Radix-2 Karatsuba Filter Rule

$$\mathbf{Filt}_n(h(z)) \to \mathrm{L}^n_{\frac{n}{2}}\, \mathbf{Filt}_{\frac{n}{2}}\left(\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}\right) \cdot \mathbf{Filt}_{\frac{n}{2}}\left(\begin{bmatrix} h_0(z) & & \\ & h_1(z) & \\ & & h_0(z)+h_1(z) \end{bmatrix}\right) \cdot \mathbf{Filt}_{\frac{n}{2}+\frac{r+l-1}{2}}\left(\begin{bmatrix} 1 & -1 \\ z & -1 \\ 0 & 1 \end{bmatrix}\right) \cdot \mathrm{L}_2^{n+r+l} \quad (5.68)$$

In the case of the zero-padded signal or the periodic extension, the same rule can be easily extended to circulant matrices by applying the rule (5.46) to the above equations, commuting the periodic extension operator with the stride permutation, and finally applying Theorem 5.9. With little effort we obtain two radix-2 Karatsuba rules for a circulant transform.

Radix-2 Karatsuba Circulant Rule

$$\mathbf{C}_n(h(z)) \to \mathrm{L}^n_{\frac{n}{2}} \cdot \mathbf{C}_{\frac{n}{2}}\left(\begin{bmatrix} 1 & z^{-1} & 0 \\ -1 & -1 & 1 \end{bmatrix}\right) \cdot \mathbf{C}_{\frac{n}{2}}\left(\begin{bmatrix} h_0(z) & & \\ & h_1(z) & \\ & & h_0(z)+h_1(z) \end{bmatrix}\right) \cdot \mathbf{C}_{\frac{n}{2}}\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}\right) \cdot \mathrm{L}_2^n \quad (5.69a)$$

$$\mathbf{C}_n(h(z)) \to \mathrm{L}^n_{\frac{n}{2}}\, \mathbf{C}_{\frac{n}{2}}\left(\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}\right) \cdot \mathbf{C}_{\frac{n}{2}}\left(\begin{bmatrix} h_0(z) & & \\ & h_1(z) & \\ & & h_0(z)+h_1(z) \end{bmatrix}\right) \cdot \mathbf{C}_{\frac{n}{2}}\left(\begin{bmatrix} 1 & -1 \\ z & -1 \\ 0 & 1 \end{bmatrix}\right) \cdot \mathrm{L}_2^n \quad (5.69b)$$

We mentioned in Section 3.1.6 that the Karatsuba methods can be derived for higher radices, for example, using the approach we present in Appendix B. For example the radix-3 Karatsuba rule is given by (3.43).

Radix-3 Karatsuba Convolution Rule

$$\mathbf{Conv}_n(h(z)) \to \bar{\mathrm{L}}^{n+l+r}_{\frac{n}{3}+\lceil\frac{r+l}{3}\rceil}\, \mathbf{Conv}_{\frac{n}{3}+\frac{r+l-1}{3}}\left(\begin{bmatrix} 1 & z^{-1} & z^{-1} & 0 & 0 & 1 \\ 1 & 1 & z^{-1} & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}\right) \cdot \mathbf{Conv}_n\left(\mathrm{diag}\left(h_0(z), h_1(z), h_2(z), (h_0+h_1)(z), (h_0+h_2)(z), (h_1+h_2)(z)\right)\right) \cdot \mathbf{Conv}_{\frac{n}{3}}\left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}\right) \mathrm{L}_3^n \quad (5.70)$$

with the assumption that $3 \,|\, n$ and $3 \,|\, (l + r + 1)$.

5.2.6 Transform-domain rules

As discussed in Section 3.1.8, the transform-domain techniques for performing various convolution types provide savings in the cost of computations by invoking fast algorithms for computing trigonometric transforms, such as FFT algorithms. This reduces the cost from $\mathcal{O}\left(n^2\right)$ to $\mathcal{O}\left(n \log n\right)$ but introduces more complex data flow patterns, where the preferred Toeplitz structure of the operation is lost. The transform-domain techniques rely on convolution property of many transforms that we briefly addressed in Section 3.1.8. The best-known is the DFT convolution property (3.50) that we straightforwardly translate into the following rule

DFT Convolution Rule

$$\begin{aligned} \mathbf{C}_n(h(z)) &\rightarrow \mathbf{DFT}_n^{-1} \cdot \mathrm{diag}(\hat{\mathbf{h}}) \cdot \mathbf{DFT}_n, \\ \hat{\mathbf{h}} &= \mathrm{DFT}_n \cdot \cdot \mathbf{h}, \end{aligned} \tag{5.71}$$

where $\mathbf{h}$ is the first column of $\mathbf{C}_n(h(z))$.

This rule is often used in conjunction with the rules (5.56) and (5.57) and the application of (5.65) to compute the linear convolution of long input sequences with shorter filters. In practice, filters and signals are often real valued; hence, the DFT, being a complex transform, is not suitable for transform-domain filtering. Real transforms, such as the RDFT and DHT discussed in Section 3.1.8, are used instead. We already formulated the convolution properties of these two transforms using the matrix form in (3.53) and (3.57), so we simply rewrite them here as convolution rules.

RDFT Convolution Rule

$$\begin{aligned} \mathbf{C}_n(h(z)) &\rightarrow \mathbf{RDFT}_n^{-1} \cdot X(\hat{\mathbf{h}}) \cdot \mathbf{RDFT}_n, \\ \hat{\mathbf{h}} &= \mathbf{RDFT}_n \cdot \mathbf{h} \end{aligned} \tag{5.72}$$

DHT Convolution Rule

$$\begin{aligned} \mathbf{C}_n(h(z)) &\rightarrow \mathbf{DHT}_n^{-1} \cdot X'(\hat{\mathbf{h}}) \cdot \mathbf{DHT}_n, \\ \hat{\mathbf{h}} &= \mathbf{DHT}_n \cdot \mathbf{h} \end{aligned} \tag{5.73}$$

Again, $\mathbf{h}$ is the first column of $\mathbf{C}_n(h(z))$, the $X(\mathbf{h})$ matrix is defined in (3.54), and the matrix $X'(\mathbf{h})$ in (3.58).

5.2.7 Blocking rules

It is often preferable to block the transforms into smaller transforms to improve the locality of the computations. The basic idea is to localize the operations so that they can be either performed in-register using smaller blocks or in-cache using larger blocks of operations, similar to the technique used by ATLAS [1]. We apply blocking rules whenever the decision is made to implement the entire transform as a basic matrix-vector multiplication. The matrices are blocked into square submatrices recursively, using one or more levels of blocking. For example, it may be useful to block the computations in two recursive levels for cache and register locality, respectively.

Circulant Blocking Rule

$$\mathbf{C}_n(h(z)) \to \left[- \right]_{i=0}^{\frac{n}{b}} \left[\begin{array}{c|c} & \\ \end{array} \right]_{j=0}^{\frac{n}{b}} \mathbf{T}_b\Big(h(z)z^{(i-j)b} \bmod (1-z^{-n})\Big), \tag{5.74}$$

where b is the size of the basic block. The matrix can be recursively blocked by blocking the Toeplitz transforms using the following rule.

Toeplitz Blocking Rule

$$\mathbf{T}_n(h(z)) \to \left[- \right]_{i=0}^{\frac{n}{b}} \left[\begin{array}{c|c} & \\ \end{array} \right]_{j=0}^{\frac{n}{b}} \mathbf{T}_b\Big(h(z)z^{(i-j)b}\Big) \tag{5.75}$$

Finally, we present the blocking rule for the filter transform described in similar format.

Filter Blocking Rule

$$\begin{aligned} \mathbf{Filt}_n(h(z)) \to & \mathrm{I}_{\lfloor \frac{n}{b} \rfloor} \otimes_{l+r} \left(\left[\begin{array}{c|c} & \\ \end{array} \right]_{i=0}^{\lceil \frac{l+r}{b} \rceil} \mathbf{T}_b\big(h(z)z^{l-ib}\big) \oplus^k \mathbf{T}_k\Big(h(z)z^{l-\lceil \frac{l+r}{b} \rceil b-k}\Big) \right) \\ & \oplus_{l+r} \mathbf{T}_{b_1}\big(h(z)z^{l-ib_1}\big) \oplus^{k_1} \mathbf{T}_{k_1}\left(h(z)z^{l-\lceil \frac{l+r}{b_1} \rceil b_1-k_1}\right) \end{aligned} \tag{5.76}$$

where

$$\begin{aligned} k &= (l+r+b) \bmod b \\ k_1 &= \begin{cases} (l+r+b_1) \bmod b_1, & b_1 \neq 0 \\ 0, & b_1 = 0 \end{cases} \\ b_1 &= n \bmod b \end{aligned}$$

This rule avoids blocking of zero coefficients. There is yet another, more natural method for blocking the filter transforms. It represents the combination of the blocking strategies and filter nesting techniques.

Filter Nested Blocking Rule

$$\mathbf{Filt}_n(h(z)) \to \left(\left[\begin{array}{c|c} & \\ \end{array} \right]_{j=0}^{\frac{l+r+1}{b}} \mathbf{Filt}_b(h_j(z)) \right) \left(\mathrm{I}_{\frac{l+r+1}{b}} \otimes_{n-1} \mathrm{I}_{n+b-1} \right) \tag{5.77}$$

where

$$h_i(z) = h_{l-ib} \cdot z^{ib-l} + \cdots + h_{l-(i+1)b+1} \cdot z^{(i+1)b-l-1}$$

If $b \nmid (l+r+1)$ then a simple modification of the above rule will take care of the remainder of the division – an additional filter with less than b coefficients. This rule, in combination with the overlap-save rule (5.56), allows blocking of filters into smaller size and smaller length filters.

5.3 Concluding remarks

In this chapter, we have introduced several definitions for filtering operations: 1) convolution of infinite signals on finite number of outputs; 2) linear convolution of compactly supported signals; 3) convolution of infinitely extended signals including circular convolution; 4) banks of filters for different filter definitions. All definitions are represented as transforms that can be included in SPIRAL's formula generator. We investigated important properties, such as the composition of

finite filters and distribution of signal extensions, that help us to define breakdown rules for different filter definitions.

From Chapter 3, we captured most important algorithms for fast computation of FIR filters in an extended set of decomposition rules that, when expanded into rule trees, covers the whole space of possibilities. In addition, we designed rules that have no impact on the computational cost of the algorithms but lead to algorithms that have structure more suitable for implementation on computer platforms whose hardware architectures share common features, such as multi-level memory hierarchy and register banks. A few examples include blocking rules, nesting rules, and overlap-add/save (OA/OS) rules. We emphasize once more that the OA/OS rules do not have to be combined with the circulant rule (5.65) to compute the convolutions using circular convolutions, in contrast to most references to OA/OS methods in the literature. We consider them as simple convolution blocking methods for input and output locality, where the smaller filters are then computed using any of the available methods, not just the circular convolution, by applying different rules recursively.

Search space. The large set of rules we presented in this chapter creates a comprehensive search space of algorithms by combining and applying the rules recursively in as many ways as possible. To reduce the search space and the search time, it is important to limit the applicability of certain rules to specific transform sizes and parameters. For example, it is clear that the circulant rule (5.65) will not be efficient unless the circulant transform is dense (i.e., has most non-zero elements). For sparse circulants, it might be more efficient to apply the extended filter rule (5.48) and compute the circulants through filter transforms, or simply implement them in a straightforward manner using their definition. We can avoid searching for unnecessary rule trees by limiting the circulant rule to be applicable only for circulant transforms with, say, at most half of the entries being zero. Such limits can be set for many rules, thus limiting the search space and speeding up the search. However, the restrictions should not be hard-wired but determined empirically for different classes of computer platforms.

Practical issues. The mathematical framework we developed in this chapter has to be implemented in several steps:

1. First, all the new constructs, mathematical operators, and matrices need to be defined and implemented in SPIRAL's formula generator, and verified for mathematical correctness using a symbolic algebra system;

2. For translation into code, programming constructs for all mathematical objects should either be defined through translation templates or expressed using lower level mathematical structures on the formula optimization level, which are suitable for optimization of programming structures (see Chapter 2). The main challenge is to draw a connection between mathematical formulas and elements of a computer program, such as loop code, and be able to combine the programming structures in larger formulas in an efficient way. For more information we refer to [39].

3. After implementing the rules in the formula generator and enabling the translation into code, the rules need to be verified on the code level for all possible expansions. Code verification is explained in Section 2.4.

4. There are different issues involving optimization of search strategies. One concern is the existence of infinite search loops that can occur when two rules are interconnected by their children transforms. This problem needs to be addressed and solved by restricting the application of the rules depending on the transform parameters, such as the size of the transform and the length of polynomials. For example, the potential loop between rules (5.65) and (5.48) should be resolved by restricting the circulant rule to larger sizes (because transform-domain rules are more efficient for larger sizes as we shall see in Chapter 7), where at the same time restricting the rule (5.48) to cases either when polynomial length is much shorter than the size of the transform or when the size of the circulant is smaller.

   Another problem is the redundant search over rule trees that are different but conceptually yield the same code. For example, each of the blocking rules in Section 5.2.7 breaks down a transform into Toeplitz matrices. Most of these matrices have the same structure and only different entries, so there is no reason to decompose them differently. However, the search engine regards each transform with different parameters separately, which creates unnecessarily large search spaces. The search engine should search only over one of the similar transforms and apply the same decomposition strategy to all others during the translation into code. This can be done by treating the specific values of the parameters, such as filter coefficients, as abstract data structures that only differ in size and not the content. The search engine then treats all transforms, such as Toeplitz matrices in blocking rules, as the same object for searching and storing in hash tables to avoid repeating searches.

We address and solve the above issues to be able to run effective search and obtain experimental results for the resulting algorithms. Before we present these results, we first introduce discrete wavelet transforms and rules for their fast implementation.

CHAPTER 6

DISCRETE WAVELET TRANSFORMS AND RULES

In Chapter 3 we reviewed elements of the theory of wavelets, explained the foundations of the discrete wavelet transforms (DWTs), and discussed its importance in digital signal processing. We also provided a brief overview of the most important methods and algorithms for efficient implementation of the DWTs through different decomposition techniques.

We now concentrate on defining the DWTs as SPIRAL transforms and capturing all reviewed algorithms using the rule formalism. We build on the framework for FIR filters developed in the previous two chapters and design rules for the DWTs that link the space of DWT algorithms with the space of algorithms for FIR filters. Thus, the breakdown rules presented in this chapter span the space of DWT algorithms by combining and recursively applying the DWT rules and the rules we introduced in the previous chapter.

We first define different versions of the finite DWT for different signal models, cast all DWT definitions as matrices, and use the mathematical language we developed up to this point to connect the DWTs with the framework for FIR filters. We next formulate popular DWT algorithms as breakdown rules that span the entire search space of competitive alternatives. By integrating these rules in SPIRAL, we enable automatic generation of the whole space of available DWT algorithms and their automatic implementation using SPIRAL's platform adaptive code generator. We will see that only a small set of rules is sufficient to describe well the space of different algorithms in conjunction with the rules for FIR filters and other convolution transforms. By the end of this chapter, we complete the development of the mathematical framework required to enable SPIRAL to generate and tune code for filtering and wavelet kernels. With the help of the SPIRAL system, we believe our approach will automatically produce code of quality comparable with the hand-tuned numerical libraries.

6.1 Discrete Wavelet Transforms

As discussed in Chapter 3, the DWT arises from the wavelet expansion theory and the multiresolution analysis. Separate form the continuous-time viewpoint, the DWT can be seen simply as an expansion of a discrete-time signal into discrete bases that span multiresolution vector spaces. The discrete wavelets are formed from the scaling and wavelet base change relations going from a coarser to a finer level of resolution defined in equations (3.63) and (3.66). The base change equations provide the means to compute the expansion coefficients at the coarser scale given the coefficients at a finer scale using relation (3.71).

In the case of a finite sequence of length n, the recursion can be performed at most $\lceil \log_2 n \rceil$ times, since the resulting sequence is downsampled by 2 at each recursion level. The DWT can be defined either as this full recursion or, as we will do, for an arbitrarily chosen number of levels $j \in \{1, \ldots \lceil \log_2 n \rceil\}$. Furthemore, depending on how we choose to treat finite signals, we shall define different types of DWTs: nonperiodic DWT that uses linear convolutions, periodic DWT using circular convolutions, and generalized DWT that uses arbitrary signal extensions.

Chapter 3 discussed that the DWT can be seen as the filter bank in Figure 3.3, where the impulse response coefficients of the lowpass and the highpass filters are the scaling and the wavelet function coefficients $\{h_k\}$ and $\{g_k\}$. Since filtering is a linear operation, so is the DWT, which can therefore be represented as a banded matrix if the data sequence is finite. As we discussed in Section 5.1, there are several different approaches to band the infinite filter matrix (5.1). Depending on the signal model at the boundaries, we consider alternative DWT and inverse DWT (IDWT) definitions:

- **The finite DWT and IDWT** for infinite signals is the most general definition of the finite DWT. Here, no signal model is assumed. Transforms are rectangular matrices in this case. The input signal, transform coefficients, and the reconstructed signal are all of different lengths.
- **The extended DWT and IDWT** are defined for signals extended at the boundaries. The type of signal extension is defined by the signal model. Transforms are rectangular but the input and the reconstructed signals have exactly the same length.
- **The nonperiodic DWT and IDWT** are a special case of the extended case where the extension is obtained by zero padding, and the wavelet system is orthogonal. The signal is of finite support.
- **The periodic DWT and IDWT** are defined as square matrices. There is no redundancy in the transform domain since the transform coefficients are of the same length as the input. Signals are periodically extended.

We start with the most intuitive definition for the special case of orthogonal wavelets that uses the convolution and the filter transforms.

6.1.1 Nonperiodic DWT

We assume that the input sequence has compact support of length n and the filters are FIR with length l. Then, each of the filter blocks in Figure 3.3 is a linear convolution on a finite input that can be represented by the convolution transform on n points.

Consider the orthogonal wavelet system with the lowpass filter $h(z)$ and the highpass filter $g(z)$ defined by

$$\begin{aligned} h(z) &= h_{l_1} z^{-l_1} + \cdots + h_0 + \cdots + h_{-r_1} z^{r_1} \\ g(z) &= g_{l_2} z^{-l_2} + \cdots + g_0 + \cdots + g_{-r_2} z^{r_2} \end{aligned}, \tag{6.1}$$

with the coefficients given by the scaling equation (3.63) and the wavelet equation (3.66), recursively. The orthogonality condition (3.81) given on page 62 says that all filters (analysis and synthesis, lowpass and highpass) are of the same degree.

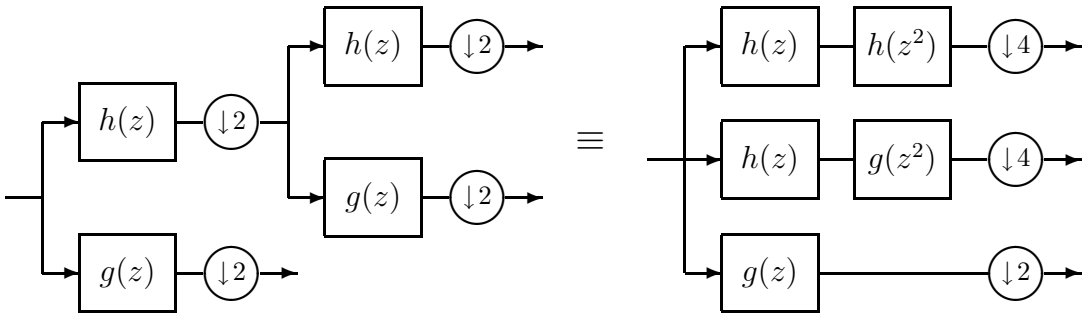


Figure 6.1: Computing channel filters using the Noble identities.

Since the input is of finite support, the filters in the first stage of the analysis filter bank are represented by convolution transforms $\mathbf{Conv}_n(h(z))$ and $\mathbf{Conv}_n(g(z))$.

Single-stage nonperiodic DWT. The *single stage nonperiodic* DWT is defined as

$$\mathbf{DWT}_{n,1}(h(z), g(z)) \rightarrow \begin{bmatrix} (\downarrow 2)_{n+l_1+r_1}^{t_1} \mathbf{Conv}_n(h(z)) \\ (\downarrow 2)_{n+l_2+r_2}^{t_2} \mathbf{Conv}_n(g(z)) \end{bmatrix}, \tag{6.2}$$

where $(\downarrow 2)_n$ is the downsampling matrix defined in (4.62) and the offsets t_i are defined as

$$t_i = r_i \bmod 2. \tag{6.3}$$

We intentionally cast equation (6.2) as a rule since it represents the *base case rule* for the nonperiodic DWT.

To obtain the exact form of a multistage DWT, consider the two stages of the filter bank shown on the left side of Figure 6.1. If we apply the Noble identity from Figure 3.7 to the next stage of the decomposition, we obtain the fused filters for each of the channels shown on the right side of Figure 6.1. The procedure can be repeated for an arbitrary number of stages j to obtain full filters for each channel. Each filter can again be represented by the convolution transform and then composed together using the results of Lemma 5.5. In this way, we obtain the definition of the *j-stage nonperiodic* DWT.

$$\mathbf{DWT}_{n,j}(h(z), g(z)) = \begin{bmatrix} \left(\downarrow 2^j\right)_{m_j}^{t_j} \mathbf{Conv}_n\left(\prod_{k=0}^{j-1} h(z^{2^k})\right) \\ \left(\downarrow 2^j\right)_{m_{j-1}}^{t_{j-1}} \mathbf{Conv}_n\left(g(z^{2^{j-1}}) \prod_{k=0}^{j-2} h(z^{2^k})\right) \\ \vdots \\ (\downarrow 8)_{m_2}^{t_2} \mathbf{Conv}_n\big(g(z^4)\, h(z^2)\, h(z)\big) \\ (\downarrow 4)_{m_1}^{t_1} \mathbf{Conv}_n\big(g(z^2)\, h(z)\big) \\ (\downarrow 2)_{m_0}^{t_0} \cdot \mathbf{Conv}_n(g(z)) \end{bmatrix}. \tag{6.4}$$

Each composition of the form $\mathbf{Conv}_n\left(g(z^{2^{i-1}} \prod_{k=0}^{i-1} h(z^{2^k})\right)$ for the i-th channel is implemented using (5.38), which also determines the output size m_i. The offsets t_i are determined the same way as for the single-stage case.

To clarify this definition, we recall that the convolution transform is defined as an infinite Toeplitz matrix operating on inputs with finite support. Another way to see this is to assume that the signal is zero-padded to a length sufficient for the computation. This explains the definition of the nonperiodic DWT given in (6.4): it is the j-stage DWT of a signal of support n, or the j-stage DWT of a signal on n points extended outside by zero padding.

The transform (6.4) is a rectangular matrix with the output size m larger than the input size n. The size m is a function of n, j, and l, and is determined by the composition rule (5.38) and the definition of the downsampling matrix (4.62). There is an obvious redundancy associated with the DWT computed in this manner [104]. It can be avoided either by periodization or at an additional cost per stage of the expansion [105].

EXAMPLE 6.1. *Let* $h(z) = h_1 z^{-1} + h_0 z^0 + h_{-1} z^1 + h_{-2} z^2$, $g(z) = g_2 z^{-2} + g_1 z^{-1} + g_0 z^0 + g_{-1} z^1$, *and let the number of input points be* $n = 6$. *Then from* (6.2) *we have*

$$\mathbf{DWT}_{6,1}(h(z), g(z)) = \begin{bmatrix} h_{-2} & . & . & . & . & . \\ h_0 & h_{-1} & h_{-2} & . & . & . \\ . & h_1 & h_0 & h_{-1} & h_{-2} & . \\ . & . & . & h_1 & h_0 & h_{-1} \\ . & . & . & . & . & h_1 \\ g_0 & g_{-1} & . & . & . & . \\ g_2 & g_1 & g_0 & g_{-1} & . & . \\ . & . & g_2 & g_1 & g_0 & g_{-1} \\ . & . & . & . & g_2 & g_1 \end{bmatrix}.$$

6.1.2 Inverse Nonperiodic DWT

Using the same logic as when deriving the forward DWT from the analysis filter bank, we obtain the inverse DWT from the synthesis part. Again, we use Noble identities to collapse all filters that belong to the same channel into a single filter and a single upsampler. A 2-stage synthesis filter bank example is shown in Figure 6.2. After upsampling by two, the transform coefficents are filtered by the time-reversed lowpass and highpass synthesis filters $\tilde{h}(z)$ and $\tilde{g}(z)$ that satisfy the perfect reconstruction property (3.82). After applying this upsample-filtering procedure in two stages, the original signal $x(z)$ is fully reconstructed. On the right side of Figure 6.2, the synthesis bank is collapsed into three channels with one upsampler and a cascade of filters. The two representations are equivalent, and we use the collapsed channels to define the inverse DWT as a single matrix.

Again, we are faced with the problem of banding the infinite matrix that represents the filter bank structure on the left of Figure 6.2. Consider first a single stage of the reconstruction filter bank. Assuming that the input signal $\{x_k\}$ is of finite support n, the reconstructed signal has to be $\{x_k\}$ and, hence, also has the support length n. We have discussed in Section 5.1 that the infinite matrix is reduced to the filter transform matrix (5.2) if we limit the number of output points. The filter bank stage then consists of upsampling, filtering with the lowpass synthesis filter $\tilde{h}(z)$ and the

highpass synthesis filter $\tilde{g}(z)$ defined by

$$\begin{aligned}\tilde{h}(z) &= \tilde{h}_{\tilde{l}_1} z^{-\tilde{l}_1} + \cdots + \tilde{h}_0 + \cdots + \tilde{h}_{-\tilde{r}_1} z^{\tilde{r}_1} \\ \tilde{g}(z) &= \tilde{g}_{\tilde{l}_2} z^{-\tilde{l}_2} + \cdots + \tilde{g}_0 + \cdots + \tilde{g}_{-\tilde{r}_2} z^{\tilde{r}_2},\end{aligned} \tag{6.5}$$

and summing the result to form the output. However, since the wavelet system has to satisfy the perfect reconstruction condition, the analysis and the synthesis filters are related. Using the relationship established in (3.79) on page 62, we can represent the synthesis filters by

$$\begin{aligned}\tilde{h}(z) &= g_{-r_2} z^{-r_2-1} - \cdots + g_{-1} z^{-1} - g_0 + g_1 z - \cdots + g_{l_2} z^{l_2-1} \\ \tilde{g}(z) &= h_{-r_1} z^{-r_1-1} + \cdots - h_{-1} z^{-1} + h_0 - h_1 z + \cdots - h_{l_1} z^{l_1-1}.\end{aligned} \tag{6.6}$$

Of course, from the orthogonality condition (3.80), for orthogonal wavelets

$$\tilde{h}(z) = h(z), \qquad \tilde{g}(z) = g(z). \tag{6.7}$$

Single-stage nonperiodic IDWT. From the above discussion, the *single-stage nonperiodic* DWT is defined as

$$\mathbf{IDWT}_{n,1}\Big(\tilde{h}(z), \tilde{g}(z)\Big) \to \Big[\, \mathbf{Filt}_n\Big(\tilde{h}(z^{-1})\Big)\,(\uparrow 2)^{\tilde{t}_1}_{n+\tilde{l}_1+\tilde{r}_1} \quad \mathbf{Filt}_n\big(\tilde{g}(z^{-1})\big)\,(\uparrow 2)^{\tilde{t}_2}_{n+l_2+\tilde{r}_2} \,\Big], \tag{6.8}$$

where $(\uparrow 2)_n$ is the upsampling matrix defined in (4.63), and the offsets are defined as

$$\tilde{t}_i = \tilde{r}_i \bmod 2. \tag{6.9}$$

From the definition of the upsampling matrix (4.63), we can determine the required input size for the matrix in (6.8).

$$\tilde{m}_i = \left\lceil \frac{\tilde{l}_i + n + \tilde{r}_i}{2} - \tilde{t}_i \right\rceil, \qquad m = \tilde{m}_1 + \tilde{m}_2 \tag{6.10}$$

However, we know that, for orthogonal wavelets, the analysis and synthesis filters are equal (see (6.7)), and so the input size of (6.8) exactly matches the output size of the forward DWT (6.2).

