Scalable Block Processing Algorithms

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 4 september 2008 om 16.00 uur

door

Martinus Gerardus van der Horst

geboren te Boxmeer
Dit proefschrift is goedgekeurd door de promotor:
prof.dr.ir. C.H. van Berkel

Copromotor:
prof.dr. J.J. Lukkien
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2008-18

Printing: Printservice Technische Universiteit Eindhoven

Cover design: Oranje Vormgevers, Eindhoven

ISBN: 978-90-386-1337-6

A catalogue record is available from the Eindhoven University of Technology Library

© Copyright 2008, M.G. van der Horst
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission from the copyright owner.
Contents

1 Introduction 1
   1.1 Introduction ........................................... 1
      1.1.1 Workloads are increasing ............................ 1
      1.1.2 Processing power is increasing ....................... 4
      1.1.3 Applying the processing power ....................... 5
   1.2 Stream processing systems ............................... 6
      1.2.1 Finite state machines ............................... 6
      1.2.2 Classes of stream processing systems ............... 8
   1.3 Algorithms ........................................... 10
      1.3.1 Functional correctness ............................. 10
      1.3.2 Block processing .................................... 11
      1.3.3 Latency ............................................ 12
      1.3.4 Memory ............................................. 12
      1.3.5 Complexity ......................................... 13
      1.3.6 Iteration period and throughput ...................... 15
      1.3.7 Summary ............................................ 17
   1.4 Hardware ............................................. 18
      1.4.1 SISD ............................................... 18
      1.4.2 VLSI ............................................... 19
      1.4.3 SIMD ............................................... 21
   1.5 Scalability ........................................... 23
      1.5.1 Scalability classes .................................. 23
      1.5.2 Realizing linear scalability ......................... 25
   1.6 Research Goals ......................................... 26
   1.7 Thesis outline ......................................... 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Existing methods</td>
<td>29</td>
</tr>
<tr>
<td>2.1</td>
<td>Introduction</td>
<td>29</td>
</tr>
<tr>
<td>2.2</td>
<td>Algebraic state update equation</td>
<td>30</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Matrices</td>
<td>31</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Higher order recursions</td>
<td>33</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Arbitrary FSMs</td>
<td>34</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Note on finite precision arithmetic</td>
<td>36</td>
</tr>
<tr>
<td>2.3</td>
<td>Pipeline structures</td>
<td>37</td>
</tr>
<tr>
<td>2.4</td>
<td>Linear look-ahead</td>
<td>40</td>
</tr>
<tr>
<td>2.5</td>
<td>Logarithmic look-ahead</td>
<td>44</td>
</tr>
<tr>
<td>2.6</td>
<td>Parallel prefix look-ahead</td>
<td>48</td>
</tr>
<tr>
<td>2.7</td>
<td>Block post-computation</td>
<td>51</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Arbitrary FSMs</td>
<td>51</td>
</tr>
<tr>
<td>2.7.2</td>
<td>FSMs with recurring states</td>
<td>54</td>
</tr>
<tr>
<td>2.8</td>
<td>Conclusion</td>
<td>56</td>
</tr>
<tr>
<td>3</td>
<td>Solution strategies</td>
<td>59</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>59</td>
</tr>
<tr>
<td>3.2</td>
<td>Infinite memory span strategy</td>
<td>60</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Single stage strategy</td>
<td>61</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Two stage strategy</td>
<td>73</td>
</tr>
<tr>
<td>3.3</td>
<td>Finite memory span strategy</td>
<td>80</td>
</tr>
<tr>
<td>3.3.1</td>
<td>VLSI implementation</td>
<td>80</td>
</tr>
<tr>
<td>3.3.2</td>
<td>SIMD implementation</td>
<td>82</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Other strategies</td>
<td>84</td>
</tr>
<tr>
<td>3.4</td>
<td>Associative window computation strategy</td>
<td>86</td>
</tr>
<tr>
<td>3.5</td>
<td>Conclusion</td>
<td>89</td>
</tr>
<tr>
<td>4</td>
<td>Associative window computations</td>
<td>93</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>93</td>
</tr>
<tr>
<td>4.2</td>
<td>Tiling</td>
<td>94</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Tiles</td>
<td>94</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Tile-sets</td>
<td>95</td>
</tr>
<tr>
<td>4.3</td>
<td>Associative window computations</td>
<td>97</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Definition of an associative window computation</td>
<td>97</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Mapping between grids and streams</td>
<td>100</td>
</tr>
<tr>
<td>4.4</td>
<td>From window to look-ahead computation</td>
<td>102</td>
</tr>
<tr>
<td>4.5</td>
<td>From look-ahead to window computation</td>
<td>104</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Basic look-ahead computation</td>
<td>105</td>
</tr>
</tbody>
</table>
CONTENTS

4.5.2 Parallel prefix based look-ahead ........................................... 106
4.5.3 Function composition ....................................................... 108
4.6 Conclusion .................................................................................. 109

5 Low-cost associative window computations ................................ 111
  5.1 Introduction ................................................................................ 111
  5.2 Calculation graph ....................................................................... 112
  5.3 Generator graph ......................................................................... 115
    5.3.1 Preliminaries ........................................................................ 116
    5.3.2 Canonical generator graphs .................................................. 119
    5.3.3 Generator graph composition ............................................... 124
  5.4 Generator graph construction .................................................... 128
    5.4.1 Basic top-down approach ..................................................... 129
    5.4.2 Partitioning top-down approach .......................................... 131
    5.4.3 Auxiliary functions ............................................................. 132
  5.5 Heuristic algorithms ................................................................... 132
    5.5.1 Divide and conquer .............................................................. 133
    5.5.2 Maximum reuse ................................................................... 135
    5.5.3 van Herk-Gil-Werman algorithm ......................................... 140
    5.5.4 Harrington’s algorithm ........................................................ 141
    5.5.5 Other heuristics .................................................................. 145
  5.6 Conclusion .................................................................................. 146

6 Rank order filtering ................................................................. 147
  6.1 Introduction ................................................................................ 147
  6.2 Rank order filters ........................................................................ 148
  6.3 Sliced merge operator ................................................................ 150
  6.4 Implementation of the sliced merge .......................................... 153
    6.4.1 The cost of merging networks ............................................. 154
    6.4.2 Mirroring ............................................................................ 158
  6.5 The ORM heuristic ................................................................. 161
    6.5.1 Informal description ............................................................ 161
    6.5.2 Algorithm specification ....................................................... 163
    6.5.3 Examples of generated graphs .......................................... 169
  6.6 Results ....................................................................................... 171
  6.7 Comparison ................................................................................. 176
    6.7.1 Dilation filter ....................................................................... 176
    6.7.2 Median filter ........................................................................ 178
    6.7.3 Window sorters ................................................................... 181
CONTENTS

6.8 Conclusion ................................................................. 184

7 Implementation ......................................................... 187
7.1 Introduction .............................................................. 187
7.2 SIMD Processors ......................................................... 188
  7.2.1 Common properties ................................................. 189
  7.2.2 Memory access ..................................................... 190
  7.2.3 Topology ............................................................. 193
  7.2.4 Example: Embedded Vector Processor ......................... 198
7.3 Implementing pipelines ............................................... 199
  7.3.1 Approach .......................................................... 199
  7.3.2 Example: Infinite impulse response filter ..................... 200
  7.3.3 Discussion ........................................................ 206
7.4 Implementing generator graphs ......................................... 207
  7.4.1 Approach .......................................................... 207
  7.4.2 Example: Rank order filtering .................................. 215
  7.4.3 Discussion ........................................................ 218
7.5 Look-ahead computations ............................................... 219
  7.5.1 Approach using strided memory access ......................... 219
  7.5.2 Approach using shuffle operations ............................. 222
  7.5.3 Example: Infinite impulse response filter ..................... 231
  7.5.4 Discussion ........................................................ 233
7.6 Conclusion .............................................................. 233

8 Conclusion .............................................................. 235

A Notation ................................................................. 241
A.1 Mathematical notation ............................................... 241
  A.1.1 Types ............................................................. 241
  A.1.2 Eindhoven Notation ............................................. 244
  A.1.3 Sets ............................................................... 245
  A.1.4 Lists .............................................................. 245
  A.1.5 Groups, semi-groups and semi-rings ......................... 247
  A.1.6 Asymptotic notation ............................................ 247
A.2 SIMD Pseudo-code ................................................... 248
  A.2.1 Variables and types ............................................. 248
  A.2.2 Statements ....................................................... 249
  A.2.3 Extensions ....................................................... 251
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>B Tools</strong></td>
<td>255</td>
</tr>
<tr>
<td>B.1 OWS tools</td>
<td>255</td>
</tr>
<tr>
<td>B.2 WinLA tool</td>
<td>256</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>257</td>
</tr>
<tr>
<td><strong>Samenvatting</strong></td>
<td>259</td>
</tr>
<tr>
<td><strong>Curriculum Vitae</strong></td>
<td>261</td>
</tr>
<tr>
<td><strong>Bibliography</strong></td>
<td>263</td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td>271</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Introduction

A stream processing system is a system that processes a stream of input data and produces a stream of output data. These types of systems are often encountered in signal processing applications, and many examples are found in common consumer electronic devices like mobile phones and DVD players, where they convert an input stream of bits into an output stream suitable for the human senses. The amount of data processed per second by these systems is very large; a 3G cellular phone is capable of processing up to 2 million bits per second [71] and a dual layer DVD can contain up to 68 billion bits [68], which are processed at a rate over 10 million bits per second. These processing rates are large and increase with each new standard, resulting in an increased workload for stream processing systems. In this thesis we discuss this increase in workload, the increasing processing power of the hardware used to implement them, and especially how to combine these two effectively.

1.1.1 Workloads are increasing

We have depicted some stream processing standards in Figure 1.1. The position of the standards in Figure 1.1 indicates the workload for a system that implements that standard. The horizontal axis of the graph corresponds to the sample rate of the system, which is the rate at which data is going in or coming out of the system. Clearly a high data rate corresponds with a high workload.
Figure 1.1: Overview of various stream processing systems
1.1. INTRODUCTION

The sample rate is, however, not solely responsible for the workload; we also need the amount of work per sample, which is indicated by the vertical axis.

We get the workload of a standard in terms of operations per second by multiplying its two coordinates. This allows us to draw the isolines in the graph, representing coordinates that have the same workload. So, a stream processing system that adheres to the GSM standard has a workload that is approximately the same as that of a system that adheres to the DAB standard, namely a little below 100 MOPS (mega operations per second).

Note that the number of operations is notoriously difficult to obtain, as it depends on the hardware platform and in many cases the content of the input stream. Therefore the positions of the standards in Figure 1.1 should be taken as indications. They were obtained using the work of various authors [1, 2, 8, 11, 21, 59, 60, 61, 80], and their works can be consulted for the reasoning behind the number of operations per sample.

It is interesting to note that the newer standards tend to the higher workloads. The, relatively, new MPEG-4 and H.264 standards require, for example, a lower bit-rate to encode the same quality of video as MPEG-2, but pay for this efficiency by a significant increase in the number of operations, which increases the workload. The newer standards therefore require more processing power.

In fact, this progress is captured by Gilder’s law, which states that the total bandwidth of communication systems will triple every year for the next 25 years\footnote{that would be from 1997 to 2022} [30]. This increase in bandwidth is not only the result of an increase in the number of communication systems, but also the result of an increase in the bandwidth of individual links. This increase is partially accomplished by advances in hardware that provide us with higher capacity communication channels, resulting in more data being transmitted per second and therefore systems that have to process more samples per second. The advances in hardware technology are not the only thing that enables the increase in bandwidth, however. Advances in compression algorithms, like arithmetic coding, enable us to store more data in fewer bits. Furthermore, error correcting algorithms like turbo codes enable us to communicate these bits reliably at rates that approach the Shannon limit. All these advances in compression and error correction algorithms come at a price; the amount of processing per bit increases, resulting in more operations per sample for systems that execute these algorithms. So, in the future we require systems that process more samples per second and perform more operations per sample than before, resulting in systems with a higher workload.
1.1.2 Processing power is increasing

The hardware that performs up to 100 MOPS (mega operations per second), as required by the MP3 and GSM standards, is cheap and readily available. High-end, general purpose, single core processors can perform several GOPS (giga operations per second), and although these processors are relatively expensive now, their prices will drop eventually. Furthermore, at the time of writing, we need only wait for a few years to obtain general purpose, single core processors that are capable of 10 GOPS. General purpose, single core processors are not the only source of processing power, however. Higher processing rates are obtained by exploiting parallelism: instead of a general purpose, single core processor the data is processed by multiple, and possibly dedicated, processor cores.

So, processing power also increases along two axes. First there is the speed of the hardware, or clock frequency. This speed has been increasing exponentially for the last couple of years and according to the ITRS [41] it will keep increasing exponentially in the years to come, as shown in Figure 1.2(a). Second, Moore’s law indicates an exponential increase, not for the speed, but for the number of transistors on a single chip, which concerns the second axis of our processing power graph, namely the amount of hardware in terms of number of transistors.

The exponential increase of the number of transistors on a single chip is also expected to hold in the years to come [41], as shown in Figure 1.2(b). These graphs indicate that the number of transistors on a single chip is expected to double every 3 years\(^2\), and the clock frequency is expected to double every 4

\(^2\)Moore’s law originally stated that the number of transistors doubles every year [57], it
years plus a few months. Although the number of transistors cannot be easily translated in a number of operations, these graphs do indicate that the total processing power of a single chip is expected to increase with approximately 50% every year.

Time will tell whether this increase in processing power is enough to keep up with the growing demand for processing power. It is clear however, that the increase in speed alone, doubling every four years and a few months, is rapidly becoming inadequate to keep up with the workload that is tripling every year. We also have use the increasing amount of hardware to satisfy the demand for processing power. Chip manufacturers have recognized this problem and started using the increasing number of transistors to put multiple cores on a single chip.

1.1.3 Applying the processing power

The problem is to put the available processing power to good use for a given stream processing system.

Utilizing only one dimension of the available processing power, namely the speed of hardware, is no problem. We can use the same algorithms or hardware designs to achieve a system with higher throughput and lower latency by simply using faster hardware. Since the speed of hardware has been increasing exponentially, this approach has been working well so far.

Utilizing the other dimension, namely the amount hardware, by means of parallelism is more of a problem since there is no simple method to increase the performance of a system given more hardware. Given the increase in processing power requirements, however, we are forced to find a way.

Block processing algorithms provide a solution to this problem. A stream processing system based on a block processing algorithm does not process the data elements of the stream one by one, but processes them block by block, hence the name. These block processing algorithms benefit from increases in hardware speed just like any other algorithm. Faster hardware means that blocks are processed faster, resulting in a higher throughput and lower latency.

Unlike other algorithms, however, block processing algorithms can also benefit from increases in the amount of hardware. When extra hardware is available it can be utilized by increasing the block size such that the extra hardware is used to process the extra elements of the blocks. Ideally, the throughput of the algorithm is proportional to the amount of hardware used to implement it. Block processing algorithms that have this property are said to scale linearly.

\footnote{was later adjusted to every two years \cite{12}, and it seems that another adjustment is in order.}
Block processing algorithms that scale linearly make full use of the processing power provided by a hardware on which they are implemented.

Not all block processing algorithms scale linearly, however, and designing those that do is not a trivial matter. In this thesis we therefore solve this problem, at least theoretically, by presenting methods for designing scalable block processing algorithms and ways to implement them on existing hardware platforms. Furthermore, we put these theoretical method into practice to check their performance on existing stream processing systems and hardware. First, however, formalize our definitions in the next few sections, after which we formulate our research goals more precisely in Section 1.6.

1.2 Stream processing systems

The systems we consider in this thesis are stream processing systems. Stream processing systems read an infinite stream of input data and process it to produce an infinite stream of output data.

In practice the stream of data is not really infinite, but there is no practical benefit to designing algorithms for large but finite streams, whereas the theory becomes simpler when infinite streams are considered. For this reason we consider infinite streams in this thesis.

In contrast to [75], we only consider stream processing systems that have one input and one output stream. We can model multiple streams, however, by using input and output alphabets containing tuples. Furthermore, our stream processing systems can be connected together to form networks of stream processing systems, as described in [75].

In this section we give a more formal specification of stream processing systems and distinguish several classes. A detailed overview of the mathematical notation used in these formal descriptions can be found in Appendix A.

1.2.1 Finite state machines

The Turing machine is generally the model of choice for computer systems. Any real-world stream processing system, however, has a finite amount of memory, and can therefore also be represented by a finite state machine (FSM). In this section we present our reasons for modeling stream processing systems as FSMs, and discuss the formal representation of FSMs.

The first reason for modeling stream processing systems as FSMs is that the two are similar. Finite state transducers, a type of FSM, are in fact stream
1.2. STREAM PROCESSING SYSTEMS

processing algorithms, since they translate an input string into an new output string. The only difference between the two is that FSMs operate on strings of finite length, while stream processing systems operate on infinite streams. This difference causes no practical problems, however, so modeling stream processing systems like linear filters, Viterbi (de)coders, etc. as FSMs is possible.

The second reason that use FSMs instead of Turing machines, is that FSM are simpler, since a Turing machine includes an FSM. In this thesis we show that the FSM model is adequate for designing block processing algorithms with linear scalability, so there is no reason to introduce the formalism of Turing machines.

For these reasons we use FSMs to model our stream processing systems. In the remainder of this section we formalize our notation for FSMs and the way in which they model stream processing systems.

A stream processing system has an input stream and an output stream. The elements of the input stream are members of the set $\Sigma$, called the input alphabet and the elements of the output stream are members of the set $\Gamma$, called the output alphabet. The input and output streams themselves are represented by functions, respectively:

\[
\text{in} : \mathbb{N} \rightarrow \Sigma \quad (1.1)
\]
\[
\text{out} : \mathbb{N} \rightarrow \Gamma \quad (1.2)
\]

So $\text{in}(0)$ represents the first input, $\text{in}(1)$ the second, etc. For now we consider 1-dimensional input and output streams, but in Chapter 4 we will expand our definitions to include data organized in multi-dimensional grids.

We consider the stream processing systems that compute the output stream as specified by some finite state machine (FSM), also called a finite state transducers. This FSM is represented by the tuple $(\Sigma, \Gamma, S, s_0, \delta, \omega)$, where $\Sigma$ and $\Gamma$ denote the input and output alphabet, $S$ the set of states, $s_0$ the starting state of the machine, $\delta : S \times \Sigma \rightarrow S$ the state update function and $\omega : S \rightarrow \Gamma$ the output function. This tuple $(\Sigma, \Gamma, S, s_0, \delta, \omega)$ specifies the following relation between inputs, outputs and state of the machine:

\[
s(0) = s_0 \quad (1.3)
\]
\[
s(i + 1) = \delta(s(i), \text{in}(i)) \text{ for } i \in \mathbb{N} \quad (1.4)
\]
\[
\text{out}(i) = \omega(s(i + 1)) \text{ for } i \in \mathbb{N} \quad (1.5)
\]

This definition corresponds with a so-called Moore machine. Furthermore, we have adopted the same convention as in [10], namely that the machine does not produce an output before the first input has been read.
Note that we use indices in our notation, unlike and many other text books. We do this because we are also interested in the exact relation between the inputs and outputs, and we will need indices to express these relations later on. So, although we could use index-less notation in some cases, we use an indexed notation throughout this thesis for consistency.

1.2.2 Classes of stream processing systems

The stream processing systems that can be modeled by an FSM, as described in the previous section, belong to the FSM class of systems. Lin and Messerschmitt defined two subclasses of this class in [53]:

A finite memory span FSM has a present state determined by a finite number of past inputs; otherwise, the FSM has an infinite memory span.

We propose to formalize this definition in the following manner. We define the extended state transition function \( \delta^* : S \times L(\Sigma) \rightarrow S \), where \( L(\Sigma) \) denotes the set containing all lists over the alphabet \( \Sigma \). The extended state transition function returns the state of the system for a given starting state and a sequence, or list of inputs in the following manner:

\[
\delta^*(s, l) = \begin{cases} 
  s & \text{if } l = [] \\
  \delta^*(\delta(s, \sigma), l_1) & \text{if } l = [\sigma] + l_1
\end{cases} \tag{1.6}
\]

This allows us to define the memory span \( W \) of a system as:

\[
W = (\max s_1, s_2, l : s_1 \neq s_2 \land \delta^*(s_1, l) \neq \delta^*(s_2, l) : \#l) + 1 \tag{1.7}
\]

where \( \max \) denotes the maximum operator, and \( \#l \) denotes the length of the list \( l \). Equation (1.7) specifies that the memory span is equal to the length of the longest sequence of inputs that still leads to different states, given different starting states, plus one. This means that the state of the system in any point in time can be derived by examining the last \( W \) inputs.

We use this memory span to divide the systems into different classes. Since it can be difficult to determine the actual memory span of a system, we classify the systems according to an upper bound for their memory span. We use \( FSM_\infty \) to denote the class of systems with a memory span \( W \) that is infinite or less, i.e., all systems \( (FSM_\infty = FSM) \). The next class is \( FSM_* \), which contains all systems with a memory span that is finite, i.e., the finite memory span systems \( (FSM_* \subset FSM_\infty) \).
1.2. **STREAM PROCESSING SYSTEMS**

Systems that have an infinite $W$ are called infinite memory span systems, and therefore belong to the class $FSM_{\infty} \setminus FSM_s$. To derive the state of such a system for a certain point in time the entire input stream until that point has to be examined. Examples of systems that fall into this class are infinite impulse response filters (IIR filters), arithmetic (de)coders and checksum generators (MD5, CRC, parity check, etc.). These systems are sometimes called recursive systems, since they are commonly specified by a recursive equation. Note, however, that a system specified by a recursive equation does not necessarily have an infinite memory span, as we show in Section 3.2.