EXAMPLE 6.2. *We continue Example 6.1 and design the inverse DWT. Since the system is orthogonal, the synthesis filters* $\tilde{h}(z) = \tilde{h}_1 z^{-1} + \tilde{h}_0 z^0 + \tilde{h}_{-1} z^1 + \tilde{h}_{-2} z^2$, *and* $\tilde{g}(z) = \tilde{g}_2 z^{-2} + \tilde{g}_1 z^{-1} + \tilde{g}_0 z^0 + \tilde{g}_{-1} z^1$ *are equal to the analysis filters* $h(z)$ *and* $g(z)$. *From* (6.8) *we obtain*

$$\mathbf{IDWT}_{6,1}(h(z), g(z)) = \begin{bmatrix} h_{-2} & h_0 & . & . & . & g_0 & g_2 & . & . \\ . & h_{-1} & h_1 & . & . & g_{-1} & g_1 & . & . \\ . & h_{-2} & h_0 & . & . & . & g_0 & g_2 & . \\ . & . & h_{-1} & h_1 & . & . & g_{-1} & g_1 & . \\ . & . & h_{-2} & h_0 & . & . & . & g_0 & g_2 \\ . & . & . & h_{-1} & h_1 & . & . & g_{-1} & g_1 \end{bmatrix}$$

The nonperiodic DWT and IDWT matrices are rectangular of dimensions $m \times n$ and $n \times m$, respectively, where $m = \tilde{m}_1 + \tilde{m}_2$ as defined in (6.10). The perfect reconstruction condition is satisfied as

$$\mathbf{IDWT}_{n,1}(h(z), g(z)) \cdot \mathbf{DWT}_{n,1}(h(z), g(z)) = \mathrm{I}_n \tag{6.11}$$

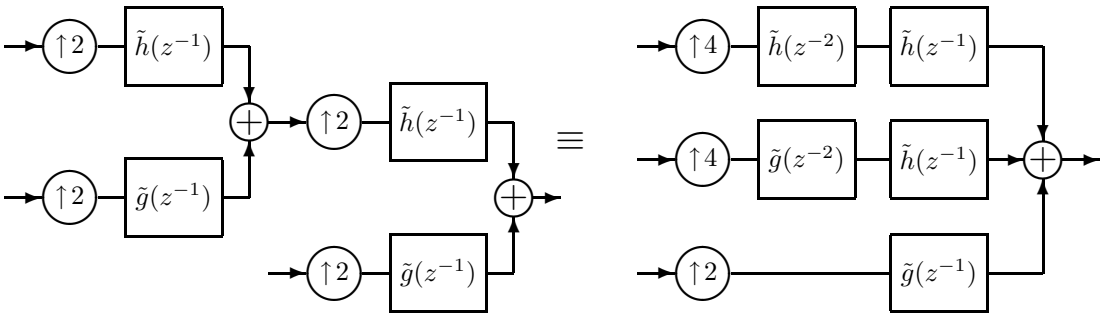


Figure 6.2: Computing synthesis channel filters using Noble identities.

Similar to the procedure we developed for obtaining the forward multistage DWT from the filter bank tree, we again apply the Noble identities to multiple stages of the synthesis filter bank to collapse the filters in different channels as shown in Figure 6.2. From the diagram on the right side of the image and the matrix (6.8), we obtain the *j-stage inverse nonperiodic* DWT.

$$\begin{aligned}\mathbf{IDWT}_{n,j}\Big(\tilde{h}(z),\tilde{g}(z)\Big) = \Big[&\mathbf{Filt}_n\Big(\textstyle\prod_{k=0}^{j-1}\tilde{h}(z^{-2^k})\Big)\,(\uparrow 2^j)_{m_j}\ \mid\\ &\mathbf{Filt}_n\Big(\tilde{g}(z^{-2^{j-1}})\textstyle\prod_{k=0}^{j-2}\tilde{h}(z^{-2^k})\Big)\,(\uparrow 2^j)_{m_{j-1}}\ \mid\\ &\vdots\\ &\mathbf{Filt}_n\Big(\tilde{g}(z^{-4})\,\tilde{h}(z^{-2})\,\tilde{h}(z^{-1})\Big)\,(\uparrow 8)_{m_2}\ \mid\\ &\mathbf{Filt}_n\Big(\tilde{g}(z^{-2})\,\tilde{h}(z^{-1})\Big)\,(\uparrow 4)_{m_1}\ \mid\\ &\mathbf{Filt}_n\big(\tilde{g}(z^{-1})\big)\,(\uparrow 2)_{m_0}\Big]\end{aligned} \tag{6.12}$$

The filter transforms in this definition are horizontally stacked as opposed to the vertical stacking of convolution transforms in (6.4). The composition of polynomials in each filter transform is obtained by applying rule (5.39). Also, the filter input dimensions m_i match exactly the convolution output dimensions in (6.4). The inverse nonperiodic DWT matrix is the left inverse of the forward nonperiodic DWT matrix for a j-stage decomposition. It is given by

$$\mathbf{IDWT}_{n,j}\Big(\tilde{h}(z),\tilde{g}(z)\Big)\cdot\mathbf{DWT}_{n,j}(h(z),g(z)) = \mathrm{I}_n\,. \tag{6.13}$$

6.1.3 Finite DWT and IDWT of infinite and extended signals

The previous section defined the forward and the inverse DWT under the assumption that the input signal has finite support and that the wavelet system is orthogonal. We now relax these conditions and let the input signal to remain infinite and the wavelets to be only biorthogonal. It is still possible to define a finite DWT matrix by limiting the number of output points and assuming that the wavelets have compact support, i.e., that the filters are FIR.

Consider the single-stage DWT of the infinite signals. Let the lowpass $h(z)$ and highpass $g(z)$ FIR filters be defined as in (6.1). A single-stage DWT is the matrix representation of the single

stage of the filter bank tree shown in Figure 3.3 on page 57. The signal is filtered by the lowpass and highpass filters and then downsampled by two. We showed the representing matrix W in (3.73) on page 55. Consider also the synthesis filter bank in Figure 3.4 that satisfies the perfect reconstruction condition. Let the synthesis FIR filters $\tilde{h}(z)$ and $\tilde{g}(z)$ be defined as in (6.5). Since the wavelet system satisfies the perfect reconstruction (PR) condition (3.82), we can write

$$\begin{bmatrix} \vdots \\ x_0 \\ x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{n-1} \\ \vdots \end{bmatrix} = \begin{bmatrix} \ddots & \vdots & \vdots & & \\ & \tilde{h}_0 & \tilde{g}_0 & & \\ & \tilde{h}_1 & \tilde{g}_1 & \vdots & \vdots \\ & \tilde{h}_2 & \tilde{g}_2 & \tilde{h}_0 & \tilde{g}_0 \\ & \vdots & \vdots & \tilde{h}_1 & \tilde{g}_1 \\ & \tilde{h}_{l-1} & \tilde{g}_{l-1} & \vdots & \vdots \\ & \vdots & \vdots & & \\ & & & \tilde{h}_{l-1} & \tilde{g}_{l-1} \\ & & & \vdots & \vdots \quad \ddots \end{bmatrix} \begin{bmatrix} \ddots & \vdots & \vdots & & \vdots & & \\ & h_0 & h_1 & \cdots & h_{l-1} & & \\ & g_0 & g_1 & \cdots & g_{l-1} & & \\ & & & h_0 & \cdots & h_{l-2} & h_{l-1} \\ & & & g_0 & \cdots & g_{l-2} & g_{l-1} \\ & & & & \vdots & \vdots & \vdots \quad \ddots \end{bmatrix} \begin{bmatrix} \vdots \\ x_0 \\ x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{n-1} \\ \vdots \end{bmatrix}. \quad (6.14)$$

Furthermore, the filters are related as (3.82), or more precisely as (6.6).

To band the infinite matrix, we limit ourselves to only n output points. Since the filters are FIR, we require only a finite number of input points to perform this operation. Starting from the reconstructed sequence, we can backtrace through the two matrices in (6.14) to determine how many input points of $\{x_k\}$ are required to expand and reconstruct the same sequence on the interval $[0, n-1]$.

Note that we used interleaved lowpass and highpass filters in the infinite case for convenience to be able to write the equations in matrix form. As before, in the finite case, we group together the rows pertaining to the same filter.

Since we are interested in n reconstructed points, the reconstruction matrix will be exactly the same as with the nonperiodic inverse DWT (6.8). The number m of transform coefficients needed to implement the inverse is given in (6.10). That is also the output size of the forward DWT; however, since we dropped the assumption that the signal is compactly supported, the number of required input points for this case is no longer n and we need to determine it.

The number of required output points is m. Before downsampling, the number of points is the output length n plus the number of extended points of the synthesis filters to the left and right $\tilde{l}_i$ and $\tilde{r}_i$, respectively. Because the synthesis filters are time-reversed, the right extension of the synthesis lowpass $\tilde{r}_1$ is added to the left extension of the analysis lowpass l_1, whereas the left synthesis extension $\tilde{l}_1$ is added to the right analysis extension r_1, which becomes the total extension length. The same is true for the highpass filters. The perfect reconstruction condition and the relations (3.82) guarantee that

$$\begin{aligned} l_{ext} &= \tilde{r}_1 + l_1 = \tilde{r}_2 + l_2 \\ r_{ext} &= \tilde{l}_1 + r_1 = \tilde{l}_2 + r_2. \end{aligned} \quad (6.15)$$

This is the number of additional samples of the input signal that need to be determined to the left and right of the $[0, n-1]$ interval. The exact number of required input points is larger than n and

is given by the following formula

$$n_{ext} = l_{ext} + n + r_{ext}. \tag{6.16}$$

We now define the *single-stage* DWT of an infinite-duration signal as

$$\mathbf{DWT}^*_{n,1}(h(z), g(z)) \rightarrow \left((\downarrow 2)^{\tilde{t}_1}_{n+\tilde{l}_1+\tilde{r}_1} \oplus (\downarrow 2)^{\tilde{t}_2}_{n+\tilde{l}_2+\tilde{r}_2}\right) \mathbf{Filt}_{N,L,R}\left(\begin{bmatrix} h(z) \\ g(z) \end{bmatrix}\right), \tag{6.17}$$

where

$$N = \begin{bmatrix} n+\tilde{l}_1+\tilde{r}_1 \\ n+\tilde{l}_2+\tilde{r}_2 \end{bmatrix} \quad L = \begin{bmatrix} \tilde{r}_1+l_1 \\ \tilde{r}_2+l_2 \end{bmatrix} \quad R = \begin{bmatrix} \tilde{l}_1+r_1 \\ \tilde{l}_2+r_2 \end{bmatrix}$$

and $\tilde{t}_i$ are defined in (6.9).

The definition of the matrix of filters $\mathbf{Filt}_{N,L,R}([h_{i,j}(z)])$ with variable sizes and extensions that appears in this equation was given in (5.32). However, because of the identities (6.15), it follows that

$$\tilde{l}_1 + \tilde{r}_1 + l_1 + r_1 = \tilde{l}_2 + \tilde{r}_2 + l_2 + r_2. \tag{6.18}$$

This important relationship between filter lengths implies that the width of the filter transforms in (6.17) is the same for both filters and enables the following most general definition of the DWT.

Single-stage DWT. The *single stage* DWT is defined by

$$\mathbf{DWT}^*_{n,1}(h(z), g(z)) \rightarrow \left((\downarrow 2)^{\tilde{t}_1}_{n+\tilde{l}_1+\tilde{r}_1} \oplus (\downarrow 2)^{\tilde{t}_2}_{n+\tilde{l}_2+\tilde{r}_2}\right) \begin{bmatrix} \mathbf{Filt}_{n+\tilde{l}_1+\tilde{r}_1}(h(z)) \\ \mathbf{Filt}_{n+\tilde{l}_2+\tilde{r}_2}(g(z)) \end{bmatrix}. \tag{6.19}$$

This definition represents the *base case rule* for the finite DWT.

Single-stage IDWT. The *single-stage inverse* DWT on n points is the same as in the nonperiodic case, which we repeat here for completeness.

$$\mathbf{IDWT}^*_{n,1}\left(\tilde{h}(z), \tilde{g}(z)\right) \rightarrow \left[\, \mathbf{Filt}_n\left(\tilde{h}(z^{-1})\right) (\uparrow 2)^{\tilde{t}_1}_{n+\tilde{l}_1+\tilde{r}_1} \quad \mathbf{Filt}_n\left(\tilde{g}(z^{-1})\right) (\uparrow 2)^{\tilde{t}_2}_{n+l_2+\tilde{r}_2} \,\right]. \tag{6.20}$$

We emphasize that both the DWT and the IDWT defined this way are rectangular. The size of $\mathbf{DWT}^*_{n,1}(h(z), g(z))$ is $m \times n_{ext}$ whereas the size of $\mathbf{IDWT}^*_{n,1}\left(\tilde{h}(z), \tilde{g}(z)\right)$ is $n \times m$.

In Example 6.2 we provided the single-stage nonperiodic IDWT matrix, which is the same as the general single-stage IDWT matrix. We now revisit the same example and find the single-stage finite DWT defined in (6.19).

EXAMPLE 6.3. *Let the analysis filters be* $h(z) = h_1 z^{-1} + h_0 z^0 + h_{-1} z^1 + h_{-2} z^2$, $g(z) = g_2 z^{-2} + g_1 z^{-1} + g_0 z^0 + g_{-1} z^1$, *and let the number of desired output points be* $n = 6$. *From the analysis and the synthesis filters, we have*

$$\begin{array}{lll} \tilde{l}_1 = l_1 = 1, & \tilde{r}_1 = r_1 = 2, & \tilde{t}_1 = 0 \\ \tilde{l}_2 = l_2 = 2, & \tilde{r}_2 = r_2 = 1, & \tilde{t}_2 = 1 \end{array}. \tag{6.21}$$

From equation (6.19), *we get*

$$\mathbf{DWT}^*_{6,1}(h(z), g(z)) \rightarrow \left((\downarrow 2)_9 \oplus (\downarrow 2)^1_9\right) \begin{bmatrix} \mathbf{Filt}_9(h(z)) \\ \mathbf{Filt}_9(g(z)) \end{bmatrix}$$

$$\mathbf{DWT}^*_{6,1}(h(z),g(z)) = \begin{bmatrix} h_1 & h_0 & h_{-1} & h_{-2} & . & . & . & . & . & . & . & . \\ . & . & h_1 & h_0 & h_{-1} & h_{-2} & . & . & . & . & . & . \\ . & . & . & . & h_1 & h_0 & h_{-1} & h_{-2} & . & . & . & . \\ . & . & . & . & . & . & h_1 & h_0 & h_{-1} & h_{-2} & . & . \\ . & . & . & . & . & . & . & . & h_1 & h_0 & h_{-1} & h_{-2} \\ . & g_2 & g_1 & g_0 & g_{-1} & . & . & . & . & . & . & . \\ . & . & . & g_2 & g_1 & g_0 & g_{-1} & . & . & . & . & . \\ . & . & . & . & . & g_2 & g_1 & g_0 & g_{-1} & . & . & . \\ . & . & . & . & . & . & . & g_2 & g_1 & g_0 & g_{-1} & . \end{bmatrix}.$$

This is the most general definition of the finite DWT since it can be applied to an input signal of arbitrary length. In Section 6.1.1 we introduced the nonperiodic DWT under the assumption that the input signal has compact support, or equivalently, that the signal is zero-padded outside of the main interval $[0, n-1]$. However, using definition (6.19), we observe that

$$\begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix} = \mathbf{IDWT}_{n,1}\Big(\tilde{h}(z),\tilde{g}(z)\Big)\,\mathbf{DWT}^*_{n,1}(h(z),g(z)) \begin{bmatrix} x_{-l_{ext}} \\ \vdots \\ x_0 \\ \vdots \\ x_{n-1} \\ \vdots \\ x_{r_{ext}} \end{bmatrix} \tag{6.22}$$

The input signal requires additional l_{ext} and r_{ext} points defined in (6.15). It is very important to understand that the additional points do not affect the output since the perfect reconstruction condition holds for any infinite signal. One interpretation is that we can take the input signal on the interval $[0, n-1]$, extend it to the required number of points in any way we desire, and still be able to reconstruct the output on the interval using (6.22).

Single-stage extended DWT. We define the *single-stage extended* DWT as

$$\mathbf{DWT}^{f_l,f_r}_{n,1}(h(z),g(z)) = \mathbf{DWT}^*_{n,1}(h(z),g(z))\,\mathrm{E}^{f_l,f_r}_{n,l_{ext},r_{ext}} \tag{6.23}$$

where $\mathrm{E}^{f_l,f_r}_{n,l_{ext},r_{ext}}$ is the extension matrix defined in (4.68) on page 83, the extension lengths l_{ext} and r_{ext} are defined in (6.15), and f_l and f_r specify the type of the extension to the left and right of the input signal, respectively. The $\mathbf{DWT}^{f_l,f_r}_{n,1}(h(z),g(z))$ is an $m \times n$ matrix, where m is defined in (6.10).

From the above discussion, we conclude that the extended DWT satisfies the perfect reconstruction condition

$$\begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix} = \mathbf{IDWT}_{n,1}\Big(\tilde{h}(z),\tilde{g}(z)\Big)\,\mathbf{DWT}^{f_l,f_r}_{n,1}(h(z),g(z)) \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix} \tag{6.24}$$

for any give f_l and f_r. If, for example, we use the zero-padding extension defined in (4.71) and if the wavelet system is orthogonal (3.80) then (6.23) equals the single-stage nonperiodic DWT defined in (6.2).

$$\mathbf{DWT}_{n,1}^{\text{zero}}(h(z), g(z)) = \mathbf{DWT}_{n,1}(h(z), g(z)) \tag{6.25}$$

for orthogonal wavelets.

EXAMPLE 6.4. *We continue with Example 6.3 and apply the zero extension to* $\mathbf{DWT}_{6,1}^{*}(h(z), g(z))$. *From* (6.15) *we obtain* $l_{ext} = 3$ *and* $r_{ext} = 3$. *Then,*

$$\mathbf{DWT}_{6,1}^{*}(h(z), g(z))\, \mathrm{E}_{6,3,3}^{zero} = \begin{bmatrix} h_{-2} & . & . & . & . & . \\ h_0 & h_{-1} & h_{-2} & . & . & . \\ . & h_1 & h_0 & h_{-1} & h_{-2} & . \\ . & . & . & h_1 & h_0 & h_{-1} \\ . & . & . & . & . & h_1 \\ g_0 & g_{-1} & . & . & . & . \\ g_2 & g_1 & g_0 & g_{-1} & . & . \\ . & . & g_2 & g_1 & g_0 & g_{-1} \\ . & . & . & . & g_2 & g_1 \end{bmatrix} = \mathbf{DWT}_{6,1}(h(z), g(z))$$

and we obtain the single-stage nonperiodic DWT *for this example since the wavelet system is orthogonal.*

Perfect reconstruction is possible regardless of the type of extension by using the definition of the DWT we provided here. At the first glance, this might strike the reader as a little odd; however, the perfect reconstruction condition guarantees that all signals of the form

$$x = \begin{bmatrix} x_{-l} & \cdots & x_{-1} & 0 & \cdots & 0 & x_n & \cdots & x_{n+r} \end{bmatrix}$$

are in the null space of the finite DWT and, hence, have no effect on the reconstructed result.

In constructing a multi-stage extended DWT, one needs to be careful with matching the dimensions of the subsequent stages because of the complicated relationships between the lengths of the filters and required expansion coefficients. We defer this discussion until we introduce recursive rules for the computation of the DWT, which greatly simplifies the construction of the multi-stage DWT.

6.1.4 Periodic DWT and IDWT

We have introduced two definitions of the finite DWT and IDWT: the extended and the nonperiodic as important special cases. Common to these definitions is the fact that the length of the input sequence is not preserved after each stage of the decomposition. The output of a single stage DWT is longer than the input, i.e., the DWT matrix is rectangular and creates extra transform coefficients. The additional coefficients are redundant in the sense that exactly n coefficients are sufficient to reconstruct the input of length n [105]. In addition to the cost of allocating additional memory for these extended coefficients, another drawback of rectangular DWTs is that the error in the transform domain tends to be amplified at the reconstruction due to the existence of the null space of the DWT.

It is advantageous to construct a square DWT and IDWT that still satisfy the perfect reconstruction condition and, possibly, exhibit other desirable properties such as orthogonality, biorthogonality, maximally flat bases, etc. This is essentially the problem of designing wavelet bases on an interval that satisfy such properties, where special treatment is given to wavelets at the interval boundaries [91, 106].

Consider the extended DWT defined in (6.23) and the corresponding perfect reconstruction equation given in (6.24). Further assume that the input size n is divisible by 2. The forward DWT is a rectangular matrix of size $m \times n$ where the output length m is given by (6.10). The input size is n and the input is extended to n_{ext} points in (6.16) using any extension method.

Now, let the extension of the input signal be periodic, i.e., $\mathrm{E}^{f_l,f_r}_{n,l_{ext},r_{ext}} = \mathrm{E}^{\mathrm{per,per}}_{n,l_{ext},r_{ext}}$. It is easy to show that, if the input signal is periodic, the ouput of the filter-downsample block $(\downarrow 2)_n \mathbf{Filt}_n(h(z))$ will also be periodic with period $n/2$ and, hence, both the lowpass and highpass outputs will be periodic signals. To simplify the problem, we consider only the lowpass part of the DWT output (the top half of (6.23)). If the lowpass output signal is $\mathbf{y}$ periodic with period $n/2$, then it can be represented by considering only the main period and extending it periodically:

$$\begin{bmatrix} y_{\tilde{l}_{ext}} \\ \vdots \\ y_0 \\ \vdots \\ y_{\frac{n}{2}-1} \\ \vdots \\ y_{\tilde{r}_{ext}} \end{bmatrix} = \mathrm{E}^{\mathrm{per}}_{n,\tilde{l}_{ext},\tilde{r}_{ext}} \begin{bmatrix} y_0 \\ \vdots \\ y_{\frac{n}{2}-1} \end{bmatrix}$$

where $\tilde{l}_{ext} = \left\lfloor \frac{\tilde{l}_1}{2} \right\rfloor$ and $\tilde{r}_{ext} = \left\lceil \frac{\tilde{r}_1 - n \bmod 2}{2} \right\rceil$. This implies that only the output on the interval $[0, n/2-1]$ needs to be computed. The values outside this interval can be found by periodic replication of this interval. To compute these required output values, according to (6.19), the forward extended DWT can be reduced to

$$\mathbf{DWT}^{\mathrm{per}}_{n,1}(h(z), g(z)) = \begin{bmatrix} (\downarrow 2)_n \mathbf{Filt}_n(h(z)) \\ (\downarrow 2)_n \mathbf{Filt}_n(g(z)) \end{bmatrix} \mathrm{E}^{\mathrm{per}}_{n,l_m,r_m}, \tag{6.26}$$

where $l_m = \max_i\{l_i\}$ and $r_m = \max_i\{r_i\}$.

Single-stage periodic DWT. The resulting matrix is now square and provides the definition of the *single-stage periodic* DWT.

$$\mathbf{DWT}^{\mathrm{per}}_{n,1}(h(z), g(z)) \to \begin{bmatrix} (\downarrow 2)_n \mathbf{C}_n(h(z)) \\ (\downarrow 2)_n \mathbf{C}_n(g(z)) \end{bmatrix}. \tag{6.27}$$

We go further and fuse the extension matrix into the lowpass IDWT part $\mathbf{Filt}_n\left(\tilde{h}(z^{-1})\right)(\uparrow 2)^{t_1}_{n+\tilde{l}_1+\tilde{r}_1}$ (see (6.8)) and apply the identity $(\uparrow 2)_{n+l+r}\,\mathrm{E}^{\mathrm{per}}_{n,l,r} = \mathrm{E}^{\mathrm{per}}_{2n,2l,2r}\,(\uparrow 2)_n$ to obtain

$$\begin{aligned} \mathbf{Filt}_n\left(\tilde{h}(z^{-1})\right)(\uparrow 2)^{\tilde{t}_1}_{n+\tilde{l}_1+\tilde{r}_1}\,\mathrm{E}^{\mathrm{per}}_{n/2,\tilde{l}_{ext},\tilde{r}_{ext}} &= \mathbf{Filt}_n\left(\tilde{h}(z^{-1})\right)\mathrm{E}^{\mathrm{per}}_{n,\tilde{l}_1,\tilde{r}_1}\,(\uparrow 2)_n \\ &= \mathbf{C}_n\left(\tilde{h}(z^{-1})\right)(\uparrow 2)_n \end{aligned}. \tag{6.28}$$

Single-stage periodic IDWT. The same procedure can be applied to the highpass portion of the filter bank, which leads to the definition of the *inverse periodic* DWT.

$$\mathbf{IDWT}_n^{\mathrm{per}}\Big(\tilde{h}(z),\tilde{g}(z)\Big) \to \Big[\,\mathbf{C}_n\Big(\tilde{h}(z^{-1})\Big)\,(\uparrow 2)_n \quad \mathbf{C}_n\big(\tilde{g}(z^{-1})\big)\,(\uparrow 2)_n\,\Big]. \tag{6.29}$$

Note that $\mathbf{IDWT}_n^{\mathrm{per}}\Big(\tilde{h}(z),\tilde{g}(z)\Big)$ is a square matrix.

Using the above procedure, we have derived the periodic DWT and IDWT from the extended DWT and IDWT using the periodic extension and the property that periodic signals stay periodic even after filtering and downsampling. From the perfect reconstruction (PR) equation (6.24) it follows immediately that

$$\mathbf{x} = \mathbf{IDWT}_{n,1}^{\mathrm{per}}\Big(\tilde{h}(z),\tilde{g}(z)\Big)\,\mathbf{DWT}_{n,1}^{\mathrm{per}}(h(z),g(z))\,\mathbf{x}. \tag{6.30}$$

This form of the DWT is particularly clean since both matrices are square so there is no redundancy in the representation.

Another advantage of the periodic DWT is that the orthogonality of the wavelet bases is preserved by using periodically extended boundary wavelets [106]. This means that the IDWT is the transpose of the DWT

$$\mathbf{IDWT}_{n,1}^{\mathrm{per}}\Big(\tilde{h}(z),\tilde{g}(z)\Big) = \mathbf{DWT}_{n,1}^{\mathrm{per}}(h(z),g(z))^T \tag{6.31}$$

However, it is well known that the periodic extension of signals introduces discontinuities which in turn heavily affect wavelet coefficients at higher (finer detail) scales [107].

Let the length of the input signal be $n = 2^s$. The *j-stage periodic* DWT and IDWT are defined in a manner similar to the multi-stage nonperiodic DWT and IDWT in (6.4) and (6.12). As an example, we provide the matrix form of the forward j-stage periodic DWT

$$\mathbf{DWT}_{n,j}(h(z),g(z)) = \begin{bmatrix} (\downarrow 2^j)_n\,\mathbf{C}_n\Big(\prod_{k=0}^{j-1} h(z^{2^k})\Big) \\ (\downarrow 2^j)_n\,\mathbf{C}_n\Big(g(z^{2^{j-1}})\prod_{k=0}^{j-2} h(z^{2^k})\Big) \\ \vdots \\ (\downarrow 8)_n\,\mathbf{C}_n\big(g(z^4)\;h(z^2)\;h(z)\big) \\ (\downarrow 4)_n\,\mathbf{C}_n\big(g(z^2)\;h(z)\big) \\ (\downarrow 2)_n\cdot\mathbf{C}_n(g(z)) \end{bmatrix}. \tag{6.32}$$

6.2 Breakdown Rules for DWT Algorithms

We visited several major approaches used for efficient implementation of the DWT in Section 3.2. After introducing several related definitions of the DWT and the IDWT in the first part of this chapter, we try to capture the most important techniques for implementing the wavelet transforms as a set of rules that will enable automatic generation of numerous possible algorithms and also enable effective translation into various versions of executable code. We start with the basic recursion

suggested by the Mallat equations (3.71) and the filter bank tree interpretation of the DWT, and proceed with the other methods discussed in Section 3.2.

6.2.1 Mallat recursive rules

By construction, the multiresolution analysis and the recursive Mallat equations (3.71) provide an efficient way to compute the DWT in a recursive way. The filter bank interpretation shown in Figure 3.3 provides the most useful intuition about this recursive implementation. Each filter bank stage of the filter tree represents a single stage of the DWT expansion. The lowpass output is further filtered and downsampled at the next expansion stage. The procedure can be captured using a breakdown rule that we will call the *Mallat rule*. Depending on which definition we choose from the previous sections, we have closely related versions of the Mallat rule.

Consider again the filter bank tree in Figure 3.3 representing the DWT with the lowpass filter $h(z)$ and the highpass filter $g(z)$ defined by (6.1). The single-stage nonperiodic DWT is given by (6.2). The nonperiodic Mallat rule defines the recursion on the lowpass output and how to compute the nonperiodic DWT coefficients at a lower level given the coefficients at the next higher level.

NONPERIODIC DWT MALLAT RULE

$$\mathbf{DWT}_{n,j}(h(z),g(z)) \rightarrow (\mathbf{DWT}_{m_1,j-1}(h(z),g(z)) \oplus \mathrm{I}_{m_2}) \cdot \mathbf{DWT}_{n,1}(h(z),g(z)), \tag{6.33}$$

where the lowpass and the highpass output lengths m_1 and m_2 are given by

$$m_i = \left\lfloor \frac{l_i}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{r_i - n \bmod 2}{2} \right\rceil, \qquad i = 1,2. \tag{6.34}$$

The single stage nonperiodic $\mathbf{DWT}_{n,1}(h(z),g(z))$ was defined in (6.2).

The inverse nonperiodic DWT is computed similarly using the synthesis bank filters defined in (6.5). Again, we assume that $l_i = \tilde{l}_i$ and $r_i = \tilde{r}_i$ for both filters. The Mallat rule is then

NONPERIODIC IDWT MALLAT RULE

$$\mathbf{IDWT}_{n,j}\Big(\tilde{h}(z),\tilde{g}(z)\Big) \rightarrow \mathbf{IDWT}_{n,1}\Big(\tilde{h}(z),\tilde{g}(z)\Big) \cdot \Big(\mathbf{IDWT}_{m_1,j-1}\Big(\tilde{h}(z),\tilde{g}(z)\Big) \oplus \mathrm{I}_{m_2}\Big) \tag{6.35}$$

where m_1 and m_2 are again given by (6.34).

In Section 6.1.3, we deferred the definition of the multi-stage finite and extended DWTs because the recursive equations provide a cleaner and more understandable form. We use the definitions of the single-stage extended DWT and IDWT in (6.23) and (6.20) to define the multi-stage versions through the Mallat rule.

EXTENDED DWT MALLAT RULE

$$\mathbf{DWT}^{f_l,f_r}_{n,j}(h(z),g(z)) \rightarrow \Big(\mathbf{DWT}^{f_l,f_r}_{\tilde{m}_1,j-1}(h(z),g(z)) \oplus \mathrm{I}_{\tilde{m}_2}\Big) \cdot \mathbf{DWT}^{f_l,f_r}_{n,1}(h(z),g(z)), \tag{6.36}$$

where

$$\tilde{m}_i = \left\lceil \frac{\tilde{r}_i}{2} \right\rceil + \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{\tilde{l}_i - n \bmod 2}{2} \right\rceil.$$

The single-stage extended $\mathbf{DWT}^{f_l,f_r}_{n,1}(h(z),g(z))$ is defined using equations (6.23) and (6.19).