Examples of finite memory span systems, i.e. class $FSM_s$ systems, are finite impulse response filters (FIR filters) and rank order filters. These systems are also called window computation systems, since their state can be determined by applying a function to a sliding window over the input stream. More formally, for every system in class $FSM_s$ there exist some function $f$ such that:

$$\forall i : i \in \mathbb{N} : s(i + 1) = f(in(i), in(i - 1), \ldots, in(i - W + 1))$$  \hspace{1cm} (1.8)

where we assume that $in(i)$ for $i < 0$ has some default value from $\Sigma$. So the state, and therefore also the output, of an $FSM_s$ machine are determined by the last $W$ inputs. Therefore there also exists some function $g$ such that:

$$\forall i : i \in \mathbb{N} : out(i) = g(in(i), in(i - 1), \ldots, in(i - W + 1))$$  \hspace{1cm} (1.9)

We now define a subclass of the $FSM_s$ systems based on this function $g$. The systems that belong to this class have $\Sigma = \Gamma = D$ and a very specific function $g$, namely one that aggregates inputs using an associative operator like so:

$$out(i) = (\circ j : 0 \leq j < W : in(i - j)) \text{ for } i \in \mathbb{N}$$  \hspace{1cm} (1.10)

where $D$ is closed under the binary operation $\circ$, and $\circ$ is associative, i.e., $(D, \circ)$ is a semi-group. Note that $\circ$ is not necessarily commutative, which means that (1.10) also specifies the aggregation order$^3$, namely from $j = 0$ to $j = W - 1$. Although systems belonging to this class perform “window computations using an associative aggregation operator”, that is a rather long name. Therefore we simply call them associative window computation systems, and denote them by $FSM_s^{\circ}$, even though there is nothing associative about the windows themselves.

---

$^3$Eindhoven notation normally does not imply an ordering, but for non-commutative operators we use the convention that the dummy variables traverse the domain in ascending order. See also Appendix A.1.2.
Examples of \( FSM \) systems are systems that compute the moving sum and rank order filters. These examples are discussed in more detail in Chapter 3 and Chapter 6 of this thesis. Especially the moving sum is interesting because it is a very simple system that allows us to demonstrate the differences between the classes of systems. Furthermore, many systems contain components that perform associative window computations, most notably the systems that perform look-ahead computations, this is discussed in Section 4.5.

1.3 Algorithms

Algorithms form the link between the specification of the system and the actual implementation on a piece of hardware by describing the way in which basic operations are used to derive the output stream from the input stream.

This description can be in natural language, but there are many other possibilities like pseudo-code or a graphical description in the form of block diagrams, signal-flow graphs or data-flow graphs [65].

In this section we opt to represent algorithms as finite state machines with extra annotation. This allows us to introduce, and formally define, the performance measures of the algorithm, and the restrictions imposed by various hardware platforms on which such an algorithm could be implemented. Furthermore, the other algorithm representations can be converted to this representation and this representation is suited for representing both VLSI and SIMD algorithms, as described in Section 1.4.

Although our representation is useful for our formal definitions, it is not well suited for representing algorithms in a manner that makes them easy to understand for human beings. So, in the rest of the thesis we revert to more traditional representations like pseudo-code and block diagrams.

1.3.1 Functional correctness

Representing algorithms as FSMS makes it easy to specify functional correctness of the algorithm, since the specification of the stream processing systems is also modeled as an FSM. So, when \( \text{out}_{\text{spec}} \) is the output stream of the FSM used to represent the specification and \( \text{out}_{\text{alg}} \) the output stream of the FSM used to represent the algorithm, then the algorithm is functionally correct if for some \( L \in \mathbb{N} \) the following holds for all input streams:

\[
(\forall i : i \in \mathbb{N} : \text{out}_{\text{spec}}(i) = \text{out}_{\text{alg}}(i + L))
\]
1.3. ALGORITHMS

That is, both FSMs produce the same outputs when they are fed same input stream, though the algorithm may produce some start-up noise first. Of course, it is not trivial to actually check the functional correctness of a given algorithm, which is why we focus on designing functionally correct algorithms by deriving them from the systems specification.

1.3.2 Block processing

An algorithm represented by an FSM has a limited throughput. The algorithm produces one output per state update, and there is a definite minimum to the time needed to perform a state update. So, there is a maximum number of outputs per unit of time, i.e., a maximum to the throughput. To allow for higher throughputs we need to consider algorithms that produce multiple outputs per state update. We call these algorithms block processing algorithms, because they do not process a single input and output at a time, but blocks containing $L$ of them.

This means that the inputs and outputs of the algorithm are vectors of length $L$. So the input and output alphabets of the FSM become:

$$
\Sigma_{alg} = \Sigma_{spec}^L \\
\Gamma_{alg} = \Gamma_{spec}^L
$$

We still use (1.11) to specify functional correctness, but now we use a mapping of the streams of the block processing systems onto the single input-output systems. Because the streams of the block processing system consists of blocks we use two indices to indicate a single element of the stream, namely the index of the block and the index of the element inside that block. The streams then map onto each other in the following manner:

$$
in(i)(j) = in(Li + j) \\
out(i)(j) = out(Li + j)
$$

for every $i \in \mathbb{N}$ and $0 \leq j < L$. Note that we differentiate between the stream of blocks and the stream of single values by using a bold font for the stream of blocks.

Using these definitions we now introduce the performance measures of the algorithm.
1.3.3 Latency

Latency is the time between the production of an output and the arrival of the last input on which that output depends. We assume that the latency of the specification FSM is minimal, i.e. zero. Therefore the $L$ from (1.11) is the latency of the algorithm measured, not in time units, but in the number of outputs. Once a sample rate or throughput is available the latency can be converted to an actual time. In this thesis, however, we measure the latency in the number of outputs.

For most block processing algorithms the latency is a constant number of blocks. This means that the latency in terms of number of outputs depends on the block size $L$. So, as a block processing algorithm is scaled to achieve a higher throughput, the latency tends to increase at the same, or higher, rate as the block size. This is a drawback of the block processing algorithm, but some kind of trade-off has to be made to achieve high throughput.

1.3.4 Memory

Algorithms are designed with a set of data-types $D$ in mind. This set is chosen, or given, based on the hardware platform on which the algorithm will be implemented. For low-level VLSI design there may be only one data-type, i.e. the boolean ($D = \{\mathbb{B}\}$), but usually the data-types consist of the booleans, and subsets of the integers and reals.

Note, that the data-types are all finite. It is often convenient to ignore this during algorithm design, but when the algorithm is implemented in hardware the range and/or precision of integers and reals is limited.

The inputs and outputs of the specification FSM are of one of the data types, i.e. $\Sigma_{\text{spec}} \in D$ and $\Gamma_{\text{spec}} \in D$, or they are tuples of one or more data types. This last case is used in, for example, VLSI designs where the only data type is $\mathbb{B}$, but where the input is a vector of bits representing a number.

More importantly, these data-types are also used to define variables, or registers, which specify the state-space of the FSM representing the algorithm. This state-space then has the following form for some $\mathcal{M} \in \mathbb{N}$ and $d_i \in D$ for $0 \leq i < \mathcal{M}$:

$$S_{\text{alg}} = (\times i : 0 \leq i < \mathcal{M} : d_i) \quad (1.16)$$

Note that this is one of the extra annotations required for our algorithm representation: the way in which the state-space is divided into registers.

The state therefore consists of $\mathcal{M}$ memory locations, or registers, where register $i$, for $0 \leq i < \mathcal{M}$, can hold a value from the data type $d_i$. The number
\( \mathcal{M} \) is then used to measure the memory requirements of the algorithm in terms of number of registers.

In the next section we show how the state update function of the FSM can be represented using basic, simple operations. These basic operations can compute results that have to be stored in temporarily. Therefore the memory of the system is sometimes divided into a number of inter-iteration (\( \mathcal{M}_s \in \mathbb{N} \)) and intra-iteration (\( \mathcal{M}_t \in \mathbb{N} \)) registers such that \( \mathcal{M} = \mathcal{M}_s + \mathcal{M}_t \).

The inter-iteration registers contain the data that has to be stored between the state updates of the FSM, i.e., these registers represent, in some manner, the state of the FSM specifying the stream processing system. The intra-iteration registers contain the data that is used to compute a single state update, i.e., these registers represent the temporary variables in an algorithm. Therefore the data in the inter-iteration registers needs to be stored between iterations, while the data in the intra-iteration registers needs only be stored during a single iteration.

The distinction between inter- and intra-iteration registers can be important when implementing the algorithm, as discussed in Section 1.4.2.3, but for now we treat all memory locations equally.

### 1.3.5 Complexity

The complexity of an algorithm specifies the number of operations it performs during execution. For stream processing system this poses a problem, since we have infinite streams and therefore require an infinite number of operations. So we use the complexity per state update, or complexity per block instead, i.e., the number of \( \mathcal{O}(1) \) operations required to produce one block of outputs.

To count these operations we must first define them. Just as for data-types, algorithms are designed with a specific set of \( \mathcal{O}(1) \) operations \( \mathcal{O} \) in mind. Again, this set is chosen, or given, based on the hardware platform on which the algorithm will be implemented.

For low-level VLSI design there may be only some gate operations plus the identity function (to represent a wire), i.e. \( \mathcal{O} = \{ \text{id, and, or, not} \} \), but usually the standard arithmetic and logical operators are available.

The operations in the set \( \mathcal{O} \) are functions accepting one or more arguments with data-types from \( \mathcal{D} \), and producing one or more results which also have a data-type from \( \mathcal{D} \). These operations are used to describe the algorithm in a step by step manner. We represent each single step of the algorithm by a basic state update function, i.e., a function that describes the way in which the state of the system changes after executing the step of the algorithm. Formally, these
basic state update functions (basic SUFs) are functions that take the value of
the appropriate number of registers from the state, apply one operation from
$O$, and store the produced result(s) in one or more registers of the state.

To define basic SUFs precisely we use the function \texttt{select}, which selects a
sub-list from a list or vector given a list of indices:

\begin{equation}
\texttt{select} : (\mathbb{L}(N) \times \mathbb{L}(A)) \to \mathbb{L}(A)
\end{equation}

\texttt{select}(i, l) = [i : i \in i : l(i)] \quad (1.17)

So, for example, select([0, 2, 3], [a, b, c, d]) = [a, c, d].

A basic SUF is a function $\delta_{w,o,r}$ that takes a state and input and produces
a new state, where $o \in O$ is an operation that takes $\#r$ inputs and produces
$\#w$ results. The vectors $r$ and $w$ contain, respectively, the memory addresses,
i.e. registers, from which the operands are read and to which the results are
written. The basic SUF then specifies a state update for a state $s \in S_{alg}$ and a
block of inputs $\sigma \in \Sigma_{alg}$, such that the memory locations $w$ are updated with
the results of the application of operation $o$ to the values stored in the memory
locations specified by $r$:

\begin{equation}
\texttt{select}(w, \delta_{w,o,r}(s, \sigma)) = o(\texttt{select}(r, s + \sigma)) \quad (1.18)
\end{equation}

Furthermore, the basic SUF does not update any other memory locations. So, if $u$ contains the memory locations that are unaffected by the operation, i.e.
$u = [i : 0 \leq i < M \land i \notin w : i]$, then we have:

\begin{equation}
\texttt{select}(u, \delta_{w,o,r}(s, \sigma)) = \texttt{select}(u, s) \quad (1.19)
\end{equation}

For example: $\delta_{[0]+,[1,2]}$ represents the basic SUF that reads two values from
memory locations 1 and 2, adds them together and writes the result to memory
location 0. So, if $M = 2$ and $L = 4$ we have $\delta_{[0]+,[1,2]}([3, 5], [7, 11, 13, 17]) = [12, 5]$. Note that the block of inputs is simply concatenated with the memory
so that a memory locations $M$ and higher are mapped onto the block of inputs.

In summary, a basic SUF represents a simple operation, namely a read from
a register, applying some kind of basic $O(1)$ operation from $O$ and writing the
results back to a register. These basic SUFs can be used to model an instruction
executed on a processor, or a simple component in a VLSI circuit.

By combining these basic SUFs, using function composition, it is possible
to form the SUF of the FSM representing the block processing algorithm. Note
that the function composition for SUFs is done in the following manner:

\begin{equation}
(\delta_0 \circ \delta_1)(s, \sigma) = \delta_0(\delta_1(s, \sigma), \sigma) \quad (1.20)
\end{equation}
1.3. ALGORITHMS

These basic SUFs are combined using the function composition of (1.20) to form $\delta_{alg}$, i.e., the state update function of the algorithm. This composition is represented by a binary tree. Each node in this tree represents a SUF. The leaves represent basic SUFs and all other nodes represent the SUF obtained by applying function composition according to (1.20) on its children. The root of the SUF tree then represents $\delta_{alg}$. The complexity of the SUF can be determined by simply counting the number of leaves of the SUF tree, i.e., counting the number of basic SUFs and thus the number of basic operations required per state update.

To make sure that we only have to analyze the SUF to find the complexity of the algorithm, we demand that the output function does not perform any operations. In other words, the output function $\omega$ is a simple projection function that selects registers from the state and uses their values as an output. More formally:

$$\omega_{alg}(s, in) = select(l, s)$$ (1.21)

for some list $l$ with $0 \leq l(i) < M$ for all $0 \leq i < L$.

Since the way in which $\delta_{alg}$ can be composed from basic SUFs is not always obvious, we consider the SUF tree as part of the extra annotation for our algorithm representation. In fact, this SUF tree is generally the most important part of the algorithm, as its leaves correspond directly to statements in a piece of pseudo-code, to the blocks in a block diagram or to the edges in a signal flow graph.

With these properties of the FSM, we define the complexity per block ($C$) as the number of leaves in the SUF tree, i.e., the number of operations required to compute one block of outputs.

1.3.6 Iteration period and throughput

Next to memory requirements, latency and complexity we are also interested in the throughput of the algorithm. The throughput is the number of outputs produced in a certain time unit. The FSM produces $L$ outputs with each state update, so the throughput equals $L$ divided by the time it takes to perform a state update. The time required to perform a state update is known as the iteration period.

To compute the iteration period we need more information. First of all, we need to attach a time cost to all of the operations in $O$. For sake of simplicity we assume that all the basic operations have a cost of 1 time unit, though a more detailed computation is possible if necessary. Secondly, we need to annotate the
SUFS tree by tagging each function composition with a label. This label indicates whether the composition is “sequential” or “parallel”. A sequential composition is denoted as $\delta_1 \circ \delta_0$, which indicates that the operations specified by SUF $\delta_0$ are performed before the operations specified by $\delta_1$. A parallel composition is denoted as $\delta_1 \circ || \delta_0$, which indicates that the operations specified by SUFs $\delta_0$ and $\delta_1$ are performed simultaneously.

Note that these labels do not change the way SUFS are composed. The composition still works as specified in (1.20), the labels are simply extra annotation that is used to compute the iteration period and throughput of the algorithm.

There are rules to labeling the function compositions. Any function composition can be labeled as sequential, but a function composition that is labeled as parallel has to adhere to certain properties. Simply put, the operations in $\delta_0$ and $\delta_1$ in the function composition $\delta_0 \circ || \delta_1$ should not interfere with each other.

More formally, when both functions are basic SUFS, i.e. $\delta_0 = \delta_{w_0,o_0,r_0}$ and $\delta_1 = \delta_{w_1,o_1,r_1}$, the composition $\delta_0 \circ \delta_1$ may be labeled as a parallel if the following holds:

\[
(\forall i : i \in w_0 : i \not\in w_1 \land i \not\in r_1) \quad (1.22)
\]

\[
(\forall i : i \in w_1 : i \not\in w_0 \land i \not\in r_0) \quad (1.23)
\]

meaning that neither of the basic SUFS writes to a register that is used by the other basic SUF. This is to make sure that any interleaved execution of the basic SUFS results in the same state update. In other words, a composition labeled as parallel means that the composition is commutative:

\[
\delta_0 \circ \delta_1 = \delta_1 \circ \delta_0 \quad (1.24)
\]

However (1.24) is only sufficient when the basic SUFS represent atomic actions on the hardware platform. This need not be the case, a classic example of this is the parallel execution of two $x:=x+1$ operations, which, on many architectures, can result in increasing the value of variable $x$ by either 1 or 2. Therefore we use (1.22) and (1.23) to verify whether a parallel label is allowed.

There is also the case where either or both of the two functions $\delta_0$ and $\delta_1$ are not a basic SUF, but a composite function like so:

\[
\delta_0 = (\circ i : 0 \leq i < N_0 : \delta_{0,i}) \quad (1.25)
\]

\[
\delta_1 = (\circ i : 0 \leq i < N_1 : \delta_{1,i}) \quad (1.26)
\]

with $3 \leq N_0 + N_1 \leq 4$. This means that $\delta_0$ is composed of two SUFS ($N_0 + N_1 = 3$), or $\delta_1$ is composed of two SUFS ($N_0 + N_1 = 3$), or both $\delta_0$ and $\delta_1$ are composites ($N_0 + N_1 = 4$).
In such cases the composition $\delta_0 \circ \delta_1$ may be labeled as a parallel if the following recursive predicate holds:

$$(\forall i, j : 0 \leq i < N_0 \land 0 \leq j < N_1 : \delta_{0,i} \circ \delta_{1,j} \text{ may be labeled as parallel}) \quad (1.27)$$

In other words it should also be possible to label any combination of the composing functions from either function as parallel. The recursion stops there if the composing functions are basic SUFs, otherwise the recursion continues all the way to the basic SUFs. Note that this specification essentially tells us that any possible execution order of the basic SUFs has to be checked to see that (1.22) and (1.23) are not violated. Checking the parallel composition in this manner is not feasible in practice, but other techniques exist, like those discussed in [22].

After the labeling, we can calculate the time it takes to perform one state update, i.e., the iteration period $I$. The time for a SUF that is composed using a parallel composition equals the maximum of the times of its composing functions. The time for a SUF that is composed using a sequential composition equals the sum of the times of its composing functions. The time for a SUF that is non-composite, i.e. a basic SUF, equals the time of the operation it represents. So by assigning a value to the leaves of the SUF tree and applying either a maximum or addition operator at each node, depending on the label, we obtain the iteration period $I$ of the SUF.

The iteration period is used to obtain the throughput of the algorithm; $L$ outputs are produced in $I$ time units, so the throughput $T$ in outputs per time unit is:

$$T = \frac{L}{I} \quad (1.28)$$

### 1.3.7 Summary

To summarize, an algorithm specifies the way in which the outputs of a system are computed. In this section we use an FSM to represent the algorithm, together with extra annotation. This extra annotation consists of a block size $L$, a set of data types $D$, a set of operations $O$ and a binary tree representing the construction of the state update function of the FSM from basic operations. Furthermore, the state of the FSM is a tuple, with each element representing a register capable of containing the value of one data type. Additionally, the SUF tree is annotated to indicate the operations that are to be performed in parallel and which operations are to be performed sequentially.

From this representation it is possible to derive several cost and/or performance measures of a system based on that algorithm. First of all there is latency
(\mathcal{L})$, which is measured in the number of outputs. Secondly we have memory requirement (\mathcal{M}), which is the number of registers required by the algorithm. Thirdly there is the complexity (\mathcal{C}), which, for stream processing systems, measures the number of operations per block of outputs. And lastly, there are the iteration period (\mathcal{I}) in time units and the throughput (\mathcal{T}) in number of outputs per time unit.

1.4 Hardware

Algorithms are generally designed with a specific hardware platform in mind. This is reflected by the set of data-types and operations that are used in the algorithm, but also in terms of extra demands on the way these operations are used to form the algorithm. In this section we briefly discuss the different hardware platforms, their characteristics and their demands on the algorithms designed for them.

According to Flynn's taxonomy \cite{25} computer architectures can be divided into four categories, namely Single Instruction Single Data (SISD), Multiple Instruction Single Data (MISD), Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD). Next to implementing algorithms on a computer there is also the possibility to implement it in dedicated hardware, i.e. as a VLSI system, which we also consider.

Systems which we do not consider are the MISD type, since are virtually unheard of, and the MIMD type of systems. MIMD systems receive a lot of attention but are generally more suited for large, complex, computations where processors can do a significant amount of work before communication between them is necessary, while the stream processing systems that we consider perform relatively simple computations on a large amount of data. Besides, both VLSI and SIMD architectures can be seen as special cases of the MIMD architecture and it is relatively easy to port algorithms designed for SIMD systems to MIMD systems, albeit with some overhead.

1.4.1 SISD

The Single Instruction Single Data (SISD) processor is the common sequential processor. The processor executes the algorithm by performing each operation in order, which means that the SUF compositions are all labeled as sequential.

The consequence of this is that the complexity and the iteration period such an algorithm are the same. Since the complexity of an algorithm is at least
linear in $L$, i.e. at least 1 operation per output is needed, this means that the throughput is some small number. On the bright side the latency is generally minimal, since it cannot be traded to improve any of the other measures.

We mention this type of hardware platform because it is very common and designing algorithms for these processors has already been studied for a long time. Without parallelism, however, the throughput is limited, therefore this platform is only of interest to us as a reference for performance.

In contrast to the SISD processor, the other two hardware platforms that we consider do allow for parallelism, which is why we focus on them in this thesis.

### 1.4.2 VLSI

The first type of platform that we focus on is dedicated hardware, or VLSI circuits. This platform is interesting because the performance requirements of many of the systems we consider necessitate an implementation in dedicated hardware. Furthermore, the emergence of Field Programmable Gate Arrays (FPGAs) has made VLSI circuits available for a wide range of applications.

#### 1.4.2.1 Restrictions

In VLSI systems there are no restrictions on which operations can be executed in parallel, other than the ones mentioned earlier in Section 1.3.6. This means that any algorithm can be implemented as a VLSI circuit, even if it was not designed for it. However, because of the way in which we implement such algorithm we usually try to label as much compositions as possible as parallel and try to reduce the iteration period of the algorithm to a minimum, i.e., one operation.

The reduction of the iteration period, to as far as one operation, is usually achieved by a technique known as pipelining. Pipelining works by adding so-called pipelining registers to the hardware. These registers are placed such that the cycle time of the circuit is reduced, i.e. between two sequential components, but such that the functionality remains the same. An easy way of pipelining a circuit while ensuring that the functionality remains the same is cut-set retiming \[65\]. Note that decrease in cycle time, and therefore increase in throughput, comes at the cost of increased memory usage and increased latency.

#### 1.4.2.2 Hardware cost

An important measure for algorithms implemented in this manner is the amount of processing hardware needed. To express this measure in terms of a number of
CHAPTER 1. INTRODUCTION

gates or as an area requires that the algorithm is actually implemented. We can also estimate the amount of hardware, however, by looking at the measures of the algorithm. The amount of processing hardware required to implement the algorithm can be estimated by considering the complexity $C$ of the algorithm, and its iteration period $T$.

In the most simple case the algorithm is implemented by instantiating a hardware component for every basic SUF in the SUF tree. This results in a amount of hardware of $\Theta(C)$, since we need $C$ operations to compute a single block of outputs\(^4\). It is also possible to reduce this number by employing techniques like time-multiplexing the hardware. Ideally this can reduce the amount of hardware needed to $\Omega(C/T)$, since we need only $C/T$ operations per unit of time. So, either we instantiate a hardware component for every operation per block of outputs and reuse the hardware for each block, or we instantiate a hardware component for every operation per time unit and reuse the hardware each time unit. These two options result in an upper and a lower bound of $O(C)$ and $\Omega(C/T)$ respectively for the amount of hardware.

In this thesis we only present VLSI algorithms that have a constant iteration period, i.e., $T \in \Theta(1)$. Therefore the upper and lower bound for the amount of hardware coincide, resulting in an amount of hardware that is always $\Theta(C)$.

1.4.2.3 Sequential compositions

Note that, when an algorithm is implemented by instantiating a hardware component for each basic SUF in the SUF tree, there are two options for implementing a sequential composition $\delta_1 \circ \delta_0$.

First, a sequential composition can mark the boundary between clock cycles, such that the components corresponding to $\delta_0$ operate in one (or more) clock cycles first and the components corresponding to $\delta_1$ operate in one (or more) clock cycles after that. If this method of implementation is used for all sequential compositions, then the iteration period $T$ of the algorithm will correspond to the number of clock cycles needed by the VLSI implementation to compute one block of outputs.

Second, the sequential composition can be implemented such that $\delta_0$ and $\delta_1$ operate in the same clock cycle, provided that the registers that are both read by $\delta_1$ and written by $\delta_0$ are intra-iteration registers. In such cases these intra-iteration registers can be replaced by simple wires. Furthermore, if this implementation is used for all sequential compositions, then the iteration pe-

\(^4\)Note that we use the $\Theta$, $O$ and $\Omega$ notations from [17], also explained in Appendix A.1.6.
1.4 HARDWARE

The period \( T \) of the algorithm will correspond to the clock cycle length and the VLSI implementation will produce a block of outputs every clock cycle.

In actual implementations a combination of both options is generally used. As mentioned before, however, the VLSI algorithms that we present in this thesis have a constant iteration period (\( T \in \Theta(1) \)). Therefore we will solely use the second option in implementing our VLSI algorithms, resulting in VLSI implementations that produce one block of \( L \) outputs per clock cycle, and each clock cycle will have a length of \( \Theta(1) \) time units.

1.4.3 SIMD

The second type of platform on which we focus in this thesis is the SIMD processor. There is renewed interest in using SIMD processors for signal processing tasks. Not only are there many SIMD extensions to general purpose processors (MMX [69], SSE [75], 3DNow! [62]), but using an SIMD processor as a co-processor in embedded systems has also gained interest [4, 47].

1.4.3.1 Restrictions and properties

Algorithms designed for this processor are similar to those for the SISD platform in the sense that many of the compositions are sequential. However, some parallel compositions are also allowed, namely on \( P \) basic SUFs that perform the same operation, but on different registers. Hence the name single instruction multiple data. The number \( P \) indicates the number of processing elements (PEs) of the SIMD processor. SIMD processors are also called vector processors, since they support operations on vectors of data in addition to the common SISD operations. Therefore the number \( P \) is also known as the vector length of the processor.

In many cases an algorithm for an SIMD processor is designed such that \( L = P \), but this need not be the case. In this thesis we also discuss examples where the block size is a multiple of the vector length, i.e. \( L = IP \), for some \( I \in \mathbb{N}^+ \). This so-called increment \( I \) is introduced so that some constant factors can be optimized [38]. Note that the increment \( I \) should not be confused with the iteration period \( T \).

The complexity of an SIMD algorithm is at most \( O(IP) \), namely when all the operations of the algorithm are vector operations. The complexity is at least \( \Omega(I) \), namely when the algorithm does not use any vector operations. Of course there is not much point in designing an SIMD algorithm that does not utilize vector operations. Therefore, in practice, a significant fraction of the operations
in an SIMD algorithm are vector operations, which means that the complexity is $\Theta(IP)$. When we look at the amount of hardware needed to implement an SIMD algorithm as a VLSI circuit, as described in Section 1.4.2 we get a minimum of $\Omega(\frac{C}{P}) = \Omega(\frac{IP}{P}) = \Omega(P)$. Note that this amount of hardware matches neatly with the amount of hardware needed to implement an SIMD processor with $P$ processing elements.

### 1.4.3.2 Supported operations

There is a large variety of SIMD processors, and most of them have additional limitations on the vector operations. In most cases, for example, the vector operations can only be performed when the vector is stored in memory such that elements that are adjacent in the vector are also in adjacent positions in the memory. In fact, this restriction is so common that we assume this requirement holds for all SIMD algorithms we design.

A more detailed discussion on the supported operations and restrictions on them for various SIMD processors can be found in Chapter 7. At this point we already want to note two operations, however, since they are used in this thesis before Chapter 7.

The first operation is called strided memory access. If an SIMD processor supports strided memory access then it is capable of reading and writing the elements of a vector from and to memory locations that have a certain distance between them. This distance is called a stride, and most SIMD processors only support a stride of one, i.e., vector elements must be stored in adjacent memory locations. An SIMD processor that supports strided memory access, however, supports any stride.

Strided memory access is a useful operation, but it is not available on modern SIMD processors. Old supercomputers that support vector operations generally performed them by pipelining the operation onto a single piece of hardware, which made implementation of the strided memory access easy. Modern SIMD processors, however, actually use $P$ PEs in parallel to perform vector operation, which makes implementing the strided memory access operation significantly harder. Though the strided memory access operation is not common, it is very useful and we discuss its use for block processing algorithms in this thesis.

The second operation used regularly in this thesis is the so-called shuffle operation. Most SIMD processor are limited in the way they can rearrange the elements of a vector. Generally only shift or rotate operations are possible, which move or rotate all the elements of a vector in one direction over the PEs.
1.5. SCALABILITY

SIMD processors that support the shuffle operation, however, are capable of permuting the elements of the vector in any possible way. This is a very useful operation that has been implemented on both the AltiVec [26] and the EVP [4].

Without these special operations, SIMD processors are one of the most restricted forms of parallel processing. This makes it interesting to see what kind of performance can be obtained on SIMD processors without these operations, and for which systems such operations result in better, or more specifically, scalable algorithms.

1.5 Scalability

In this thesis we are not concerned with fixed algorithms, but rather with algorithms that are parameterized by the block size $L$. Altering the block size of an algorithm affects its performance measures. A larger block size can lead to a higher throughput, but is also likely to lead to a higher complexity, latency, memory usage and a larger amount of processing hardware. We are interested in the way these measures affect each other, in particular the way in which the throughput of the algorithm scales with respect to the amount of processing hardware.

Ideally the throughput of the algorithm should be proportional the amount of hardware used to implement it, i.e. its cost, so that we can obtain any throughput by using a proportional amount of hardware.

Note that the scalability of an algorithm can also be used to optimize other system characteristics. A scalable SIMD algorithm, for example, can be implemented with the same throughput on two different processors: a fast processor with a small $P$, or a slow processor with a large $P$. A slow processor is generally cheaper and more energy efficient than a faster processor. In this way the scalability can be used to optimize cost or energy efficiency, instead of throughput.

1.5.1 Scalability classes

To measure the scalability of an algorithm we express the throughput as a function of the amount of processing hardware. So, in case of a VLSI algorithm the throughput $T$ is expressed as a function of the complexity $C$, and in case of an SIMD algorithm the throughput $T$ is expressed as a function of the number of PEs $P$.

The domain of such a function is unlikely to be continuous, especially in the case of VLSI algorithms. After all, allowing for only one extra operation per
block, for example, is not likely to result in a greater throughput. Despite this we can still gather important information from the function. In particular we can use it to divide the scalability of the algorithm into one of two categories.

The first category contains the algorithms that do not scale. The throughput of algorithms in this category cannot be increased, no matter the amount of hardware that is used. The scalability function, i.e., the function expressing the throughput in terms of the amount of processing hardware, of such an algorithm has a maximum. In many cases the scalability function for such an algorithm is only defined for a single point, namely for the amount of hardware in which the algorithm can be implemented.

The second, and more interesting, category is formed by those algorithms that do scale. The throughput of algorithms in this category can be increased, given enough hardware. For such algorithms the scalability function has no maximum. We divide this category of scalable algorithms into two sub-categories.

The first of these sub-categories is formed by those that scale linearly. For these algorithms the scalability function is a linear function of the amount of hardware. For VLSI algorithms this means that $T \in \Theta(C)$, and for SIMD algorithms this means that $T \in \Theta(P)$. For such algorithms the cost in terms of processing hardware grows linearly with the throughput. In other words, to double the throughput of such an algorithm we simply double the amount of hardware used in the implementation. Considering that the number of transistors on a chip is growing exponentially, this class of algorithms is very interesting.

Algorithms with linear scalability have an interesting property, namely that the complexity per output is constant, i.e., $\frac{C}{T} \in \Theta(1)$. We derive this property taking the scalability function and using $T = \frac{L}{P}$. For SIMD algorithms the scalability function, for algorithms that scale linearly, is $\Theta(P)$, i.e., $T \in \Theta(P)$. This means that $T = \frac{L}{P} \in \Theta(P)$, and implies that $L \in \Theta(IP)$. For SIMD algorithms we also known that $C \in \Theta(IP)$, therefore the complexity per output is constant, i.e. $\frac{C}{T} \in \Theta(1)$. In a similar manner we derive the property for VLSI algorithms; there the scalability function for algorithms that scale linearly specifies $T \in \Theta(C)$. This means that $\frac{L}{T} \in \Theta(C)$, and we known that $I \in \Theta(1)$ for VLSI algorithms in this thesis, therefore the ratio between complexity $C$ and block size $L$ is constant, i.e. $\frac{C}{T} \in \Theta(1)$. Note that this property can also be derived for VLSI algorithms with a larger iteration period, namely algorithms where the amount of hardware has been reduced to $\Omega(\frac{L}{T})$. So, instead of checking the throughput as a function of the amount of hardware to see if an algorithm has linear scalability, we can also compute the complexity per output. If the algorithm scales and its complexity per output is constant then the algorithm
1.5. SCALEABILITY

has linear scalability.

This complexity per output, which we express in operations per output (opo), is a good means of comparing individual block processing algorithms when they scale linearly. When multiple block processing algorithms exhibit linear scalability, we consider the one with the lowest complexity per output to be the best, since it requires the least amount of hardware to achieve a certain throughput.

The second sub-category of algorithms is formed by those that have super-linear scalability. These algorithms can achieve any throughput by increasing the amount of hardware, but the law of diminishing returns makes high throughputs unfeasible. In such cases it is often interesting to examine the scalability function in more detail to see how fast the cost grow with the throughput.

1.5.2 Realizing linear scalability

The algorithms that scale linearly are the most interesting. These algorithms allow us to achieve any throughput with a proportional cost in terms of processing hardware. At least, that is the theory, but can these algorithms be realized in practice?

The answer is that this is possible, if one is willing to accept the other costs like latency and memory, which probably do not scale linearly. The throughput may be limited by the I/O bandwidth though, since the implementation needs L inputs and outputs every iteration period. Exactly how this I/O bandwidth is realized is beyond the scope of this thesis, but some options are to have enough pins on the chip to transfer each element of the block individually, or a serial-to-parallel and parallel-to-serial component that converts the sequential input stream to a stream of blocks and the output stream of blocks into a sequential output stream. In any case, it is quite pointless to require a throughput that is higher than the maximum realizable bandwidth.

The only problem with our algorithms could be posed by the wire length. The model we use for our algorithms does not allow us to easily define the length of the wires in the implementation. Long wires introduce additional delays, and may therefore influence the clock frequency/iteration period of the implementation. For relatively small block sizes, and therefore small amount of hardware, however, the delay introduced by long wires will be small compared to the delay of the processing elements. Furthermore, the pipelining of many of the VLSI algorithms in this thesis results in algorithms with a wire length that is constant with regard to the block size and amount of hardware. The SIMD algorithms may form a problem, however. A simple SIMD processor that
only supports communication between neighboring PEs does not contain long wires, but an SIMD processor that supports the shuffle operation does. So, although the wire length does not influence the clock frequency (and therefore throughput) of an implementation for relatively small amounts of hardware, it should be examined when extremely large amounts of hardware are used.

1.6 Research Goals

The problem that is discussed and solved in this thesis is, in short, the problem of obtaining high-throughput stream processing systems using slow hardware components. In particular we obtain implementations such that the throughput is proportional to the amount of hardware used, i.e., the implementations scale linearly.

Linear scalability is, of course, relatively easy to obtain for certain stream processing systems. Stream processing systems from class $FSM_*$ compute their outputs based on a finite number of inputs, therefore multiple outputs can be computed simultaneously by having each piece of hardware work on a single output in parallel.

There are other stream processing systems, however, that pose more of a challenge. Systems from class $FSM_\infty$ pose a problem, since they require all preceding inputs to compute a single output. Conceptually this means that the computations for the outputs of an $FSM_\infty$ system become more complex after each output computation. In practice, however, parts of the computation of the previous output are stored in the state of the system and reused in the computation of the next output. This ensures that the complexity of the system does not grow out of control, but also introduces a dependency between the output computations that makes it hard to compute them in parallel which, in turn, makes it hard to design a scalable block processing algorithm for the system.

In the implementation of many $FSM_*$ systems a similar construction is used to reduce the complexity. Take a system that computes the moving sum, i.e., a system whose output is the sum of the last $W$ inputs. The moving sum system belongs to the $FSM_*$ class, and can therefore be implemented with linear scalability by summing $W$ inputs for every output. In practice, however, such a system is implemented by taking the previous output, adding a new input and subtracting an old input. This latter implementation has a constant complexity per output, while the implementation with linear scalability has a complexity of $W$ per output. The SISD implementation, however, has
1.6. RESEARCH GOALS

dependencies between its outputs that makes it hard to turn it into a scalable block processing algorithm, just like systems of the $FSM_{\infty}$ class.

This is why we are interested in the $FSM_{\odot}$ class of systems. These systems form a sub-class of the $FSM_*$ systems, and the output computation is a simple aggregation operation. This makes it relatively easy to examine the reuse between outputs for such a system, and therefore these systems form an excellent starting point for our research.

This starting point leads us, after first examining alternatives in the literature, to the results in this thesis. These results are summarized in the following list:

- An overview of the existing methods for obtaining scalable block processing algorithms. This overview shows that certain methods resulting in VLSI implementations exist, but none of them are directly suited for SIMD processors.

- The first method for designing scalable block processing systems for any stream processing system given scalable block processing algorithms for $FSM_{\odot}$-class systems.

- The first generic approach for designing scalable block processing algorithms for $FSM_{\odot}$-class systems. We also show how several existing algorithms fit in the notation used by our approach.

- A block processing algorithm, called the maximum reuse algorithm, based on our generic approach for $FSM_{\odot}$-class systems in general. The maximum reuse algorithm is the first algorithm that scales linearly and that is also suited for implementation on currently available SIMD processors.

- An efficient, scalable block processing algorithm for rank order filters (an $FSM_{\odot}$ system), illustrating how our generic approach can be adapted to take system-specific features into account to result in better algorithms for a specific system. The resulting filtering algorithms are found to have better performance characteristics than existing rank order filter algorithms.

- An overview of SIMD processors features that are necessary to implement a scalable block processing algorithm.

- The first method for implementing some of the existing VLSI implementations on an SIMD processor while maintaining linear scalability.
A method for implementing the algorithms designed by our generic approach, both in VLSI and on an SIMD processor. This method therefore complements the available implementation methods of stream processing system in VLSI, and it introduces scalable implementations of stream processing systems to currently available SIMD processors.

1.7 Thesis outline

Chapter 2 of this thesis gives an overview of existing techniques and design methods for obtaining scalable block processing algorithms for stream processing systems. Although none of the techniques result in an algorithm suited for SIMD, they do result in at least a VLSI algorithm, which can form a valuable first step towards an SIMD algorithm. These techniques are then discussed in more detail in Chapter 3 where they are applied to a simple, yet interesting, example system, namely one that computes a moving sum.

In Chapter 4 we introduce associative window systems, i.e. systems from the \( FSM^\odot \) class, in more detail. Chapter 4 also shows a method for reducing the problem of designing a block processing algorithm for any system to the problem of designing a block processing algorithm for an associative window computation. Since window computations are embarrassingly parallel, designing a block processing algorithm for them is rather trivial. Obtaining an efficient algorithm, however, is not.

Therefore the focus of Chapter 5 is on designing efficient algorithms for window computations with a constant-cost operator. The techniques presented in that chapter can be applied to most window computations, and therefore to most of our design problems. However, it is also possible that the operator has a non-constant cost, i.e., the cost depends on the value of the operands. An example of such a system, namely rank order filters, is studied in Chapter 6.

Once an algorithm has been designed it still has to be implemented. Chapter 7 describes this final step in the design of the system. The chapter presents the results of applying our methods to practical problems like IIR and rank order filters, and implementing them on an existing hardware platform, namely the EVP. This chapter also presents a conversion method to convert some of the existing VLSI algorithms to SIMD algorithms.

Finally, Chapter 8 concludes this thesis, though the reader may also wish to refer to Appendix A for a summary of common terms and notations.
Chapter 2

Existing methods

2.1 Introduction

In this chapter we present an overview of existing methods to design scalable block processing algorithms for $FSM_{\infty}$ systems. These methods from the literature are, for the most part, based on so-called look-ahead computations, also known as block pre-computations.

Look-ahead computations are named such because these computations are used to advance the state of a system by $L$ steps, in contrast to a state update equation, which is used to advance the state of an algorithm by a single step. These computations therefore look multiple steps ahead. More formally, a look-ahead computation results in the state of the system $s(L(i + 1))$, when using $s(Li)$ and $in(i)$ as inputs\(^1\). Ideally the look-ahead computation has a lower complexity than $L$ successive applications of the state update equation, enabling us to design a scalable block processing algorithm around it.

The look-ahead computations are based on the algebraic state update equation, which we discuss in Section 2.2 of this chapter. In Section 2.3 we discuss the pipeline structure, which is heavily used in the so-called linear look-ahead computation from Section 2.4 and the block-post computation method from Section 2.7. Next to the linear look-ahead technique we also discuss the logarithmic look-ahead in Section 2.5 and the parallel prefix look-ahead in Section 2.6 before concluding this chapter in Section 2.8.

\(^1\)Technically such a computation with $L = 1$ is also a look-ahead computation, but not a very interesting one. Therefore we generally assume that $L > 1$. 
2.2 Algebraic state update equation

To apply any of the look-ahead computations discussed in this chapter, and to obtain a block processing algorithm, the state update equation of the system must be written in a specific form. We refer to this form as the algebraic state update function and discuss its properties in this section.

Definition 2.2.1 (Algebraic state update equation). The algebraic state update equation, as presented by Fettweis et al. in [23], is:

\[ s(i + 1) = (a(i) \otimes s(i)) \oplus b(i) \text{ for } i \in \mathbb{N} \]  

(2.1)

where \((\mathbb{D}, \otimes)\) and \((\mathbb{D}, \oplus)\) are semi-groups, i.e., the set \(\mathbb{D}\) is closed under the binary operations and the operations are associative. Furthermore, \(\otimes\) must be left distributive over \(\oplus\), i.e.:

\[ (\forall a, b, c : a, b, c \in \mathbb{D} : a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)) \]  

(2.2)

Note that the specification of an entire system consists of more than a single state update equation. The specification also contains equations expressing \(a(i)\) and \(b(i)\) in terms of the inputs \(\text{in}(i)\) of the system and an equation that expresses the output \(\text{out}(i)\) in terms of the state \(s(i + 1)\), i.e., \(\omega\). It is the recursive state update equation, however, that causes problems in parallelizing the system. Any non-recursive equations, specifying either \(a\) or \(b\) in terms of the input, or the output function \(\omega\) that specifies the outputs in terms of \(s\), can be modelled as an \(FSM^*\) system and implemented at any throughput using block processing\(^2\).