Since the extended single-stage IDWT is exactly equal to the single-stage nonperiodic IDWT, then the extended IDWT Mallat rule is the same as (6.35). When applied recursively j times, the rule (6.36) indirectly defines the matrix form of the j-stage extended DWT that we skipped in Section 6.1.3. The recursion terminating matrix is the same as the single-stage extended DWT given in (6.23).

Finally, the Mallat rule for the periodic DWT and IDWT has a particularly clean form that follows immediately from the single-stage periodic $\mathbf{DWT}^{\mathrm{per}}_{n,1}(h(z), g(z))$ and $\mathbf{IDWT}^{\mathrm{per}}_{n,1}\left(\tilde{h}(z), \tilde{g}(z)\right)$ defined in (6.27) and (6.29), respectively.

Periodic DWT Mallat Rule

$$\mathbf{DWT}^{\mathrm{per}}_{n,j}(h(z), g(z)) \rightarrow \left(\mathbf{DWT}^{\mathrm{per}}_{n,j-1}(h(z), g(z)) \oplus \mathrm{I}_n\right) \cdot \mathbf{DWT}^{\mathrm{per}}_{n,1}(h(z), g(z)) \tag{6.37}$$

Periodic IDWT Mallat Rule

$$\mathbf{IDWT}^{\mathrm{per}}_{n,j}\left(\tilde{h}(z), \tilde{g}(z)\right) \rightarrow \mathbf{IDWT}^{\mathrm{per}}_{n,1}\left(\tilde{h}(z), \tilde{g}(z)\right) \left(\mathbf{IDWT}^{\mathrm{per}}_{n,j-1}\left(\tilde{h}(z), \tilde{g}(z)\right) \oplus \mathrm{I}_n\right) \tag{6.38}$$

For the periodic case, all matrices are square of dimensions $n \times n$ and the additional size parameters and the offsets need not be computed. Furthermore, if the wavelet bases are orthogonal, the periodic IDWT Mallat rule is simply the transpose of (6.37).

Practical considerations. It should be clear that the downsampling after filtering is not very efficient since half of the already computed samples are then wasted. A speedup of two is achieved by fusing the donwsampling operator with the filtering stage. The fusion is done in SPIRAL at the formula optimization level where the downsampler is represented using the gather operators introduced on page 82. We provide one example of this fusion for illustrative purposes.

Consider again the Mallat rule (6.37) for the periodic DWT in (6.32). Without the loss of generality, we assume that the supports of $h(z)$ and $g(z)$ are the same, i.e., that $l_1 = l_2 = l$ and $r_1 = r_2 = r$. Using definitions (5.34) and (5.24), we can represent the filtering stage as

$$\mathbf{C}_n\left(\begin{bmatrix} h(z) \\ g(z) \end{bmatrix}\right) = \begin{bmatrix} \mathrm{I}_n \otimes_{l+r} (h_l, \ldots, h_{-r}) \\ \mathrm{I}_n \otimes_{l+r} (g_l, \ldots, g_{-r}) \end{bmatrix} \cdot \mathrm{E}^{\mathrm{per}}_{n,l,r}\,.$$

If we fuse $(\downarrow 2)_n$ in each half of this matrix we get

$$\left((\downarrow 2)_n \oplus (\downarrow 2)_n\right) \mathbf{C}_n\left(\begin{bmatrix} h(z) \\ g(z) \end{bmatrix}\right) = \begin{bmatrix} \mathrm{I}_n \otimes_{l+r-1} (h_l, \ldots, h_{-r}) \\ \mathrm{I}_n \otimes_{l+r-1} (g_l, \ldots, g_{-r}) \end{bmatrix} \cdot \mathrm{E}^{\mathrm{per}}_{n,l,r-1}\,. \tag{6.39}$$

Example 6.5. *Let $h(z) = h_1 z^{-1} + h_0 z^0 + h_{-1} z^1 + h_{-2} z^2$, $g(z) = g_1 z^{-1} + g_0 z^0 + g_{-1} z^1 + g_{-2} z^2$, and the number of input points be $n = 6$. Then, we have*

$$(\downarrow 2)_6\, \mathbf{C}_6\left(\begin{bmatrix} h(z) \\ g(z) \end{bmatrix}\right) = \begin{bmatrix} \mathrm{I}_3 \otimes_2 (h_1, \ldots, h_{-2}) \\ \mathrm{I}_3 \otimes_2 (g_1, \ldots, g_{-2}) \end{bmatrix} \cdot \mathrm{E}^{per}_{6,1,1} = \begin{bmatrix} h_0 & h_{-1} & h_{-2} & . & . & h_1 \\ . & h_1 & h_0 & h_{-1} & h_{-2} & . \\ h_{-2} & . & . & h_1 & h_0 & h_{-1} \\ g_0 & g_{-1} & g_{-2} & . & . & g_1 \\ . & g_1 & g_0 & g_{-1} & g_{-2} & . \\ g_{-2} & . & . & g_1 & g_0 & g_{-1} \end{bmatrix}.$$

The upsampling operator can be similarly fused into the synthesis filter bank.

Fusing the up/downsampling in the filtering matrices avoids redundant operations and saves half of the cost. A disadvantage of this method is that it destroys the Toeplitz structure of the filtering stage and the incorporated filter transforms. Consequently, none of the rules for filter transforms described in Chapter 5 is applicable and the implementation of the DWT has to fall back to the straightforward time-domain algorithms and blocking strategies.

On the other hand, the polyphase representation discussed in Section 3.2.3 avoids redundant operations, preserves the filter transform structure, and allows application of numerous FIR filter rules. We capture the polyphase method in a set of rules we define next.

6.2.2 Polyphase rules

According to the Noble identities in Figure 3.7 on page 61, one stage of the filter bank can be represented using the polyphase notation shown in Figure 3.8.

Consider again the polynomials $h(z)$ and $g(z)$ downsampled to even and odd factors $h_e(z)$, $h_o(z)$, $g_e(z)$, and $g_o(z)$ according to (3.38). The single-stage nonperiodic DWT can be decomposed into a matrix of convolution transforms using the following rule

Polyphase nonperiodic DWT rule

$$\mathbf{DWT}_{n,1}(h(z), g(z)) \to \mathbf{Conv}_N \left(\begin{bmatrix} h_e(z) & h_o(z) \\ g_e(z) & g_o(z) \end{bmatrix} \right) \bar{\mathrm{L}}_2^n, \tag{6.40}$$

where

$$N = \left[\left\lceil \frac{n}{2} \right\rceil \quad \left\lfloor \frac{n}{2} \right\rfloor \right],$$

and where $\bar{\mathrm{L}}_2^n$ is either the linear permutation (4.52) or the stride permutation (4.46) if $2 | n$. In the latter case, the rule reduces to

$$\mathbf{DWT}_{n,1}(h(z), g(z)) \to \mathbf{Conv}_{\frac{n}{2}} \left(\begin{bmatrix} h_e(z) & h_o(z) \\ g_e(z) & g_o(z) \end{bmatrix} \right) \mathrm{L}_2^n .$$

The generalized matrix of convolution transforms in (6.40) was defined in (5.37) on page 100.

Example 6.6. *We apply the polyphase rule to the nonperiodic* DWT *in Example 6.4. We first determine the downsampled filters for $h(z)$ and $g(z)$ given in Example 6.3 using equations* (3.38) *on page 43.*

$$\begin{aligned} h_e(z) &= h_0 + h_{-2} z, & h_o(z) &= h_1 z^{-1} + h_{-1} \\ g_e(z) &= g_2 z^{-1} + g_0, & g_o(z) &= g_1 z^{-1} + g_{-1} \end{aligned}.$$

From, (6.40) *we obtain*

$$\mathbf{DWT}_{6,1}(h(z),g(z)) = \begin{bmatrix} h_{-2} & . & . & . & . & . \\ h_0 & h_{-2} & . & h_{-1} & . & . \\ . & h_0 & h_{-2} & h_1 & h_{-1} & . \\ . & . & h_0 & . & h_1 & h_{-1} \\ . & . & . & . & . & h_1 \\ g_0 & . & . & g_{-1} & . & . \\ g_2 & g_0 & . & g_1 & g_{-1} & . \\ . & g_2 & g_0 & . & g_1 & g_{-1} \\ . & . & g_2 & . & . & g_1 \end{bmatrix} \mathrm{L}_2^6 = \mathbf{Conv}_3\left(\begin{bmatrix} h_e(z) & h_o(z) \\ g_e(z) & g_o(z) \end{bmatrix}\right) \mathrm{L}_2^6 .$$

A similar rule can be obtained for the IDWT by considering the synthesis filter bank shown in Figure 3.4 and the synthesis polyphase matrix in (3.78).

Consider the synthesis filters $\tilde{h}(z)$ and $\tilde{g}(z)$ and define their downsampled versions: $\tilde{h}_e(z)$, $\tilde{h}_o(z)$, $\tilde{g}_e(z)$, $\tilde{g}_o(z)$, using equation (3.38), as before. The polyphase rule for the nonperiodic IDWT is then

Polyphase nonperiodic IDWT rule

$$\mathbf{IDWT}_{n,1}(h(z),g(z)) \to \bar{\mathrm{L}}^n_{\lceil \frac{n}{2} \rceil}\, \mathbf{Filt}_N\left(\begin{bmatrix} \tilde{h}_e(z) & \tilde{g}_e(z) \\ \tilde{h}_o(z) & \tilde{g}_o(z) \end{bmatrix}\right), \tag{6.41}$$

where N and $\bar{\mathrm{L}}^n_{\frac{n}{2}}$ are defined the same way as in (6.40). In the case $2|n$, we have

$$\mathbf{IDWT}_{n,1}(h(z),g(z)) \to \mathrm{L}^n_{\frac{n}{2}}\, \mathbf{Filt}_{\frac{n}{2}}\left(\begin{bmatrix} \tilde{h}_e(z) & \tilde{g}_e(z) \\ \tilde{h}_o(z) & \tilde{g}_o(z) \end{bmatrix}\right).$$

Example 6.7. *Consider again Example 6.2. We compute the downsampled synthesis filters as*

$$\begin{aligned} \tilde{h}_e(z) &= \tilde{h}_0 + \tilde{h}_{-2}z, & \tilde{h}_o(z) &= \tilde{h}_1 z^{-1} + \tilde{h}_{-1} \\ \tilde{g}_e(z) &= \tilde{g}_2 z^{-1} + \tilde{g}_0, & \tilde{g}_o(z) &= \tilde{g}_1 z^{-1} + \tilde{g}_{-1} \end{aligned}.$$

Then, we have

$$\mathbf{IDWT}_{6,1}\left(\tilde{h}(z),\tilde{g}(z)\right) = \mathrm{L}_3^6 \begin{bmatrix} \tilde{h}_{-2} & \tilde{h}_0 & . & . & . & \tilde{g}_0 & \tilde{g}_2 & . & . \\ . & \tilde{h}_{-2} & \tilde{h}_0 & . & . & . & \tilde{g}_0 & \tilde{g}_2 & . \\ . & . & \tilde{h}_{-2} & \tilde{h}_0 & . & . & . & \tilde{g}_0 & \tilde{g}_2 \\ . & \tilde{h}_{-1} & \tilde{h}_1 & . & . & \tilde{g}_{-1} & \tilde{g}_1 & . & . \\ . & . & \tilde{h}_{-1} & \tilde{h}_1 & . & . & \tilde{g}_{-1} & \tilde{g}_1 & . \\ . & . & . & \tilde{h}_{-1} & \tilde{h}_1 & . & . & \tilde{g}_{-1} & \tilde{g}_1 \end{bmatrix}.$$

The polyphase rules for the extended DWT follow a similar pattern. However, special consideration must be given to the input/output sizes of the downsampled filters.

Polyphase extended DWT rule

$$\mathbf{DWT}^{f_l,f_r}_{n,1}(h(z),g(z)) \to \mathbf{Filt}_{N,L,R}\left(\begin{bmatrix} h_e(z) & h_o(z) \\ g_e(z) & g_o(z) \end{bmatrix}\right) \bar{\mathrm{L}}^{n_{ext}}_{2,t}\, \mathrm{E}^{f_l,f_r}_{n,l_{ext},r_{ext}}, \tag{6.42}$$

where n_{ext}, l_{ext}, and r_{ext} are given by (6.16) and (6.15), the offset $t = \max_i\{\tilde{r}_i + l_i\} \bmod 2$, the matrices

$$N = \begin{bmatrix} m_1 & m_1 \\ m_2 & m_2 \end{bmatrix}, \quad L = \begin{bmatrix} \left\lfloor \frac{\tilde{r}_1+l_1}{2} \right\rfloor & \left\lceil \frac{\tilde{r}_1+l_1}{2} \right\rceil \\ \left\lfloor \frac{\tilde{r}_2+l_2}{2} \right\rfloor & \left\lceil \frac{\tilde{r}_2+l_2}{2} \right\rceil \end{bmatrix}, \quad R = \begin{bmatrix} \left\lfloor \frac{\tilde{l}_1+r_1}{2} \right\rfloor & \left\lfloor \frac{\tilde{l}_1+r_1-1}{2} \right\rfloor \\ \left\lfloor \frac{\tilde{l}_2+r_2}{2} \right\rfloor & \left\lfloor \frac{\tilde{l}_2+r_2-1}{2} \right\rfloor \end{bmatrix},$$

and the sizes m_i are defined in (6.10).

By invoking the properties (4.73) on page 84, in the case of zero-padding, we can write

$$\bar{\mathrm{L}}_{2,t}^{n_{ext}} \, \mathrm{E}^{\mathrm{zero}}_{n,l_{ext},r_{ext}} = \left(\mathrm{E}^{\mathrm{zero}}_{\lceil \frac{n}{2} \rceil, \lceil \frac{l_{ext}}{2} \rceil, \lceil \frac{r_{ext}-n \bmod 2}{2} \rceil} \oplus \mathrm{E}^{\mathrm{zero}}_{\lfloor \frac{n}{2} \rfloor, \lfloor \frac{l_{ext}}{2} \rfloor, \lfloor \frac{r_{ext}+n \bmod 2}{2} \rfloor} \right) \bar{\mathrm{L}}_2^n \,. \tag{6.43}$$

It can be proven that, from here, the polyphase extended DWT rule (6.42) becomes the nonperiodic polyphase rule (6.40).

A similar manipulation can be performed for the periodic extension using the identities (4.76). This leads to the periodic polyphase rules we define next.

Consider the analysis filter bank with filters $h(z)$ and $g(z)$, and let the input size n satisfy $2 \,|\, n$. For the single-stage periodic DWT (6.27) the following rule applies.

Polyphase periodic DWT rule

$$\mathbf{DWT}^{\mathrm{per}}_{n,1}(h(z), g(z)) \to \mathbf{C}_{\frac{n}{2}} \left(\begin{bmatrix} h_e(z) & h_o(z) \\ g_e(z) & g_o(z) \end{bmatrix} \right) \mathrm{L}_2^n \,. \tag{6.44}$$

Similarly, we can define the polyphase rule for the periodic IDWT using the definition (6.29).

Polyphase periodic IDWT rule

$$\mathbf{IDWT}^{\mathrm{per}}_{n,1}\left(\tilde{h}(z), \tilde{g}(z)\right) \to \mathrm{L}_{\frac{n}{2}}^n \, \mathbf{C}_{\frac{n}{2}} \left(\begin{bmatrix} \tilde{h}_e(z) & \tilde{g}_e(z) \\ \tilde{h}_o(z) & \tilde{g}_o(z) \end{bmatrix} \right). \tag{6.45}$$

As the final note in this subsection, we mention that the polyphase rules enable each stage of the DWT to be computed using filter rules and, at the same time, avoid implementing redundant operations. The greatest impact is achieved when specialized filter rules, such as transform-domain and Karatsuba rules, lead to efficient algorithms for FIR filters. The polyphase rules for the DWT are therefore seen as the gateway to efficient FIR filter algorithms.

However, the polyphase rules as well as the Mallat rules do not themselves reduce the arithmetic cost. Lattice and lifting factorizations are also available and they asymptotically achieve a cost reduction by 2. We capture the rules that enable both factorizations in the remainder of this chapter.

6.2.3 Lattice factorization rules

If the wavelet bases are orthogonal there is a very efficient way to decompose each stage of the expansion using the lattice factorization we introduced in Section 3.9. Since wavelets are orthogonal, the analysis filters $h(z)$ and $g(z)$ defined in (6.1) are of equal even length. The rotation coefficients $\alpha_j = \tan\theta_j$ in (3.87) are determined by solving the recursive equations (3.89) on page 63. These equations assumed that the filters are causal, but this is not necessary. At each step of the recursion,

the coefficients α_j are chosen so that the Laurent degree of $h^{(j)}(z) - \alpha_j g^{(l)}(z)$ is reduced by one, which means canceling the highest power of either z or z^{-1}. Because of the paraunitary property of the filters, this procedure actually cancels two degrees and the algorithm terminates after $\frac{l+r+1}{2}$ steps. If there is a need to cancel the highest degree of z then the second equation in (3.89) requires z^2 rather than z^{-2}

$$(1+\alpha_j^2)z^2 g^{(j-1)}(z) = \alpha_j h^{(j)}(z) + g^{(j)}(z),$$

and the delay operators in (3.86) will be

$$\Delta(z^{-1}) = \begin{bmatrix} 1 & 0 \\ 0 & z^1 \end{bmatrix}. \tag{6.46}$$

The rules for lattice decomposition follow directly the form presented in Figure 3.2.4. We start with the rule for nonperiodic DWT and IDWT.

LATTICE NONPERIODIC DWT RULE

$$\mathbf{DWT}_{n,1}(h(z),g(z)) \to \beta\,\mathbf{Conv}_N\left(\begin{bmatrix} 1 & \alpha_0 \\ -\alpha_0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & z \end{bmatrix}\begin{bmatrix} 1 & \alpha_1 \\ -\alpha_1 & 1 \end{bmatrix}\cdots\begin{bmatrix} 1 & 0 \\ 0 & z^{-1} \end{bmatrix}\begin{bmatrix} 1 & \alpha_J \\ -\alpha_J & 1 \end{bmatrix}\right)\bar{\mathrm{L}}_2^n, \tag{6.47}$$

where

$$N = \left[\left\lceil\frac{n}{2}\right\rceil \;\; \left\lfloor\frac{n}{2}\right\rfloor\right]$$

The composition of the convolution transforms is performed according to the results of Lemma 5.5 on page 101 and equation (5.38), trivially adapted for matrices of convolution transforms.

The lattice factorization rule for the nonperiodic inverse DWT is easily obtained by simply transposing and time-reversing (6.47) since the wavelets are orthogonal and, hence, the polyphase matrices are orthogonal (see page 62).

LATTICE NONPERIODIC IDWT RULE

$$\mathbf{IDWT}_{n,1}(h(z),g(z)) \to \beta\,\bar{\mathrm{L}}_{\lceil\frac{n}{2}\rceil}^n\,\mathbf{Filt}_N\left(\begin{bmatrix} 1 & -\alpha_J \\ \alpha_J & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & z^{-1} \end{bmatrix}\cdots\begin{bmatrix} 1 & -\alpha_1 \\ \alpha_1 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & z \end{bmatrix}\begin{bmatrix} 1 & -\alpha_0 \\ \alpha_0 & 1 \end{bmatrix}\right) \tag{6.48}$$

The form of the lattice factorization is especially clean in the case of the periodic DWT and IDWT. The convolution and filter matrices become circulant matrices, and all products are between square matrices. In this case, we require that $2|n$.

LATTICE PERIODIC DWT RULE

$$\mathbf{DWT}_{n,1}^{\mathrm{per}}(h(z),g(z)) \to \beta\,\mathrm{L}_2^{\frac{n}{2}}\,\mathbf{C}_n\left(\begin{bmatrix} 1 & \alpha_0 \\ -\alpha_0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & z \end{bmatrix}\begin{bmatrix} 1 & \alpha_1 \\ -\alpha_1 & 1 \end{bmatrix}\cdots\begin{bmatrix} 1 & 0 \\ 0 & z^{-1} \end{bmatrix}\begin{bmatrix} 1 & \alpha_J \\ -\alpha_J & 1 \end{bmatrix}.\right) \tag{6.49}$$

According to Theorem 5.9 on page 103 the composition of the circular transforms is the same as the circular transform of the compositions

$$\mathbf{C}_n(h(z)\cdot g(z)) = \mathbf{C}_n(h(z))\cdot\mathbf{C}_n(g(z))$$

and together with the definition of the matrix of circulants (5.34), the composition in (6.49) is performed by assigning to each element of the matrices in the lattice decomposition a circulant transform with the element as its parameter. For example,

$$\mathbf{C}_n\left(\begin{bmatrix} 1 & \alpha_0 \\ -\alpha_0 & 1 \end{bmatrix}\right) = \begin{bmatrix} \mathbf{C}_n(1) & \mathbf{C}_n(\alpha_0) \\ \mathbf{C}_n(-\alpha_0) & \mathbf{C}_n(1) \end{bmatrix} = \begin{bmatrix} \mathrm{I}_n & \alpha_0 \mathrm{I}_n \\ -\alpha_0 \mathrm{I}_n & \mathrm{I}_n \end{bmatrix} \tag{6.50}$$

The lattice factorization for the periodic inverse DWT is again obtained by transposing and time-reversing (6.49).

LATTICE PERIODIC IDWT RULE

$$\mathbf{IDWT}^{\mathrm{per}}_{n,1}(h(z), g(z)) \rightarrow \beta \mathrm{L}_2^{\frac{n}{2}} \mathbf{C}_n\left(\begin{bmatrix} 1 & -\alpha_J \\ \alpha_J & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & z^{-1} \end{bmatrix}\cdots\begin{bmatrix} 1 & -\alpha_1 \\ \alpha_1 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & z \end{bmatrix}\begin{bmatrix} 1 & -\alpha_0 \\ \alpha_0 & 1 \end{bmatrix}.\right) \tag{6.51}$$

As we mentioned in Section 3.9, the lattice factorization of orthogonal filter banks reduces the computational cost asymptotically to one half of the polyphase implementation cost. Another advantage of the decomposition is that the rounding error of the rotation coefficients does not affect the perfect reconstruction condition because the error is cancelled in the synthesis bank decomposition. The lattice factorization, however, cannot be applied to biorthogonal wavelets and filter banks. In that case, the lifting scheme factorization is used instead.

6.2.4 Lifting scheme rules

The lifting scheme for constructing and decomposing biorthogonal filter banks reduces the arithmetic cost of the DWT. We reviewed the basics of the lifting steps factorization in Section 3.2.5, where we explained the factoring algorithm based on the Euclidean algorithm for polynomials. Similarly to the lattice factorization, the lifting scheme starts with the polyphase matrix

$$P(z) = \begin{bmatrix} h_e(z) & h_o(z) \\ g_e(z) & g_o(z) \end{bmatrix}$$

where $h_e(z),\ h_o(z),\ g_e(z),\ g_o(z)$ are the downsampled analysis filters $h(z)$ and $g(z)$ defined in (6.1). The polyphase matrix is decomposed into alternating primal and dual lifting steps (3.91). The factorization is not unique due to the fact that the Laurent polynomial division is not unique [19]. We provide the analysis of the degrees of freedom and the number of different lifting schemes in Appendix A. At this point we mention that the number L of lifting steps varies with different lifting schemes for the same $P(z)$, but is bounded by $L < \max\{\deg(h), \deg(g)\}$.

The factorization scheme was illustrated in Figure 3.10. The structure is very similar to the lattice factorization. The rules for lifting factorization for different DWT definitions can be defined straightforwardly using the properties of the bounded filter matrices as in the case of lattice factorization.

LIFTING NONPERIODIC DWT RULE

$$\mathbf{DWT}_{n,1}(h(z), g(z)) \rightarrow \mathbf{Conv}_N\left(\begin{bmatrix} \alpha & 0 \\ 0 & \frac{1}{\alpha} \end{bmatrix}\begin{bmatrix} 1 & s_{\frac{L}{2}}(z) \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ t_{\frac{L}{2}}(z) & 1 \end{bmatrix}\cdots\begin{bmatrix} 1 & s_0(z) \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ t_0(z) & 1 \end{bmatrix}\right)\bar{\mathrm{L}}_2^n, \tag{6.52}$$

where

$$N = \left[\left\lceil \frac{n}{2} \right\rceil \;\; \left\lfloor \frac{n}{2} \right\rfloor \right],$$

The initial lifting step and the order of subsequent steps is determined by the filters $h(z)$ and $g(z)$. For example, if $\deg(h_e) > \deg(h_o)$ the first step is dual, otherwise it is primal. Also, if $2 \nmid L$, the last lifting step in (6.52) does not exist. As we mentioned with the lattice factorization, the composition of the convolution transforms is performed according to the results of Lemma 5.5.

The inverse lifting scheme can be obtained by applying the factoring algorithm described in Section 3.2.5 to the synthesis polyphase matrix $\tilde{P}(z)$ defined in (3.78). However, by recognizing that the perfect reconstruction condition (3.82) holds after the factorization into lifting steps, the inversion can be performed by inverting each lifting step matrix individually. This is a trivial procedure since the inverse matrix of either of the lifting steps (3.91) is obtained by simply changing the sign of the lifting filters. Hence, we obtain the lifting scheme for the nonperiodic IDWT.

LIFTING NONPERIODIC IDWT RULE

$$\mathbf{IDWT}_{n,1}\Big(\tilde{h}(z), \tilde{g}(z)\Big) \rightarrow \bar{\mathrm{L}}^{n}_{\lceil \frac{n}{2} \rceil}\, \mathbf{Filt}_N \left(\begin{bmatrix} 1 & 0 \\ -t_0(z) & 1 \end{bmatrix} \begin{bmatrix} 1 & -s_0(z) \\ 0 & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & 0 \\ -t_{\frac{L}{2}}(z) & 1 \end{bmatrix} \begin{bmatrix} 1 & -s_{\frac{L}{2}}(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\alpha} & 0 \\ 0 & \alpha \end{bmatrix} \right) \tag{6.53}$$

The periodic DWT and IDWT lifting rules follow directly

LIFTING PERIODIC DWT RULE

$$\mathbf{DWT}^{\mathrm{per}}_{n,1}(h(z), g(z)) \rightarrow \mathbf{C}_n \left(\begin{bmatrix} \alpha & 0 \\ 0 & \frac{1}{\alpha} \end{bmatrix} \begin{bmatrix} 1 & s_{\frac{L}{2}}(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ t_{\frac{L}{2}}(z) & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & s_0(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ t_0(z) & 1 \end{bmatrix} \right) \mathrm{L}^{n}_{2} \tag{6.54}$$

LIFTING PERIODIC IDWT RULE

$$\mathbf{IDWT}^{\mathrm{per}}_{n,1}\Big(\tilde{h}(z), \tilde{g}(z)\Big) \rightarrow \mathrm{L}^{n}_{\frac{n}{2}}\, \mathbf{C}_n \left(\begin{bmatrix} 1 & 0 \\ -t_0(z) & 1 \end{bmatrix} \begin{bmatrix} 1 & -s_0(z) \\ 0 & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & 0 \\ -t_{\frac{L}{2}}(z) & 1 \end{bmatrix} \begin{bmatrix} 1 & -s_{\frac{L}{2}}(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\alpha} & 0 \\ 0 & \alpha \end{bmatrix} \right) \tag{6.55}$$

where we require that $2 \,|\, n$.

As we noted above, the lifting rule can take different forms depending on the chosen factorization scheme. We allow the generation of all possible lifting schemes by invoking all possible Laurent polynomial divisions. For example, for the Daubechies 9/7 biorthogonal wavelets used in the JPEG2000 standard, there are 27 different lifting schemes, of which only one preserves the symmetry and, hence, the linear phase of the filters. In this case, there are either four or five lifting steps, each with the filters of degree one or two. Different lifting schemes will obviously provide different run times. We investigate these effects and report the results in Chapter 7

6.3 Concluding Remarks

We started this chapter by introducing four different definitions of the discrete wavelet transform (DWT): the nonperiodic DWT, the finite DWT of infinite signals, the extended DWT, and the

periodic DWT. Each definition arises from different assumptions on the processed signal, similar to the models we considered for the FIR filters in Chapter 5. For the nonperiodic DWT, the input signal has a finite support. The second model assumes that the signal is of infinite length and that the DWT is computed on a finite number of points. The extended DWT assumes that the signal is infinitely extended outside of the main interval $[0, N-1]$. Depending on the type of extension, the effects on the boundary filters of the DWT filter bank will be different. The DWT matrix is rectangular for all of the above definitions, leading to a redundant wavelet representation. The last definition assumes infinite periodic extension of signals and leads to a square DWT transform. In this case, the redundancy is avoided in the transform domain; however, the somewhat regular structure of the redundant definitions is slightly violated by cyclically wrapping the boundary filters.

For each definition we designed a set of decomposition rules in Section 6.2 that captures most of the known algorithms for efficient computation of the DWT. We covered the algorithms introduced in Chapter 3. We break the presented algorithms into two classes:

- The direct implementation algorithms with $\mathcal{O}(n)$ cost based on Mallat rules introduced in Section 6.2.1
- The factoring algorithms that achieve asymptotic reduction of the number of operations by two based on the relation between filter coefficients arising from the perfect reconstruction condition. These include the lattice factorization methods with rules introduced in Section 6.2.3 and the lifting scheme factorization methods with rules presented in Section 6.2.4.