It is possible to restructure any FSM in such a manner that it adheres to the algebraic state update form [64]. This process of restructuring is based on the idea that \(\otimes\) and \(\oplus\) can stand for relatively complicated operations. In Section 2.2.1 it is shown that these operations can stand for matrix multiplication and addition respectively. In Sections 2.2.2 and 2.2.3 we use this to restructure higher order recursions and arbitrary FSMs into the form of the algebraic state update equation.

During the restructuring of the state update equation we have to keep an eye on the complexity of the \(\otimes\) and \(\oplus\) operations, since the complexity of the look-ahead computations, presented later in this chapter, are expressed in terms these operations. Therefore the complexities of these operations can have a large impact on the complexity of the look-ahead computations.

\(^2\)Methods for designing scalable block processing algorithms for \(FSM^*\) systems are discussed in Chapter 3.
2.2. ALGEBRAIC STATE UPDATE EQUATION

2.2.1 Matrices

Equation (2.1) can express a recursion on scalars, but $\otimes$ and $\oplus$ can also express matrix multiplications and additions respectively. This was shown by Fettweis et al. in [23], which we summarize here.

The paper [23] shows that for any given semi-ring $(\tilde{D}, \odot, \oslash)$ the semi-ring $(D, \oplus, \otimes)$ can be formed where $D$ is the set of matrices with elements from $\tilde{D}$ and the matrix operations are defined in the usual way:

\[
(A \otimes B)_{i,j} = (\odot k :: A_{i,k} \oslash B_{k,j}) \quad (2.3)
\]

\[
(A \oplus B)_{i,j} = A_{i,j} \oplus B_{i,j} \quad (2.4)
\]

A semi-ring $(D, \oplus, \otimes)$ has the property that $(D, \oplus)$ is a commutative semi-group, $(D, \otimes)$ is a semi-group and that $\otimes$ distributes over $\oplus$. Therefore, a recursion on matrices of the form (2.1) based on the semi-ring $(D, \oplus, \otimes)$ satisfies the requirements of the algebraic state update equation.

This construction of matrix operations can be used, for example, to model a Viterbi decoder as demonstrated in [24]. Consider the example trellis from [27] in Figure 2.1. Each element of the state vector $s(i)$ of the Viterbi decoder corresponds to a node in column $i$ of the trellis. The value of an element in the state vector represents the length of the shortest path leading to its corresponding node. In Figure 2.1 this length is represented by the number inside the node. To compute $s(i+1)$ we “multiply” the current state with a matrix containing the branch metrics shown next to the arrows in Figure 2.1. The matrix multiplication is done by using addition for $\oplus$ and the minimum operator for $\odot$. This results in a matrix multiplication where the multiplication of a row of a matrix with the state vector results in the basic “add, compare, select” operator of a Viterbi decoder. For our specific example of Figure 2.1.
this results in:

\[
\mathbf{s}(i + 1) = \mathbf{a}(i) \otimes \mathbf{s}(i) = \begin{pmatrix}
0 & 2 & \infty & \infty \\
\infty & \infty & 1 & 1 \\
2 & 0 & \infty & \infty \\
\infty & \infty & 1 & 1
\end{pmatrix} \otimes \begin{pmatrix}
2 \\
3 \\
2 \\
1
\end{pmatrix} = \begin{pmatrix}
2 \\
3 \\
2
\end{pmatrix}
\] (2.5)

We do need the unity element for the minimum operator here, i.e. \( \infty \), but it poses no problem to add it to \( \mathbb{D} \), or to use a suitable large value already in \( \mathbb{D} \) in its stead.

This approach increases the complexity of the \( \otimes \) and \( \oplus \) operators, since they no longer operate on scalar values. The complexity of the operators depends on the size of the matrices and vectors used in the equation, and the specific implementation used for the matrix multiplication. Multiplying two \( N \times N \) matrices may take \( \Theta(N^3) \) operations, though matrix multiplication operations with a complexity of \( O(N^{2.807}) \) [77] and \( O(N^{2.376}) \) [16] also exist.

Note that the matrices used in the matrix multiplication may have special properties that allow for a more efficient matrix multiplication. The property that the matrices for Viterbi decoding, like the one from (2.5), contain relatively many \( \infty \) symbols can be used to reduce the complexity of the matrix multiplication significantly. A “multiplication” with \( \infty \) requires no operation to implement, since \( \infty \) is the identity operator of our multiplication operator.

Such a property, however, can only be used in matrix multiplications where matrices are used that adhere to that property. Unfortunately the look-ahead computations, which we present later in this chapter, require matrix multiplications with matrices that are the product of several matrices. To reduce the complexity of the matrix multiplications used in a look-ahead computation, we can therefore only rely on properties that are maintained under matrix multiplication.

The property that the matrix contains many \( \infty \) symbols is not maintained under matrix multiplication. In each multiplication the number of \( \infty \) symbols can only decrease, resulting eventually in a matrix without any \( \infty \) symbols in it. An example of a property that \emph{is} maintained under matrix multiplication is a block diagonal form, i.e., a sparse matrix whose only non-zero values are contained in blocks around the diagonal of the matrix. The sparsity of such matrices allows us to implement the matrix multiplication with low complexity. Furthermore, the block diagonal form is maintained under matrix multiplication, allowing us to use the same low complexity implementation for all matrix multiplication in the look-ahead computation. We discuss this use of the block diagonal form in more detail in Section 3.2.1.1.
2.2. Higher order recursions

The construction of matrix operations from the previous section is also useful to turn higher order recursions into the algebraic state update form of (2.1). This was also presented in [23] and we summarize it here.

Consider a semi-ring \((\mathbb{D}, \oplus, \otimes)\) and an \(N\)-th order recursion:

\[
s'(i + 1) = (\oplus j : 0 \leq j < N : a'_j(i) \otimes s'(i - j)) \oplus b'(i)
\]  

(2.6)

We rewrite it by choosing, for example:

\[
s(i) = \begin{pmatrix}
  s'(i) \\
  s'(i - 1) \\
  \vdots \\
  s'(i - N + 1)
\end{pmatrix}
\]

\[
b(i) = \begin{pmatrix}
  b'(i) \\
  0 \\
  \vdots \\
  0
\end{pmatrix}
\]

(2.7)

\[
a(i) = \begin{pmatrix}
  a'_0(i) & a'_1(i) & \cdots & a'_{N-1}(i) \\
  1 & 0 & \cdots & 0 \\
  \vdots & \ddots & \ddots & \vdots \\
  0 & \cdots & 0 & 1 \\
\end{pmatrix}
\]

(2.8)

where 0 and 1 stand for the unity elements of the \(\oplus\) and \(\otimes\) operator respectively. A semi-ring \((\mathbb{D}, \oplus, \otimes)\) does not necessarily contain these identity elements, but it is always possible to expand the semi-ring in such a way that it does.

The definitions from (2.7) specify that the vector \(s(i)\) computed by (2.1) contains, as its first element, \(s'(i)\) from (2.6). Therefore this method allows us to rewrite a system specified by a higher order recursion to the algebraic state update form such that it computes the new state.

The state space form used to model linear filters [67] is, in fact, obtained when this construction method is used. A second order low-pass filter specified by, for example:

\[
\text{out}(i) = 1.14\text{out}(i-1) - 0.42\text{out}(i-2) + 0.07\text{in}(i) + 0.14\text{in}(i-1) + 0.07\text{in}(i-2)
\]

(2.8)

is written in the algebraic state update form by choosing, for example:

\[
a(i) = \begin{pmatrix}
  1.14 & -0.42 \\
  0 & 1
\end{pmatrix}
\]

\[
b(i) = \begin{pmatrix}
  0.07\text{in}(i) + 0.14\text{in}(i-1) + 0.07\text{in}(i-2) \\
  0
\end{pmatrix}
\]

(2.9)
CHAPTER 2. EXISTING METHODS

This choice corresponds to the so-called direct form I architecture for IIR filters [70]. Other choices are also possible, as discussed in Section 3.2.1. This approach increases the complexity of the \( \otimes \) and \( \oplus \) operators, since they now operate on matrices and vectors of size \( N \times N \) and \( N \) respectively, where \( N \) is the order of the recursion. The matrices used in (2.7) and (2.9) contain relatively many zeroes and ones, which allows us to implement multiplications with these matrices more efficiently than multiplications with arbitrary matrices. This property that the matrices contain many zeroes is, however, not maintained when the matrices are multiplied. Therefore we cannot rely on this property to reduce the complexity of our matrix multiplications when we implement the look-ahead computations presented later on in this chapter.

2.2.3 Arbitrary FSMs

Any FSM with \( N \) states can be restructured to adhere to the algebraic state update equation [64]. In this section we demonstrate this procedure. We make use of the matrix operations from Section 2.2.1 and define (\( \tilde{D} \), \( \boxplus \), \( \boxdot \)) in the following manner:

\[
\tilde{D} = \{0, 1\} \tag{2.10}
\]

\[
a \boxplus b = a + b \mod 2 \tag{2.11}
\]

\[
a \boxdot b = a \cdot b \mod 2 \tag{2.12}
\]

That is, the \( \boxplus \) operation is a XOR operation and the \( \boxdot \) is an AND operation.

The state \( s(i) \) is a vector of length \( N \) and it encodes the state of the FSM using one-hot encoding. This means that the FSM still has \( N \) possible states, but now they are encoded using the \( N \) vectors of length \( N \) that have a Hamming distance of 1 to the zero vector. In other words the state vector \( s(i) \) is a vector that contains a single one and \( N - 1 \) zeroes. So, when \( 0 \leq j < N \) and \( s(i)(j) = 1 \) holds, this means that the FSM is in the state \( j \) at time \( i \).

The state update function is written in the form of \( s(i+1) = (a(i) \otimes s(i)) \oplus b(i) \) by choosing \( b(i) \) to be the zero vector and by choosing \( a(i) \) to be a \( N \times N \) matrix according to:

\[
a(i)_{k,l} = \begin{cases} 1 & \text{if } \delta(l, \text{in}(i)) = k \\ 0 & \text{if } \delta(l, \text{in}(i)) \neq k \end{cases} \tag{2.13}
\]

In effect, the matrix \( a(i) \) is the transition matrix of the FSM for input \( \text{in}(i) \). This choice for the matrix \( a(i) \) ensures that our encoding of the state \( s(i) \) is mapped on the correct encoding of the state \( s(i + 1) = \delta(s(i), \text{in}(i)) \).
2.2. **ALGEBRAIC STATE UPDATE EQUATION**

As an example we have the FSM in Figure 2.2. This FSM computes a parity check of an input stream that consists of zeroes and ones. We can turn it into the algebraic state update form by choosing:

\[
a(i) = \begin{cases} 
\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \text{if } \text{in}(i) = 0 \\
\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} & \text{if } \text{in}(i) = 1 
\end{cases} \tag{2.14}
\]

Although it is useful to be able to write any FSM in the form of the algebraic state update equation, this method increases the complexity of the \( \otimes \) operator considerably. For many systems this representation is the best we can do, but in some cases we are able to find a representation that results in operators with a much lower complexity. One might notice, for example, that the FSM from Figure 2.2 can also be written in the algebraic state update form by choosing:

\[
s(i + 1) = s(i) \oplus \text{in}(i) \tag{2.15}
\]

where \( \oplus \) stands for modulo 2 addition. This latter choice is preferable because the \( \oplus \) operation used here is less complex than the \( \otimes \) operation needed when (2.14) is used.

In summary, the recursive state update function of any finite state machine can be written in the algebraic state update form of (2.1). This results in a significant increase in the complexity of the \( \otimes \) operators, since it has to operate on matrices and vectors of size \( N \times N \) where \( N \) is the number of states of the FSM. Therefore other, more efficient, ways of representing state update function of the system in the algebraic state update form are, when possible, preferred.
2.2.4 Note on finite precision arithmetic

Finite precision arithmetic plays a role when the look-ahead computations presented in rest of this chapter are implemented. The look-ahead computations based on the algebraic state update equation are valid mathematically, but an implementation on hardware that performs finite precision arithmetic produces problems. The cause of this problem is that operators like addition and multiplication lose their associativity when finite precision arithmetic effects, like saturation and round-off, come into play. These effects therefore violate our requirement that the operators in the algebraic state update form are associative.

Finite precision arithmetic is not always a problem. Linear time invariant (LTI) systems are generally designed with round off effects in mind. The rounding-off is modelled as noise inserted into the system and methods for analyzing noisy system are available [43, 58].

In LTI systems a certain level of noise is acceptable. There are systems, however, where no noise can be tolerated. A good example of this is arithmetic coding. Arithmetic coding is a data compression technique that achieves a compression rate near the Shannon limit. The mathematical equations specifying its operation are those of a linear, second order, time-varying system [63]:

\[ s(i + 1) = \begin{pmatrix} p(i, \text{in}(i)) & 0 \\ q(i, \text{in}(i)) & 1 \end{pmatrix} s(i) \]  \hspace{1cm} (2.16)

where \( p(i, j) \) is the probability of \( \text{in}(i) = j \) and \( q(i, j) \) is the cumulative probability, i.e., \( q(i, j) = \sum k : 0 \leq k < j \quad p(i, k) \). The compressed data for the input up until \( i \) is the shortest bit string representing a number between \( s(i)(0) \) and \( s(i)(0) + s(i)(1) \).

Any noise introduced in this system would destroy the possibility of decompressing the data. Therefore arithmetic coding standards, like the H.264 standard [56], specify explicitly what type of round-off must be used and at what point in the computation it should be used.

Unlike an ordinary multiplication, a multiplication that includes round-off is not associative. So, although (2.16) is in the form of Equation (2.1), it does not fulfill the associativity requirement of the algebraic state update form for practical arithmetic coding standards.

It is still possible to obtain a block processing algorithm for arithmetic coding by applying the methods of Section 2.2.3 or Section 2.7. These methods, however, have an overhead factor equal to the number of states of the FSM, which is very large compared to the algorithm that would be based on the \( 2 \times 2 \) matrix multiplications of (2.16).
Finite precision arithmetic effects can, therefore, have a considerable impact on the design of block processing algorithms. Fortunately, many systems of practical value are either tolerant to low levels of noise, like linear filters and Viterbi decoders, or they are based on finite precision arithmetic in the first place, like checksums and rank order filters.

### 2.3 Pipeline structures

The algebraic state update equation is one ingredient used in the design of scalable block processing algorithms. Another important ingredient, which we discuss in this section, is a pipeline structure. Pipeline structures are used in both the linear look-ahead method discussed in the next section and the block post-computation method discussed in Section 2.7. A pipeline structure is a number of components connected in series, separated by registers.

Any type of component can be used to construct a pipeline. In this section we use the $C_{δ,ω,1}$ components from Figure 2.3(a) as an example, because a pipeline structure based on these components is used for both the linear look-ahead computation and the block post-computation method.

A $C_{δ,ω,L}$ component is a component that takes $L$ inputs $\text{in}(i)$ and the current state $s(i)$, and produces the state $s(i + L)$ and the $L$ outputs $\text{out}(i)$. We construct such a $C_{δ,ω,L}$ component by using $C_{δ,ω,1}$ components, which take only one input $\text{in}(i)$ and the current state $s(i)$, and produce the next state $s(i + 1) = δ(s(i), \text{in}(i))$ and one output $\text{out}(i) = ω(s(i + 1))$. These $C_{δ,ω,1}$ components are placed in series in such a manner that the state produced by one component is used as the starting state of the next component in the series, as shown in Figure 2.3(b). By using $L$ of such components we obtain a $C_{δ,ω,L}$ component.

We assume that the complexity of a $C_{δ,ω,1}$ component is $Θ(1)$ and that it
CHAPTER 2. EXISTING METHODS

Figure 2.4: A non-scaling block processing algorithm

takes \( \Theta(1) \) time units for such a component to produce its outputs given its inputs. This means that the complexity of a \( C_{\delta, \omega, L} \) component is \( \Theta(L) \) and that it takes \( \Theta(L) \) time units for such a component to produce its outputs given its inputs.

A \( C_{\delta, \omega, L} \) component can be used to obtain a block processing algorithm, as shown in Figure 2.4. This block processing algorithm, however, does not scale. The computation steps represented by the \( C_{\delta, \omega, L} \) component cannot be performed in parallel, since there is a dependency between the computations of the state update function in each \( C_{\delta, \omega, 1} \) component. The state update function has to be evaluated sequentially \( L \) times for each block of inputs. This means that the iteration period, i.e. the time required to process a single block, is \( I \in \Theta(L) \). Therefore the throughput of the algorithm is \( T = \frac{I}{L} \in \Theta(1) \), since an increase in the size of the blocks causes a proportional increase in the computation time of the \( C_{\delta, \omega, L} \) component. In short, the algorithm based on the \( C_{\delta, \omega, L} \) component in Figure 2.4 does not scale, since we can increase the amount of hardware in the implementation by increasing the block size \( L \), but this does not result in any increase of the throughput.

To make the algorithm scalable the critical path of such a series can be reduced by pipelining it, as shown in Figure 2.5. A register is placed wherever a dashed line intersects a data path. Note this results in \( \Theta(L^2) \) registers, since there are \( L - 1 \) lines, or cut-sets, and each one intersects \( L + 1 \) data paths. We denote a component constructed in this manner by \( pC_{\delta, \omega, L} \), since it is a pipelined version of \( C_{\delta, \omega, L} \).

The pipeline registers in \( pC_{\delta, \omega, L} \) act as a buffer, delaying the data that travels along the arrow by one iteration or, equivalently, \( L \) sample periods. This pipelining breaks the dependency between the \( C_{\delta, \omega, 1} \) components, resulting in an iteration period that is independent of the block size \( L \). Therefore an algorithm based on a component \( pC_{\delta, \omega, L} \) achieves a throughput of \( \Theta(L) \) and scales linearly, at a price of \( \Theta(L^2) \) pipeline latches and a latency of \( \Theta(L^2) \) outputs.
Figure 2.5: Pipelined version of $C_{\delta,\omega,4}$ from Figure 2.3(b)

<table>
<thead>
<tr>
<th>VLSI circuit</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>$L$ (\Theta(L^2)) outputs</td>
</tr>
<tr>
<td>Memory</td>
<td>$M$ (\Theta(L^2)) registers</td>
</tr>
<tr>
<td>Complexity</td>
<td>$C$ (\Theta(L)) operations/block</td>
</tr>
<tr>
<td>Iteration period</td>
<td>$I$ (\Theta(1)) time units</td>
</tr>
<tr>
<td>Throughput</td>
<td>$T$ (\Theta(L)) outputs/time unit</td>
</tr>
<tr>
<td>Scalability</td>
<td>linear, (\Theta(1)) operations per output</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIMD program</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr. of PEs</td>
<td>$P$ (L = IP)</td>
</tr>
<tr>
<td>Latency</td>
<td>$L$ (\Theta(PL)) outputs</td>
</tr>
<tr>
<td>Memory</td>
<td>$M$ (\Theta(PL)) registers</td>
</tr>
<tr>
<td>Complexity</td>
<td>$C$ (\Theta(L)) operations/block</td>
</tr>
<tr>
<td>Iteration period</td>
<td>$I$ (\Theta(I)) time units</td>
</tr>
<tr>
<td>Throughput</td>
<td>$T$ (\Theta(P)) outputs/time unit</td>
</tr>
<tr>
<td>Scalability</td>
<td>linear, (\Theta(1)) operations per output</td>
</tr>
<tr>
<td>Remarks</td>
<td>Requires strided memory access</td>
</tr>
</tbody>
</table>

Table 2.1: Characteristics of $pC_{\delta,\omega,L}$
The performance characteristics of the $pC_{\delta,\omega,L}$ component are shown in Table 2.1. The SIMD program, and its characteristics, are obtained by emulating the VLSI circuit on the SIMD processor. In the SIMD program each PE of the SIMD processor emulates $I$ consecutive components of the pipeline. The details of this SIMD program are discussed in Section 7.3.1.

The $pC_{\delta,\omega,L}$ component, however, is not a block processing algorithm in itself. Apart from a block of inputs the component also needs a starting state before it is able to produce a block of outputs.

The component without pipelining, i.e. $C_{\delta,\omega,L}$, produces $s((i+1)L)$ in iteration $i$, which is exactly the state that is required in the next iteration of that component. The pipelined component $pC_{\delta,\omega,L}$, however, produces $s(L(i - L + 2))$, but it still requires $s(L(i + 1))$ for its next iteration.

Feeding the produced state into a buffer and back into the $pC_{\delta,\omega,L}$ component, therefore results in an algorithm that does not behave according to the system specification\(^3\) for $L > 1$.

So, to complete the block processing algorithm something more is needed, as we discuss in the next section and Section 2.7. The performance characteristics of a block processing algorithm that contains a pipeline structure are a combination of the ones in Table 2.1 and those of the other components of the algorithm. The scalability of the block processing algorithm depends entirely on those other components, since the $pC_{\delta,\omega,L}$ component already scales linearly.