The Mallat rules implement the DWTs in a straightforward manner, preserving the regular filter bank structure. The advantage of these rules is the regularity of the computations that can be performed with excellent efficiency on most platforms. On the other hand, the factoring rules lead to the reduction of the arithmetic cost of algorithms. However, these algorithms have a longer critical path and, typically, higher memory requirements.

Search space. All the DWT rules we presented create a search space of algorithms by expanding a DWT at the top level of the rule tree into either smaller size DWTs or into one of the filtering transforms, such as convolution, filter, and circulant transforms. This is important to emphasize because the whole space of algorithms for FIR filters is embedded into the space of algorithms for DWTs. Mallat rules provide means to reduce the multi-stage DWTs into the single-stage DWTs, at which point the other DWT rules can be applied. After the DWT rules are applied, the children transforms are one of the filtering transforms. From that point, any of the applicable rules for filtering transforms can be applied. We investigate the efficiency of all DWT methods by varying the top-level rule to expand the search space in different ways.

Practical considerations. Since the search space for filtering transforms is embedded in the space of DWT algorithms, most of the discussion from the end of Chapter 5 apply to DWT implementations. However, we add a few more notes about the implementation of the DWT algorithms in SPIRAL.

- The fusion of the downsampling operator and the filter bank stage is done at the formula optimization level using the gather and scatter operators [39]. However, if the boundary

filters are different, as in the case of the periodic DWT, the fusion has to be taken care of by introducing special optimization rules that achieve looping of both the downsampled boundary filters and the main filters.

- Polyphase rules permute the data in order to restore the regular Toeplitz structure of the filter transforms. The readressing of the data comes at a price that has to be offset by the advantage afforded by the FIR filter algorithms. In most cases the manipulation is justified only if the filter algorithms significantly reduce the cost of computations. For that reason, the polyphase rule is usually seen as the gateway to transform-domain methods.

- The factoring rules, such as the lifting scheme, are applied in multiple stages. The number of stages depends on the number of lifting steps and they are computed consecutively. Ideally, the lifting scheme can be implemented in-place, where the temporary vector of results is kept the same after each stage of the factorization. However, the current SPIRAL framework does not support generation of in-place implementations. This might have a slowing effect on the factoring algorithms, especially for large transform sizes.

In the next chapter, we conduct selected experiments to investigate the efficiency of the proposed methods and decomposition rules. We present the results and discuss the implications on the automatic generation and tuning of FIR filter and DWT algorithms and implementations.

CHAPTER 7

EXPERIMENTAL RESULTS

In Chapter 2, we described the SPIRAL system that we use to automatically implement and tune code for FIR filters and discrete wavelet transforms (DWTs). We introduced key concepts of the mathematical framework and the rule formalism that allows SPIRAL to automatically generate and modify algorithms, discussed the methodology behind the translation of mathematical formulas into code, verification of the implementation, performance evaluation, and finally reviewed the available search techniques that are aimed at optimizing the implementations for a given transform.

In Chapter 3, we reviewed the basic filtering and wavelet processing operations and fast algorithms for their implementation found in the literature. We defined these operations as finite-dimensional matrices that perform the convolution operation on finite, extended, and infinite sequences, and the corresponding definitions for DWTs. In chapters 4 through 6, we developed a new framework for representing these transforms in SPIRAL and designed a large set of rules that spans a comprehensive space of algorithms that includes both the reviewed algorithms and the whole space of new algorithms designed for faster implementation on a computer. We described how we implemented the set of rules in SPIRAL's formula generator, how to test them for correctness on the formula and code levels, and how we modified the search techniques to accommodate for new transforms and new algorithms.

Extending SPIRAL's rule formalism to filtering and wavelet functions and their algorithms, enables us to use SPIRAL to automatically generate the entire space of algorithms and implementations that we believe covers all known techniques for fast implementation of FIR filters and DWTs. SPIRAL further enables us to implement all of these algorithms, evaluate them, and search for their optimized code.

In this chapter, we use the framework developed in the previous chapters to perform a set of experiments to demonstrate the efficiency and the portability of our automatically generated code. We benchmark our code to assess the performance and present results that demonstrate the diversity of optimized solutions, even on platforms with almost identical architecture.

7.1 Overview of Experiments and Platforms

We conduct experiments and group them into two categories:

- Performance benchmarks;

- Comparison of the best found methods, algorithms, or implementations on a single platform, or across different platforms and compilers.

More specifically, we distinguish between benchmark experiments, where the goal is to evaluate our tuned code for efficiency, and experiments comparing different approaches and options to determine the correlation between different methods and advantages or limitations of the target platform.

Our goal in this chapter is to perform comprehensive performance tests of our code generator on multiple computer platforms, as well as to explore the efficiency of various computational methods on a range of platforms to determine their practicality. The practical value of our system will be demonstrated by the quality of produced code and by often surprising implementation solutions found by our automatic optimizer, which are hardly ever considered by a human programmer.

The first and most important point we would like to bring across is that our automatically generated and optimized code performs on par with the best hand-coded libraries on the platforms that were used in our experiments. Even though that alone is enough to justify our approach of automatic code generation and automatic tuning, we emphasize that our approach produces surprising solutions that bring further insight into the efficiency of many chosen implementation methods.

For a signal processing expert, and even for many well-versed software programmers, the best found implementations of filtering and wavelet algorithms can seem very unintuitive. We list below some of the observations supporting this fact.

- Lower arithmetic cost of an algorithm does not necessarily translate into a faster run time, i.e., the operation count is not a reliable predictor of runtime performance;
- Different realizations of the same algorithm can lead to a considerable discrepancy in performance, even when the arithmetic cost is the same. There are several reasons that can explain this observation: different data flow patterns, intricate memory bandwidth problems, compiler and used compiler options, among other reasons;
- The space of possible competitive implementations is very large and cannot be searched exhaustively; however, it is highly beneficial to search over a comprehensive set of alternatives because the spread in performance indicates that a considerable speedup of up to several factors is achieved by search;
- The best found algorithms can be very different across different computer platforms, which implies that code tuned for one platform is not portable and that the optimization has to be repeated for every platform individually.

Hence, high performance can be achieved only through either automatic implementation and search or by a painstaking hand-coding effort at the hardware level performed by algorithm and hardware experts.

Benchmarks. To measure the quality of the generated code, we benchmark the run times of the best found implementations. We use two evaluation criteria in parallel.

- Benchmark implementations against the theoretical peak performance on the target machine;

- Benchmark implementations against Intel Performance Primitives (IPP), Intel's hand-coded libraries that include digital filtering and wavelet functions.

In the first approach, we measure the efficiency of implementations by computing the number of executed floating point operations per second, commonly expressed in millions of operations using the unit of MFLOPS:

$$\text{Operation rate (MFLOPS)} = \frac{\text{Total number of floating point operations } \times 10^{-6}\text{(Mflop)}}{\text{Run time (s)}} \tag{7.1}$$

This measure will be compared to the theoretical upper bound in MFLOPS that can be achieved on the target platform, or as we call it, the *theoretical peak performance*. The performance can also be expressed as a percentage of the peak performance. As an example, highly optimized BLAS routines can achieve up to 80% of the peak performance, whereas optimized FFT routines usually achieve at most 50%.

We also benchmark our automatically generated implementations against hand-tuned computer vendor implementations provided by Intel's IPP (see Section 1.1.3). We present the results as

$$\text{Relative run time} = \frac{\text{SPIRAL run time (s)}}{\text{IPP run time (s)}} \tag{7.2}$$

IPP libraries are optimized using specific properties of the supported architectures, e.g., SSE extensions on some Intel processors, and also provide a library of hand-coded implementations for unspecified, generic platforms.

Comparison of methods and implementation choices. We test several implementation strategies governed by the the top-level rules on multiple architectures, compare them among themselves, and thoroughly investigate their runtime performance across a range of transform sizes, parameters, platforms, compilers, and compiler options. Organization of the experiments and obtained results is based on the chosen

- target computer platform,
- computational method,
- general purpose compiler, and
- compiler options.

We demonstrate the dependence of the results on each of these different factors, even across experiments run on similar platforms and/or with similar compilers and compiler options. We show that the performance of any given method on any given platform is highly unpredictable. Different methods and implementation options are found even on platforms with very similar architecture, and more importantly, all of our breakdown rules play an important role in generating the best found implementations on one platform or the other.

Platforms and compilers. We perform experiments and obtain results across a set of selected computer platforms with different architectural features. We list the platforms and their specifications in Table 7.1. Our motivation for selecting this set of platforms is to diversify the hardware on

Table 7.1: Computer platforms used for experiments.

Name	Processor	Core	Speed	L_1/L_2 Cache	OS	Compiler
Xeon-1.7	Xeon	Galatin	1.7 GHz	8KB/256KB	Linux 2.4.16	GNU C 3.2.1
Athlon-1.73	Athlon XP 2100+	Palomino	1.733 GHz	64KB/512KB	Win XP	Intel C++ 8.1
P4-1.6-lin	Celeron	Mobile	1.6 GHz	8KB/256KB	Linux 2.6.5	GNU C Intel C++ 8.1
P4B-3.0-lin	Pentium 4	Northwood B	3.0 GHz	8KB/512KB	Linux 2.4.21	GNU C
P4B-3.0-win	Pentium 4	Northwood B	3.0 GHz	8KB/512KB	Win XP	Intel C++ 8.1
P4C-3.2-win	Pentium 4	Northwood C	3.2 GHz	8KB/512KB	Win XP	Intel C++ 8.1
Macintosh	Power PC	G4e 7455	933 MHz	32KB/256KB	Mac OS X	GNU C
P4E-3.6	Pentium 4	Prescott	3.6 GHz	16KB/1MB	Win XP	Intel C++ 8.1

which we run our experiments. This allows us to test the automatic tuning capability of our system across a spectrum of general-purpose computer platforms and investigate and gain more insight in efficiency of specific methods on specific platforms. With SPIRAL, all of this is achieved with little additional effort by installing the system on the target platform, and setting and running the desired experiments.

The chosen platforms range from Intel Pentium 4 family of processors with different clock speeds, cache sizes, and different design of processor cores, running either Windows XP or Linux operating systems, to AMD Athlon and Apple Macintosh platforms that have, for example, larger cache sizes, or, in the case of Athlon, two floating point units.

For faster reference, we give all of our eight platforms unique shorthand names as specified in Table 7.1. Names comprise of platform identifiers separated by dashes. The first identifier is usually the name of the processor, followed by its clock rate, and, optionally, the name of the operating system. The second column in the table specifies the name of the processor with the specific core name in column 3. We also provide clock rate for the processor as well as the size of level one L_1 and level two L_2 cache memories. We note that L_1 cache size in the table refers to data cache only — instruction cache size is not provided. Finally, we specify the operating system running the platform, and the type of C compiler used.

We mentioned in Chapter 2 that the SPL compiler provides translation of formulas representing algorithms either into C or Fortran code. For the purpose of this chapter, we only use C language as the target code. Choosing the compiler, as well as various compiler flags can have a considerable impact on performance. The especially difficult problem is choosing optimal compiler flags. First, there can be as many as 500 different compiler flags for a single compiler. Second, choosing the right set of options depends on the target computer platform, the compiled program, and even on other chosen options since specific optimization features can be inter-dependent. If the choice of the compiler flags is not clear, one option is to perform empirical search over possible combinations [108]. However, for our experiments, we use a set of compiler flags empirically proven to provide satisfactory performance on selected platforms. We provide a set of compiler flags for both Intel

C++ compiler and GNU C compiler for different platforms, all summarized in Table 7.2.

Table 7.2: Compilers and compiler options.

Name	Compiler	Flags
gcc-O1-3.3	GNU C 3.3.1	`-O1 -fomit-frame-pointer -malign-double -fstrict-aliasing -mcpu=pentiumpro`
gcc-O1-3.2	GNU C 3.2.1	`-O1 -malign-double -fstrict-aliasing -mcpu=pentium4`
gcc-O6-3.2	GNU C 3.2.1	`-O6 -fomit-frame-pointer -malign-double -fstrict-aliasing -mcpu=pentiumpro`
gcc-mac-O1	GNU C 3.3 (Apple)	`-O1 -fomit-frame-pointer -std=c99 -mcpu=7450`
gcc-mac-O3	GNU C 3.3 (Apple)	`-O3 -fomit-frame-pointer -std=c99 -fast -mcpu=7450`
icc-8.0-lin	Intel C++ 8.0	`-O -axK`
icc-8.1-lin	Intel C++ 8.1	`-O -tpp7 -march=pentium4`
Intel-win	Intel C++ 8.1	`/O2 /Qc99 /Qrestrict /G7`
Intel-SSE3	Intel C++ 8.1	`/O2 /Qc99 /Qrestrict /G7 /QxKWP`

We end this section by noting that, for all our experiments, we use dynamic programming (DP) search to find the optimized solution. For some experiments, search times can be considerable, and DP provides a suitable tradeoff between the quality of the generated code and the speed of search.

7.2 Runtime Performance Benchmarks

Our first task is to determine the quality of the code produced by our system. To do that, we benchmark the best found implementations generated by SPIRAL against the hand-tuned IPP routines provided by Intel. We perform benchmarks for both the best found FIR filter transform code as well as the best DWT code. We call IPP routines from within SPIRAL's evaluation module to obtain the run times for both systems using the same measurement method for a fair comparison.

For FIR filters, we time the IPP library routine `FIR` for randomly generated taps. It is important to emphasize that the IPP's `FIR` function implements an FIR filter that is equivalent to our extended filter transform for a zero-padded input signal.

$$\mathbf{Filt}_n^{\text{zero}}(h(z)) \tag{7.3}$$

For the purpose of timing IPP and our code, we will always use random causal filters, i.e.,

$$h(z) = h_k z^{-k} + \cdots + h_0$$

Similarly, for the DWTs, we time the IPP library routine `DWT` that implements the extended DWT with zero-padded input extension, i.e.,

$$\mathbf{DWT}_n^{\cdot}(h(z), g(z), \text{zero}) \tag{7.4}$$

where the lowpass $h(z)$ and the highpass $g(z)$ filters depend on the chosen wavelet system.

Performance benchmarks against IPP on non-Intel platforms. For non-Intel platforms, we use IPP libraries designed for high performance on generic Intel and other platforms. These

libraries do not make use of special instructions such as SSE and SSE2, which makes them suitable for comparison with our scalar code.

We first present results for Athlon-1.73. Using IPP libraries on an AMD Athlon platform is well justified for two reasons:

1. There are no available hand-tuned filtering and wavelet libraries for Athlon platforms;
2. Athlon processors are designed and optimized to efficiently run Intel Pentium III and Pentium 4 programs.

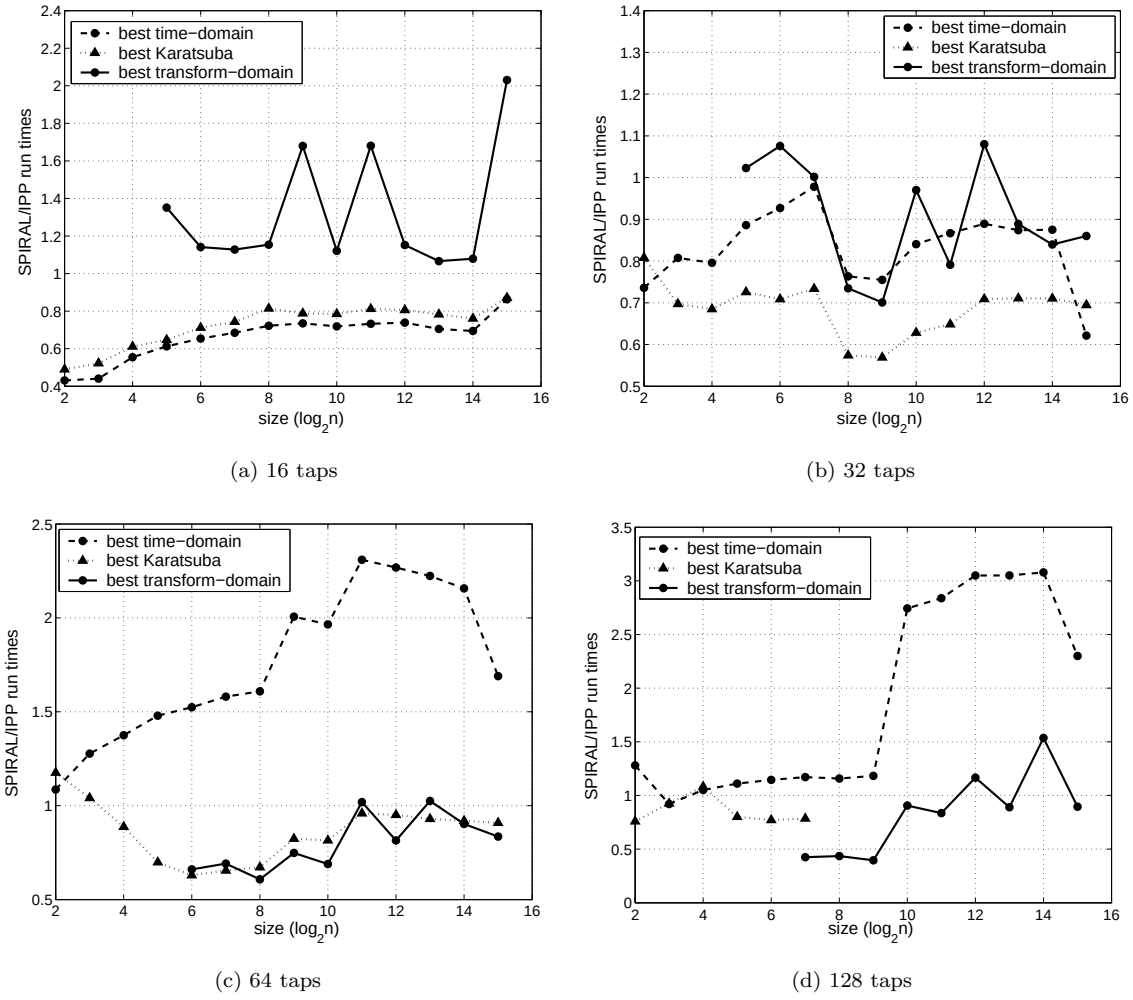
Figure 7.1 shows four plots with the relative run times vs. the IPP run times defined in (7.2) for FIR filters with 16, 32, 64, and 128 taps as a function of the size of the input data sequence. On each plot there are three lines showing the relative run times for three different methods against the run time of IPP's `FIR` routine. The lower the relative run time of a method the better. Relative run times below one show that our generated code is faster than the IPP code. For now, we are only interested in the best found implementation in SPIRAL, which is shown as the bottom line in each plot. The results are obtained through DP search using Intel-win compiler.

For a filter with 16 coefficients, our best code outperforms IPP by about 30% for larger filter sizes and up to 60% for smaller filters. In this case, the best blocking strategy outperforms Karatsuba and transform-based methods, which we will discuss in more detail later in this chapter. For a filter of length 32, all methods run faster than the IPP routine for most sizes. However, the Karatsuba method is a clear winner, reducing the run time of IPP by as much as 40% but mostly by 30% for all sizes. Graph 7.1(c) shows the same comparison for a filter of length 64. For this length, the transform-domain approach is competitive with the best Karatsuba method, and, combined, the best found approach outperforms the IPP implementation by as much as 40%. Finally, in graph 7.1(d) we show that our automatically generated code obtained for a 128-tap filter runs faster than the IPP implementation by about 50% for smaller sizes and stays competitive with the IPP for sizes larger then or equal to 2^{12}. In this case, SPIRAL finds the transform-based techniques as the best implementation.

This experiment shows two important advantages of our approach on Athlon:

1. Automatically generated implementations are competitive, and for most cases outperform IPP hand-coded libraries.
2. Search over the entire space of implementations finds a combination of different techniques for different filter lengths, such as flexible blocking strategies and Karatsuba methods, as the best solution. SPIRAL automates this search that would otherwise be almost impossible to perform by hand due to the extent of the implementation space.

Next, we benchmark our code for DWTs against IPP's `DWT` routine again on Athlon-1.73. In Figure 7.2, we compare the run times of the best found implementation by SPIRAL and the IPP code for three different wavelet systems: 1) rational 5/3 wavelets with lowpass and highpass filters of length 5 and 3, respectively; 2) Daubechies 9/7 wavelets with filter lengths of 9 and 7, respectively; and 3) Orthogonal Daubechies 30 wavelets with both filters of length 30.



(a) 16 taps

(b) 32 taps

(c) 64 taps

(d) 128 taps

Figure 7.1: Comparison of different filtering methods and IPP FIR function on Athlon-1.7 (lower is better).

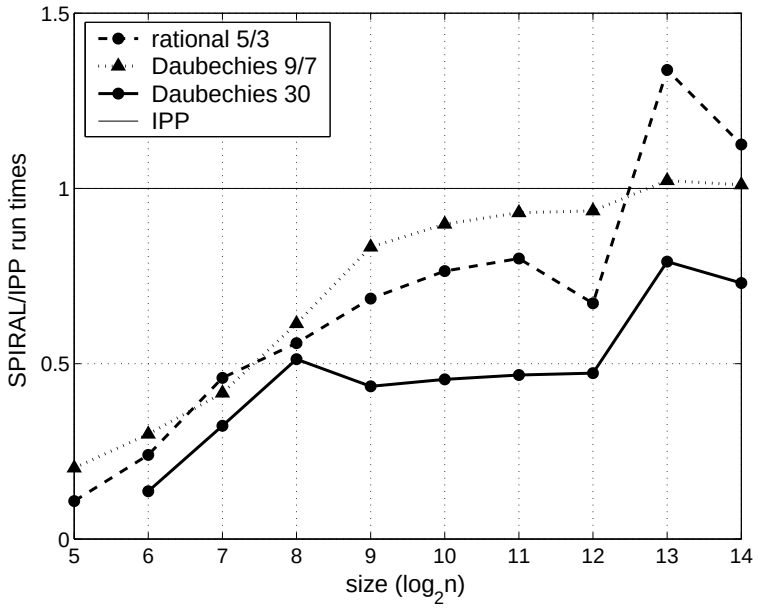


Figure 7.2: Comparing the best found DWT implementation and IPP code on Athlon-1.73 (lower is better)

The results are shown for DWT sizes between 2^5 and 2^{11}. For all three wavelets, our code outperforms the IPP code for almost all sizes. For rational 5/3 wavelet, SPIRAL generated code is faster by about 5 times for size 32 because of overheads inherent to the IPP implementation. For medium sizes the improvement varies between 50% and 20% until size 2^{13}, where the IPP code runs faster by about 20%. For Daubechies 9/7 wavelet, the results are similar except that even for larger sizes, the best found SPIRAL code still stays competitive with IPP. On the other hand, for Daubechies 30, our code performs better than IPP for all sizes, running in most cases about two times faster.

Benchmarks against IPP on Intel platforms. The results from the Athlon platform demonstrate the advantage of our automatic tuning system over the hand-coded libraries provided by Intel designed to achieve high performance on a generic computer platform. We now perform benchmark tests on the Intel's native platforms, for which the IPP routines are better tuned. We choose P4B-3.0-win with Intel-win compiler for evaluation.

In Figure 7.3, we show the run time comparison of our best found implementation and the Intel's hand-tuned scalar-code IPP routine `FIR` for implementing FIR filters. As before for the FIR filter experiments, in this figure lower lines mean better performance. For a filter of length 32, our best found implementation outperforms the IPP routines consistently across all considered sizes (2^2 – 2^{14}), reducing the run times between 20% and 40%. For longer filters, SPIRAL outperforms IPP scalar code for smaller filter sizes, but is slower for larger sizes, although it stays competitive with at most 20% slower run times. This is not surprising keeping in mind that the IPP libraries are tuned to schedule code optimally for their native architecture, even for scalar code, leading to considerable advantages for larger sizes for which memory bandwidth becomes the bottleneck.

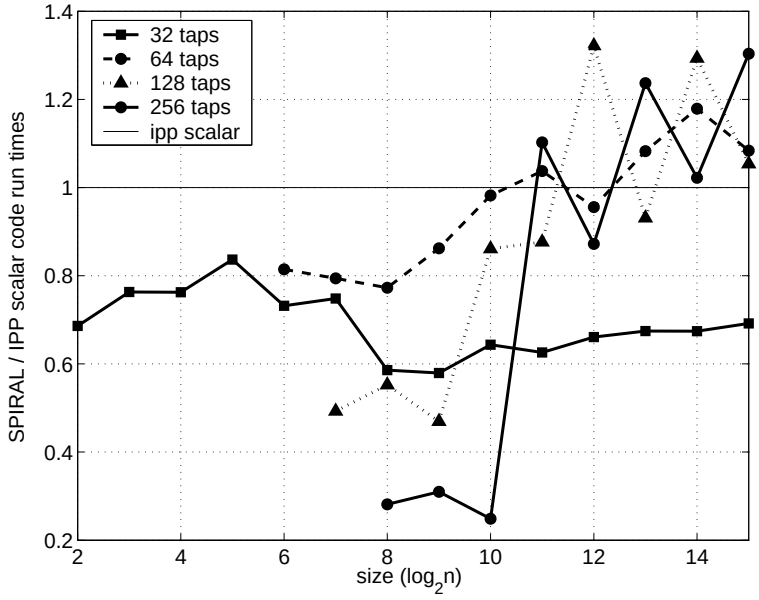


Figure 7.3: Comparing the best found filter code and IPP filter code on P4B-3.0-win

In Figure 7.4, we present results of another comparison of our code and IPP filter routines, this time for the DWT. As before, we measure the run time of our best found implementation and IPP's `DWT` function for three wavelets: rational 5/3, Daubechies 9/7, and Daubechies 30. The results on the Intel Pentium 4 machine are surprisingly similar to Athlon-1.73. For all three wavelets and for most of the DWT sizes, SPIRAL generated code outperforms the IPP routine. The improvement for Daubechies 30 wavelets is most notable and ranges between speedups of four times for smaller sizes and about 30% for larger sizes.

> Benchmarks against the hand-tuned IPP libraries show that our automatically generated and tuned code is competitive with and often outperforms the hand-coded routines provided by the vendor of the target platform.

FLOPS performance. To fully assess the quality of the generated code we compute the number of floating point operations per second for each of the best found implementations and compare them with the theoretical peak performance on the target platform. Hence, in constrast to the run time plots, higher lines in graphs mean better performance. The number of floating point operations is estimated from the asymptotic cost of the implemented algorithms; hence, the units we used we refer to as the pseudo FLOPS or, more precisely, pseudo MFLOPS.

Figure 7.5 shows MFLOPS performance plots on P4-3.0-win and Athlon-1.73 platforms for FIR filters using only time-domain methods for all sizes. On this plots, higher line corresponds to higher performance. We observe that, for filter lengths 16, 32, and 128, the performance is close to 2200 MFLOPS which is about 75% of the peak performance of 3000 MFLOPS that can be achieved if one floating point operation is performed per each cycle. On the right plot we see the results for

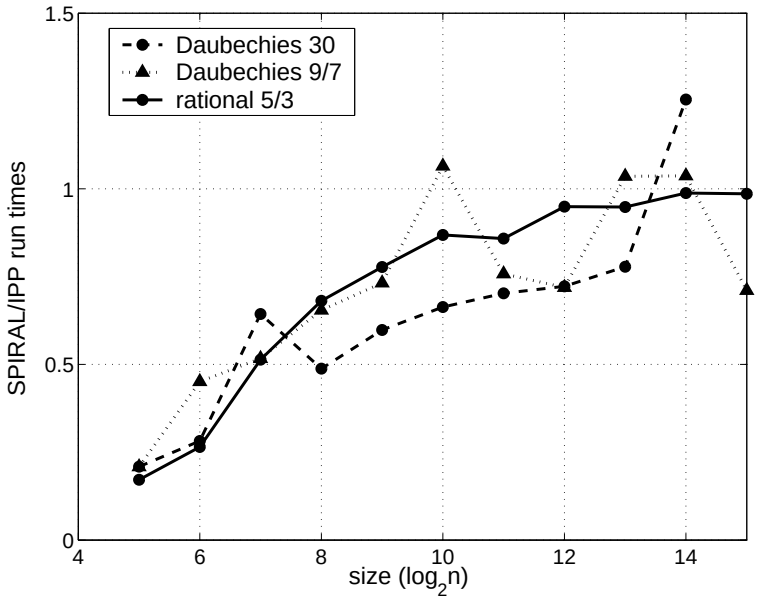
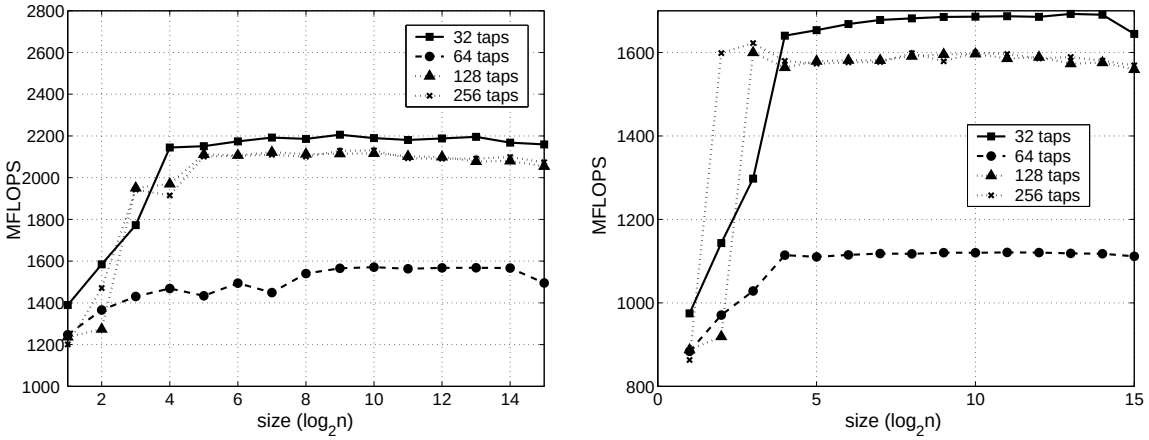


Figure 7.4: Comparing the best found DWT code and IPP DWT code on P4B-3.0-win for three wavelets: rational 5/3, Daubechies 9/7, and Daubechies 30

Athlon-1.73 that run close to 1700 MFLOPS or one floating point operation per cycle. However, Athlon provides two floating point units that can work in parallel, so the actual percentage of the peak performance is closer to 50%. For a filter of length 64, the performance drops rapidly on all platforms for reasons that are not completely clear. By inspecting the best generated rule trees, we observed that, only for filters of length 64, the best found blocking strategy uses basic Toeplitz blocks of size 16×16. In all other cases, the basic blocks are of size 8×8, which seems more efficient. One explanation of why the blocks 8×8 are not efficient in this special case could be that extensive conflict cache misses occur because of the blocking effect for this particular size.

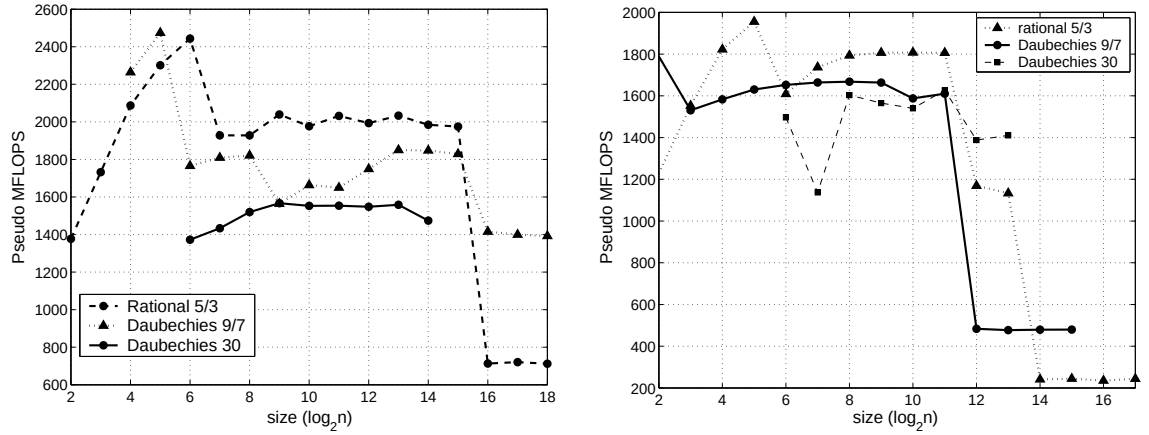
Similar performance plots are obtained for the best found code for DWTs. Figure 7.6 shows the MFLOPS results for the same platforms as for the FIR filter case: P4B-3.0-win, and Athlon-1.73. Again, higher lines indicate better performance. We implement the periodic DWT defined in (6.27) and compute the performance for the method that invokes Mallat rule (6.37) on the top level. On P4B-3.0-win platform, performance stabilizes around 70% for the rational 5/3 wavelets and around 50% for Daubechies 30 wavelets. The maximum is achieved for sizes 32 and 64 for rational 5/3 and Daubechies 9/7 wavelets, respectively, and exceeds 80% of the peak performance. We note that there is a sharp drop in performance for sizes 2^{16} and larger due to the cache boundary. On Athlon-1.73, the performance stays close to 50% of the peak performance for all three considered wavelets. However, in this case there is a sharp drop in performance already at size 2^{12}.

Next, we analyze the performance of several implementation approaches on different platforms and with different compiler options. We intend to show that our system generates surprising solutions using all the available rules we developed in Chapters 5 and 6. We also present results that show that the best found code heavily depends on the target platform, target compiler, and compiler



(a) P4B-3.0-win (b) Athlon-1.73

Figure 7.5: Performance of the best found FIR filter code in MFLOPS



(a) P4B-3.0-win (b) Athlon-1.73

Figure 7.6: Performance of the best found DWT code in MFLOPS (higher is better)

options. Finally, we show the spread in performance between different fast algorithms to exemplify the possible improvement that can be obtained by search.

7.3 FIR Filter Methods Across Multiple Platforms

In the first part of our set of experiments, we search and generate code for FIR filter transforms. We gradually introduce different implementation methods to investigate the performance and effectiveness of the breakdown rules we designed in Chapter 5. We test the performance of the main implementation methods on multiple platforms to emphasize the diversity of the generated solutions. In addition to different computer platforms, we test performance for different compilers and compiler options.

7.3.1 Setup of experiments

Our test transform for these experiments will be the FIR filter transform $\mathbf{Filt}_n(h(z))$ defined in (5.2) on page 93. At this point, we recommend reviewing some of the basic definitions and the notation for FIR filters in Chapters 3 and 5. We recall that the filter transform is defined by its input size n and filter coefficients or filter taps h_i specified by the polynomial $h(z)$. We call the degree (3.2) of the Laurent polynomial $h(z)$ the filter length k. We perform experiments for different filter sizes and filter lengths. The exact values of the filter taps are irrelevant for obtaining the run times as long as they are different from zero and one. For all our experiments, we set the filter taps randomly to double precision numbers except zero and one.

We roughly divide all experiments into several parts:

1. *Time-domain experiments* use breakdown rules for FIR filter transform that implement filters using blocking, nesting, or divide-and-conquer (Karatsuba) methods.

2. *Karatsuba experiments* include the Karatsuba rule (5.68) that reduces the arithmetic cost of time-domain methods from $\mathcal{O}\left(n^2\right)$ to $\mathcal{O}\left(n^{\log_2 3}\right)$;

3. *Transform-domain experiments* apply the RDFT convolution rule (5.72) to compute the filter in the frequency domain with $\mathcal{O}\left(n \log_2 n\right)$ cost.

7.3.2 Time-domain methods

Given a filter transform $\mathbf{Filt}_n(h(z))$ of size n and filter length k, the straightforward implementation is to design a loop over all n outputs as in the rule

$$\mathbf{Filt}_n(h(z)) \rightarrow \mathrm{I}_n \otimes_{k-1} (h_0, \ldots, h_{k-1}) \tag{7.5}$$

and possibly loop over filter taps stored in a table. However, storing and retrieving filter coefficients from a table is not very efficient. For reasonable length filters, coefficients can be included directly in code:

```
for i = 0..n-1
   y[i] = h[0]*x[0+i]+h[1]*x[1+i]+...+h[n-1]*x[n-1+i]
end
```

This implementation is the most straightforward, yet reasonably good implementation of the transform $\mathbf{Filt}_n(h(z))$ and typically one chosen by a human programmer with little knowledge about compilers.

Time-domain blocking methods. This is our starting point. We design this implementation in our framework by allowing only one implementation of the overlap-save rule (5.56) for $s = 1$ and set the degree of loop unrolling to be at least the filter length k. We use this approach as the baseline method for the first experiment.

We compare different time-domain algorithms relative to the baseline method. We perform these experiments on Xeon-1.7 with gcc-O1-3.2 compiler flags. Figure 7.7 shows relative run times with respect to the baseline method for various filter sizes along the x-axis. Experiments were first run

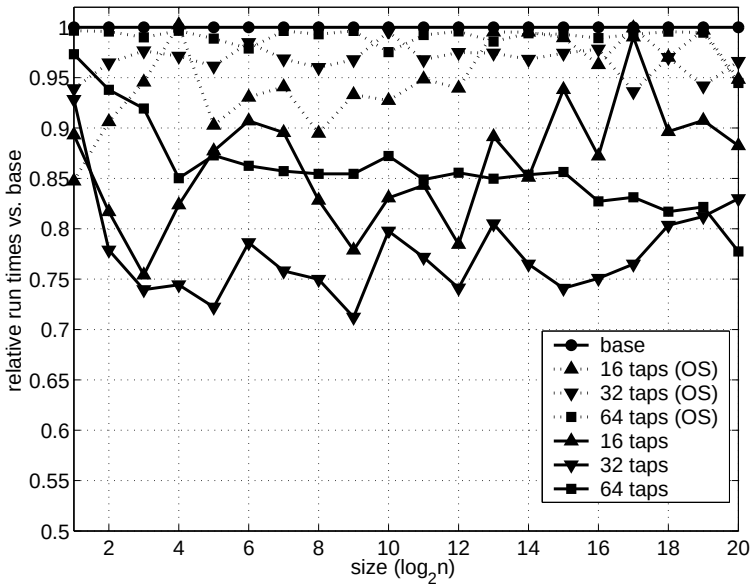


Figure 7.7: Comparison of the time-domain methods on Xeon-1.7 (lower is better)

by turning on only overlap-save (OS) rule. We ran the experiment for filters with 16, 32, and 64 taps. By only applying the OS rule the run times improve by up to 10% for the 16-tap filter and slightly less for other filters.

Next, we allow blocking (5.76) and nesting rules (5.77). Both rules allow flexible recursive blocking strategies by invoking recursively either the blocking rule for Toeplitz transforms (5.75) or again the nesting rule. The run times in Figure 7.7 show a significant improvement over the baseline method by as much as 30% for a filter with 32 taps. On this platform, the nesting rule is never found, and the blocking wins as the best implementation strategy. We note that the overlap-add (OA) rule was also tried by the search; however, we consistently observe that the OA rule leads to slower run times than OS rule on all platforms, concluding that output locality is more important than the input locality. For that reason we perform all experiments that require overlapped methods using only OS rule.

Compiler deficiencies. The reason for choosing very simplified compiler options gcc-O1-3.2 is that the GNU C compiler cannot optimize well large portions of unrolled code. Turning on more sophisticated compiler optimization techniques can violate the regular structure of data flow for filter implementations. To demonstrate the deficiency of the GNU C compiler on Linux, we repeat the above experiment for OS method, which has the most regular structure of all filter algorithms, but we now turn on sophisticated compiler options gcc-O6-3.2. In Figure 7.8 we see how detrimental extensive optimizations of GNU C compiler can be on filters of smaller size. The performance stays slightly worse even for larger filters, suggesting that more "intelligent" compiler optimization techniques should not be taken for granted and used in all situations. With Xeon-1.7 and GNU C compiler, we determine that `-O1` and a few additional flags specified in Table 7.2 lead to better performance.

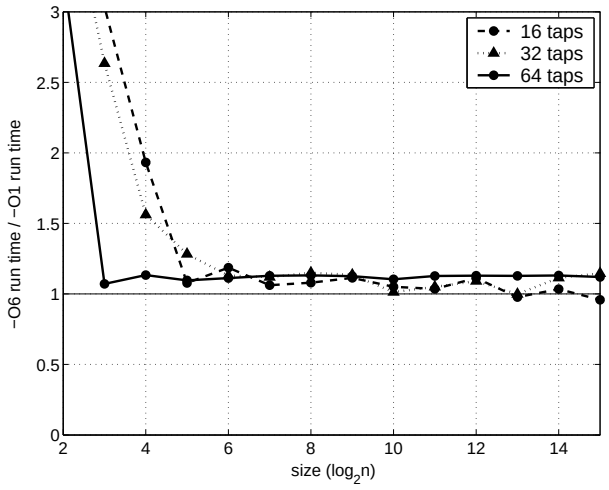


Figure 7.8: Blocking methods for gcc-O6-3.2 and gcc-O1-3.2 on Xeon-1.7

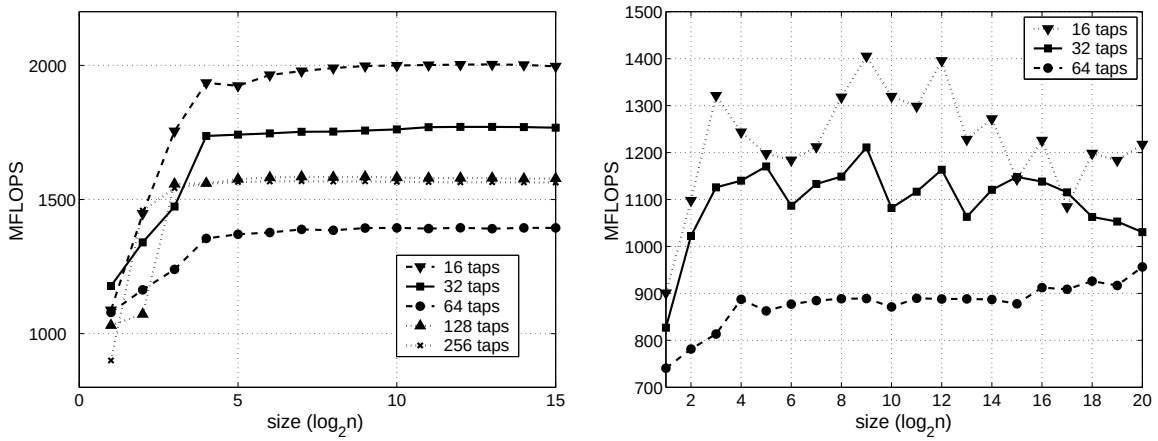
FLOPS performance of time-domain methods. The same experiment was run in parallel on four different platforms: Xeon-1.7 with gcc-O1-3.2, Athlon-1.73, P4B-3.0-win, and P4e-3.6, all with Intel-win compiler flags. We showed the FLOPS plots for time-domain methods on P4B-3.0-win and Athlon-1.73 platforms in Figure 7.5. In Figure 7.9, we show the MFLOPS results for flop/s for P4E-3.6 and Xeon-1.7.

We observe that Intel C++ compiler is very consistent for larger size filters, where the performance stays almost exactly the same for all sizes. This suggests that the compiler applies intensive optimization methods and unifies implementations; thus, for different sizes, compiled code has very similar structure and, therefore, similar performance. On the other hand, the GNU C compiler produces erratic behavior, as we can see on the Xeon plot. This is not surprising since the optimizations are set to almost a minimum and the performance depends highly on the structure of the algorithm.

7.3.3 Karatsuba methods

The next set of experiments is designed to investigate the efficiency of Karatsuba, or divide-and-conquer methods we discussed in Section 3.1.6. Even though Karatsuba methods operate also in time domain, when we refer to time-domain methods we mean only blocking and nesting strategies that preserve the cost of the straightforward filter implementation. Karatsuba methods reduce the cost for every recursive implementation of the decomposition (for more details see Appendix B). To enable the Karatsuba approach and combine it with the best time-domain algorithms, we turn on the radix-2 Karatsuba rule specified in (5.68) on page 110 and leave all time-domain rules from the previous experiments active. By doing this, we allow search to explore the algorithm space where Karatsuba techniques can be applied on arbitrary levels and combined with best overlap-save and blocking/nesting strategies.

Figure 7.10 compares run times for the best found time-domain method and the best Karat-



(a) P4E-3.6

(b) Xeon-1.7

Figure 7.9: Performance in MFLOPS for best found time-domain method.

suba method for Athlon-1.73 and Macintosh platforms with compilers Intel-win and gcc-O1-mac, respectively. On Athlon, the best blocking strategy outperforms Karatsuba only for filter length 16. For filter lengths 32, 64, and 128, the best rule tree is usually a combination of overlap-save on the top level, Karatsuba decomposition for two or three recursion, and some blocking strategy on smaller filters. In Table 7.3, we provide two examples of rule trees found by DP search on Athlon-1.7 platform. The first rule tree decomposes a 64-tap filter of size 32 into smaller filters using two levels of Karatsuba decomposition and computes small filters using overlap-save technique. The second rule tree, shown on the right in Table 7.3, decomposes a 64-tap filter of size 1024 into a filter of size 128, applies three levels of Karatsuba decomposition, and finally implements smaller filters using two levels of blocking, first into Toeplitz blocks of size 4 and then of size 2.

On both platforms, Karatsuba methods achieve speedup of more than two times when compared to the best time-domain methods for 64-tap filters, and for 32-tap filters on Macintosh platform. For shorter filters, time-domain methods achieve high performance rates, whereas Karatsuba methods reduce the computational cost but have more complicated data flow patterns. However, for Macintosh platform, Karatsuba methods win even for length 16 filters. However, we report that Karatsuba method was not found on Xeon-1.7 platform with gcc-O1-3.2 compiler. In that particular case, the blocking strategies were superior over Karatsuba methods for all sizes, suggesting that the intricate structure of the recursion in divide-and-conquer approach is not appropriately handled by the compiler. The regular structure of the convolution operation and efficient blocking strategies to improve locality are preferable for this platform and compiler combination.

We investigate the efficiency of Karatsuba methods w.r.t different methods after we obtain experiments for the transform-domain algorithms in the next section.

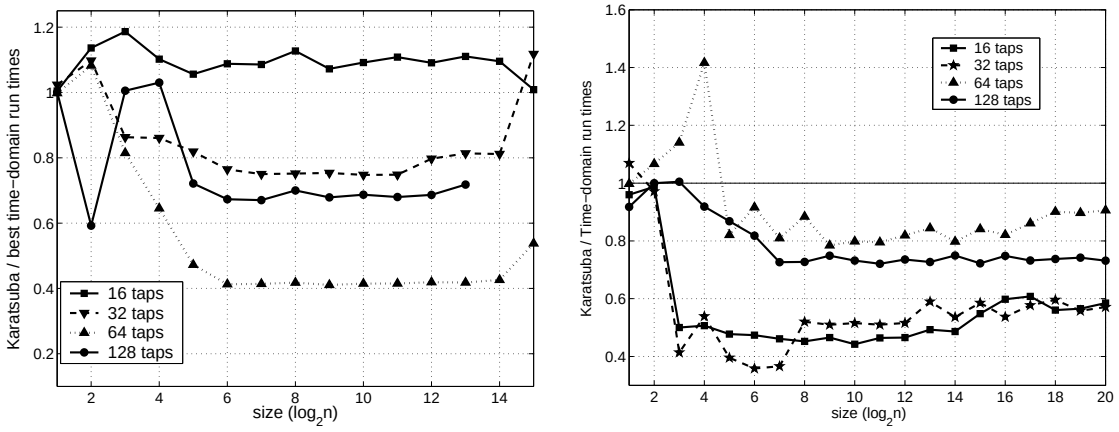
7.3.4 Transform-domain methods

The computational cost of implementing FIR filters can be further reduced by using transform-based algorithms. For large sizes, the cost of the implementation is of $\mathcal{O}(n \log n)$ as we discussed

Table 7.3: Two rule tree examples for Karatsuba method found by search

```
ruletree := RuleFilt_Karatsuba(
  Filt(32, 64, -63),
    RuleFilt_Karatsuba(
    Filt(16, 32, -31),
      RuleFilt_Karatsuba(
      Filt(8, 16, -15),
        RuleFilt_OverlapSave(
        Filt(4, 8, -7),
        RuleFilt_Base(Filt(2, 8, -7)) ) ) )
```

```
ruletree := RuleFilt_OverlapSave(
  Filt(1024, 64, -63),
  RuleFilt_OverlapSave(
    Filt(128, 64, -63),
    RuleFilt_Karatsuba(
      Filt(64, 64, -63),
      RuleFilt_Karatsuba(
        Filt(32, 32, -31),
        RuleFilt_Karatsuba(
          Filt(16, 16, -15),
          RuleFilt_Blocking(
            Filt(16, 16, -15),
            RuleToeplitz_Base(Toeplitz(7)),
            RuleToeplitz_Blocking(
              Toeplitz(7),
              RuleToeplitz_Base(Toeplitz(3))))))
```



(a) Athlon-1.73

(b) Macintosh

Figure 7.10: Comparison of Karatsuba methods and the best blocking/nesting strategy.

in Section 3.1.8. In this section, we implement filters based on the real discrete Fourier transform (RDFT) convolution property. We implement FIR filters using the circulant rule (5.65) to embed filters into circulant transforms and then using rule (5.72) to compute the circulant transform in the transform domain. Approximate cost for the RDFT is $\mathcal{O}(2.5n \log n)$, where n is the size of the transform, leading to the total cost of $\mathcal{O}(5n \log n)$ including both RDFT and inverse RDFT. We investigate the performance of implementations based on this approach and compare them with the best time-domain and Karatsuba methods we found so far.

Comparison of computational costs of time-domain and transform-domain methods gives us a rough estimate of the runtime performance. For time-domain methods, the arithmetic cost is exactly $2n \cdot k - n$, where n is the size of the filter output (the number of rows of the matrix) and k is the number of filter coefficients. This cost is lower than the cost of transform-based methods for two cases: 1) filter size n is small, and 2) filter length k is much smaller than n. We mentioned in

Section 3.1.5 that the latter case can be avoided by using block convolution methods such as the overlap-save. The basic idea is to segment the filter matrix into filters of block size $b = n/s$ so that $b \sim k$, and then apply transform methods on filter matrices that are more "dense".

Following the above discussion, we design our experiments to allow the overlap-save rule (5.56) to reduce the size of a filter n to b so that it is close to the filter length k, and then apply circulant and RDFT rules to compute the filter in the transform-domain. There is an inherent tradeoff in how to efficiently segment the filter matrix before applying the RDFT transform method. On one hand, there is a compulsory overhead in implementing filters in the transform domain arising from the embedding rule (5.45) equal to $k-1$ additional rows needed to complete the circulant matrix. It is clearly advantageous to choose b much larger than $k-1$ to reduce the relative cost incurred by the overhead. On the other hand, the cost of implementing the filter using the transform method increases with b and has to be lower than the direct implementation with the cost $2bk-k$. In this case, it is advantageous to reduce the size b to make transform-domain approach more efficient. In other words, when a filter is embedded into a circulant matrix, the number of zero entries of the circulant matrix is proportional to b/k. Less zero entries in the matrix lead to more efficient transform-domain methods when compared to the time-domain implementation. This tradeoff between the sparsity of the circulant matrix and the overhead incurred by filter embedding determines the optimal segmentation for the transform-based approach.

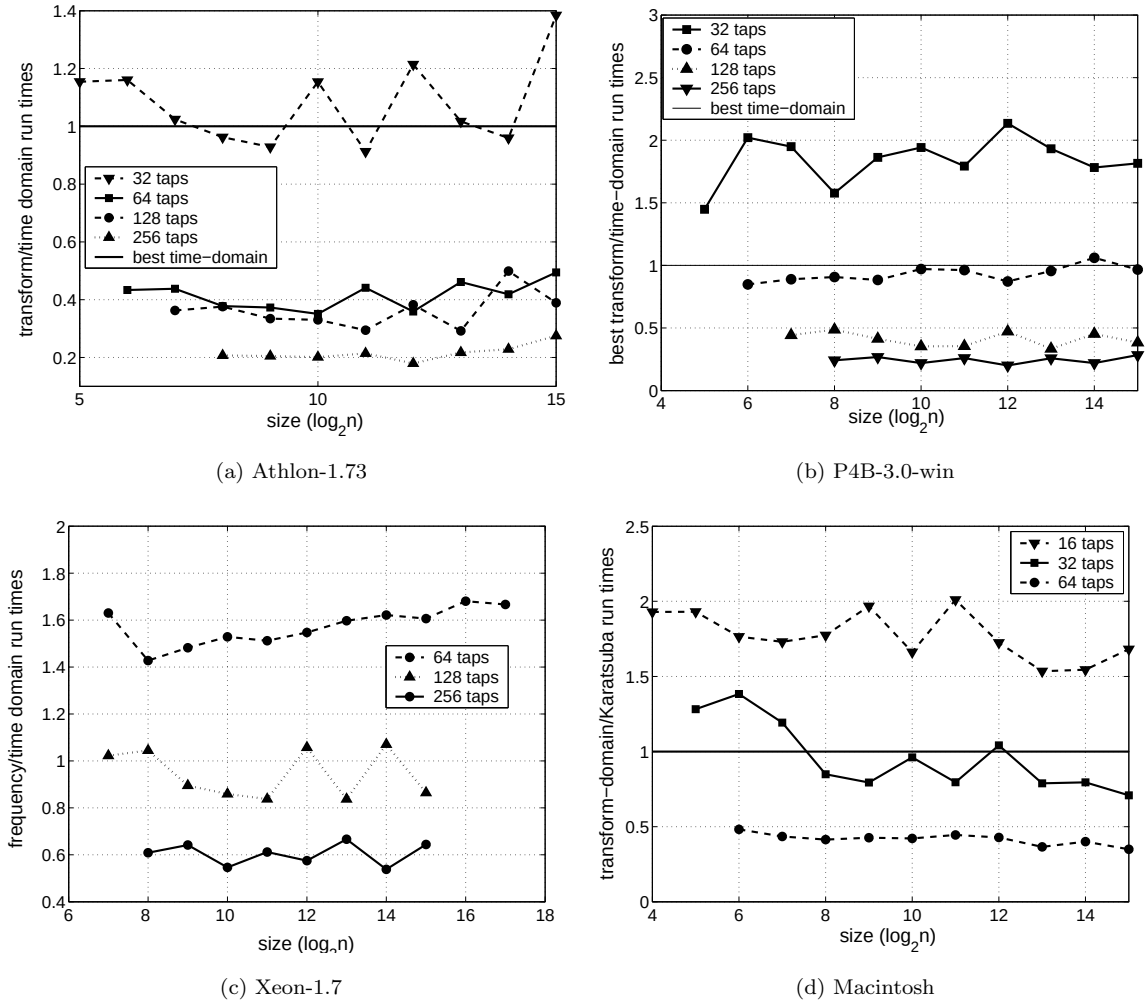
DP search finds the best overlapped filter blocks and the best RDFT convolution rule (5.72) for the target platform. In Figure 7.11 we show results of experiments run on four different platforms:

1. Athlon-1.73 with Intel-win compiler
2. P4B-3.0-win with Intel-win compiler
3. Xeon-1.7 with gcc-O1-3.2 compiler
4. Macintosh with gcc-mac-O1 compiler

The graphs show relative run times of the best found transform-domain methods normalized by the run time of the best found time-domain methods for filter sizes 2^1 to 2^{15}. Different lines in graphs show the run time comparison for filters of different lengths.

On Athlon-1.7, for a 32-tap filter, the transform-domain method is competitive with the best found time-domain method, each being better for particular filter sizes. For longer filter lengths, our transform domain implementation clearly wins over the best blocking/nesting strategy with the increasing speedup as the filter length increases. For a 128-tap filter, the best transform-domain method is about five times faster. On the other hand, for P4B-3.0-win platform, the 32-tap filters are still running much faster, almost 2 times, using the best available time-domain method, typically the best recursive blocking strategy. However, similar to the Athlon platform, a 64-tap filter implementation is faster in the transform domain, and the advantage increases with the length of the filter, as expected.

The results obtained on Xeon-1.7 platform showed a different picture. For shorter filter lengths, the transform-domain approach is clearly inferior to the best found blocking strategy we reported in the previous section shown in Figure 7.7. Even for a 64-tap filter, the run times were slower



(a) Athlon-1.73

(b) P4B-3.0-win

(c) Xeon-1.7

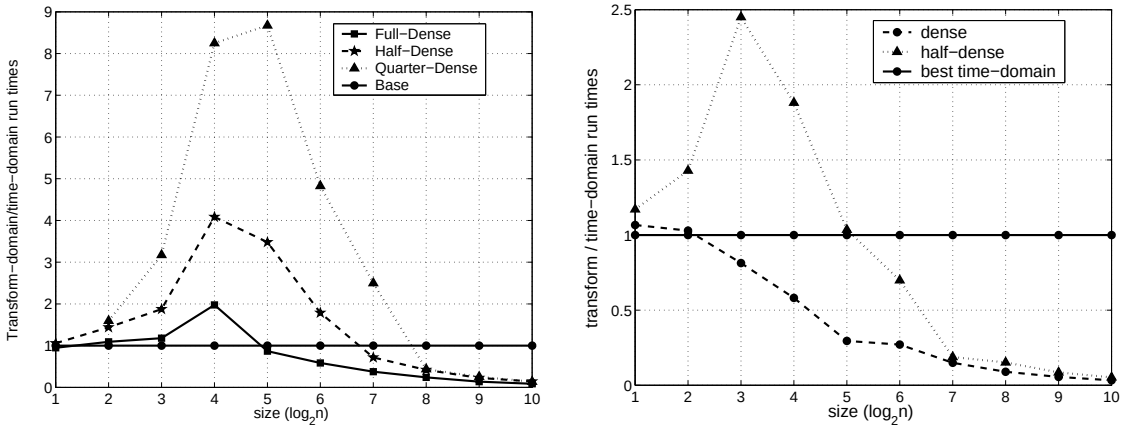
(d) Macintosh

Figure 7.11: Comparison of the best found time-domain and transform-domain methods.

by about 50%, in contrast to P4B-3.0-win and Athlon-1.7. The first filter length for which the transform-domain approach becomes competitive is 128.

In stark contrast to Xeon results, on Macintosh platform, similar to Athlon-1.7, the transform-domain methods compete with the best found blocking and nesting strategies already for filter of length 32. The significant difference of results on different platforms can be attributed to the capability of used compilers, especially when compared between Xeon-1.7 and P4B-3.0-win because they both feature Intel processors with similar design. Another reason are the different hardware features. For example, both Athlon and Macintosh feature large L_1 cache and large register banks, allowing a compiler to do a much better job of scheduling the implementation.