Note that, we assume that the basic $C_{\delta,\omega,1}$ components have a complexity and a delay of $\Theta(1)$ and pass only one data element from one component to the next. This is not necessarily true for every pipeline structure, since many of them are constructed from more complex components. Any extra complexity of the components used to construct the pipeline affects the performance characteristics of the pipeline structure, but it does not affect the scalability. As long as the complexity and delay of the basic components are independent of $L$ the linear scalability of the block processing algorithm is maintained.

### 2.4 Linear look-ahead

The linear look-ahead computation is obtained by unrolling the algebraic state update equation (2.1) a total of $L$ times. This results in:

$$s(L(i + 1)) = (\alpha(Li, L) \otimes s(Li)) \oplus \beta(Li, 0, L) \text{ for } i \in \mathbb{N} \land L \in \mathbb{N}^+ \quad (2.17)$$

\(^3\)Such algorithms, however, are used to run $L$ identical systems on the same piece of hardware by interleaving their input streams and de-interleaving the output stream.
2.4. LINEAR LOOK-AHEAD

Equation (2.17) allows us to look-ahead \( L \) steps in the time it takes to perform one \( \otimes \) and one \( \oplus \) operation. Therefore it can be used, in combination with the pipeline structure \( p_{C_{\delta,\omega,L}} \) from the previous section, to construct the block processing algorithm shown in Figure 2.6. The block marked “Look-ahead” computes the new state based on the old state in the time it takes to perform one \( \otimes \) and one \( \oplus \) operation using (2.17). Of course, this block must also compute \( \alpha(L_i, L) \) and \( \beta(L_i, 0, L) \).

The appropriate values of \( \alpha \) and \( \beta \) are computed in a pipelined manner at the cost of \( 2L - 2 \otimes \) operations and \( L - 1 \oplus \) operations. This is done by another pipeline structure, but instead of using \( C_{\delta,\omega,1} \) as the basic components for the structure we construct it from the components \( C_{\otimes,\oplus,1} \) depicted in Figure 2.7(a). We have marked the inputs of the \( \oplus \) and \( \otimes \) operators in Figure 2.7(a) to indicate which of its inputs is the first and second operands, since the operations are not necessarily commutative.

When these \( C_{\otimes,\oplus,1} \) components are placed in a pipeline of length \( L \), the last component computes \( \alpha(L_i, L) \) and \( \beta(L_i, 0, L) \), i.e. the terms needed for the look-ahead computation. An example of such a pipeline for \( L = 4 \) is shown in Figure 2.7(b) where \( 1_{\otimes} \) and \( 1_{\oplus} \) denote the unity elements for the \( \otimes \) and \( \oplus \) operators respectively. Note that the \( a(L_i + j) \) and \( b(L_i + j) \) values enter the pipeline structure in reversed order, i.e. the last component of the pipeline receives those with \( j = 0 \), while the first component receives those with \( j = L - 1 \).

By combining all the components in the right manner we obtain the linear look-ahead based algorithm depicted in Figure 2.8. The mapping component takes the inputs of the system and produces the values of \( a \) and \( b \) as required by
CHAPTER 2. EXISTING METHODS

\[ \alpha(L_i, L - j - 1) \otimes \alpha(L_i, L - j) \]

\[ \beta(L_i, j + 1, L) \oplus \beta(L_i, j, L) \]

(a) A basic $C_{\otimes, \oplus, 1}$ component

\[ C_{\otimes, \oplus, 1} \rightarrow \rightarrow \rightarrow C_{\otimes, \oplus, 1} \rightarrow \rightarrow \rightarrow C_{\otimes, \oplus, 1} \rightarrow \rightarrow \rightarrow C_{\otimes, \oplus, 1} \]

\[ \alpha(4i, 4) \]

\[ \beta(4i, 0, 4) \]

(b) A $C_{\otimes, \oplus, 4}$ component

Figure 2.7: The pipeline that computes (2.18) and (2.19) for $L = 4$

\[ \text{Mapping} \]

\[ \alpha(L_i, L) \otimes \beta(L_i, 0, L) \]

\[ s(L(i + 1)) \]

\[ s(i) \]

\[ C_{\delta, \omega, L} \]

\[ \text{out}(i) \]

Figure 2.8: A block processing algorithm based on linear look-ahead
2.4. LINEAR LOOK-AHEAD

<table>
<thead>
<tr>
<th>VLSI circuit</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency $L$</td>
<td>$\Theta(L^2)$ outputs</td>
<td></td>
</tr>
<tr>
<td>Memory $M$</td>
<td>$\Theta(L^2)$ registers</td>
<td></td>
</tr>
<tr>
<td>Complexity $C$</td>
<td>$\Theta(L)$ operations/block</td>
<td></td>
</tr>
<tr>
<td>Iteration period $I$</td>
<td>$\Theta(1)$ time units</td>
<td></td>
</tr>
<tr>
<td>Throughput $T$</td>
<td>$\Theta(L)$ outputs/time unit</td>
<td></td>
</tr>
<tr>
<td>Scalability</td>
<td>linear, $\Theta(1)$ operations per output</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIMD program</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr. of PEs $P$</td>
<td>$L = IP$</td>
<td></td>
</tr>
<tr>
<td>Latency $L$</td>
<td>$\Theta(PL)$ outputs</td>
<td></td>
</tr>
<tr>
<td>Memory $M$</td>
<td>$\Theta(PL)$ registers</td>
<td></td>
</tr>
<tr>
<td>Complexity $C$</td>
<td>$\Theta(L)$ operations/block</td>
<td></td>
</tr>
<tr>
<td>Iteration period $I$</td>
<td>$\Theta(I)$ time units</td>
<td></td>
</tr>
<tr>
<td>Throughput $T$</td>
<td>$\Theta(P)$ outputs/time unit</td>
<td></td>
</tr>
<tr>
<td>Scalability</td>
<td>linear, $\Theta(1)$ operations per output</td>
<td></td>
</tr>
<tr>
<td>Remarks</td>
<td>Requires strided memory access</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Characteristics of a linear look-ahead block processing algorithm

the $C_{\otimes,\oplus,L}$ component. The complexity and other performance characteristics of the mapping component varies from system to system. We assume, however, that it does not influence the performance characteristics of the rest of the algorithm, as shown in Table 2.2. For functions that map the inputs directly on a $a$ and $b$ value this assumption is reasonable, for other functions we might need to design a separate block processing system that has linear scalability.

The performance characteristics in Table 2.2 are obtained after pipelining the block diagram in Figure 2.8. The pipelining for both the $C_{\otimes,\oplus,L}$ and $C_{\delta,\omega,L}$ is done as explained in Section 2.3. The characteristics in Table 2.2 are the same as those of the pipeline structure in Table 2.1 since the linear look-ahead consists mainly of two pipeline structures. The only difference in performance characteristics is in the constants, but those are obscured by the $\Theta$ notation.

There are two major drawbacks of this form of look-ahead. Both are the result of the pipelining structure. First of all, the pipelining introduces a large number of registers and a latency to match. Secondly, only when we allow strided memory access operations is it possible to implement the algorithm in a scalable manner on an SIMD processor. We have demonstrated this for IIR filters in [38] and we discuss it in more detail in Section 7.3.2.

The algorithm as presented in this section is also known as the incremental
block computation described by Parhi and Messerschmitt in [67]. In their paper they apply it to the linear time invariant systems, like IIR filters, since these systems are common and they have a constant \( a(i) \), i.e. \( a(i) = a(i + 1) \) for \( i \in \mathbb{N} \), so \( \alpha \) can be pre-computed.

The literature also contains other algorithms based on the linear look-ahead computation, like [54] and [72] for IIR filters. These implementations, however, do not exhibit linear scalability. The VLSI algorithm from [54] has super-linear scalability, since it has a complexity of \( C \in \Theta(L^2) \) operations per block, which means that the amount of hardware grows quadratically with the throughput. The SIMD algorithm from [72] is based on this VLSI algorithm and it maps these \( \Theta(L^2) \) operations onto \( \Theta(L) \) operations on vectors of length \( L \), i.e., \( P = L \). This results in an algorithm that does not scale at all, since the iteration period is \( \Theta(L) \), resulting in a throughput of \( \frac{L}{\Theta(L)} \in \Theta(1) \), i.e., a throughput that is constant regardless of the amount of hardware used. The reason that the algorithms from [54] and [72] do not scale linearly is that the state produced by the linear look-ahead computation is used to compute the entire block of outputs directly. This direct computation dominates the complexity of the algorithm, and therefore also the scalability of the algorithm. Since this direct computation does not scale linearly, it makes the linear scalability of the look-ahead computation irrelevant, and results in an algorithm that does not scale.

### 2.5 Logarithmic look-ahead

The logarithmic look-ahead computation [23] does not exhibit linear scalability, but it avoids a pipeline structure and the associated disadvantages. The logarithmic look-ahead computation is obtained by applying a divide and conquer strategy to the linear look-ahead computation. For simplicity we assume that \( L = 2^l \) for some \( l \in \mathbb{N} \). This results in the same formula for look-ahead as for the linear look-ahead (2.17):

\[
s(L(i + 1)) = (\alpha(Li, L) \odot s(Li)) \oplus \beta(Li, L) \text{ for } i \in \mathbb{N} \land L \in \mathbb{N}^+
\]

but now \( \alpha \) and \( \beta \) are computed as follows:

\[
\alpha(i, j) = \begin{cases} a(i) & j = 1 \\ a(i + \frac{j}{2}, \frac{j}{2}) \odot \alpha(i, \frac{j}{2}) & j > 1 \end{cases} \tag{2.21}
\]

\[
\beta(i, j) = \begin{cases} b(i) & j = 1 \\ (\alpha(i + \frac{j}{2}, \frac{j}{2}) \odot \beta(i, \frac{j}{2})) \oplus \beta(i + \frac{j}{2}, \frac{j}{2}) & j > 1 \end{cases} \tag{2.22}
\]
2.5. LOGARITHMIC LOOK-AHEAD

Figure 2.9: Logarithmic look-ahead with $L = 8$
In Figure 2.9(a) the notation
\[
\gamma(i, j) = (\alpha(i, j), \beta(i, j))
\] (2.23)
is used in the depiction of an algorithm that performs a logarithmic look-ahead computation. The blocks in the diagram represent operations on tuples and this operation is depicted in Figure 2.9(b). The diagram shows that the \(\alpha\) and \(\beta\) are computed in \(\log_2 L\) stages, with each stage containing \(L\) operations. These stages are followed by a stage containing the recursive loop that calculates a tuple containing the new state. Note that the “…” symbol, which is the product of several \(a\)’s, and which is computed in this last stage is not needed. The “…” is included so that the picture contains uniform components, but computations involving “…” can be omitted from the implementation.

The look-ahead computation from Figure 2.9(a) is then used as shown in Figure 2.10 to obtain a block processing algorithm. This algorithm consists of three components. The first component maps the inputs on the \(a\) and \(b\), and therefore \(\gamma\), values. This component is the same as the one used in the linear look-ahead computation from the previous section. The second component is the look-ahead computation shown in Figure 2.9(a). The third component uses the \(L\) states produced by the look-ahead computation to compute \(L\) outputs in parallel by applying the output function \(\omega\) to each of the states individually.

To map the block diagram from Figure 2.9(a) onto an SIMD processor we have to schedule the operations of the algorithm on the \(P\) processing elements (PEs) of the processor. The simplest way to schedule the operations is obtained by choosing \(L = P\). This choice enables us to schedule each stage in a different time slot, and to schedule each row of the block diagram on a different PE.

An advantage of this look-ahead computation is that it computes a block of \(L\) new states, instead of a single new state like the linear look-ahead computation. This means that the computation of intermediate states does not have to take place in a pipeline structure, since each output can be computed directly from the state on which it depends. Because of the absence of this pipeline the algorithm requires less memory and introduces less latency. Furthermore, in the absence of the pipeline structure the SIMD variant does not require strided
2.5. LOGARITHMIC LOOK-AHEAD

<table>
<thead>
<tr>
<th>VLSI circuit</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency $L$</td>
<td>$\Theta(L \log L)$ outputs</td>
</tr>
<tr>
<td>Memory $M$</td>
<td>$\Theta(L \log L)$ registers</td>
</tr>
<tr>
<td>Complexity $C$</td>
<td>$\Theta(L \log L)$ operations/block</td>
</tr>
<tr>
<td>Iteration period $T$</td>
<td>$\Theta(1)$ time units</td>
</tr>
<tr>
<td>Throughput $T$</td>
<td>$\Theta(L)$ outputs/time unit</td>
</tr>
<tr>
<td>Scalability</td>
<td>super-linear</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIMD program</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr. of PEs $P$</td>
<td>$L = P$</td>
</tr>
<tr>
<td>Latency $L$</td>
<td>$\Theta(L)$ outputs</td>
</tr>
<tr>
<td>Memory $M$</td>
<td>$\Theta(L)$ registers</td>
</tr>
<tr>
<td>Complexity $C$</td>
<td>$\Theta(L \log L)$ operations/block</td>
</tr>
<tr>
<td>Iteration period $T$</td>
<td>$\Theta(\log L)$ time units</td>
</tr>
<tr>
<td>Throughput $T$</td>
<td>$\Theta(\frac{L}{\log L})$ outputs/time unit</td>
</tr>
<tr>
<td>Scalability</td>
<td>super-linear</td>
</tr>
</tbody>
</table>

Table 2.3: Characteristics of logarithmic look-ahead

A summary of the characteristics of this look-ahead computation is shown in Table 2.3. The difference between the VLSI and SIMD variants is caused by pipelining. The VLSI algorithm can be pipelined such that the throughput becomes $\Theta(L)$ at the cost of $\Theta(L \log L)$ pipeline registers. This is accomplished by placing $L$ pipeline latches between each of the stages of the computation. The SIMD algorithm is not pipelined, since accessing the pipeline registers would only increase the number of operations without reducing the iteration time.

The logarithmic look-ahead computation is not encountered often by itself, but the so-called “pipelined incremental block” computation of Parhi and Messerschmitt [67] is a combination of the linear look-ahead computation and the logarithmic look-ahead computation. The pipelined incremental block computation is obtained by first applying linear look-ahead to (2.1) and then applying logarithmic look-ahead to (2.17).

In summary, the logarithmic look-ahead computation has low latency and does not require strided memory access for SIMD implementation. This makes the logarithmic look-ahead computation suited for a large number of systems and hardware platforms. Its scalability is super-linear, but the cost increase compared to the throughput is reasonable; doubling the block size adds only a constant to the number of operations per output.
2.6 Parallel prefix look-ahead

The operation $\odot$ on tuples, used in Figure 2.9(a) in the previous section, is also useful for expressing the look-ahead computation as a so-called scan or prefix operation. This allows us to apply the available parallel prefix algorithms to look-ahead computations, resulting in an algorithm with the lowest complexity so far.

Expressing the look-ahead computation as a prefix operation is done in the following way. First the $\odot$ operator is defined as:

$$(a, b) \odot (c, d) = (a \otimes c, (a \otimes d) \oplus b) \quad (2.24)$$

Note that this operator is associative because $\otimes$ and $\oplus$ are associative and $\otimes$ left-distributes over $\oplus$.

We use the shorthand

$$c(i) = (a(i), b(i)) \quad (2.25)$$

and the $\odot$-operator to write:

$$(\alpha(i, L) \odot \ldots, s(i + L)) = (\odot : 0 \leq j < L : c(i + L - 1 - j)) \odot (\ldots, s(i))$$

for $i \in \mathbb{N} \land L \in \mathbb{N}^+ \land \ldots \in \mathbb{D}$

This equation can be used as the basis for a look-ahead computation, since it shows the relation between the state $s(i + L)$ and the state $s(i)$.

Note that Fettweis et al. [23] assume that "..." in (2.26) is the unity element for the $\otimes$ operation, i.e., $\ldots \otimes r = r$. Equation (2.26), however, is just as valid for any other $\ldots \in \mathbb{D}$, and since we are only interested in the second element of the tuple, i.e. the state, it is not necessary to have, or introduce, a unity element for $\otimes$ in the set $\mathbb{D}$.

The computation of the states $s(i + j)$ for $1 \leq j \leq L$ from $s(i)$ according to (2.26) is done by using a so-called prefix or scan computation. The prefix computation is performed on the list of $[c(i + L - 1), \ldots, c(i + 1), c(i), (\ldots, s(i))]$, scanning from right to left and using the reduction operator $\odot$. The result of this computation is the list $[(\alpha(i, L) \otimes \ldots, s(i + L)), \ldots, (\alpha(i, 1) \otimes \ldots, s(i + 1)), (\ldots, s(i))]$. From this resulting list we then extract the new states of the system and use them to compute the outputs of the system.

There are many algorithms for the parallel computation of such a prefix operation [9, 49]. The optimum parallel prefix algorithms, w.r.t. complexity and critical path, have the following characteristics. First of all, the block diagram contains $\Theta(L)$ operators, so the complexity of the parallel prefix algorithm is
2.6. PARALLEL PREFIX LOOK-AHEAD

Figure 2.11: A parallel prefix algorithm used for look-ahead computation
VLSI circuit

<table>
<thead>
<tr>
<th>Latency</th>
<th>$L$</th>
<th>$\Theta(L \log L)$ outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>$M$</td>
<td>$\Theta(L \log L)$ registers</td>
</tr>
<tr>
<td>Complexity</td>
<td>$C$</td>
<td>$\Theta(L)$ operations/block</td>
</tr>
<tr>
<td>Iteration period</td>
<td>$T$</td>
<td>$\Theta(1)$ time units</td>
</tr>
<tr>
<td>Throughput</td>
<td>$T$</td>
<td>$\Theta(L)$ outputs/time unit</td>
</tr>
<tr>
<td>Scalability</td>
<td></td>
<td>linear, $\Theta(1)$ operations per output</td>
</tr>
</tbody>
</table>

SIMD program

| Nr. of PEs | $P$ | $L = P$ |
| Latency    | $L$ | $\Theta(L)$ outputs |
| Memory     | $M$ | $\Theta(L)$ registers |
| Complexity | $C$ | $\Theta(L)$ operations/block |
| Iteration period | $T$ | $\Theta(\log L)$ time units |
| Throughput | $T$ | $\Theta(\frac{L}{\log L})$ outputs/time unit |
| Scalability | | super-linear |

Table 2.4: Characteristics of prefix look-ahead

$\Theta(L)$. Secondly, the depth of the diagram, i.e., maximum number of operations on the path from any input to any output, is $\Theta(\log L)$, which also forms the critical path of the algorithm.

One parallel prefix algorithm, from [23], applied to the look-ahead computation is shown in Figure 2.11. This look-ahead computation can be used in the same way as the logarithmic look-ahead computation in Figure 2.10 to obtain a block processing algorithm.

A parallel prefix based algorithm is well-suited for VLSI implementation, as shown in Table 2.3. Implementing a parallel prefix algorithm in SIMD, however, is problematic. The depth of the block diagram is $\Theta(\log L)$, so at least $\Theta(\log L)$ of the $\Theta(L)$ operations in the algorithm depend on each others results. This means that the schedule of the $\Theta(L)$ operations of the algorithm on the $P$ processing elements of the SIMD processor consists of at least $\Theta(\log L)$ time units. Therefore we obtain the performance characteristics for an SIMD implementation as shown in Table 2.4.

To implement the parallel prefix algorithm on an SIMD processor we could also pipeline the algorithm, breaking the dependency between the operations, to such an extend that each pipeline stage contains only one operation. Since this would require $\Theta(L)$ cut-sets of the algorithm and each cut-set would intersect at least $L$ data-paths, this would result in $\Theta(L^2)$ pipeline registers. This ap-
proach would then result in performance characteristics similar to those of the linear look-ahead approach of Section 2.4. In contrast to the linear look-ahead approach, however, the algorithm might also need shuffle operations in addition to strided memory accesses. The reason for this is that the operations, and their data dependencies, are not necessarily distributed regularly between the different pipeline stages.

In short the parallel prefix look-ahead computation is suited for VLSI implementation, but not for SIMD implementation. Table 2.4 shows that the VLSI algorithm for prefix look-ahead is better than that of the logarithmic look-ahead, because the complexity is less. The SIMD variant, however, is not much better than the SIMD variant of the logarithmic look-ahead. It does have a lower complexity, but this is not reflected in the throughput of the algorithm.

In [13] Chatterjee et al. make a case to include scan operations as primitives on SIMD hardware platforms. This would allow a more scalable implementation of the SIMD algorithm on that hardware platform. However, the \( \odot \)-operation is a problem-specific reduction operator, so the chances of a generic SIMD processor supporting the right scan operation for a specific system are very small.

## 2.7 Block post-computation

Apart from the look-ahead computations there is another way to obtain a block processing algorithm for an arbitrary stream processing systems. The look-ahead computations are used to compute the state(s) of the system before computing the block of outputs, therefore it is also known as block pre-computation. In this section we discuss the opposite of block pre-computation, namely block post-computation.

### 2.7.1 Arbitrary FSMs

Lin and Messerschmitt show in [53] that any FSM can be implemented as a block processing algorithm with linear scalability using so-called block post-computation, which we discuss in this section. They use the \( pC_{\delta, \omega, L} \) components from Section 2.3. These components are pipelines that, given a block of inputs and a starting state, compute a block of outputs.

Instead of doing a look-ahead computation to determine which starting state should be fed into the pipeline, however, the block post-computation method “guesses” the starting state. A wrong guess would break the computation, so
the success of the block post-computation approach is guaranteed by making
$N = \# S$ attempts, i.e., one attempt for every state in $S$.