Cross-over between time and transform-domain methods. We conclude that the cross-over point between the time-domain and the transform-domain methods depends heavily on the chosen computer platform and compiler and has to be determined empirically since the arithmetic cost comparison only provides a very rough estimate of the actual performance. We investigate this further by designing experiments that focus on determining the performance of transform-domain



(a) Xeon-1.7

(b) P4B-3.0-win

Figure 7.12: Comparison of time-domain and transform-domain methods for circulant matrices.

algorithms for circulant matrices, that represent circular convolutions used in transform-domain methods. We explore cases of circulants that are fully "dense" in the sense that they contain no zero entries, "half-dense" circulants with half of the entries being zero, and similarly defined "quarter-dense" circulants. The motivation for performing these experiments is that the circulant matrices constructed from overlap-save and circular convolution techniques have at least half of their entries equal to zero, as we discussed earlier in this section, and possibly more zeros if the overhead arising from the embedding is a deciding factor in efficiency of these methods.

Circular convolution. We search over different time-domain strategies for full-dense, half-dense, and quarter-dense circulant transforms, which include the blocking rule (5.74) and rule (5.66) that implements the circulant through the filter transform. We then compare the results with the best found frequency domain implementation using rule (5.72). To investigate the discrepancy between the results obtained for time-domain and frequency-domain methods on different platforms, we choose two platforms: Xeon-1.7, and P4B-3.0-win.

The first experiment compares the run time of the full-dense circulant transform implemented in the time domain using the blocking rule (5.74), and in the transform domain using rule (5.72). The graph 7.12(a) shows that, for Xeon-1.7, the blocking methods outperform the transform method up to size 2^4. Starting from size 2^5, the transform method is better and steadily improves for larger sizes, as expected by its cost of only $\mathcal{O}(n \log n)$. We saw in Figure 7.11 that, for FIR filter transform, the cross-over point between time and transform-domain methods is at the filter length 128. This is not surprising because we have to keep in mind that the transform-domain methods for filters induce circulant matrices with at least half of their entries zero, and additional overhead. We implement half-dense and quarter-dense circulants using rules (5.74) and (5.66) and again compare the run times with the transform methods. For half-dense circulants, the transform method outperforms time-domain methods only for size 128 and higher, whereas for quarter-dense circulants the cross-over point is before size 256. This result suggests that the filters are implemented in the transform domain by reducing large filters into filters of size similar to the filter length $b \sim k$, leading to

circulants that are close to being half dense.

We perform the same experiment with fully dense and half-dense circulants on P4B-3.0-win platform. From graph 7.11(b), we observe that the transform method outperforms time-domain methods for filter lengths larger than 64. Graph 7.12(b) shows the results for half-dense and fully dense circulants. For fully dense circulants, the crossover point is between sizes 4 and 8, whereas for half-dense circulants just above size 32. This is again consistent with the results obtained from FIR filter experiments.

7.3.5 Comparison of all methods for FIR filters

Determining the right implementation strategy is highly dependent on the chosen computer platform and used compiler. To summarize, we compare results using three different approaches: 1) time-domain methods including overlap-save, blocking, and nesting rules, 2) Karatsuba methods that combine Karatsuba decomposition rule with best time-domain methods, and 3) transform-domain methods using RDFT convolution rule (5.72). In Figure 7.1, we show the comparison of all three methods on Athlon-1.73 for four different filter lengths: 16, 32, 64, and 128.

For a filter with 16 coefficients, the best blocking strategy outperforms both Karatsuba methods and the transform-based methods, with the latter being considerably slower as we expected based on the earlier results we showed in Figure 7.11. For a filter of length 32 the transform and time-domain methods compete against each other, as we have seen in experimental results comparing time and transform-domain methods on Athlon in Figure 7.11. However, the Karatsuba method is a clear winner for this filter lengths. The graph 7.1(c) shows that the transform-domain approach is competitive with the best Karatsuba method for filter length 64. Finally, in graph 7.1(d), we show the results obtained for a 128-tap filter. The transform-domain method for this filter length outperforms all other methods.

The Athlon example shows that the best found implementation depends on the filter size and length and that all included rules and methods are used in constructing the optimized implementation at some point. In Table 7.4, we illustrate the diversity of considered platforms by specifying the best found implementation for all different filter lengths.

Table 7.4: Best found methods for FIR filter transform of various lengths on different platforms.

	16-tap	32-tap	64-tap	128-tap
Xeon-1.7	Blocking	Blocking	Blocking	RDFT
Athlon-1.73	Karatsuba/Blocking	Karatsuba	Karatsuba/RDFT	RDFT
P4B-3.0-win	Blocking	Karatsuba	RDFT	RDFT
Macintosh	Karatsuba	Karatsuba	RDFT	RDFT
P4E-3.6	Blocking	Karatsuba	Karatsuba	RDFT

We note that the best found methods are different across different filter lengths for all listed platforms on which we ran experiments. As two extreme cases we emphasize results for Xeon-1.7 and Macintosh. On Xeon, blocking strategies are fastest up to filter length 128, after which the

transform-domain methods take over. Karatsuba methods are never found on Xeon. In contrast, methods based on Karatsuba rule are fastest on Macintosh for filter lengths 16 and 32. For a filter of length 64 and higher, the RDFT-based approach is providing the best performance.

7.4 DWT Methods on Multiple Platforms

In Section 7.2, we performed benchmarks for our automatically generated DWT code. We saw that our code either outperforms the hand-coded IPP function `DWT` or stays competitive for most DWT sizes and considered wavelet bases. In this section, we further investigate the efficiency of specific methods for implementing the DWTs across different platforms and different wavelet bases. Our goal is not to determine the efficiency of a particular method on any given platform but to demonstrate the richness of algorithms found by our automatic implementation and search and show that various factors affect the optimal solution for a particular platform. We perform experiments for the periodic DWT defined in (6.27) for three different wavelet bases:

- Rational 5/3 biorthogonal symmetric wavelet where the lowpass and the highpass filters are given as
$$h(z) = -\tfrac{1}{8}z^2 + \tfrac{1}{4}z + \tfrac{3}{4} + \tfrac{1}{4}z^{-1} - \tfrac{1}{8}z^{-2}$$
$$g(z) = \tfrac{1}{4}z^2 - \tfrac{1}{2}z + \tfrac{1}{4}$$
- Daubechies 9/7 biorthogonal wavelets used in JPEG2000 standard [17].
- Daubechies 30 orthogonal wavelets with coefficients given in [87]

7.4.1 Evaluation of DWT implementation methods

We compared three different methods determined by the top-level DWT rule:

1. *Mallat method.* We use rule (6.37) to compute the $\mathbf{DWT}^{\mathrm{per}}_{n,j}(h(z), g(z))$ for the set of chosen wavelets
2. *Polyphase method.* We use the Polyphase rule (6.44) to decompose $\mathbf{DWT}^{\mathrm{per}}_{n,1}(h(z), g(z))$ into four circulant transforms. This rule is a gateway to the entire space of FIR filtering algorithms, including transform-domain algorithms and Karatsuba methods.
3. *Lifting scheme.* Rule (6.54) is used to decompose $\mathbf{DWT}^{\mathrm{per}}_{n,1}(h(z), g(z))$ into lifting steps. This method reduces the arithmetic cost of the computation asymptotically by 50%.

Figure C.2 shows three graphs with relative run times of polyphase and lifting scheme methods with respect to the run time of Mallat method for three different wavelets on P4B-3.0-win platform. For all experiments the degree of unrolling was set to 1024, i.e., for blocks of size 32×32. The top graph displays the results for rational 5/3 wavelet. For smaller sizes 2^2 to 2^5, the lifting scheme achieves the fastest run times; however, for all larger sizes, Mallat method is the best found approach. In this case, the lifting scheme method includes two lifting steps and reduces the arithmetic cost by about 30% but is still slower for larger transforms because increased critical path offsets the cost advantage.

For Daubechies 9/7 wavelets, the liftin4g scheme features three lifting steps and still performs slower than Mallat method, except for sizes 2^8 and 2^9. It is interesting to note that, in this case, the polyphase method plays important role for smaller size (16–128). Since the downsampled filter lengths for the polyphase method are very short (4 and 3 taps for highpass even and odd filters, respectively), most of the sophisticated filtering techniques do not apply in this case. It is therefore surprising that the polyphase method outperforms Mallat method because it mainly serves as a gateway to advanced filtering techniques. Since code is completely unrolled for sizes 128 and smaller, it is probable that the polyphase rule schedules the computations in a way that is preferable for P4B-3.0-win architecture and Intel-win compiler.

Finally, graph C.2(c) shows results for Daubechies 30 wavelet, which features longer highpass and lowpass filters with 30 taps. In this case, lifting scheme outperforms all other methods and provides the speedup of more than two times over Mallat method. Polyphase approach also improves over Mallat method since, in this case, sophisticated filtering techniques, such as Karatsuba methods, are found. However, the filter lengths are still not enough for transform-domain methods to be efficient, and on this platform and this wavelet basis, the lifting scheme approach is the best implementation strategy.

We perform the same experiment on Athlon-1.73 and the results are shown in Figure C.3 in Appendix C. The comparison of methods follows a similar pattern as for P4B-3.0-win platform; however, we point out to several differences. Both the lifting scheme and the polyphase methods outperform Mallat method for Daubechies 9/7 wavelet for sizes up to 2^8. For Daubechies 30 wavelet, the lifting scheme is the best approach for sizes 2^7 to 2^{12}, but for larger DWT sizes polyphase method with better filtering techniques is a better solution.

On Macintosh, even for rational 5/3 wavelet, the lifting scheme implementation provides faster run times up to size 2^6. These results are shown in Figure C.4 in Appendix C. However, for Daubechies 9/7 wavelet, the polyphase method is fastest for sizes 2^6 to 2^{11}. Furthermore, it is interesting to note that the lifting and the polyphase methods compete for Daubechies 30 for most sizes. The fastest implementation alternates between these two methods as the size of the transform increases.

Performance of different lifting scheme strategies. We perform experiments to determine the performance of various lifting schemes that can be derived for the same DWT based on the degrees of freedom in dividing Laurent polynomials. Detailed analysis and explanation of how to obtain different lifting schemes is provided in Appendix A. It turns out that different factorizations of the DWT into lifting steps leads to different implementations and different run times. To illustrate the concept, we measure the run times of all possible lifting schemes for the DWT with Daubechies 9/7 wavelet on Xeon-1.7. Out of all 27 possible lifting schemes, only one preserves the symmetry of the lifting step filters. All lifting schemes have either four or five lifting steps. In Figure C.1 we show the histogram of the run times obtained for all lifting scheme factorizations. We emphasize that the spread in run times is almost a factor of five between the fastest and the slowest lifting scheme. Among the fastest lifting schemes is the symmetric one, which also have only four lifting steps. Note that some of the schemes are measured twice and appear two times in the histogram.

7.5 Compiler Issues

We have already seen that the run times and the best found algorithms vary significantly between different, and even very similar architectures. We now investigate the effects of different compilers and compiler options on performance.

The first experiment compares the run times of the best found time-domain and the best-found Karatsuba algorithms compiled by two compilers: icc-8.0-lin and gcc-O1-3.3 (see Table 7.2). Figure 7.13 shows the relative run times on P4-1.6-lin platform obtained by dividing the run time obtained by the gcc compiler and the run time when the icc compiler is used. Interestingly, when the best blocking and nesting strategies are implemented, the Intel icc compiler provides faster run time, and more so when the filter is shorter. For a filter of length 16, Intel compiler reduces the run time by as much as 30% over the gcc compiled code. On the other hand, for Karatsuba methods, the GNU compiler produces faster code for both filter lengths.

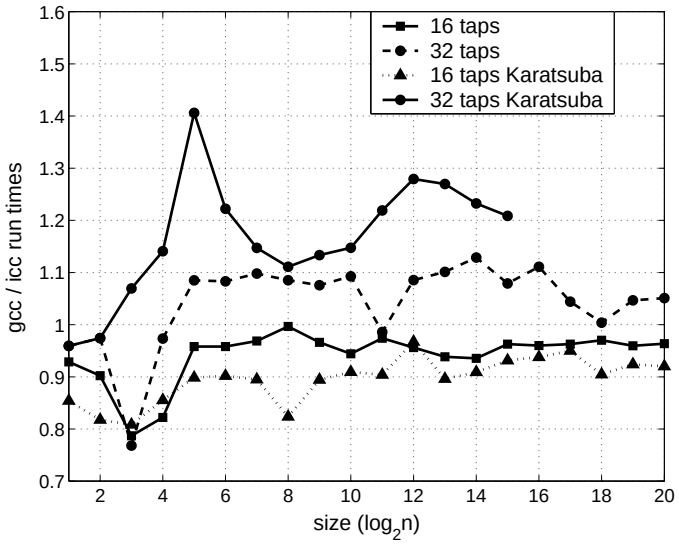
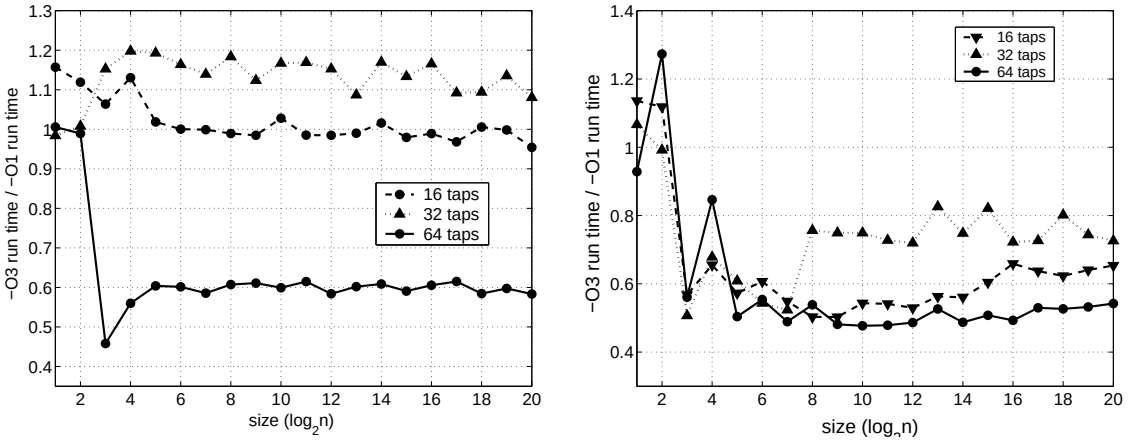


Figure 7.13: Comparison of run times for time-domain and Karatsuba methods compiled by icc-8.0-lin and by gcc-O1-3.3 compilers on P4-1.6-lin platform

In the second experiment, we compare the results on Macintosh for time-domain and Karatsuba methods for FIR filters obtained for the same GNU C compiler using two different compiler options: gcc-mac-O1 and gcc-mac-O3. Figure 7.14 shows measured relative run times of code compiled using gcc-mac-O3 options with respect to code compiled using gcc-mac-O1 options. The left-hand side graph illustrates the results for time-domain methods. For shorter filters, lower compiler optimization options gcc-mac-O1 achieve faster run times, but for a 64-tap filter, gcc-mac-O3 options achieve the speedup over gcc-mac-O1 by almost two times. For Karatsuba methods, however, gcc-mac-O3 improves the run time for all three filter lengths. A speedup of two times is obtained for a 64-tap filter, but much less for a filter of length 32.

We showed that the choice of compiler and used compiler options can considerably affect the runtime performance. Even more interesting is the fact that the improvement is sometimes unpredictable and that more sophisticated compiler optimization techniques can lead to slower run

times, as was the case for blocking and nesting strategies shown in graph 7.14(a). This provides yet another dimension in searching the space of implementations. Even though our framework does not provide the mechanism for searching over different compiler options, experiments like these can help in determining reasonable compiler flags to achieve high performance.



(a) time-domain methods

(b) Karatsuba methods

Figure 7.14: Effect of compiler options gcc-mac-O1 and gcc-mac-O3 on the run time of: (a) time-domain methods, and (b) Karatsuba methods for filters on Macintosh platform.

7.5.1 Compiler vectorization

Finally, we conduct a set of experiments on P4E-3.6 platform that provides a set of special vector instructions. To fully utilize vector instruction sets, special optimizations have to be performed on the algorithmic level to identify basic computational blocks that can be efficiently vectorized [47]. General purpose compilers, such as Intel C++ 8.1, provide code optimizations that operate on the code level and attempt to utilize special instructions to speed up code. P4E-3.6 provides 2-way vectorizing instructions for double precision code, which we use in our experiments. This means, theoretically, a speedup of two times is achievable by using vector instructions.

The experiments on P4E-3.6 are run using Intel C++ 8.1 compiler with Intel-SSE3 compiler flags that enable vector instructions assembled from scalar code. Figure 7.15 shows the speedups for FIR filters of lengths 16, 32, 64, and 128 obtained using internal compiler vectorization methods. We observe that the maximum speedup does not exceed 15% for most sizes. For a 64-tap filter, the speedup obtained by compiler vectorization methods is almost negligible. This suggests that the vectorization has to be performed on the algorithmic level to enable compiler to make full use of special instructions, similar to largely successful efforts for the DFT transform algorithms in [47].

7.6 Concluding Remarks

In this chapter, we performed experiments with our framework for FIR filters and DWTs integrated into the SPIRAL system to determine the quality of our automatically generated and platform-tuned

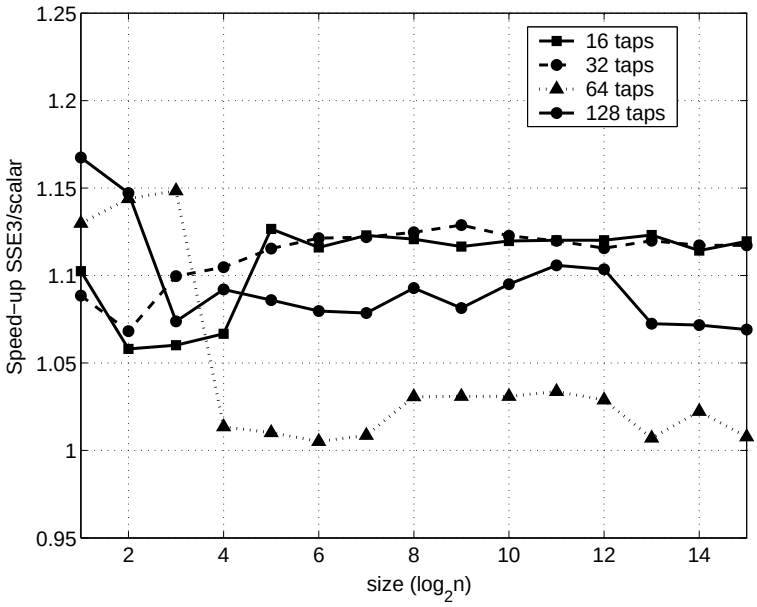


Figure 7.15: SSE3 vectorization speedup for FIR filters obtained by Intel-SSE3 compiler

code. Our goal was to demonstrate the main advantages of our system that tunes software implementations for filtering and wavelet kernels to different architectures. At this point, we summarize these advantages.

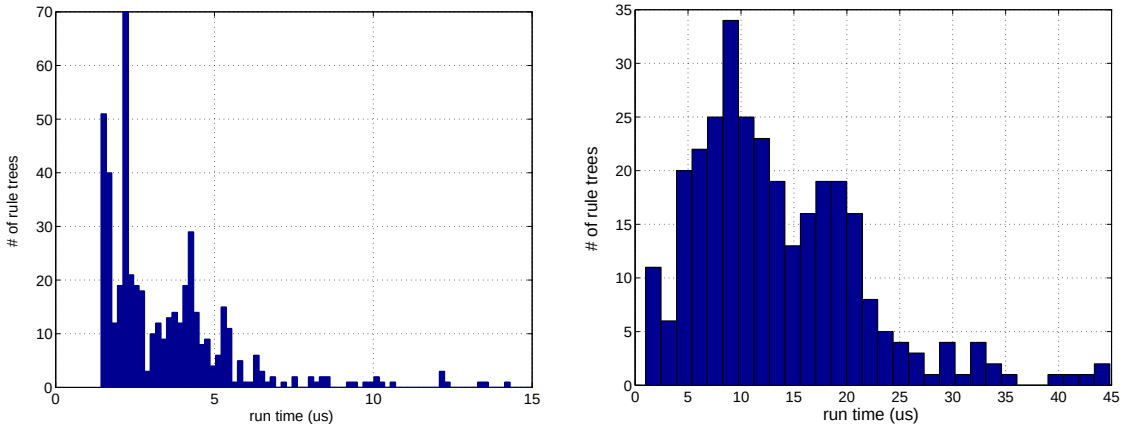
1. Our framework, together with the SPIRAL system, enables automatic generation of the entire space of fast algorithms and implementation choices providing the user with the readily available C or Fortran code.

2. The system automatically tunes code for filtering and wavelet transforms to the platform on which it they are to be implemented.

3. Our automatically generated and tuned code for FIR filters and DWTs *competes or even outperforms* hand-tuned software routines developed by algorithm and architecture experts and teams of programmers employed by Intel. The benchmarks were conducted on both Intel and non-Intel platforms.

4. Recursive application of rules we defined in our framework generates a space of algorithms that is created by a surprising combination of the existing algorithms found in the literature. It is conceivable that some of these algorithms generated by a machine have never been tried before by human programmers, even though all of the captured methods are well-known to signal processing experts. The richness and the extent of the search space is often enough to match the human ingenuity in achieving high performance.

To illustrate the richness of our search space and the scale of the performance gains obtained by search, we generate a large number of random rule trees that describe different algorithms for



(a) Filter $\mathbf{Filt}_{128}(h(z))$

(b) $\mathbf{DWT}^{\mathrm{per}}_{128}(h(z), g(z))$

Figure 7.16: Run times of randomly generated rule trees for: (a) FIR filter transform of size 128 and 17 filter taps on P4C-3.2, and (b) Daubechies 9-7 DWT of size 64 on Xeon-1.7.

implementing FIR filters and DWTs. We choose a filter transform of size 128 with 16 filter coefficients, switch on all available rules, and randomly generate around 700 rule trees. We compile and measure all rule trees and create a histogram of obtained run times shown on the left side of Figure 7.16. We note that there is a difference of almost an order of magnitude between the slowest and the fastest rule tree, which implies that the search can improve the speed of the execution by several times. As a side issue, we mention that the random rule tree generator does not generate a uniform distribution and favors simple rule trees, such as the base case rule (5.49). Since for this small and short filter transforms base case rule provides already reasonable run time, the whole histogram is skewed slightly towards faster rule trees. The same experiment is performed for the periodic DWT of size 128 with Daubechies 9/7 wavelet. Graph 7.16(b) shows the run time spread histogram for about 300 randomly generated rule trees. In this case, the performance spread is even more dramatic than for FIR filters. The slowest and the fastest rule tree have run times that are different by more than 30 times.

CHAPTER 8

CONCLUSION

Automatic code generation and tuning is one of the most exciting and challenging problems in the high-performance computing community. It frees software developers from tedious and time-consuming tasks of tuning the software implementations to a specific target platform. Automatic tuning creates portable numerical libraries that achieve high performance across most of the current hardware platforms and, with little additional effort, across many future architectures.

The SPIRAL system provides a suitable framework for automatic generation and tuning of implementations for digital signal processing (DSP) transforms. In Chapter 2, we discussed many features and modules of SPIRAL, including the generator of formulas representing the algorithms, formula compiler, evaluation module, and search engine. Together, these modules enable automatic generation of algorithms, translation into code, and optimization of implementations for the target computer platform by search.

To enable SPIRAL to generate tuned filtering and wavelet kernels, we designed a new framework to represent FIR filters and discrete wavelet transforms (DWTs) in SPIRAL and to formulate the existing fast algorithms using the rule formalism. We designed the breakdown rules to allow the formula generator to span the space of algorithms that we believe are good candidates for the fastest run times on most general purpose computer platforms.

We expressed most of the known algorithms in a unique form using the mathematical language we developed in Chapters 4, 5, and 6. We design more than 40 rules to span the entire space of possible candidates for the fastest algorithm. For example, rules (5.56) and (5.57) defined on page 106 in combination with rule (5.65) on page 109 provide important block convolution methods known in the literature as the overlap-save and the overlap-add methods. Unlike the explanations and formulas used in the literature (e.g., [38]), the rules clearly demonstrate that the two methods are each other's transposes and that they block the computations for input and output locality. Further, they exactly define the operations that need to be performed and capture the structural information of the computations that the SPL compiler can use to, for example, determine the size of the loops and how to loop the computations.

We implemented the transforms and the rules in the SPIRAL's formula generator, paying special attention to optimizing the rules so that they can be properly handled by the formula optimization block (see Figure 2.1 on page 14). One such optimization is fusion of the downsampling operator shown in equation (6.39). Next, we verified the rules for mathematical correctness using the SPIRAL's verification and debugging tools (see Section 2.4). When necessary, we designed templates

for translating the mathematical formulas into the programming language and also verified whether the formulas produced the desired code.

For some rules, such as the blocking rules from Section 5.2.7, we adjusted the storing methods and the search strategies to enable more efficient optimization and faster convergence. Further speedup of search was accomplished by restricting the applicability of rules to transforms for which the rule is truly efficient. For example, Filter Circulant rule (5.65) was restricted to transform sizes n and filter lengths k whose ratio n/k and k/n is below certain threshold. In the first case, the rule would lead to circulants that are too sparse, whereas in the latter case, to a large overhead.

By accomplishing the above tasks, we enabled SPIRAL to generate code for FIR filters, including code for linear and circular convolutions, and filter banks, as well as code for a class of orthogonal and biorthogonal discrete wavelet transforms for different signal models. We next highlight the contributions of the work presented in this thesis.

8.1 Major Contributions

By enabling automatic generation of platform-adapted implementations for FIR filters and the DWTs using SPIRAL, we have considerably closed the gap between: 1) existing software libraries for numerical implementation of digital filters and DWTs that provide portability but limited performance, and 2) hand-tuned libraries for filtering and wavelet kernels that achieve high performance for a small set of chosen platforms usually provided by the hardware vendor.

We summarize our achievements in this thesis.

1. We **automatically generate** C or Fortran code for
   - linear convolution of FIR filters and compact support signals.
   - linear convolution of FIR filters and infinite or infinitely extended signals on finite number of output points.
   - circular convolution or any type of generalized convolution, such as symmetric convolution.
   - single-stage and multi-stage DWT for
     (a) finite support signals,
     (b) infinite and infinitely extended signals, and
     (c) periodically extended signals.
   - DWTs for any type of orthogonal and biorthogonal wavelets.
2. Our automatically generated code is obtained by searching in a comprehensive space of algorithms and implementation choices in order to achieve high performance. Benchmark tests show that our code is **competitive**, and often **outperforms hand-coded libraries** developed by teams of programmers and algorithm and architecture experts. The impact of such automated high-performance code generator system on the development of numerical software libraries is substantial. The **cost** of developing numerical libraries can be **substantially reduced** by deploying our system on current and future computer platforms and avoid the

continuous cycle of coding, testing, and debugging for each and every newly developed architecture.

3. Our system **automatically implements most well-known algorithms** for FIR filtering including:

   - overlap-save and overlap-add methods,
   - transform-domain methods, including the complex and real DFT, and the discrete Hartley transform methods,
   - radix-n divide-and-conquer or Karatsuba recursive methods,
   - blocking and nesting techniques for cache and register locality,

   and algorithms for DWTs including:

   - efficient direct implementation using Mallat recursions,
   - polyphase implementation of the filter bank,
   - lifting scheme algorithms for all possible factoring schemes,
   - lattice factoring algorithms.

   Code is generated and available to the user for "free" enabling fast implementation and testing of most available methods.

4. Furthermore, the combination of all of the above algorithms using recursive application of breakdown rules enables **new algorithms** to be automatically developed and tested. Search in the comprehensive space of implementations often finds solutions that are hardly ever considered by a human programmer.

By developing a suitable framework for representing FIR filtering and DWT algorithms and integrating this framework into SPIRAL, we enabled the SPIRAL system to automatically generate, translate and search for most efficient implementations on the platform of choice. We believe this provides the *first* truly automatic performance tuning system for filtering and wavelet numerical kernels. Our system generates code that is optimized for the target platform by considering a vast space of alternative solutions. The end result is that our system achieves very high performance comparable with hand-optimized numerical libraries for FIR filters and DWTs.

8.2 Limitations of the Current Framework

We enabled SPIRAL to generate code for most of the well-known methods for implementing filtering and wavelet algorithms. However, we point out to several limitations of the current SPIRAL framework, as well as limitations in scope of our FIR filter and DWT framework.

The mathematical framework and the rule formalism we developed for FIR filters and the DWTs is sufficient to represent most of the existing algorithms for one-dimensional convolution operators and discrete wavelet transforms.

Wavelet packets. In addition to the presented material, with a slight modification of the rules introduced in Chapter 6, our framework is able to generate code for arbitrary wavelet packets. For

example, Mallat rule (6.37) can be modified so that it expands the high-pass branch of the wavelet tree shown in Figure 3.3.

$$\mathbf{DWT}^{\mathrm{per}}_{n,j}(h(z),g(z)) \rightarrow \left(\mathrm{I}_n \oplus \mathbf{DWT}^{\mathrm{per}}_{n,j-1}(h(z),g(z))\right) \cdot \mathbf{DWT}^{\mathrm{per}}_{n,1}(h(z),g(z)) \tag{8.1}$$

Combining these two rules, we can expand the tree into any possible wavelet packet tree.

Multi-dimensional transforms. Even though we provide rules for generation of algorithms for 1-D transforms and filters, a simple extension of the rules as

$$\begin{aligned} \mathbf{T}^{1-D}_n &\rightarrow \mathcal{R}_{type}\left\{\mathbf{T}'_r,\ldots,\mathbf{T}^{(n)}_t\right\} \\ \mathbf{T}^{2-D}_n &\rightarrow \mathcal{R}_{type}\left\{\mathbf{T}'_r,\ldots,\mathbf{T}^{(n)}_t\right\} \otimes \mathcal{R}_{type}\left\{\mathbf{T}'_r,\ldots,\mathbf{T}^{(n)}_t\right\} \end{aligned} \tag{8.2}$$

will generate code for 2-D or any n-dimensional convolution and DWT as long as the filters or the wavelet bases are separable [16, 109]. This approach is only one of various implementation strategies for 2-D transforms, and often suboptimal. Additional research effort is required to allow automatic optimization of code for 2-D filters and DWTs. Furthermore, the current framework does not support non-separable multi-dimensional transforms.