Specifically, the $pC_{\delta,\omega,L}$ component is duplicated $N$ times and each instance
uses a different starting state $s_i$ such that $S = \{s_i \mid 0 \leq i < N\}$. Each $pC_{\delta,\omega,L}$
produces a pair consisting of a state and a block of outputs. These state-output
pairs are fed into a multiplexer which selects the pair produced by the $pC_{\delta,\omega,L}$
component that started with the correct initial state $s(Li)$. This component
produces $s(L(i + 1))$, which is then used by the multiplexer to make its choice
in the next iteration.

The block diagram of this algorithm is shown in Figure 2.12. Note that we as-
sume that the states $s_0, s_1, \ldots s_{N-1}$ are represented by the numbers $0, 1, \ldots N - 1$. If this is not the case an extra component is needed to map the state $s(Li)$
onto such a number so that the multiplexer can select the correct state-output
pair.

The performance characteristics of this algorithm are shown in Table 2.5 and
they are similar to those of the pipeline structure from Table 2.1. The difference
is that the pipeline structure has been duplicated $N$ times, which means that

![Figure 2.12: A block processing algorithm for FSMs with linear scalability](image-url)
2.7. BLOCK POST-COMPUTATION

<table>
<thead>
<tr>
<th>VLSI circuit</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>Memory</td>
<td>$\mathcal{M}$</td>
</tr>
<tr>
<td>Complexity</td>
<td>$\mathcal{C}$</td>
</tr>
<tr>
<td>Iteration period</td>
<td>$\mathcal{T}$</td>
</tr>
<tr>
<td>Throughput</td>
<td>$T$</td>
</tr>
<tr>
<td>Scalability</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIMD program</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr. of PEs</td>
<td>$P$</td>
</tr>
<tr>
<td>Latency</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>Memory</td>
<td>$\mathcal{M}$</td>
</tr>
<tr>
<td>Complexity</td>
<td>$\mathcal{C}$</td>
</tr>
<tr>
<td>Iteration period</td>
<td>$\mathcal{T}$</td>
</tr>
<tr>
<td>Throughput</td>
<td>$T$</td>
</tr>
<tr>
<td>Scalability</td>
<td></td>
</tr>
<tr>
<td>Remarks</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5: Characteristics of a block processing algorithm based on block post-computation

The memory size and complexity of the implementations is multiplied by $N$.

The difference in the iteration period and throughput between Table 2.4 and 2.5 for the VLSI implementation are caused by the multiplexer. This multiplexer has to select one state-output pair from $N$ such pairs, which takes $\Theta(\log N)$ time. This means that the multiplexer is on the critical path of the implementation and therefore determines its iteration time and throughput. For the SIMD implementation the difference between the iteration times from the two tables is a factor $N$, and therefore the difference in throughput is a factor $\frac{1}{N}$. This factor is introduced because the processor now needs to emulate $N$ pipelines in every iteration instead of only one.

So, the algorithm based on block post-computation scales, since we can increase the block size to get an increased throughput, and it scales linearly because the complexity per output is a constant $\Theta(N)$. The overhead of a factor $N$ can grow extremely large, however, since the number of states $N$ of the FSM is extremely large in many practical cases.

---

4Note that some pipeline registers can be removed by sharing them between the inputs of the $pC_{\delta,\omega,L}$ components. Unfortunately this sharing cannot be applied to the outputs.
CHAPTER 2. EXISTING METHODS

2.7.2 FSMs with recurring states

The overhead introduced by the method from the previous section is considerable. Some FSMs have states that are visited repeatedly, however, which allows for a more efficient algorithm than the one from the previous section. In [53] Lin and Messerschmitt describe how this property can be exploited by a method which they call bit-positioning. We briefly summarize their findings in this section.

Consider an FSM that has a state, say $s'$, such that between every visit of the FSM to that state there are at most $M$ inputs. In other words, in the state transition graph of the FSM there is a node $s'$ that is on every cycle and the length of the longest cycle is $M$. An example of FSMs that have this property are those that are used to decode Huffman codes. For those FSMs $s'$ is the root of the decoding tree and $M$ is the length of the longest code word, i.e., the depth of the decoding tree.

The basic idea is not to “guess” what the correct starting state would be, but at what point in time the recurring state was visited. Since it is visited at least once every $M$ inputs, there are $M$ choices.
2.7. BLOCK POST-COMPUTATION

<table>
<thead>
<tr>
<th>VLSI circuit</th>
<th>SIMD program</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Latency</strong></td>
<td>$L = IP - M$</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>$\Theta(MP(M + L) + M)$ registers</td>
</tr>
<tr>
<td><strong>Complexity</strong></td>
<td>$\Theta(M^2 + ML)$ operations/block</td>
</tr>
<tr>
<td><strong>Iteration period</strong></td>
<td>$\Theta(\log M)$ time units</td>
</tr>
<tr>
<td><strong>Throughput</strong></td>
<td>$\Theta(L) / \log M$ outputs/time unit</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>linear, $\Theta(M + \frac{M^2}{L})$ operations per output</td>
</tr>
<tr>
<td><strong>Nr. of PEs</strong></td>
<td>$P$</td>
</tr>
<tr>
<td><strong>Latency</strong></td>
<td>$\Theta(PL)$ outputs</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>$\Theta(MP(M + L) + M)$ registers</td>
</tr>
<tr>
<td><strong>Complexity</strong></td>
<td>$\Theta(M^2 + ML)$ operations/block</td>
</tr>
<tr>
<td><strong>Iteration period</strong></td>
<td>$\Theta(MI)$ time units</td>
</tr>
<tr>
<td><strong>Throughput</strong></td>
<td>$\Theta(\frac{L}{\log M} - \frac{1}{L})$ outputs/time unit</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>linear, $\Theta(M + \frac{M^2}{L})$ operations per output</td>
</tr>
<tr>
<td><strong>Remarks</strong></td>
<td>Requires strided memory access</td>
</tr>
</tbody>
</table>

Table 2.6: Characteristics of a block post-computation algorithm for FSMs with a recurring state

Specifically, we instantiate $M$ pipelines, all of which use the state $s'$ as their starting state. The difference between these pipelines and those from the previous section is that they also produce an output $p(i + 1)$ such that $p(i + 1)$ is the smallest non-negative integer for which $s(i + 1 - p(i)) = s'$ holds. In effect, $p(i + 1)$ describes the number of steps since the last visit to the recurring state $s'$ before step $i + 1$. We call the pipelines that produce this extra output $pC'_{\delta,\omega,L}$ components.

Another difference between the system from this section and the previous section is that the $pC'_{\delta,\omega,L}$ components in this algorithm have different block sizes. The block size ranges from $L$ to $L + M - 1$, see Figure 2.13. The output of component $pC'_{\delta,\omega,L}$ is used when $s'$ was visited exactly on the boundary between blocks, while the $pC'_{\delta,\omega,L+1}$ component handles the case where the state was visited one input before the block boundary, etc.

The number of $C'_{\delta,\omega,1}$ components required for this method is $\Theta(M^2 + ML)$. Furthermore, the pipelining requires $M + L$ cut-sets, each of which intersects the $\Theta(M + L)$ inputs and outputs of each of the $M$ components. Therefore we require $\Theta(M(M + L)^2)$ pipeline registers to hold the inputs and outputs of the components. Additionally, a buffer of size $M$ is needed to buffer the last $M$
inputs of each block, since they are needed in addition to the inputs of the next block in the next iteration. So the total amount of memory is \( \Theta(M(M+L)^2 + M) \) registers. The pipelining also results in a latency of \( \Theta(M + L) \) blocks, i.e. one for each cut-set, which corresponds to \( \Theta((M + L)L) \) outputs. These performance characteristics have been summarized in Table 2.6.

This method also results in an algorithm with a high overhead, but in those cases where this method can be applied \( M \) tends to be relatively small compared to the overhead \( N = \#S \) of the approach presented in the previous section. For Huffman decoding, for example, we have \( M \in O(\log N) \). Therefore the algorithm designed in this manner is often preferable to the one from Section 2.7.1.

Although Lin and Messerschmitt do not explore it, it is possible to generalize this optimization to include FSMs that visit one state from a certain subset of states at least once every \( M \) steps, effectively creating a combination of the method from Section 2.7.1 and this section.

### 2.8 Conclusion

In this chapter we have presented an overview of methods to design block processing algorithms for stream processing systems of class \( FSM_\infty \). Some of these methods result in algorithms that scale linearly, i.e., algorithms where the amount of hardware needed is proportional to the throughput obtained. All of them, however, have some kind of drawback.

The block post-computation method results in algorithms that scale linearly and applies to any FSM, but it also has a huge overhead factor for nearly all practical FSMs. This factor equals the number of states in the system. Therefore block post-computation is only practical when the system has a small number of states \( N \), i.e., for \( N < L \). When the FSM has recurring states the overhead can be reduced somewhat, but it is still considerable and can only be applied if the FSM actually has recurring states.

There are also the so-called look-ahead computations, also known as block pre-computation methods. Any recursion of the form (2.1), i.e. the algebraic state update equation, can be evaluated using a block pre-computation system. It is possible to put any FSM in that form, but the operations needed in that case are not \( \Theta(1) \) and the overhead introduced in this manner depends on the number of states of the FSM. Therefore look-ahead computations are not better suited than the block post-computation for general FSMs. Look-ahead computations are suited, however, for systems that can be represented as (2.1) with operations
of relatively small or $O(1)$ complexity.

The linear look-ahead computation provides an algorithm that scales linearly for systems based on the algebraic state update equation. It has the drawbacks that the memory requirements and the latency grow quadratically with the block size. Furthermore, to implement the linear look-ahead computation on an SIMD processor we require so-called strided memory access operations to preserve the linear scalability of the implementation.

The logarithmic look-ahead computation is a compromise between the scalability and the drawbacks of the linear look-ahead computation. A logarithmic look-ahead computation scales super-linearly, but the costs grow relatively slow with the throughput, and it results in relatively low latency and memory requirements.

The prefix look-ahead computation has even better performance characteristics than the logarithmic look-ahead, when it is implemented in VLSI. It has the same low latency and low memory requirements as the logarithmic look-ahead computation, and it scales linearly. When implemented on an SIMD processor, however, the linear scalability is lost.

In summary, a large number of methods are covered by the literature. These methods provide algorithms that scale linearly when implemented on dedicated hardware. None of them, however, produces block processing algorithms that scale linearly and that can be implemented on typical SIMD processors.
Chapter 3
Solution strategies

3.1 Introduction

Depending on the properties of a system, and therefore the class to which a system belongs, various solution strategies can be used to obtain a block processing algorithm that scales linearly. In the previous chapter some strategies have been discussed for systems of class $FSM_\infty$. In this chapter we discuss solution strategies for each of the classes, ranging from the very general $FSM_\infty$ class to the more specific $FSM_*^\circ$ class.

We illustrate these solution strategies using a simple, yet interesting, example system. This system computes the moving sum over an input stream, i.e., its outputs are the sum of the last $W$ inputs for some window size $W \in \mathbb{N}^+$. Formally the outputs are specified by the following equation:

$$\text{out}(i) = (+ j : 0 \leq j < W : \text{in}(i - j))$$

(3.1)

The moving sum system belongs to class $FSM_*^\circ$, since its outputs are the aggregation, with an associative aggregation operator, of a finite number of inputs. Any strategy to design systems for class $FSM_*^\circ$ can therefore be illustrated using the moving sum system. We also use the system to illustrate strategies designed for other classes of systems, since the $FSM_*^\circ$ class is a subclass of both the $FSM_*$ and $FSM_\infty$ from Chapter 1.

In each section for this chapter we discuss one or more strategies that apply to a specific class of stream processing system. We illustrate those strategies by applying them to the moving sum system. We discuss only a few of the
possible strategies, since the number of design choices is very large. We focus on strategies that result in algorithms that scale linearly and that are suited for implementation in both VLSI and on SIMD processors.

### 3.2 Infinite memory span strategy

In this section we discuss two strategies that allow us to obtain algorithms that scale linearly for the moving sum system. These strategies will not use any of the properties that make the moving sum system a part of the $FSM_\infty$ and $FSM_\ast$ class of systems. So we effectively regard the moving sum system as a finite state machine based on a recursive state update function, i.e., a system of class $FSM_\infty$.

Regarding the moving sum system as a recursive system may seem surprising, but the moving sum is usually implemented on SISD processors based on the following recursive equation:

$$\text{out}(i) = \text{out}(i-1) + \text{in}(i) - \text{in}(i-W); \quad (3.2)$$

This recursive equation has the advantage that only two operations are needed, namely one addition and one subtraction, to compute one output, regardless of the window size $W$. It is possible to retain this efficiency when (3.2) is used as the basis for a block processing algorithm.

We therefore examine how Equation (3.2) can be used as a basis for a block processing algorithm. Chapter 2 describes two strategies that lead to algorithms that scale linearly and that are suited for both VLSI and SIMD. These are the linear look-ahead strategy from Section 2.4 and the block post-computation strategy from Section 2.7.1. We do not consider the block post-computation strategy, because it has a rather large overhead that would destroy any efficiency gained by basing our algorithm on (3.2). So the only strategy that we consider here is the linear look-ahead approach.

To use the linear look-ahead approach we need to write the state update equation of our system in the form of the algebraic state update equation (2.1). There are several ways of doing this, and we discuss two of them in the next two subsections. Both strategies rely, to some extent, on the form of (3.2), i.e. the form of an infinite impulse response filter, for which the so-called state space form is a representation that adheres to (2.1).

The first strategy, discussed in Section 3.2.1, translates (3.2) directly into its state space form, effectively obtaining an algorithm that computes it outputs in a single stage. The second strategy, discussed in Section 3.2.2 first splits (3.2)
into two systems one of which is an $FSM_*$ system and one which is an $FSM_\infty$ system, effectively resulting in an algorithm that computes its outputs in two stages.

### 3.2.1 Single stage strategy

Equation (3.2) has the same form as an infinite impulse response (IIR) filter of order $W$. Linear filters, like IIR filters, can be written in the so-called state space form, which adheres to the algebraic state update equation (2.1) and is therefore suitable for a look-ahead computation. In this section we demonstrate this by applying this strategy to obtain both a VLSI and an SIMD implementation for the moving sum system.

#### 3.2.1.1 State space form

In this section we consider the state space form and some of its properties. Furthermore, we write the moving sum system in this state space form and consider the look-ahead equation obtained by using the linear look-ahead approach.

The state space form of an $N$-th order linear system consists of two equations:

\[
\begin{align*}
\mathbf{s}(i + 1) &= A \mathbf{s}(i) + b \mathbf{in}(i) \\
\mathbf{out}(i) &= c^T \mathbf{s}(i) + d \mathbf{in}(i)
\end{align*}
\]  

Equations (3.3) and (3.4) can also be reduced to a single equation involving a $(W+1) \times (W+1)$ matrix:

\[
\begin{pmatrix}
\mathbf{s}(i + 1) \\
\mathbf{out}(i)
\end{pmatrix} = \begin{pmatrix} A & b \\ c^T & d \end{pmatrix} \begin{pmatrix} \mathbf{s}(i) \\ \mathbf{in}(i) \end{pmatrix}
\]  

(3.5)

Our moving sum example is a linear system of order $W$, so we have $N = W$. We choose the following values for $A$, $b$, $c$ and $d$:

\[
\begin{pmatrix} A \\ c^T \\ b \\ d \end{pmatrix} = \begin{pmatrix} 1 & 0 & \ldots & \ldots & 0 & 1 \\
1 & 0 & \vdots & \vdots & 0 \\
0 & 1 & \ddots & \vdots & \vdots \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
0 & \ldots & 0 & 1 & 0 & 0 \\
1 & 0 & \ldots & 0 & -1 & 1 \end{pmatrix}
\]  

(3.6)
This choice corresponds to the so-called direct form II filter structure for IIR filters \[70\]. There are some practical issues to take into account, but we discuss those in Section 3.5.

Other choices for \(A, b, c\) are also possible. The state representation can be chosen freely, so any linear transformation of the state results in different choices for \(A, b, c\). As long as the transformation is invertible no information is lost and it is still possible to compute the correct outputs from the transformed state. So, for every non-singular \(N \times N\) matrix \(M\) the system defined by \(A, b, c, d\) is the same as the system defined by \(A', b', c'\) and \(d'\) for \[54\]:

\[
\begin{align*}
A' &= M^{-1}AM \\
b' &= M^{-1}b \\
c' &= M^Tc \\
d' &= d
\end{align*}
\]  

(3.7)

where the relation between the two state representations is:

\[s(i) = M \, s'(i)\]  

(3.8)

Another choice for \(A, b, c\) is obtained by splitting the filter into parallel first and second order sections \[48\] and then combining them to obtain a block diagonal form for \(A\) \[54\]. Unfortunately we cannot apply this technique to our example, since it only works when the oldest output used to compute the current output is at least as old as the oldest input used to compute the current output. Systems that do not adhere to this property can be split into two parts to obtain two separate linear systems, one finite impulse response (FIR) filter and one IIR filter that does adhere to this property. For our example, however, this procedure would yield no IIR filter since the FIR part resulting from this procedure already describes the entire moving sum system. The reason for this is that the moving sum is a FIR filter, an \(FSM_\ast\) system, which we have disguised as an IIR filter.

For generic IIR filters, however, the block diagonal form with block sizes at most 2 are interesting because the sparsity of the matrix \(A\) means it takes fewer operations to implement multiplications with that matrix. The block diagonal form, and its sparsity, is maintained for every \(A^i\) for \(i \in \mathbb{N}\). This is useful since the linear look-ahead approach applied to \[3.3\] results in a computation where a power of the matrix \(A\) is used:

\[s(i + L) = A^L s(i) + B \, \text{in}(i)\]  

(3.9)
where $B$ is the $N \times L$ matrix specified by:

$$B = (A^{L-1}b \ A^{L-2}b \ \ldots \ b)$$  \hspace{1cm} (3.10)$$

For our moving sum the example matrix $A^L$ with dimensions $W \times W$, and the example matrix $B$ with dimensions $W \times L$ have the following form:

$$A^L = \begin{pmatrix} 1 & 0 & \ldots & \ldots & \ldots & \ldots & 0 \\ \vdots & \vdots & \ddots & & & & \vdots \\ 1 & 0 & \ldots & \ldots & \ldots & \ldots & 0 \\ 1 & 0 & \ldots & \ldots & \ldots & \ldots & 0 \\ 0 & 1 & \ddots & & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & & \vdots \\ 0 & \ldots & 0 & 1 & 0 & \ldots & 0 \end{pmatrix}_{L \min W}$$  \hspace{1cm} (3.11)$$

$$B = \begin{pmatrix} 1 & \ldots & \ldots & \ldots & \ldots & 1 \\ \vdots & \ddots & & & & \vdots \\ \vdots & & \ddots & & & \vdots \\ 1 & \ldots & 1 & 0 & \ldots & 0 \end{pmatrix}_{W}$$  \hspace{1cm} (3.12)$$

where max and min denote the maximum and minimum operator respectively.

These two matrices, and the linear look-ahead computation of (3.9), form the basis of the VLSI and SIMD algorithms discussed in the next two sections.

### 3.2.1.2 VLSI implementation

In this section we discuss a VLSI implementation based on linear look-ahead. This implementation consists of two parts. The first part performs the look-ahead computation specified by (3.9). The state produced by the first part is then used by the second part to compute $L$ outputs in a pipelined manner.

The first part of the VLSI implementation consists of two of the matrix-vector multiplication (MVM) components from [54] depicted in Figure 3.1(a). These components actually perform a multiply-add operation, multiplying a matrix with a vector and adding the result to another vector, but we will use the term MVM from [54].
Figure 3.1: MVM component for a $2 \times 4$ matrix
The MVM components are used to compute the look-ahead computation, which consists of adding the results of the two matrix-vector multiplications $A^L s(i)$ and $B \text{in}(i)$. Since all the multiplications in the moving sum system are with -1, 0 or 1, the MVM components can be simplified considerably.

Note that a single column of the MVM component corresponds to the components used to construct the pipeline for the linear look-ahead computation from Section 2.4. There are some differences, however. First of all, since $A$ is constant, the different powers of $A$ are not computed in the pipeline, but they are pre-computed and used as constants in the implementation. Secondly, the commutativity of the addition operator allows us to re-order the inputs. In Section 2.4 the first component in the pipeline processed the most recent input. In the MVM component the most recent input, i.e. the last element of the input vector, is processed by the last column of the MVM component.

The second part of the implementation consists of components that take a state, compute a single output and the next state, as depicted in Figure 3.2. We denote these components with $C_{\delta, \omega, 1}$, since they correspond to the components in Figure 2.3(a) from Section 2.3. In contrast to Section 2.3, however, the internal structure of the component is more detailed. A component here consists of three MVM components, a multiplier and an adder, though the multiplier can be omitted for our moving sum example.

By combining the MVM components for $A^L$ and $B$ with the $C_{\delta, \omega, 1}$ components from Figure 3.2 we obtain the implementation depicted in Figure 3.3. This algorithm corresponds to the “Incremental Block State Architecture” of...
Parhi and Messerschmitt with an increment of 1, as discussed in [67].

We analyze the performance measurements of the algorithm by examining each part of the algorithm in turn. Note that we pipeline the implementation in Figure 3.3 to reduce the critical path such that it contains only a single addition.

The MVM B component in the first part of the implementation is pipelined as shown in Figure 3.1(b). Since B is an $W \times L$ matrix, this results in $L$ cut-sets that each intersect $W + L$ data paths, namely one for each row and one for each column in the matrix. This results in a total of $\Theta(WL + L^2)$ pipeline registers.

The pipelining reduces the iteration period of the first part to $\Theta(1)$, i.e., a single addition operator. It also introduces latency, however. Each cut-set delays a block of inputs by a single iteration. Since we needed $L$ cut-sets and an iteration produces $L$ outputs, this means that the first part of the implementation introduces a latency of $\Theta(L^2)$ outputs.

Finally, the complexity of the first part of the implementation is $\Theta(WL)$, since B is a $W \times L$ matrix and the MVM component requires one multiplier and adder per element of the matrix. For the moving sum we do not need the multipliers, since the elements of the matrix B are either zero or one, but we do need the adders. The adders can be removed on those places where matrix B contains zeroes, but since the matrix contains only $\Theta(W^2)$ zeroes, see (3.12), the complexity remains $\Theta(WL)$ for $L \geq W$.

The MVM $A^L$ component and the recursive loop do little to influence the performance characteristics of the first part of the implementation. The matrix $A^L$ is very sparse, see Equation (3.11), and contains only a single non-zero element in every row. Therefore the critical path of the MVM component contains a single adder, and pipelining is unnecessary. The component does add 1 block delay to the latency, $W$ to the number of registers, and $W$ addition operations to the complexity, but adding these values to the characteristics of the MVM B

![Figure 3.3: Block processing algorithm based on the state space form](image)

Figure 3.3: Block processing algorithm based on the state space form
3.2. INFINITE MEMORY SPAN STRATEGY

<table>
<thead>
<tr>
<th>VLSI circuit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency ( L )</td>
</tr>
<tr>
<td>Memory ( M )</td>
</tr>
<tr>
<td>Complexity ( C )</td>
</tr>
<tr>
<td>Iteration period ( T )</td>
</tr>
<tr>
<td>Throughput ( T )</td>
</tr>
<tr>
<td>Scalability</td>
</tr>
</tbody>
</table>

\( \Theta(L^2) \) outputs
\( \Theta(WL + L^2) \) registers
\( \Theta(WL) \) operations/block
\( \Theta(1) \) time units
\( \Theta(L) \) outputs/time unit
linear, \( \Theta(W) \) operations per output

Table 3.1: Characteristics of the VLSI implementation of a moving sum based on the state space form and linear look-ahead

component has no effect in the \( \Theta \)-notation.

So now we turn to the second part of the implementation. This part is also pipelined to reduce the iteration period to a single addition. To compute the number of pipeline registers introduced by pipelining each \( C_{\delta,\omega,1} \) component, we consider the matrix (3.6), since these components perform a matrix-vector multiplication with that \((W+1) \times (W+1)\) matrix.

The matrix from (3.6) is very sparse. In fact it contains at most 3 non-zero values on a single row, namely the last row consisting of \( c^T \) and \( d \). So it takes only 3 cut-sets to pipeline the components and reduce the critical path of the entire pipeline to a single addition. These cut-sets intersects with \( \Theta(W) \) data paths inside the component and with \( L \) inputs and/or outputs outside of the component. Therefore the pipelining of the \( C_{\delta,\omega,1} \) components introduces \( \Theta(W + L) \) pipeline registers in the implementation per \( C_{\delta,\omega,1} \) component for our moving sum example. This results in a total of \( \Theta(WL + L^2) \) pipeline registers for the entire second part of the implementation.

This pipelining again introduces a latency, delaying blocks of outputs by one iteration for every cut-set. The pipelining takes \( \Theta(L) \) cut-sets, so the latency it introduces is \( \Theta(L^2) \) outputs.

The complexity of the second part of the implementation is relatively modest. The sparsity of the matrix in (3.6) is such that we need only 2 additions and 1 subtraction operation per \( C_{\delta,\omega,1} \) component. We obtain this number by noting that the number of addition operations needed per row of (3.6) is the number of non-zero elements in that row, minus one. The first row of (3.6) contains two non-zero values, resulting in one adder. The only other row (3.6) that contains more than one non-zero value is the last row, which contains two ones and one minus one, resulting in one adder and one subtraction operator. This results in a total complexity of \( \Theta(L) \) for the second part of our implementation.
When the performance measures of the first and second part of the implementation are combined, we obtain the results shown in Table 3.1. The numbers for the latency, number of registers and the complexity are obtained by adding the measurements of the first part to those of the second part. The iteration period depends on the critical path in the implementation and it is therefore the maximum of the iteration period of the first part and the iteration period of the second part of the implementation. The throughput of the implementation is the number of outputs produced per unit of time, so it is obtained by dividing the block size \( L \) by the iteration period of the implementation, resulting in a throughput of \( \Theta(L) \). So the algorithm scales, since we can increase the throughput by increasing the block size. Moreover, the algorithm scales linearly, since the complexity per output is constant, namely \( \Theta(W) \).

### 3.2.1.3 SIMD implementation

The incremental block state architecture of Parhi and Messerschmitt discussed in the previous section can also be adapted for SIMD processors, as we have shown in [38]. We summarize this adaptation process here for our moving sum example.

The SIMD implementation basically emulates the VLSI implementation. The first part of the VLSI implementation from Figure 3.3 is implemented by having each PE of the SIMD processor emulate one column of the MVM component. The second part of the VLSI implementation from Figure 3.3 is implemented by having each PE of the SIMD processor emulate one \( C_{\delta,\omega,1} \) component. This setup requires that we make the design choice \( P = L \). In Section 7.3.2 we also discuss the possibility to choose \( L = IP \) for any \( I \in \mathbb{N}^+ \), but we ignore that option in this section.

The first part of the VLSI implementation is described by the GCL-like pseudo-code in Listing 3.1. The annotation between the program statements contains the pre- and post-conditions of the statements using Hoare logic [37]. In this annotation we use the abstraction function \( \llbracket a \rrbracket \) to map the value of the program variables onto their mathematical counterparts. This function maps variables of the type \texttt{scalar} onto a scalar and variables of the type \texttt{vector} onto a vector of length \( P \). Arrays of scalars are also mapped onto vectors, but the length of the resulting vector corresponds to the length of the array. Furthermore, arrays of vectors are mapped onto matrices, where the rows correspond to the index in the array and the columns correspond to the vector elements. A more extensive description of our pseudo-code can be found in Appendix A.2.
3.2. *INFINITE MEMORY SPAN STRATEGY*

Listing 3.1: Part 1 of the linear look-ahead algorithm for the moving sum system as an SIMD program

```plaintext
var P, W, i, j : scalar;
  input_vec : vector;
  AL, state, new_state : array [0..W) of scalar;
  B_vec, part1_vec : array [0..W) of vector;
{\[P\] = P \land \[W\] = W \land \[i\] \in N \land \[B\_vec\] = B}
{\[AL\](w) = j such that A^L(w, j) = 1}
{Inv_0: \[state\] = s(Pi)}
{Inv_1: \[part1\_vec\](w, p) = f(i, w, p) see (3.13)}

do true →
  input_vec := strided_read(IN, P*(i+P−1), 1−P);
  \{\[input\_vec\](p) = in(P(i + P − 1) + p(1 − P))\}
  j := 0;
  do (j < W) →
    \{\[part1\_vec\](w, p) = f(i + 1, w, p + 1) for 0 ≤ w ≤ [j]\}
    \{\[part1\_vec\](w, P − 1) = (B in(i))(w) for 0 ≤ w ≤ [j]\}
    new_state[j] := state[AL[j]] + part1_vec[j](P−1);
    \{\[new\_state\](w) = s(P(i + 1))(w) for 0 ≤ w ≤ [j]\}
    part1_vec[j] := shift(part1_vec[j], −1, 0);
    \{\[part1\_vec\](w, p) = f(i + 1, w, p) for 0 ≤ w ≤ [j]\}
    j := j + 1;
  od
{Inv_1(i := i + 1)}
{\[new\_state\] = s(P(i + 1))}
  j := 0;
  do (j < W) →
    state[j] := new_state[j];
    j := j + 1;
  od
{Inv_0(i := i + 1)}
  i := i + 1;
od
```

Listing 3.1: Part 1 of the linear look-ahead algorithm for the moving sum system as an SIMD program
In Listing 3.1 we use the following definition for function $f$:

$$f(i, w, p) = \left( \sum_{k: 0 \leq k < p} B_{w,k} \in(P(i - p + P - 1) + k) \right) \quad (3.13)$$

This function describes the contents of the pipeline registers of the MVM component, where parameter $i$ indicates the number of the iteration, $w$ the row of the MVM component, and $p$ the column.

There are a few vector operations in the program in Listing 3.1. The strided memory access operation \texttt{strided\_read} is used to read the inputs into a vector. The element-wise vector multiplication and addition operations in the inner loop are used to compute the \texttt{part1\_vec} vectors. There is also the \texttt{shift} operation that shifts the elements of the vector by one position, inserting a zero value at the start of the vector in the process. For a more detailed description of these vector operations we again refer to Appendix A.2.

The second part of the algorithm consist of a pipeline that takes the state $s(Pi)$ and produces the outputs of the system. The pseudo-code for this part of the algorithm is shown in Listing 3.2. Note that this program uses the \texttt{strided\_write} operation to output the outputs, but other that, it uses the same operations as the program in Listing 3.1.

The program in Listing 3.2 does pose a problem, however. It requires the state $s(P(i+1))$, when updating the \texttt{part2\_vec} variables with the \texttt{shift} operation, since it needs to insert the state $s(P(i+1))$ at the start of the vector. This problem is solved by combining this program with the one that computes that state from Listing 3.1. The result of this combination is the block processing program depicted in Listing 3.3.

Since each line of the program in Listing 3.3 contains operations that require $\Theta(1)$ time, and some operations are repeated $W$ times in a loop, the iteration time of the program is $\Theta(W)$ time units.

Furthermore, the program produces $P$ new outputs per iteration, therefore its throughput is $\Theta(P/W)$ outputs per time unit.

The program contains $\Theta(W)$ operations per iteration that are vector operations, i.e., operations that involve a variable ending with “\_vec”. Therefore the complexity of this program is $\Theta(WP)$ operations per block.

The latency of the program is computed by noting that the outputs $\text{out}(P(i-p) + p)$ for $0 \leq p < P$ are produced in iteration $i$. The last inputs on which these outputs depend are $\text{in}(P(i-p) + p)$. These inputs are read in iteration $i$, but they are read for the first time in iteration $i - P + 1$. Therefore the latency of the program is $P - 1$ iterations, which corresponds to $\Theta(P)$ blocks, which corresponds to $\Theta(P^2)$ outputs.
3.2. INFINITE MEMORY SPAN STRATEGY

Listing 3.2: Part 2 of the linear look-ahead algorithm for the moving sum system as an SIMD program

```plaintext
var P, W, i, j : scalar;
input_vec, output_vec : vector;
A, b : array [0..W) of scalar;
part2_vec, new_vec : array [0..W) of vector;

// Constraint 1
\{ [P] = P \land [W] = W \land [i] \in N \land [b] = b \}
\{ [A](w) = j \text{ such that } A(w,j) = 1 \}
\{ Inv_0 : [part2_vec](w,p) = s(P(i-p) + p)(w) \}

do true →
  input_vec := strided_read(IN, P*i, 1-P);
  \{ [input_vec](p) = in(P(i-p) + p) \}
  output_vec := part2_vec[0] - part2_vec[W-1] + input_vec;
  \{ [output_vec](p) = out(P(i-p) + p) \}
  strided_write(OUT, P*i, 1-P, output_vec);
  j := 0;
  do (j < W) →
    new_vec[j] := part2_vec[A[j]] + (b[j]*input_vec);
    \{ [new_vec](w,p) = s(P(i-p) + p + 1)(w) \text{ for } 0 \leq w \leq [j] \}
    j := j+1;
  od
  \{ [new_vec](w,p) = s((i+1-p-1)P + p + 1)(w) \}
  j := 0;
  do (j < W) →
    part2_vec[j] := shift(new_vec[j], -1, s(P(i+1))[j]);
    \{ [part2_vec](w,p) = s((i+1-p)P + p)(w) \text{ for } 0 \leq w \leq [j] \}
    j := j+1;
  od
\{ Inv_0(i := i + 1) \}
  i := i+1;
od
```

Listing 3.2: Part 2 of the linear look-ahead algorithm for the moving sum system as an SIMD program
Listing 3.3: The complete linear look-ahead algorithm for the moving sum system as an SIMD program
3.2. INFINITE MEMORY SPAN STRATEGY

<table>
<thead>
<tr>
<th>SIMD program</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr. of PEs</td>
<td>$P$</td>
</tr>
<tr>
<td>Latency</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>Memory</td>
<td>$\mathcal{M}$</td>
</tr>
<tr>
<td>Complexity</td>
<td>$\mathcal{C}$</td>
</tr>
<tr>
<td>Iteration period</td>
<td>$\mathcal{T}$</td>
</tr>
<tr>
<td>Throughput</td>
<td>$\mathcal{T}$</td>
</tr>
<tr>
<td>Scalability</td>
<td></td>
</tr>
<tr>
<td>Remarks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$L = P$</td>
</tr>
<tr>
<td></td>
<td>$\Theta(L^2)$ outputs</td>
</tr>
<tr>
<td></td>
<td>$\Theta(WL + L^2)$ registers</td>
</tr>
<tr>
<td></td>
<td>$\Theta(WL)$ operations/block</td>
</tr>
<tr>
<td></td>
<td>$\Theta(W)$ time units</td>
</tr>
<tr>
<td></td>
<td>$\Theta(LW)$ outputs/time unit</td>
</tr>
<tr>
<td></td>
<td>linear, $\Theta(W)$ operations per output</td>
</tr>
<tr>
<td></td>
<td>Requires strided memory access</td>
</tr>
</tbody>
</table>

Table 3.2: Characteristics of the SIMD implementation of a moving sum based on the state space form and linear look-ahead

The memory used by the algorithm is, for a large part, used as a buffer for the inputs. Since there are $\Theta(P)$ iterations between the first and last time that a block of inputs is accessed, we need to store $\Theta(P)$ blocks of inputs, which results in a memory requirement of $\Theta(P^2)$ registers. We also need $\Theta(W)$ registers to store the $\mathbf{A}$, $\mathbf{AL}$, $\mathbf{b}$, state and new_state arrays. Furthermore, we need $\Theta(WP)$ registers to store the vector arrays $\mathbf{B}_{vec}$, $\mathbf{part1}_{vec}$, $\mathbf{part2}_{vec}$ and $\mathbf{new}_{vec}$, and $\Theta(P)$ registers to store the vectors $\mathbf{input1}_{vec}$, $\mathbf{input2}_{vec}$ and $\mathbf{output}_{vec}$. An additional 4 registers are needed to store the $P$, $W$, $i$ and $j$ variables, though the $P$ and $W$ variables are constant. This results in a total of $\Theta(P^2 + WP)$ registers.

An overview of the performance characteristics of this program is shown in Table 3.2. Note that since $P = L$ holds we have replaced all occurrences of $P$ in the table with $L$. This allows for an easier comparison between the performance of the SIMD program and the VLSI implementation. It turns out that the only differences between the two implementations are the iteration time, and, consequently, the throughput. The iteration time of the VLSI platform is lower because it allows for more parallelism than the SIMD platform. Other than that the performance measures are the same.

### 3.2.2 Two stage strategy

In this section we consider the option of dividing the moving sum system into two separate subsystems. This separation allows us to implement each subsystem individually.
The subsystems are numbered 0 and 1, and the first is specified by:

\[ \text{in}_0(i) = \text{in}(i) \quad (3.14) \]
\[ \text{out}_0(i) = \text{in}_0(i) - \text{in}_0(i - W) \quad (3.15) \]

and the second subsystem is specified by:

\[ \text{in}_1(i) = \text{out}_0(i) \quad (3.16) \]
\[ \text{out}_1(i) = \text{out}_1(i - 1) + \text{in}_1(i) \quad (3.17) \]

This means that the second subsystem produces the outputs of the moving sum system, i.e. when we take (3.17) and substitute (3.16), (3.15) and (3.14) we obtain (3.2). That is, \( \text{out}_1(i) = \text{out}_1(i - 1) + \text{in}(i) - \text{in}(i - W) = \text{out}(i) \).

We obtain a block processing algorithm for the moving sum system by designing a block processing algorithm for system 0 and for system 1. Ideally both algorithms use the same block size, so that they can be combined into one algorithm without requiring any complicated form of buffering.

The first subsystem, system 0, belongs to class \( FSM_* \). There is a straightforward strategy to design block processing algorithms for \( FSM_* \) systems, as we demonstrate in Section 3.3. We apply this strategy from Section 3.3 to obtain a block processing algorithm for the first subsystem.

The second subsystem, system 1, is of the \( FSM_\infty \) class and therefore we use the linear look-ahead approach from Section 2.4 to implement it. Unlike the implementation from the previous section, however, the state of the implementation for this second subsystem is not a vector. It now contains a single scalar value, namely the last output, i.e.:

\[ s_1(i + 1) = \text{out}_1(i) \quad (3.18) \]

This simplifies the implementation of the second subsystem considerably.

### 3.2.2.1 VLSI implementation

In this section we show how both subsystems are implemented in VLSI. The implementation of system 0 is straightforward, and for the system 1 we use the linear look-ahead approach.

An example implementation of the first subsystem, based on the strategy from Section 3.3, is shown in Figure 3.4. Note that the input of the subtraction components in Figure 3.4 are marked to indicate the order of the operands.
This first subsystem, implemented as indicated by Figure 3.4, has very good performance characteristics. No pipelining is needed, since the critical path consists of a single subtraction operation. The iteration period is therefore a single time unit, i.e., the time it takes to perform one subtraction operation. The latency is only one iteration, or $\Theta(L)$ outputs, the amount of memory needed is $W-1$ registers, and the throughput is $L$ outputs per time unit.

So now we turn to the second subsystem. We view this system as a 1st order IIR filter, and implement it just as we have done in Section 3.2.1. The advantage here is that the order of the filter is now independent of the window size $W$, allowing for an implementation with good performance characteristics. No multiplications are needed, and instead of vectors of length $N=W$ we only require computations involving scalars.

The block diagram for this implementation is shown in Figure 3.5. This algorithm is, again, the same as Parhi and Messerschmitt’s “Incremental Block State Architecture” with an increment of 1. Therefore the second subsystem has the same performance characteristics as the VLSI algorithm from Section 3.2.1 would have for $W=1$.

After combining these two implementations, by connecting the outputs of the implementation of first subsystem to the inputs of the implementation of the
Figure 3.5: Linear look-ahead algorithm for the second subsystem with \( L = 6 \)

<table>
<thead>
<tr>
<th>VLSI circuit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency ( \mathcal{L} )</td>
</tr>
<tr>
<td>Memory ( \mathcal{M} )</td>
</tr>
<tr>
<td>Complexity ( \mathcal{C} )</td>
</tr>
<tr>
<td>Iteration period ( \mathcal{I} )</td>
</tr>
<tr>
<td>Throughput ( \mathcal{T} )</td>
</tr>
<tr>
<td>Scalability</td>
</tr>
</tbody>
</table>

Table 3.3: Characteristics of the VLSI implementation of a moving sum based on a 2 subsystem algorithm and linear look-ahead
3.2. INFINITE MEMORY SPAN STRATEGY

Listing 3.4: Subsystem 0 as an SIMD program

```
var P, W, i : scalar;
    input_vec, oldinput_vec, output_vec : vector;
do true →
    input_vec := read(IN, P*i);
    {[input_vec[(p)] = in(P*i+p)]}
    oldinput_vec := read(IN, P*i-W);
    {[oldinput_vec[(p)] = in(P*i+p-W)]}
    output_vec := input_vec - oldinput_vec;
    {[output_vec[(p)] = out(P*i+p)]}
    write(OUT0, P*i, output_vec);
    i := i+1;
od
```

second subsystem, we obtain the implementation for the moving sum system. The performance characteristics of this implementation, as shown in Table 3.3 are a combination of those of the two subsystems.

Note that this implementation has better performance characteristics than the one from Section 3.2.1 Fewer of the performance measures contain the window size \( W \) and the number of operations per output is a small constant.

3.2.2.2 SIMD implementation

In this section we show how both subsystems are implemented in SIMD. First we present the SIMD program for system 0, and after that we use the linear look-ahead approach to construct the program for system 1. For simplicity we consider the implementations such that \( P = L \), though other choices are also possible [38].

The SIMD program for system 0 is shown in Listing 3.4. The iteration period of this program is \( \Theta(1) \) time units and it produces one block of \( P \) outputs in each iteration. Its throughput is therefore \( \Theta(P) \) and its complexity is \( \Theta(P) \) operations per block of outputs. The memory used by the program consist of the buffer for the inputs, the three vector registers \( \text{input_vec}, \text{oldinput_vec} \) and \( \text{output_vec} \), and the three scalar registers for \( P, W \) and \( i \), which results in
a total $\Theta(P + W)$ registers. The latency of the program is one iteration per block, which corresponds to $\Theta(P)$ outputs.

Note that the program in Listing 3.4 uses unaligned memory access to obtain the contents for the variable oldinput_vec. Unaligned memory access means that the address used in the memory access operation is not a multiple of $P$. Some SIMD processors support this kind of memory access, on others we have to emulate it by using two memory access operations, a few shifting operations and masked assignments. We discuss this in more detail in Section 7.2.