M-channel filter banks. Our framework allows generation of code for 2-band wavelet transforms. The generalization to M-band DWTs is straightforward for some of the rules such as the Mallat rules, but requires considerably more effort for the lifting method and adaptation of the formula optimization rules.

Special instruction sets. In Chapter 2, we briefly discussed that SPIRAL generates implementations that support vector instruction (SIMD) and fused multiply-add (FMA) instruction extensions for a few trigonometric transforms. Since the filtering and wavelet kernels exhibit different algorithm structure, additional research is required to fully automate the generation of SIMD and FMA code. Because of the regularity of the direct implementation for convolutions and filter banks, it is conceivable that the special instruction sets can be efficiently utilized, making the more sophisticated techniques, such as divide-and-conquer methods less relevant.

8.3 Future Work

Based on the work in this thesis, future research can proceed in several directions. We suggested in the previous section that there are several limitations of the current framework. We believe that a significant improvement in both the quality of implementation and the scope of the possible applications can be achieved by addressing these issues. We list some of these issues in their order of importance.

1. **Code vectorization.** Since a growing number of computer platforms supports SIMD instructions, a considerable speedup can be obtained by designing the framework that will support vector code for filtering and wavelet kernels. Vendor libraries, such as Intel's IPP, suggest that the speedup is close to the theoretical peak improvement for the particular instruction set.

2. **2-D convolutions and wavelet transforms.** 2-D convolutions and DWTs are extensively used in image processing applications, such as the JPEG2000 image coding standard [17]. The problem of tuning the implementations for 2-D transforms offers many additional opportunities for instruction parallelism and pipelining. For example, see [27, 28].

3. **Wavelet packets.** With little effort, the current framework can be extended in the direction of generating code for arbitrary wavelet packets. The system can be then used to generate efficient implementations by searching over both the fastest implementations and the best wavelet packet basis using any criterion (e.g., [83]) by including it in the cost function of the SPIRAL's optimization problem [13].

4. **Other generalizations of the current framework.** The current framework can be further extended to generate implementations for M-channel filter banks and wavelet systems, complex wavelets, second-generation wavelets, overcomplete wavelet representations (frames), and directional wavelets [110].

APPENDIX A

LIFTING SCHEME FACTORIZATIONS

The lifting scheme (LS) factorization of polyphase matrices wit the paraunitary property was introduced in Chapter 3 on page 64. The foundation of the LS is established in the common Euclidean algorithm (EA) for Laurent polynomials. Since the division of Laurent polynomials allows a certain degree of freedom, different factorization schemes can be derived for the same polyphase matrix. We next explore these degrees of freedom.

A.1 Euclidean Algorithm

Consider two polynomials $a(z)$ and $b(z)$ in $\mathbb{R}[z]$ with degrees $\deg(a) = n$ and $\deg(b) = m$, respectively. The division with a remainder can always be performed as

$$a(z) = q(z) \cdot b(z) + r(z) \tag{A.1}$$

where q(z) is the *quotient* and $r(z)$ is the *remainder*. As long as the degree of the remainder is $\deg(r) < m$, the division is unique. The division can continue recursively until it is performed without the remainder to find the greatest common divisor (gcd) of $a(z)$ and $b(z)$. The algorithm proceeds as follows

$$\begin{aligned}
&1. \quad a(z) = q_1(z) \cdot b(z) + r_1(z) \quad \rightarrow \quad \begin{bmatrix} a(z) \\ b(z) \end{bmatrix} = \begin{bmatrix} 1 & q_1(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r_1(z) \\ b(z) \end{bmatrix} \\
&2. \quad b(z) = q_2(z) \cdot r_1(z) + r_2(z) \quad \rightarrow \quad \begin{bmatrix} r_1(z) \\ b(z) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ q_2(z) & 1 \end{bmatrix} \begin{bmatrix} r_1(z) \\ r_2(z) \end{bmatrix} \\
&3. \quad r_1(z) = q_3(z) \cdot r_2(z) + r_3(z) \quad \rightarrow \quad \begin{bmatrix} r_1(z) \\ r_2(z) \end{bmatrix} = \begin{bmatrix} 1 & q_3(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} r_3(z) \\ r_2(z) \end{bmatrix} \\
&\qquad\qquad \vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&M. \quad r_{M-2}(z) = q_M(z) \cdot r_{M-1}(z) + 0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad r_{M-1} = \gcd(a(z), g(z)) \quad \rightarrow \quad \begin{bmatrix} a(z) \\ b(z) \end{bmatrix} = \begin{bmatrix} 1 & q_1(z) \\ 0 & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & 0 \\ q_M(z) & 1 \end{bmatrix} \begin{bmatrix} r_{M-1}(z) \\ 0 \end{bmatrix}
\end{aligned} \tag{A.2}$$

The Euclidean algorithm described above can be described in the matrix format as shown on the right side of (A.2). Each of the triangular polynomial matrices is one lifting step matrix. Since the polyphase matrix is paraunitary, the synthesis filters are not independent of the analysis filter s and

a slight modification of the Euclidean algorithm gives the lifting scheme (3.97) with $M+1$ lifting steps.

A.2 Division of Laurent polynomials

However, since the filters are represented by Laurent polynomials $h(z) = \sum_{k=a}^{b} h_k z^{-k} \in \mathbb{R}[z, z^{-1}]$ whose degree is defined as $\deg(h) = |b-a|$, the division in (A.1) is not unique. Consider the following example.

EXAMPLE A.1. *Consider two polynomials*

$$a(z) = -\tfrac{1}{8}z^{-1} + \tfrac{3}{4} - \tfrac{1}{8}z, \quad b(z) = \tfrac{1}{4} + \tfrac{1}{4}z.$$

There are precisely three different ways perform the division (A.1) *in this case*

$$a(z) = \left(-\tfrac{1}{2}z^{-1} + \tfrac{7}{2}\right)\left(\tfrac{1}{4} + \tfrac{1}{4}z\right) - z$$

$$a(z) = \left(-\tfrac{1}{2}z^{-1} - \tfrac{1}{2}\right)\left(\tfrac{1}{4} + \tfrac{1}{4}z\right) + 1$$

$$a(z) = \left(\tfrac{7}{2}z^{-1} - \tfrac{1}{2}\right)\left(\tfrac{1}{4} + \tfrac{1}{4}z\right) - z^{-1}$$

For standard polynomials the division proceeds by reducing the degree of the dividing polynomial at every stage by matching the highest degree with the highest degree of the divisor. With Laurent polynomials the degree reduction can occur by matching either the lowest or the highest degree at each step of the division. The upper limit on the number of possible divisions for two Laurent polynomials is established by the following lemma.

Lemma A.1 (Upper bound on the number of possible Laurent polynomial divisions). *Let $\mathbb{R}[z, z^{-1}]$ be the ring of Laurent polynomials and let $a(z), b(z) \in \mathbb{R}[z, z^{-1}]$ be two polynomials with degrees $\deg(a) = n$ and $\deg(b) = m$, where $n \geq m$. The number of different divisions with the remainder K is bounded by*

$$1 \leq K \leq n - m + 2 \tag{A.3}$$

up to a scaled monomial Cz^k.

Proof. The division with the remainder gives $a(z) = q(z) \cdot b(z) + r(z)$, where $q(z)$ is the quotient and $r(z)$ is the remainder with the minimum degree. The remainder degree is always less than m since, otherwise, we can write

$$r(z) = q_1(z) \cdot b(z) + r_1(z) \Rightarrow a(z) = (q(z) + q_1(z)) \cdot b(z) + r_1(z)$$

and $r_1(z)$ would have a lower degree. It follows directly that $\deg(bq) = n$ and that $\deg(q) = n - m$ since $\deg(bq) = \deg(b)\deg(q)$ is satisfied for Laurent polynomials. Since $r(z) = a(z) - b(z)q(z)$ it is clear that $b(z) \cdot q(z)$ has to annihilate $n - \deg(r)$ outer terms in $a(z)$. Unlike the standard polynomials where only the highest degrees are cancelled, for Laurent polynomials there is a choice between annihilating either the highest or the lowest degree terms, including all their combinations.

Let us choose the maximum degree $\deg(r) = m - 1$. In this case the number of required annihilated terms is $s = n - (m-1)$. In general, we can choose to cancel $s - t$ highest degree terms

and t lowest degree terms, where $0 \leq t \leq s$. There are $s+1$ different choices. To see that this is the upper bound, let us consider two cases

(i) $\deg r = m - 1$

Since the number of choices for cancelling the terms is $n - m + 2$ in this case, it is also the maximum number of different divisions assuming that we annihilate only one term at a time.

(ii) $\deg r < m - 1$

The long division proceeds as follows

$$\begin{aligned} &1. \quad a(z) = q_1(z) \cdot b(z) + r_1(z) && \deg(r_1) < n \\ &2. \quad r_1(z) = q_2(z) \cdot b(z) + r_2(z) && \deg(r_2) < \deg(r_1) \\ &\qquad\qquad \vdots \end{aligned}$$

until $\deg r_k < m$. Then $r(z) = r_k(z)$ and $q(z) = \sum_i q_i(z)$. Since $\deg(r_{k-1}) \geq m$ there can be at most $n-m+1$ steps in the division and exactly that many independent terms to annihilate, which brings total to $n-m+2$ choices using the same argument as for the $\deg r = m-1$ case. The difference here is that when some of the $n - m + 1$ independent terms are canceled the dependent terms also get canceled, but the maximum cannot exceed $n + m - 1$.

■

In the Example A.1 the number of different divisions meets the upper bound of $n - m + 2 = 2 - 1 + 2 = 3$.

A.3 Lifting scheme factorizations

Lemma A.2 (Upper bound on the number of lifting steps). *Given the polynomials $a(z)$ and $b(z)$ of degrees n and m, the Euclidean algorithm terminates in at most $m + 1$ steps and the maximum number of lifting steps S is bounded by $m + 2$, i.e.,*

$$M \leq \deg(b) + 1, \quad S \leq \deg(b) + 2$$

Proof. At each step of the Euclidean algorithm in (A.2) the degree of the remainder is reduced by at least one since $\deg(r_{i-1}) < \deg(r_i)$. Since the algorithm starts with $r_1(z)$ of degree at most $m-1$ and terminates when $r_M(z) = 0$, meaning that the degree of r_{M-1} has to be at least equal to zero, the number of the divisions can be at most $m+1$ with the equality when the degree of the remainder is reduced by one at each step. The number of lifting steps is then bounded by $m + 2$ according to (A.2). ■

We can now combine the two lemmas to find the upper bound on the number of possible lifting steps factorizations.

Lemma A.3 (Upper bound on the number of factorization schemes). *Given the Laurent polynomials $a(z)$ and $b(z)$ with degrees n and m, respectively, where $n > m$, the number of different lifting schemes or the ways to perform the Euclidean algorithm is bounded from the above as*

$$L \leq (n - m + 2) \cdot 3^{M-2} \tag{A.4}$$

where $M = m + 1$ is the maximum number of lifting steps.

Proof. (i) Let us first assume that at each division in the Euclidean algorithm the degree of the remainder is reduced by one, i.e., $\deg(r_{i+1}) = \deg(r_i) - 1$ starting from $r_1(z)$ with the maximum degree $m - 1$. According to Lemma A.2 the number of lifting steps is exactly $m + 1$. At each step of the EA we have a choice of how to do the division of Laurent polynomials bounded by $K \leq n - m + 2$ as suggested by Lemma A.1.

For the first division shown in the first equation of (A.2) the number of possible divisions is $N_1 \leq n - m + 2$. Since $\deg(r_1) = m - 1$, in the second step the number of possible divisions of $b(z)$ and $r_1(z)$ is $N_2 \leq m - \deg(r_1) + 2 = 3$. By assumption, the degree of each subsequent remainder is reduced by one so that the number of divisions stays bounded by $N_i \leq 2-1+2 = 3$. The last step $M = m - 1$ in the EA is trivial since $r_M(z)$ is a monomial and there is only one choice on how to perform it. When we sum the choices in all steps we obtain

$$L = \prod_{i=1}^{M} N_i \leq (n - m + 2) \cdot \underbrace{3 \cdot 3 \cdots 3}_{M-2} \cdot 1 = (n - m + 2) \cdot 3^{M-2} \tag{A.5}$$

(ii) Let us now show that this is also the upper bound if we drop the assumption of reducing the degree of the remainder by one at each step of the EA. Assume that for some step of the EA the degree is reduced by $t+1$, i.e., $\deg(r_{k+1}) = \deg(r_k) - 1 - t$. According to Lemma A.1, in the next step $r_{k-1}(z) = r_k(z)q_{k+1}(z) + r_{k-1}(z)$ the maximum number of divisions is $N_{k+1} \leq t + 3$. The algorithm can now terminate in at most $m + 1 - t$ steps leading to the upper bound of

$$L \leq (n - m + 2) \cdot \underbrace{3 \cdot 3 \cdots 3}_{M-2-t} \cdot (3 + t) < (n - m + 2) \cdot 3^{M-2} \tag{A.6}$$

and the upper bound is lower than in (A.5). Therefore, the number of choices cannot be increased by reducing the degree of the remainders by more than one, so the case (i) provides the global upper bound.

■

The number of lifting schemes grows exponentially with the degree of the polynomials. Different lifting schemes have different properties. For example, the number of lifting schemes for Daubechies 9-7 wavelets meets the upper bound of 27 different factorizations since $\deg(h_e) = 4$, $\deg(h_o) = 3$ so $L \leq (4 - 3 + 1) \cdot 3^2 = 27$. Since the filters have linear phase, it is advantageous to preserve this property for the lifting steps. However, this is possible using only one out of 27 different schemes, which is incidentally the one used in the JPEG2000 standard. Different lifting schemes are especially important for wavelet transforms mapping integers to integers [111].

For the polyphase matrices representing wavelet systems, the EA can proceed either with the even and odd lowpass or the highpass filters, as we explained in Section 3.2.5. For example, we can employ the EA on lowpass $h_e(z)$ and $h_o(z)$ defined as (3.38), where the highpass $g_e(z)$ and $g_o(z)$ can be updated at each step of the division using the relation (3.94). We end this section with an illustrative example

EXAMPLE A.2. *Consider the lowpass and the highpass filters*

$$\begin{aligned} h(z) &= -\tfrac{1}{8}z^{-2} + \tfrac{1}{4}z^{-1} + \tfrac{3}{4} + \tfrac{1}{4}z - \tfrac{1}{8}z^2 \\ g(z) &= \quad \tfrac{1}{4}z^{-2} - \tfrac{1}{2}z^{-1} + \tfrac{1}{4} \end{aligned}$$

Their downsampled versions are

$$\begin{aligned} h_e(z) &= -\tfrac{1}{8}z^{-1} + \tfrac{3}{4} - \tfrac{1}{8}z \quad & h_o(z) &= \tfrac{1}{4} + \tfrac{1}{4}z \\ g_e(z) &= \tfrac{1}{4}z^{-1} + \tfrac{1}{4} & g_o(z) &= -\tfrac{1}{2} \end{aligned}$$

We apply the Euclidean algorithm on $h_e(z)$ and $h_o(z)$. According to Lemma A.2, the maximum number of lifting steps is $S = M + 1 = m + 2 = 3$. Three different division strategies from these two polynomials were given in Example A.1. By choosing the second strategy we obtain the decomposition

$$\begin{bmatrix} h_e(z) & h_o(z) \\ g_e(z) & g_o(z) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{4} + \frac{1}{4}z \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\frac{1}{2} - \frac{1}{2}z^{-1} & 1 \end{bmatrix}$$

with only two lifting steps. This is below the upper bound. However, if we choose the first division in Example A.1 we obtain the following factorization

$$\begin{bmatrix} h_e(z) & h_o(z) \\ g_e(z) & g_o(z) \end{bmatrix} = \begin{bmatrix} -z & 0 \\ 0 & \frac{1}{2}z^{-1} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 4z & 1 \end{bmatrix} \begin{bmatrix} 1 & -\frac{1}{4} - \frac{1}{4}z^{-1} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \frac{7}{2} - \frac{1}{2}z^{-1} & 1 \end{bmatrix}$$

There are exactly three lifting steps in this case which equals the upper bound. The total number of different schemes is bounded by $L \leq (2 - 1 + 2) \cdot 3^{2-2} = 3$. There is another lifting scheme arising from the third division strategy in Example A.1, so this upper bound $L = 3$ is also reached in this example.

APPENDIX B

GENERALIZED KARATSUBA METHODS

In Chapter 3 we introduced divide-and-conquer or Karatsuba methods for decomposing filtering operations into shorter filters to reduce the arithmetic cost. In Section 3.1.6 we presented the case when the filter is split into its even and odd coefficients (3.39) leading to a radix-2 implementation. The method can be generalized in many directions. We already mentioned that even for radix-2 approach different strategies can be used to combine the products of downsampled filters and downsampled signals leading to different radix-2 schemes. For example, the scheme shown in Figure 3.2 we shall refer to as the standard form, where a different scheme can easily be obtained by transposing the graph [57]. Another way to generalize the algorithm is to increase the downsampling factor to obtain higher-radix implementations, and even further by having a different downsampling factor for the input and the output. In this section, we present a scheme to derive higher-radix Karatsuba algorithms in the standard form, and provide the cost analysis for the general case.

B.1 Standard Radix-n Karatsuba Algorithm

Consider two polynomials $h(z)$ and $x(z)$. We are interested in computing the product $y(z) = h(z) \cdot x(z)$ in an efficient way. Let us represent each of the polynomials using downsampled decomposition with the factor n.

$$h(z) = \sum_{i=0}^{n-1} h_i(z^n) z^{-i}, \quad x(z) = \sum_{i=0}^{n-1} x_i(z^n) z^{-i}, \quad y(z) = \sum_{i=0}^{n-1} y_i(z^n) z^{-i} \tag{B.1}$$

where, for example,

$$h_i(z) = \sum_{j=0}^{n-1} h_{nj+i} z^{-j} \tag{B.2}$$

By multiplying the two polynomials $h(z) \cdot x(z)$ in this form and grouping the terms we obtain the following set of equations

$$\begin{aligned}
y_0(z) &= h_0(z)x_0(z) + \left[h_1(z)x_{n-1}(z) + h_2(z)x_{n-2}(z) + \cdots + h_{n-1}(z)x_1(z)\right] z^{-1} \\
y_1(z) &= \left[h_0(z)x_1(z) + h_1(z)x_0(z)\right] + \left[h_2(z)x_{n-1}(z) + \cdots + h_{n-1}(z)x_2(z)\right] z^{-1} \\
&\vdots \qquad\qquad\qquad\qquad \vdots \\
y_n(z) &= h_{n-1}(z)x_0(z) + h_{n-2}(z)x_1(z) + \cdots + h_1(z)x_{n-2}(z) + h_1(z)x_{n-2}(z)
\end{aligned} \tag{B.3}$$

where we also made the substitution $z = z^n$.

The decomposition can be compactly represented in the matrix form

$$\mathbf{y}(z) = \begin{bmatrix} h_0(z) & h_{n-1}(z)z^{-1} & \cdots & h_2(z)z^{-1} & h_1(z)z^{-1} \\ h_1(z) & h_0(z) & \ddots & & h_2(z)z^{-1} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ h_{n-2}(z) & & \ddots & h_0(z) & h_{n-1}(z)z^{-1} \\ h_{n-1}(z) & h_{n-2}(z) & \cdots & h_1(z) & h_0(z) \end{bmatrix} \cdot \mathbf{x}(z) \tag{B.4}$$

In the rest of this presentation we shall drop the argument z to save space, assuming that all terms represent polynomials.

The main idea in the Karatsuba decomposition is to use the direct products of the form $h_i \cdot x_i$ as many times as possible by combining and expressing the cross-products as

$$h_i \cdot x_j + h_j \cdot x_i = (h_i + h_j) \cdot (x_i + x_j) - h_i \cdot x_i - h_j \cdot x_j \tag{B.5}$$

Assuming that we can precompute $h_i + h_j$ and that the direct products $h_i \cdot x_i$ have already been computed in one of other outputs, we reduce the number of multiplications from two to only one $(h_i + h_j) \cdot (x_i + x_j)$ with two new additions.

EXAMPLE B.1. *Let us examine cases when $n = 3$ and $n = 4$. When $n = 3$ we have the following set of equations.*

$$\begin{aligned} y_0 &= h_0x_0 + [h_1x_2 + h_2x_1]\, z^{-1} \\ y_1 &= [h_0x_1 + h_1x_0] + h_2x_2z^{-1} \\ y_2 &= h_2x_0 + h_1x_1 + h_0x_2 \end{aligned} \tag{B.6}$$

Using (B.5) *we can express* (B.6) *as*

$$\begin{aligned} y_0 &= h_0x_0 + [(h_1 + h_2)(x_1 + x_2) - h_1x_1 - h_2x_2]\, z^{-1} \\ y_1 &= [(h_0 + h_1)(x_0 + x_1) - h_0x_0 - h_1x_1] + h_2x_2z^{-1} \\ y_2 &= h_1x_1 + (h_0 + h_2)(x_0 + x_2) - h_0x_0 - h_2x_2 \end{aligned} \tag{B.7}$$

which in the matrix form yields

$$\mathbf{y}(z) = \begin{bmatrix} 1 & z^{-1} & z^{-1} & 0 & 0 & 1 \\ 1 & 1 & z^{-1} & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix} \operatorname{diag}\{h_0, h_1, h_2, h_0{+}h_1, h_0{+}h_2, h_1{+}h_2\} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \cdot \mathbf{x}(z) \tag{B.8}$$

Instead of 9 polynomial multiplications, 6 additions, and two shifts, radix-3 Karatsuba decomposition requires 6 multiplications, 12 additions, and 3 shifts.

In the case $n = 4$ we have the following equations

$$\begin{aligned} y_0 &= h_0x_0 + [h_1x_3 + +h_2x_2 + h_3x_1]\, z^{-1} \\ y_1 &= [h_0x_1 + h_1x_0] + [h_2x_3 + h_3x_2]\, z^{-1} \\ y_2 &= [h_2x_0 + h_1x_1 + h_0x_2] + h_3x_3z^{-1} \\ y_4 &= h_3x_0 + h_2x_1 + h_1x_2 + h_0x_3 \end{aligned} \tag{B.9}$$

that can be expressed using the cross product decomposition

$$\begin{aligned} y_0 &= h_0x_0 + [h_2x_2 + (h_1+h_3)(x_1+x_3) - h_1x_1 - h_3x_3]\, z^{-1} \\ y_1 &= [(h_0+h_1)(x_0+x_1) - h_0x_0 - h_1x_1] + [(h_2+h_3)(x_2+x_3) - h_2x_2 - h_3x_3]\, z^{-1} \\ y_2 &= [h_1x_1 + (h_0+h_2)(x_0+x_2) - h_0x_0 - h_2x_2] + h_3x_3z^{-1} \\ y_4 &= (h_0+h_3)(x_0+x_3) - h_0x_0 - h_3x_3 + (h_1+h_2)(x_1+x_2) - h_1x_1 - h_2x_2 \end{aligned} \tag{B.10}$$

The original 16 multiplications and 12 additions are reduced to 10 multiplications and 24 additions.

B.2 Radix-n Karatsuba Cost Analysis

Let the input sequence $x(z)$ be of length $N = n^r$ and the filter $h(z)$ of length $L = n^s$. The number of operations per output point for a straightforward multiplication is L multiplications and $L-1$ additions. Radix-n Karatsuba method break down the problem into multiplications of polynomials $h_i(z)$ and $x_i(z)$ of size n^{s-1} and n^{r-1}, respectively.

According to (B.3) and (B.5) and from Example B.1 we can see that the computations are broken down into three stages: input additions, multiplication with downsampled polynomials, and output additions. The number of direct products $h_i(z)x_i(z)$ is n where the number of all possible products of pairs $(h_i+h_j)(x_i+x_j)$ is $n(n-1)/2$.

1. **Input additions**. There are precisely $n(n-1)/2$ input additions of the form $x_i(z)+x_j(z)$. Since each $x_i(z)$ is N/n long, there are $N(n-1)/2$ additions in total, or

   $$C_a^i(n) = \frac{1}{2}(n-1)$$

   per output point.

2. **Output additions**. From (B.3), if n is odd there is precisely $(n-1)/2$ cross product pairs of the form $h_i \cdot x_j + h_j \cdot x_i$ that can be decomposed using (B.5). In the case n is even, the equations in (B.3) have $n/2-1$ cross-product pairs if the output index in y_i is even and $n/2$ pairs if the index is odd, making the average number of cross-product pairs again $(n-1)/2$.

   Since the decomposition in (B.5) introduces one new addition per cross-product pair, there are $(n-1)/2$ new adds in addition to the previous $n-1$ yielding a total of

   $$C_a^o(n) = \frac{3}{2}(n-1)$$

3. **Polynomial products**. There are $n(n+1)/2$ total polynomial products of downsampled polynomials. The number of required total multiplications and additions is therefore

   $$C_m^t(n,N,L) = \frac{n(n+1)}{2} \cdot \frac{N}{n} \cdot \frac{L}{n}, \qquad C_a^t(n,N,L) = \frac{n(n+1)}{2} \cdot \frac{N}{n} \cdot \left(\frac{L}{n} - 1\right)$$

   or

   $$C_m(n,L) = \frac{n+1}{2n}L, \qquad C_a^p(n,L) = \frac{n+1}{2} \cdot \left(\frac{L}{n} - 1\right)$$

   per output point

We can now summarize the total number of multiplications and additions per output point for the radix-n Karatsuba algorithm

$$\begin{aligned} C_a(n,L) &= \underbrace{2(n-1)}_{i/o} + \underbrace{\frac{n+1}{2}\cdot\left(\frac{L}{n}-1\right)}_{filters} \quad \textbf{adds} \\ C_m(n,L) &= \underbrace{\frac{n+1}{2n}L}_{filters} \qquad\qquad\qquad\qquad\quad \textbf{mults} \end{aligned} \tag{B.11}$$

EXAMPLE B.2. *In the case $n = 2$ the costs are*

$$\begin{aligned} C_a(2,L) &= 2 + \frac{3}{2}\left(\frac{L}{2}-1\right) \\ C_m(2,L) &= \frac{3}{4}L \end{aligned}$$

When the filter is very short $L = 2$ then the total cost is $C(2,2) = 3.5$ operations per point whereas the direct implementation requires only 3 operations per point. Hence, Karatsuba method should be used only when $L > 2$.

Furthermore, for any radix n it is never advantageous to use the Karatsuba algorithm when $L = n$ since in that case

$$\begin{aligned} C_a(n,n) &= 2(n-1) \\ C_m(n,n) &= \frac{n+1}{2} \end{aligned}$$

and the total cost is greater than $2n-1$ for the direct multiplication.

Of course, the radix-n Karatsuba method can be applied recursively. The number of additions and multiplications in that case is computed by solving the recursive equations

$$\begin{aligned} C_a(n,L) &= 2(n-1) + \frac{n(n+1)}{2}\cdot C_a(n,L/n) \\ C_m(n,L) &= \frac{n(n+1)}{2}\cdot C_m(n,L/n) \end{aligned} \tag{B.12}$$

For K levels of recursion we obtain

$$\begin{aligned} C_a^{(K)}(n,L) &= \sum_{k=1}^{K}\left(\frac{n(n+1)}{2}\right)^{k-1}\frac{2(n-1)}{n^{k-1}} + \left(\frac{n(n+1)}{2}\right)^{K}\cdot C_a(n,L/n^k) \\ C_m^{(K)}(n,L) &= \left(\frac{n(n+1)}{2}\right)^{K}\cdot C_m(n,L/n^K) \end{aligned} \tag{B.13}$$

If the filters at the K-th level are computed using the direct multiplications then the costs become

$$\begin{aligned} C_a^{(K)}(n,L) &= \left(\frac{n+1}{2}\right)^{K}\left(\frac{L}{n^K}-1\right) + 4\cdot\left[\left(\frac{n+1}{2}\right)^{K}-1\right] \\ C_m^{(K)}(n,L) &= \left(\frac{n+1}{2}\right)^{K}\left(\frac{L}{n^K}\right) \end{aligned} \tag{B.14}$$

Since the cost obviously decreases with every step of the recursion it is advantageous to proceed with the full recursion down to the point where the length of the downsampled filters is $L/n^K = n$,

as we discussed in Example B.2. In that case, the costs are minimized:

$$\begin{aligned} C_a^{(s-1)}(n,L) &= (n-1)\left(\frac{n+1}{2}\right)^{s-1} + 4\cdot\left[\left(\frac{n+1}{2}\right)^{s-1} - 1\right] \\ C_m^{(s-1)}(n,L) &= n\left(\frac{n+1}{2}\right)^{s-1}, \end{aligned} \tag{B.15}$$

where $L = n^s$.

For radix-2 algorithm the cost of total recursion is then

$$\begin{aligned} C_a^{(s-1)}(2,n^s) &= \left(\frac{3}{2}\right)^{s-1} + 4\cdot\left[\left(\frac{3}{2}\right)^{s-1} - 1\right] \\ C_m^{(s-1)}(2,n^s) &= 2\left(\frac{3}{2}\right)^{s-1}, \end{aligned}$$

with the total of

$$C^{(s-1)}(2,n^s) = 7\left(\frac{3}{2}\right)^{s-1} - 4$$

operations (adds and mults) as opposed to $2\cdot 2^s - 1$ for the direct multiplication.

APPENDIX C

RESULTS OF FIR FILTER TRANSFORM AND DWT EXPERIMENTS

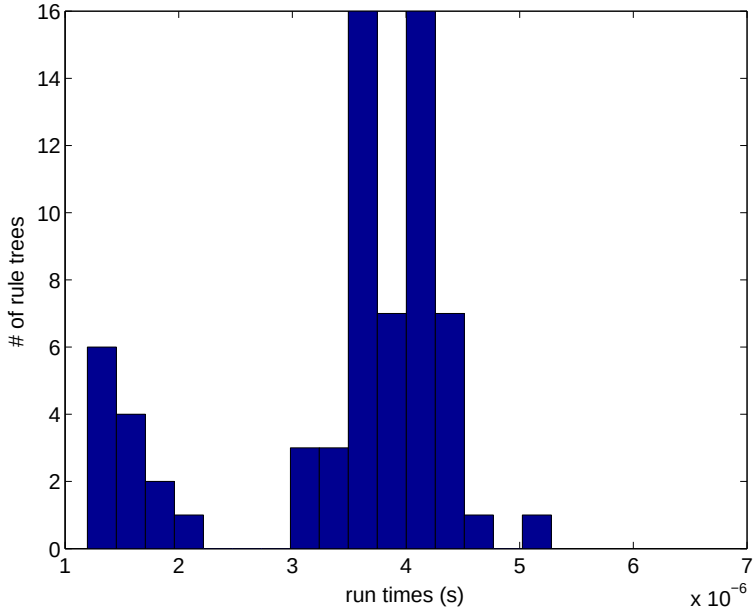
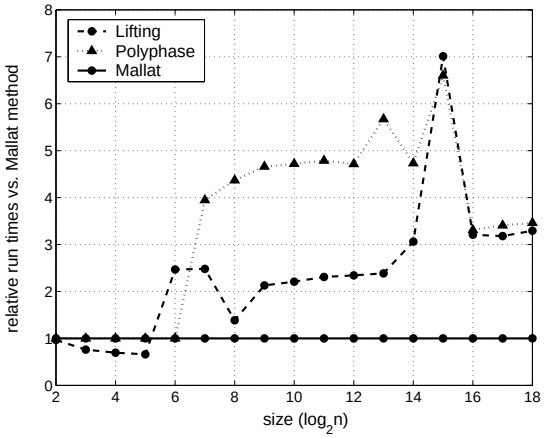
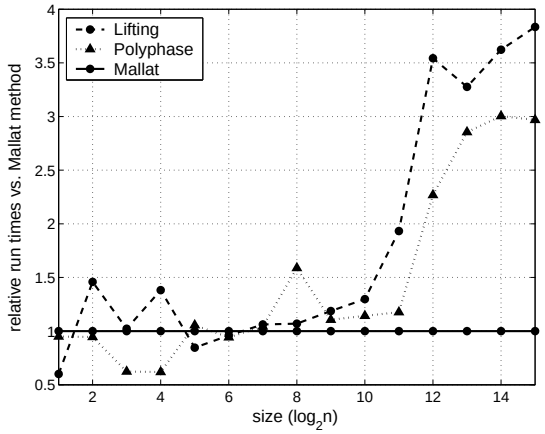


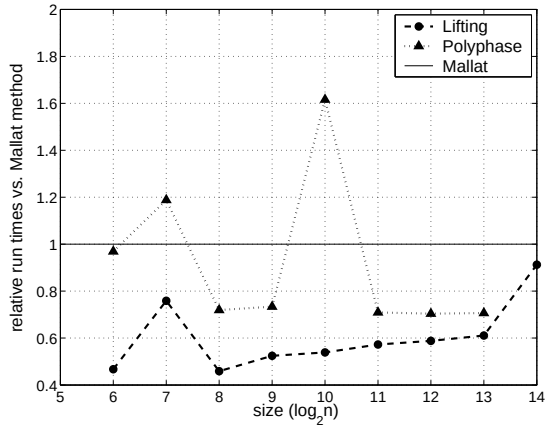
Figure C.1: Runtime comparison of all lifting scheme factorizations for Daubechies 9/7 wavelet transform



(a) rational 5/3

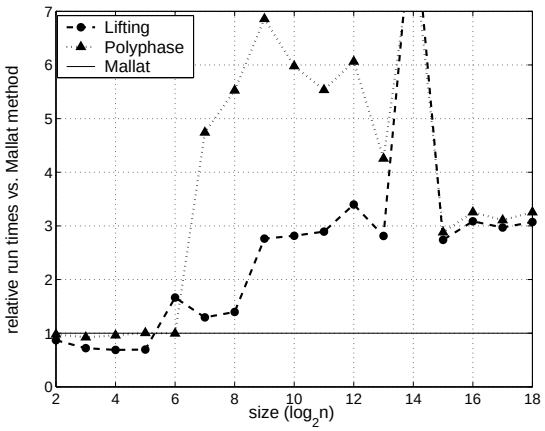


(b) Daubechies 9/7

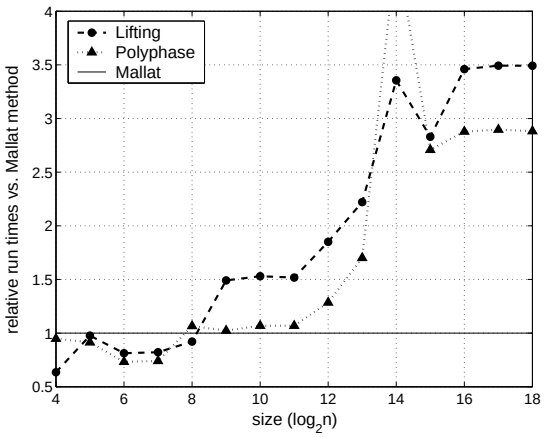


(c) Daubechies 30

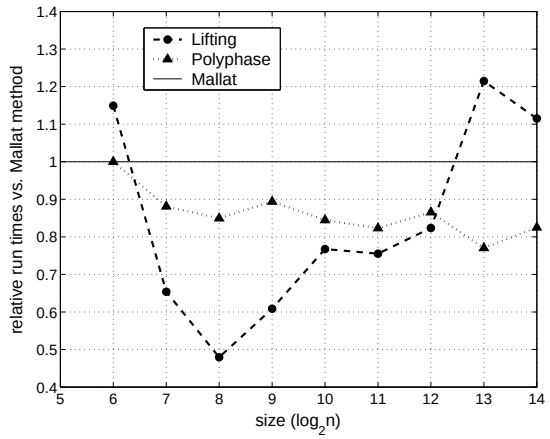
Figure C.2: Comparison of Mallat, Lifting and Polyphase rules on P4B-3.0-win.



(a) rational 5/3

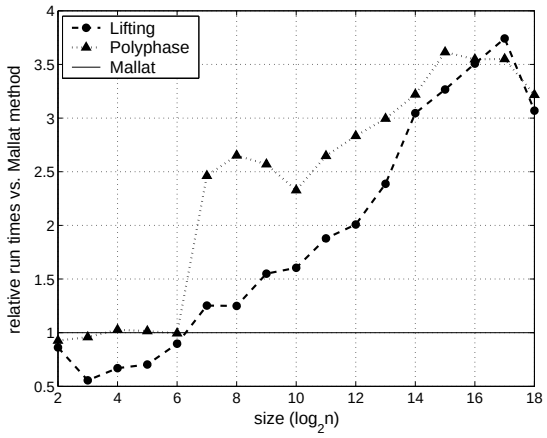


(b) Daubechies 9/7

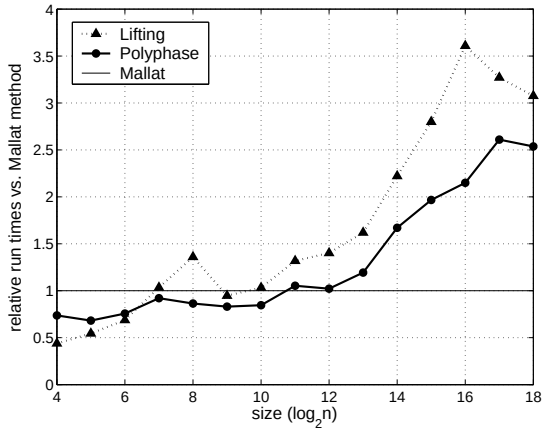


(c) Daubechies 30

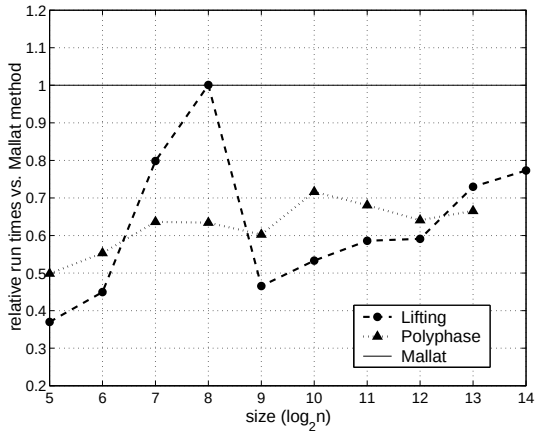
Figure C.3: Comparison of Mallat, Lifting and Polyphase rules on Athlon-1.73.



(a) rational 5/3

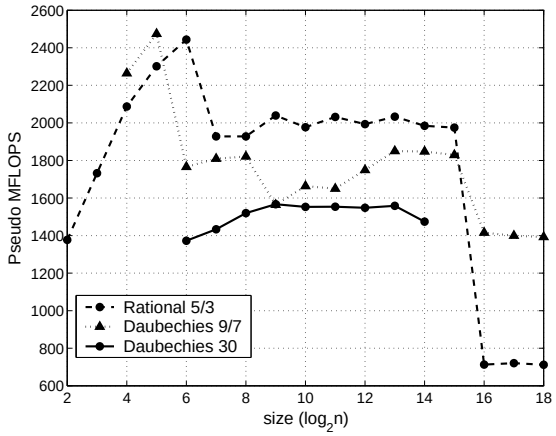


(b) Daubechies 9/7

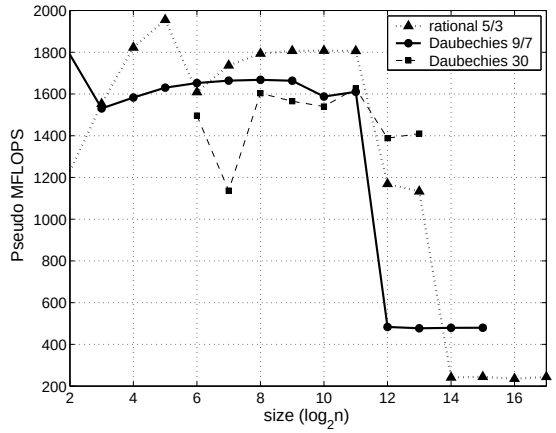


(c) Daubechies 30

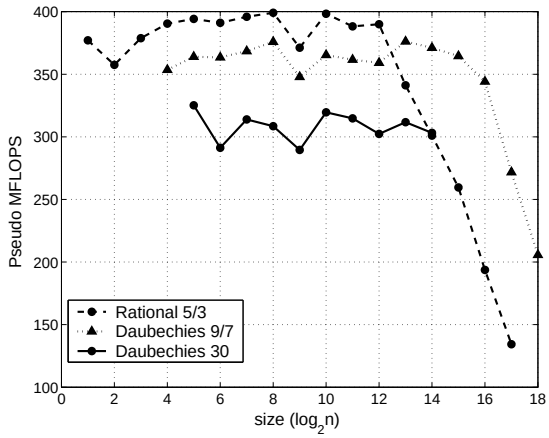
Figure C.4: Comparison of Mallat, Lifting and Polyphase rules on Macintosh.



(a) P4B-3.0-win



(b) Athlon-1.73



(c) Macintosh

Figure C.5: Performance of DWT algorithms on different platforms.

BIBLIOGRAPHY

[1] R. C. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software (ATLAS)," in *Proc. Supercomputing*, 1998. `math-atlas.sourceforge.net`.

[2] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 . `www.netlib.org/lapack/lawns/lawn147.ps`.

[3] J. Bilmes, K. Asanović, C. Chin, and J. Demmel, "Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology," in *Proc. Supercomputing*, ACM SIGARC, 1997. `www.icsi.berkeley.edu/~bilmes/phipac`.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, third ed., 1999.

[5] E.-J. Im and K. Yelick, "Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY," in *Proc.Int. Conf. on Computational Science*, pp. 127–136, 2001.

[6] R. Vuduc, J. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance optimizations and bounds for sparse matrix-vector multiply," in *Proc. Supercomputing*, (MD, USA), November 2002.

[7] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick, "Self adapting linear algebra algorithms and software," *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation*, 2005. (*t*o appear).

[8] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Int. Conf. Acoustics, Speech, and Signal Processing*, vol. 3, pp. 1381–1384, 1998. `www.fftw.org`.

[9] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE Special Issue on Program Generation, Optimization, and Adaptation*, vol. 93, no. 2, 2005.

[10] D. Mirković and S. L. Johnsson, "Automatic Performance Tuning in the UHFFT Library," in *Proc. ICCS*, LNCS 2073, pp. 71–80, Springer, 2001.

[11] J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. Prasanna, M. Püschel, and M. M. Veloso, " SPIRAL: Portable Library of Optimized Signal Processing Algorithms," 1998. `www.spiral.net`.

[12] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 21–45, 2004.

[13] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "Spiral: Code generation for DSP transforms," *Proc. IEEE Special Issue on Program Generation, Optimization, and Adaptation, Invited Paper*, 2005. to appear.

[14] C. S. Burrus, R. A. Gopinath, and H. Guo, *Introduction to Wavelets and Wavelet Transforms: A Primer*. Upper Saddle River, New Jersey: Prentice Hall, 1998.

[15] G. Strang and T. Nguyen, *Wavelets and Filter Banks*. Wesley, 1998.

[16] M. Vetterli and J. Kovačević, *Wavelets and Subband Coding*. Englewood Cliffs: Prentice Hall, 1995.

[17] International Organization for Standardization and International Electrotechnical Commission., *ISO/IEC 15444-1:2000, Information technology - JPEG 2000 image coding system - Part 1: Core coding system*.

[18] "Wavelet toolbox." MathWorks. `http://www.mathworks.com/products/wavelet`.

[19] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *Journal of Fourier Analysis and Applications*, vol. 4, no. 3, pp. 247–269, 1998.

[20] G. Fernández, S. Periaswamy, and W. Sweldens, "LIFTPACK: A software package for wavelet transforms using lifting," in *Wavelet Applications in Signal and Image Processing IV* (M. Unser, A. Aldroubi, and A. F. Laine, eds.), pp. 396–408, Proc. SPIE 2825, 1996.

[21] "Wavelet explorer." Wolfram Research. `http://www.wolfram.com/products/applications/wavelet/`.

[22] J. Buckheit, S. Chen, D. Donoho, I. Johnstone, and J. Scargle, *WaveLab Reference manual*. Technical report, 1995. `http://www-stat.stanford.edu/wavelab`.

[23] "Lastwave." `http://www.cmap.polytechnique.fr/~bacry/LastWave/`.

[24] J. E. Fowler, "QccPack: An open-source software library for quantization, compression, and coding," in *Proc. SPIE 4115* (A. G. Tescher, ed.), pp. 294–301, August 2000.

[25] M. D. Adams, *ISO/IEC 1/SC 29/WG 1, JasPer Software Reference Manual (Version 1.700.0)*. International Organization for Standardization and International Electrotechnical Commission.

[26] A. F. Breitzman, *Automatic Derivation and Implementation of Fast Convolution Algorithms*. PhD thesis, Computer Science, Drexel University, 2003.

[27] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado, "Wavelet transform for large scale image processing on modern microprocessors.," in *VECPAR*, pp. 549–564, 2002.

[28] C. Tenllado, D. Chaver, L. P. nuel, M. Prieto, and F. Tirado, "Vectorization of the 2D wavelet lifting transform using SIMD extensions," in *Proc. Int. Parallel and Distributed Processing Simposium*, April 2003.

[29] Intel Corp., *Real and Complex FIR filter Using Streaming SIMD Extensions - Intel application note AP-809*. `http://developer.intel.com`.

[30] B. Singer and M. M. Veloso, "Automating the Modeling and Optimization of the Performance of Signal Transforms," *IEEE Trans. on Signal Processing*, vol. 50, no. 8, pp. 2003–2014, 2002.

[31] G. E. Révész, *Introduction to Formal Languages*. McGraw-Hill, 1983.

[32] A. Graham, *Kronecker Products and Matrix Calculus with Applications*. New York: John Wiley & Sons, Ellis Horwood Series in Mathematics and Its Applications, 1981.

[33] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. of Computation*, vol. 19, pp. 297–301, 1965.

[34] J. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures," *IEEE Trans. on Circuits and Systems*, vol. 9, no. 4, pp. 449–498, 1990.

[35] R. Tolimieri, M. An, and C. Lu, *Algorithms for discrete Fourier transforms and convolution*. Springer, 2nd ed., 1997.

[36] C. Van Loan, *Computational Framework of the Fast Fourier Transform*. Siam, 1992.

[37] J. Johnson and M. Püschel, "In search for the optimal Walsh-Hadamard transform," in *Proc. IEEE Int. Conf. Acoust. Speech Sign. Process.*, vol. IV, pp. 3347–3350, 2000.

[38] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*. Prentice-Hall, 2nd ed., 1999.

[39] F. Franchetti, Y. Voronenko, and M. Püschel, "Loop merging for signal transforms," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2005. submitted for publication.

[40] The GAP Team, University of St. Andrews, Scotland, *GAP—Groups, Algorithms, and Programming*, 1997. `www-gap.dcs.st-and.ac.uk/~gap/`.

[41] M. Frigo, "A fast Fourier transform compiler," in *Proc. ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 169–180, 1999.

[42] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A Language and Compiler for DSP Algorithms," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 298–308, 2001.

[43] N. Rizzolo and D. Padua, "Hilo: High level optimization of FFTs," in *Proc. Workshop on Languages and Compilers for Parallel Computing*, 2004. (to appear).

[44] Y. Voronenko and M. Püschel, "Automatic Generation of Implementations for DSP Transforms on Fused Multiply-Add Architectures," in *Proc.Int. Conf. Acoustics, Speech, and Signal Processing*, 2004.

[45] F. Franchetti, H. Karner, S. Kral, and C. W. Ueberhuber, "Architecture Independent Short Vector FFTs," in *Proc.ICASSP*, vol. 2, pp. 1109–1112, 2001.

[46] F. Franchetti and M. Püschel, "A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms," in *Proc. IPDPS*, pp. 20–26, 2002.

[47] F. Franchetti and M. Püschel, "Short Vector Code Generation for the Discrete Fourier Transform," in *Proc. IPDPS*, pp. 58–67, 2003.

[48] R. E. Bellman and S. E. Dreyfuss, *Applied Dynamic Programming*. New Jersey: Princeton University Press, 1962.

[49] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[50] B. Singer and M. Veloso, "Stochastic search for signal processing algorithm optimization," in *Proc. Supercomputing*, 2001.

[51] D. Sepiashvili, " Performance models and search methods for optimal FFT implementations," Master's thesis, ECE Dept., Carnegie Mellon University, 2000.

[52] G. Haentjens, "An investigation of recursive FFT implementations," Master's thesis, ECE Dept., Carnegie Mellon University, 2000.

[53] S. A. Martucci, "Symmetric convolution and the discrete sine and cosine transforms," *IEEE Trans. on Signal Processing*, vol. 42, no. 5, pp. 1038–1051, 1994.

[54] M. Püschel and J. M. F. Moura, "Algebraic theory of signal procesing," *submited to IEEE Transactions on Image Processing*, p. 66, December 2004.

[55] M. Püschel and J. M. F. Moura, "The algebraic approach to the discrete cosine and sine transforms and their fast algorithms," *SIAM Journal of Computing*, vol. 32, no. 5, pp. 1280–1316, 2003.

[56] R. C. Agarwal and J. W. Cooley, "New algorithms for digital convolution," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. ASSP-25, no. 5, pp. 392–410, 1977.

[57] Z. J. Mou and P. Duhamel, "Fast FIR filtering: algorithms and implementations," *Signal Process.*, vol. 13, no. 4, pp. 377–384, 1987.

[58] R. C. Agarwal and C. S. Burrus, "Fast one-dimensional digital convolution by multi-dimensional techniques," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. ASSP-22, no. 1, pp. 1–10, 1974.

[59] H. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*. Springer, 1981.

[60] R. E. Blahut, *Fast Algorithms for DIgital Signal Processing*. Reading, MA: Addison-Wesley, 1985.

[61] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2. Addison-Wesley, 3rd ed., 1997.

[62] M. Vetterli, "Running FIR and IIR filtering using multirate filter banks," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 36, pp. 730–738, May 1988.

[63] Z. J. Mou and P. Duhamel, "A unified approach for the fast FIR filtering algorithms," in *Proc. ICASSP*, pp. 1914–1917, April 1988.

[64] S. Winograd, "Some bilinear forms whose multiplicative complexity depends on the field of constants," *Math. Syst. Theor.*, vol. 10, pp. 169–180, 1977.

[65] J. Hong, M. Vetterli, and P. Duhamel, "Basefield transforms with the convolution property," *Proc. of the IEEE*, vol. 82, pp. 400–412, March 1994.

[66] N.-C. Hu and O. K. Ersoy, "Fast computation of real discrete Fourier transform for any number of data points," *IEEE Trans. on Circuits and Systems*, vol. 38, pp. 1280–1292, Nov. 1991.

[67] V. Britanak and K. R. Rao, "The fast generalized discrete Fourier transforms: A unified approach to the discrete sinusoidal transforms computation," *Signal Processing*, vol. 79, pp. 135–150, 1999.

[68] R. N. Bracewell, *The Hartley Transform*. New York: Oxford University Press, 1986.

[69] P. Duhamel and M. Vetterli, "Improved Fourier and Hartley transform algorithms: applications to cyclic convolution of real data," *IEEE Trans. on Acoust. Speech and Sign. Process.*, vol. ASSP-35, pp. 818–824, June 1987.

[70] G. Bi and Y. Chen, "Fast generalized DFT and DHT algorithms," *Signal Processing*, vol. 65, pp. 383–390, March 1998.

[71] I. Daubechies, "Orthonormal bases of compactly supported wavelets," *Comm. on Pure and Appl. Math.*, vol. 4, pp. 909–996, Nov. 1988.

[72] S. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Trans. on Pattern Anal. and Machine Intell.*, vol. 11, pp. 674–693, July 1989.

[73] L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals*. Englewood Cliffs, NJ: Prentice Hall, 1978.

[74] H. Guo and S. C. Burrus, "Waveform and image compression with the Burrows Wheeler transform and the wavelet transform," in *Proc. IEEE Int. Conf. Imag. Process.*, (Santa Barbara), pp. 26–29, Oct. 1997.

[75] A. Said and W. A. Perlman, "An image multiresolution representation for lossles and lossy image compression," *IEEE Trans. on Image Processing*, vol. 5, pp. 1303–1310, Sept. 1996.

[76] D. L. Donoho, "Denoising by soft-thresholding," *IEEE Trans. on Information Theory*, vol. 41, pp. 613–627, May 1995.

[77] G. Beylkin and J. M. Keiser, "On the adaptive numerical solution for nonlinear partial differential equations in wavelet bases," *Journ. Comp. Phys.*, vol. 132, pp. 233–259, 1997.

[78] W. Dahmen, A. Durdila, and P. Oswald, eds., *Multiscale Wavelet Methods for Partial Differential Equations*. San Diego: Academic Press, 1997.

[79] G. Bachman, L. Narici, and E. Beckenstein, *Fourier and Wavelet Analysis*. New York: Springer-Verlag, 2000.

[80] R. H. Bamberger, S. L. Eddins, and V. Nuri, "Generalized symmetric extension for size-limited multirate filter banks," *IEEE trans. on image process.*, vol. 3, pp. 82–87, Jan. 1994.

[81] V. Silva and L. de Sá, "General method for perfect reconstruction subband processing of finite length signals using linear extensions," *IEEE trans. on signal processing*, vol. 47, September 1999.

[82] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*. Prentice-Hall, 1993.

[83] K. Ramchandran and M. Vetterli, "Best wavelet packet bases in a rate-distortion sense," *IEEE Trans. on Image Processing*, vol. 2, pp. 160–174, April 1993.

[84] K. Ramchandran, M. Vetterli, and C. Herley, "Wavelets, subband coding, and best bases," *Proc. of the IEEE*, vol. 84, pp. 541–559, April 1996.

[85] J. M. Saphiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Trans. on Signal Processing*, vol. 41, no. 12, pp. 3445–3462, 1993.

[86] C. Herley and M. Vetterli, "Orthogonal time-varying filter banks and wavelet packets," *IEEE Transactions on Signal Processing*, vol. 42, pp. 2650–2663, October 1994.

[87] I. Daubechies, *Ten Lectures on Wavelets*. Philadelphia, PA: SIAM, CMBS series, 1992.

[88] P. Steffen, P. N. Heller, R. A. Gopinath, and C. S. Burrus, "Theory of regular M-band wavelet bases," *IEEE Trans. on Signal Processing*, vol. 41, pp. 3497–3511, December 1993.

[89] I. W. Selesnick, "Parametrization of orthogonal wavelet systems," tech. rep., ECE Dept. and Computational Mathematics Library, Rice University, Houston, TX, May 1997.

[90] R. A. Gopinath, J. E. Odegard, and C. S. Burrus, "Optimal wavelet representation of signals and the wavelet sampling theorem," *IEEE Trans. on Circuits and Systems II*, vol. 41, pp. 262–277, April 1994.

[91] A. Cohen, I. Daubechies, and J. C. Feauveau, "Biorthogonal bases of compactly supported wavelets," *Comm. on Pure and Aplied Math.*, vol. 45, pp. 485–560, 1992.

[92] C. M. Brislawn, J. N. Bradley, R. J. Onyshczak, and T. Hopper, "The FBI compression standard for digitized fingerprint images," in *Proc. SPIE Conf. 2847, Appl. Digital Image Process. XIX*, 1996.

[93] G. Beylkin, R. R. Coifman, and V. Rokhlin, "Fast wavelet transforms and numerical algorithms," *Comm. Pure and Appl. Math.*, vol. 44, pp. 141–183, 1991.

[94] C. S. Burrus and J. E. Odegard, "Wavelet systems and zero moments," *IEEE Trans. Sign. Process.*, 1996.

[95] M. Bellanger, G. Bonnerot, and M. Coudreuse, "Digital filtering by polyphase network: Application to sample rate alteration and filter banks," *IEEE Trans.on Acoust. Speech and Sign. Process.*, vol. ASSP-24, pp. 109–114, April 1976.

[96] M. Vetterli and D. L. Gall, "Perfect reconstruction FIR filter banks: some properties and factorizations," *IEEE Trans. on Acoustics, SPeech and SIgnal Processing*, vol. ASSP-37, pp. 1057–1071, July 1989.

[97] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes*. Cambridge Univ. Press, 1989.

[98] M. Lounsbery, T. D. DeRose, and J. Warren, "Multiresolution surfaces of arbitrary topological type," *Trans. on Graphics*, vol. 16, no. 1, pp. 34–73, 1997.

[99] T. G. Marshall, "U-L block-triangular matrix and ladder realizations of subband coders," in *Proc. IEEE ICASSP*, vol. III, pp. 177–180, 1993.

[100] I. Shah and T. A. Kalker, "On ladder structures and linear phase conditions fo bi-orthogonal filter banks," in *Proc. IEEE ICASSP*, vol. 3, pp. 181–184, 1994.

[101] R. Calderbank, I. Daubechies, W. Sweldens, and B. L. Yeo, "Wavelet transforms that map integers to integers," *Applied and Computational Harmonic Analysis*, vol. 5, no. 3, pp. 332–369, 1998.

[102] V. K. Goyal, "Transform coding with integer to integer transforms," *IEEE Transactions on Information Theory*, vol. 46, pp. 465–473, March 2000.

[103] K. M. Hoffman and R. Kunze, *Linear Algebra*. Prentice-Hall, 2nd ed., 1971.

[104] K. Taswell and K. C. McGill, "Algorithm 735: Wavelet transform algorithms for finite-duration discrete-time signals," *ACM Trans. on Math. Softw.*, vol. 20, pp. 398–412, Sept. 1994.

[105] K. C. McGill and C. Taswell, "Length-preserving wavelet transform algorithms for zero-padded and linearly-extended signals," tech. rep., Veterans Affairs Medical Center, Palo Alto, CA, 1992.

[106] A. Cohen, I. Daubechies, and P. Vial, "Wavelets and wavelet transforms on an interval," *Appl. Comput. Harmon. Anal.*, vol. 1, pp. 54–81, 1993.

[107] G. Karlsson and M. Vetterli, "Extension of finite length signals for sub-band coding," *Signal Processing*, vol. 17, pp. 161–168, 1989.

[108] *ACOVEA: Analysis of computer options via evolutionary algorithm*, 2004. `http://www.coyotegulch.com/acovea/`.

[109] A. K. Jain, *Fundamentatls of Digital Image Processing*. Upper Saddle River, NJ: Prentice-Hall, 1989.

[110] F. C. A. Fernandes, R. L. C. van Spaendonck, and C. S. Burrus, "A new framework for complex wavelet transforms," *IEEE Trans. on Signal Processing*, vol. 51, pp. 1825–1837, July 2003.

[111] D. Stefaniou and I. Tabus, "Euclidean lifting schemes for I2I wavelet transform implementations," *Studies in Informatics and Control*, vol. 11, pp. 255–270, September 2002.