The second subsystem, system 1, belongs to the $FSM_\infty$ class. Just as we did for the VLSI implementation, we implement this system using the linear look-ahead approach. To this end we use the program from Listing 3.3. The second subsystem does not correspond to a $W$-th order IIR filter, however, but it corresponds to a first order IIR filter. Therefore we optimize the program from Listing 3.3 to the point where we obtain the program shown in Listing 3.5.

In Listing 3.5 the following definition for $f$ holds:

$$f(i, p) = (+k : 0 \leq k < p : \text{in}(P(i - p + P - 1) + k)) \quad (3.19)$$

The performance characteristics of the program in Listing 3.5 are the same as those of the SIMD program for the linear look-ahead approach from Section 3.2.1 with $W = 1$.

The programs for the two subsystems are combined into a single program for the complete moving sum system. Since these programs implement different systems there is no overlap between them and each single iteration of the first program is followed by a single iteration of the second program. We only have to ensure that the outputs of the first program are read as inputs by the second program, i.e., $\text{OUT0} = \text{IN1}$.

The performance characteristics for the combined program are a combination of those of the two programs for the subsystems. These performance characteristics are shown in Table 3.4.

Just as the VLSI implementation based on the two stage strategy has better performance characteristics than the VLSI implementation based directly on the state space form, the SIMD implementation based on the two stage strategy has better performance characteristics than the SIMD implementation based on the state space form. Just as for the VLSI implementation, the window size $W$ now occurs in fewer of the performance measurements. Furthermore, the algorithm still scales linearly, but now with a small, constant number of operations per output.
3.2. INFINITE MEMORY SPAN STRATEGY

\[
\begin{align*}
\text{var} & \quad P, W, i, \text{state} : \text{scalar}; \\
& \quad \text{input1_vec, input2_vec, output_vec : vector}; \\
& \quad \text{part1_vec, part2_vec : vector}; \\
\{ [P] = P \land [W] = W \land [i] \in N \} \\
\{ \text{Inv}_0 : \text{part1_vec}(p) = f(i, p) \} \\
\{ \text{Inv}_1 : \text{part2_vec}(p) = s_1(P(i - p) + p) \} \\
\{ \text{Inv}_2 : \text{state} = s_1(Pi) \} \\
\text{do true} \rightarrow \\
\quad \text{input1_vec} := \text{strided_read}(\text{IN1}, P*(i+P-1), 1-P); \\
\quad \{ [\text{input1_vec}](p) = \text{in}(P(i-p + P-1) + p) \} \\
\quad \text{input2_vec} := \text{strided_read}(\text{IN1}, P*i, 1-P); \\
\quad \{ [\text{input2_vec}](p) = \text{in}(P(i-p) + p) \} \\
\quad \text{output_vec} := \text{part2_vec} + \text{input2_vec}; \\
\quad \{ [\text{output_vec}](p) = s_1(P(i-p) + p + 1) \} \\
\quad \{ [\text{output_vec}](p) = \text{out}(P(i-p) + p) \} \\
\quad \text{strided_write}(\text{OUT}, P*i, 1-P, \text{output_vec}); \\
\quad \text{part1_vec} := \text{part1_vec} + \text{input1_vec}; \\
\quad \{ [\text{part1_vec}](p) = f(i+1, p+1) \} \\
\quad \{ [\text{part1_vec}](P-1) = (+k : 0 \leq k < P : \text{in}(Pi + k)) \} \\
\quad \text{state} := \text{state} + \text{part1_vec}(P-1); \\
\quad \{ \text{Inv}_2(i := i + 1) \} \\
\quad \text{part1_vec} := \text{shift}(\text{part1_vec}, -1, 0); \\
\quad \{ \text{Inv}_3(i := i + 1) \} \\
\quad \text{part2_vec} := \text{shift}(\text{output_vec}, -1, \text{state}); \\
\quad \{ \text{Inv}_1(i := i + 1) \} \\
\quad i := i + 1; \\
\text{od}
\end{align*}
\]

Listing 3.5: Subsystem 1 as an SIMD program
### 3.3 Finite memory span strategy

In this section we discuss a strategy for designing block processing algorithms with linear scalability for $FSM_*$ systems. We demonstrate this strategy using the moving sum system. Therefore we only use the properties that make it part of the $FSM_*$-class of systems. In other words, we only use that the outputs of the system are a function of the $W$ last inputs, but we do not use the property that this function is based on an associative aggregation operator.

Designing block processing algorithms for $FSM_*$ systems is straightforward. The outputs of an $FSM_*$ system are a function of a finite number ($W$) of preceding inputs. So a block processing algorithm is an algorithm that evaluates this function $L$ times in parallel in each iteration. In short, systems of class $FSM_*$ are embarrassingly parallel.

#### 3.3.1 VLSI implementation

In this section we consider the implementation of the block processing algorithm for the moving sum system, based on our view of this system as an $FSM_*$ system.

The function that maps the last $W$ inputs of the moving sum system onto an output does this by taking the sum of the inputs. Therefore our VLSI implementation consists of $L$ parts that each take the sum of $W$ inputs and thereby produce $L$ outputs in each iteration. A block diagram of an algorithm based on this idea is shown in Figure 3.6.

Without pipelining the algorithm in Figure 3.6 would have a critical path
Figure 3.6: $FSM_*$ based algorithm for the moving sum with $W = 5$ and $L = 6$
that consists of $W$ adders. The algorithm is therefore pipelined by taking $W - 1$ more-or-less diagonal cut-sets as indicated by the dashed lines in Figure 3.6. This reduces the critical path to a single adder.

As always, these $W - 1$ cut-sets introduce a latency in the algorithm of $\Theta(W)$ iterations. Because each iteration results in a new block of inputs, this means that the total latency of the algorithm is $\Theta(WL)$ outputs.

To compute the number of pipeline registers introduced by this pipelining, we number the cut-sets from left to right in Figure 3.6 starting at 0. Cut-set $i$ then intersects with $2L + W - 2 - i$ data paths. Each intersection between the cut-set and a data path introduces a register. So, all $W - 1$ cut-sets together introduce $\Theta(WL + W^2)$ registers.

The complexity of the algorithm is obtained by counting the number of adders, which is $\Theta(WL)$.

These performance characteristics are summarized in Table 3.5. This algorithm has a higher complexity, and therefore a higher number of operations per output, than the two stage strategy from Section 3.2.2. On the other hand it has a lower latency and memory requirements for sufficiently large block sizes, i.e. $L > W$. So, in terms of scalability the two stage strategy from Section 3.2.2 is better, but the $FSM_s$-based strategy presented here is of use in cases where the latency and memory usage are also a concern.

### Table 3.5: Characteristics of the VLSI implementation of the moving sum system using an $FSM_s$-based algorithm

<table>
<thead>
<tr>
<th>VLSI circuit</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency $L$</td>
<td>$\Theta(WL)$ outputs</td>
</tr>
<tr>
<td>Memory $M$</td>
<td>$\Theta(WL + W^2)$ registers</td>
</tr>
<tr>
<td>Complexity $C$</td>
<td>$\Theta(WL)$ operations/block</td>
</tr>
<tr>
<td>Iteration period $I$</td>
<td>$\Theta(1)$ time units</td>
</tr>
<tr>
<td>Throughput $T$</td>
<td>$\Theta(L)$ outputs/time unit</td>
</tr>
<tr>
<td>Scalability</td>
<td>linear, $\Theta(W)$ operations per output</td>
</tr>
</tbody>
</table>

3.3.2 SIMD implementation

The block processing algorithm that evaluates the $FSM_s$ output function $L$ times in parallel can also be implemented on an SIMD processor. In this section we present that implementation and discuss its performance characteristics.

Our SIMD program is based on the VLSI implementation from Figure 3.6.
3.3. FINITE MEMORY SPAN STRATEGY

The VLSI algorithm requires $WL$ addition operations. These operations have to be mapped and scheduled onto the different $P$ processing elements of an SIMD processor. By choosing $L = P$ it is possible to map the operations in the block diagram such that each row corresponds to a different processing element, and each diagonal to a different time slot. The program resulting from this mapping is shown in Listing 3.6.

The characteristics of the resulting SIMD algorithm are shown in Table 3.6. The latency is obtained by noting that the last input on which an output depends is read for the first time in the iteration in which the output is produced. Therefore the latency is a single iteration or $\Theta(P) = \Theta(L)$ outputs. The program requires several vector registers, each one consisting of $\Theta(L)$ registers plus a buffer of size $W$ registers for the inputs, resulting in a total memory requirement of $\Theta(L+W)$. The complexity is $\Theta(WL)$, since each iteration con-
CHAPTER 3. SOLUTION STRATEGIES

<table>
<thead>
<tr>
<th>Nr. of PEs</th>
<th>$P$</th>
<th>$L = P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>$\mathcal{L}$</td>
<td>$\Theta(L)$ outputs</td>
</tr>
<tr>
<td>Memory</td>
<td>$\mathcal{M}$</td>
<td>$\Theta(W + L)$ registers</td>
</tr>
<tr>
<td>Complexity</td>
<td>$C$</td>
<td>$\Theta(WL)$ operations/block</td>
</tr>
<tr>
<td>Iteration period</td>
<td>$I$</td>
<td>$\Theta(W)$ time units</td>
</tr>
<tr>
<td>Throughput</td>
<td>$T$</td>
<td>$\Theta\left(\frac{L}{W}\right)$ outputs/time unit</td>
</tr>
<tr>
<td>Scalability</td>
<td></td>
<td>linear, $\Theta(W)$ operations per output</td>
</tr>
</tbody>
</table>

Table 3.6: Characteristics of the SIMD implementation of the moving sum system using an $FSM_*$ based algorithm

...}

3.3.3 Other strategies

So far we have discussed only one strategy for designing a block processing algorithm for the moving sum system based on the properties of the $FSM_*$ class. There are other strategies, however, based on the idea that the moving sum corresponds to a FIR filter of order $W - 1$ where all the filter coefficients are 1. So, any strategy for FIR filters can also be applied to the moving sum.

One such strategy is strength reduction [65]. This technique is used to design block processing FIR filters in which multiplication operations are traded for (slightly more) addition operations. This technique is especially suited for VLSI implementations, since a component that performs an addition is generally smaller and faster than a component that performs a multiplication. The strength reduction technique does not provide us with a better algorithm for the moving sum however. The filter coefficients of the FIR filter that computes

...
the moving sum are all 1, so there are no multiplications in the FIR filter to trade with in the first place.

Another strategy is to write the FIR filter in the state space form discussed in Section 3.2.1. This state space form is based on the following equations:

\[
\begin{align*}
\mathbf{s}(i+1) &= A \mathbf{s}(i) + b \mathbf{in}(i) \quad (3.20) \\
\mathbf{out}(i) &= c^T \mathbf{s}(i) + d \mathbf{in}(i) \quad (3.21)
\end{align*}
\]

where \( A \) is a \( N \times N \) matrix, the \( b, c \) and the state \( \mathbf{s}(i) \) are vectors of length \( N \), \( d \) is a scalar, and \( N \) equals the order of the filter. For our FIR filter we would then have \( N = W - 1 \) with:

\[
\begin{pmatrix}
A & b \\
\mathbf{c}^T & d
\end{pmatrix} = \begin{pmatrix}
0 & \ldots & \ldots & \ldots & 0 & 1 \\
1 & \ddots & \ddots & \ddots & \vdots & 0 \\
0 & \ddots & \ddots & \ddots & \vdots & \vdots \\
\vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\
0 & \ldots & 0 & 1 & 0 & 0 \\
1 & \ldots & \ldots & \ldots & 1 & 1
\end{pmatrix} \quad (3.22)
\]

We have already discussed the state-space form, however, in Section 3.2.1 and we will not repeat it here. In fact, by choosing the transformation matrix \( \mathbf{M} \) such that:

\[
\mathbf{M} = \begin{pmatrix}
1 & \ldots & \ldots & 1 \\
0 & \ddots & \ddots & \ddots \\
\vdots & \ddots & \ddots & \ddots \\
0 & \ldots & 0 & 1
\end{pmatrix} \quad (3.23)
\]

the state-space form obtained by applying (3.7) to (3.6) is:

\[
\begin{pmatrix}
A & b \\
\mathbf{c}^T & d
\end{pmatrix} = \begin{pmatrix}
0 & \ldots & \ldots & \ldots & 0 & 1 \\
1 & \ddots & \ddots & \ddots & \vdots & 0 \\
0 & \ddots & \ddots & \ddots & \vdots & \vdots \\
\vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\
\vdots & \ldots & 1 & 0 & 0 & \vdots \\
0 & \ldots & 0 & 1 & 1 & 0 \\
1 & \ldots & \ldots & 1 & 0 & 1
\end{pmatrix} \quad (3.24)
\]

which is, essentially, a FIR filter.
Note that the last element of the state vector of this FIR filter is unused in (3.24), i.e., that element of the state is neither used to compute an output nor used in updating the state vector. By removing this unused state variable we get (3.22). So the implementation of our moving sum/FIR filter based on the state space form can be derived by following the strategy described in Section 3.2.1. The only difference is that the matrix \( A \) and the vector \( c \) differ slightly.

For these reasons we have only discussed the general finite memory span strategy and not these other two strategy. The first one, the strength reduction strategy, does not apply to the moving sum system and the second one, based on the state space form, is very similar to the strategy from Section 3.2.1.

### 3.4 Associative window computation strategy

So far we considered the moving sum system as a system from the \( FSM_\infty \) and \( FSM_* \) classes. In this section we consider the possibilities when we view the moving sum systems as an \( FSM_* \)-class of system, i.e., a system whose outputs are the aggregation of \( W \) inputs. Here we only provide a brief sketch of the methods used to obtain an implementation for these kinds of systems. The existing methods are discussed in more detail in \([36, 29, 34]\). In the rest of this thesis we unite these methods under a single notation, and we introduce a new method that is suitable for SIMD implementation. In this section we only provide an overview of the extraordinary performance characteristics obtained by this strategy.

Systems of the \( FSM_* \)-class, like the moving sum, aggregate a number of inputs using an associative operator. In case of the moving sum the number of inputs, or window size, is \( W \) and the operator used is the addition operator.

The class \( FSM_* \) is a sub-class of the class \( FSM_* \), and we have already shown that designing a block processing algorithm with linear scalability for \( FSM_* \) systems is straightforward. We are still interested in designing block processing algorithms for \( FSM_* \) systems because we can obtain algorithms with an even lower complexity than for the more general \( FSM_* \) systems.

Consider the moving sum system with a window size of \( W = 3 \). An algorithm for such a system requires 2 additions per output when it is designed according to the \( FSM_* \) method described in the previous section. That is, each output is the sum of three inputs, therefore requiring two additions to compute. We can,
3.4. ASSOCIATIVE WINDOW COMPUTATION STRATEGY

however, also base our algorithm on the following equations:

\[ \text{out}(2i) = \text{in}(2i - 2) + f(i) \quad (3.25) \]
\[ \text{out}(2i + 1) = f(i) + \text{in}(2i + 1) \quad (3.26) \]
\[ f(i) = \text{in}(2i - 1) + \text{in}(2i) \quad (3.27) \]

So, we compute the even outputs by adding inputs to the intermediate results represented by the function \( f \), and we compute the odd outputs by adding those same intermediate results to other inputs. Therefore we require one addition to compute each output, plus the additions needed to compute the intermediate results. The intermediate results are the sum of two inputs and require one addition to compute. For each two outputs, however, we need only a single intermediate result, therefore the cost of the addition needed to compute that intermediate result can be amortized over two outputs. So, an algorithm based on the equations above requires only 3 additions per 2 outputs, or 1.5 additions per output, which is an improvement of 25\% over the \( FSM \) based algorithm, which requires 2 additions per output.

The improvement for such a small window size \( W = 3 \) is already significant, but for larger window sizes even larger improvements can be made. In fact, in the rest of this thesis we show how algorithms for \( FSM_\circ \) class systems can be obtained with characteristics as shown in Table 3.7.

Note that Table 3.7 shows that the complexity and throughput of the algorithm do not depend on the size of the window. So, the embarrassingly parallel implementation based on the \( FSM_\circ \) strategy requires \( \Theta(W) \) operations per output, but the \( FSM_\circ \)-based strategy requires only \( \Theta(1) \). Therefore the number of operations required per output remains constant, regardless of the window size.

Note that for both the VLSI and SIMD implementation the block size can not be chosen freely. The block size has to be a multiple of a certain constant \( O(W) \). The reason for this is illustrated by our moving sum example for \( W = 3 \): we obtain the result of 1.5 operations per output by amortizing 3 operations over 2 outputs, namely one odd and one even output. So, to maintain this ratio of 1.5 operations per output we must always compute the same number of odd and even outputs in a single iteration. For \( W = 3 \), therefore, the block size \( L \) must be a multiple of 2.

The performance characteristics for the VLSI implementation in Table 3.7 are better than any other VLSI implementation we have discussed in this chapter. Therefore, if the restriction on the block sizes is admissible, this implementation is preferable over the others.
CHAPTER 3. SOLUTION STRATEGIES

<table>
<thead>
<tr>
<th>VLSI circuit</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency ( L )</td>
<td>( \Theta(L \log W) ) outputs</td>
</tr>
<tr>
<td>Memory ( M )</td>
<td>( \Theta(L \log W) ) registers</td>
</tr>
<tr>
<td>Complexity ( C )</td>
<td>( \Theta(L) ) operations/block</td>
</tr>
<tr>
<td>Iteration period ( I )</td>
<td>( \Theta(1) ) time units</td>
</tr>
<tr>
<td>Throughput ( T )</td>
<td>( \Theta(L) ) outputs/time unit</td>
</tr>
<tr>
<td>Scalability</td>
<td>linear, ( \Theta(1) ) operations per output</td>
</tr>
<tr>
<td>Remarks</td>
<td>( L = O(W)I )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIMD program</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr. of PEs ( P )</td>
<td>( L = O(W)P )</td>
</tr>
<tr>
<td>Latency ( L )</td>
<td>( \Theta(L) ) outputs</td>
</tr>
<tr>
<td>Memory ( M )</td>
<td>( \Theta(L) ) registers</td>
</tr>
<tr>
<td>Complexity ( C )</td>
<td>( \Theta(L) ) operations/block</td>
</tr>
<tr>
<td>Iteration period ( I )</td>
<td>( O(W) ) time units</td>
</tr>
<tr>
<td>Throughput ( T )</td>
<td>( \Theta(P) ) outputs/time unit</td>
</tr>
<tr>
<td>Scalability</td>
<td>linear, ( \Theta(1) ) operations per output</td>
</tr>
<tr>
<td>Remarks</td>
<td>Requires the shuffle operation</td>
</tr>
</tbody>
</table>

Table 3.7: Characteristics of the moving sum using an \( FSM_*^\circ \)-based algorithm

Comparing the SIMD implementation of this algorithm to the other SIMD implementations in this chapter is not as straightforward. When the algorithms are implemented using the same number of PEs \( P \), as opposed to implementations using the same block size \( L \), we obtain the following results: The \( FSM_*^\circ \)-based strategy from Section 3.3 is the only implementation with better latency and memory requirements. The 2-stage \( FSM_\infty \)-based strategy from Section 3.2.2 is the only implementation with a better complexity and iteration period. This fact, however, does not result in a higher throughput for the \( FSM_\infty \)-based strategy, although that strategy is the only one that can match the throughput of the \( FSM_*^\circ \)-based implementation from this section. Also, in contrast to the 2-stage \( FSM_\infty \)-based strategy, the implementation from this section does not need strided memory access operations, it needs shuffle operations instead.

So, if the latency and memory requirements are a concern, the \( FSM_* \)-based implementation from Section 3.2.2 is an option, though the \( FSM_*^\circ \)-based implementation has comparable latency and memory requirements for small window sizes \( W \). If the throughput has the highest priority, the \( FSM_*^\circ \)-based implementation is preferred, since it does not require the strided memory access and has a lower latency and memory requirement than the \( FSM_\infty \)-based implementation.
This type of $FSM^\oplus$-based implementation is especially interesting because it is possible to convert any look-ahead computation into a associative window computation, as explained in Section 4.5.1. Therefore, by performing this conversion, we can design SIMD algorithms with linear scalability that perform look-ahead computations. This effectively lowers the requirements on the SIMD processor for look ahead computations with linear scalability from the support of strided memory access to the support of the shuffle operation.

3.5 Conclusion

There are many strategies that lead to a block processing algorithm for a system that computes the moving sum. In this chapter we discussed a number of them that lead to block processing algorithms that scale linearly and that can be implemented in both VLSI and on SIMD.

Two of our strategies are based on the linear look-ahead approach, which gives good results in terms of scalability but has the drawback that the SIMD variant can only be run on a processor that supports strided memory access.

At this point we must note that there is a slight problem with the single stage implementation based on the linear look-ahead strategy from Section 3.2.1. The state $s(i)(j)$ of that implementation is specified by:

$$s(i)(j) = (\sum_{k: 0 \leq k < i-j} \text{in}(k)) \text{ for } 0 \leq j < W$$  \hspace{1cm} (3.28)

The state $s(i)(j)$ therefore equals the sum of all the inputs up to $i-j$. So, even though the input alphabet is finite, there is no limit to the range of $s(i)(j)$.

Fortunately this problem can be solved. The outputs of the algorithm depend only on the differences between the elements of the state vector. So adding, or subtracting, a constant to each element of the state vector is an operation that does not influence the outputs. Therefore we can perform a normalization step at any time when the value of the elements of the state vector become too large. Probably the easiest way to do this, is by replacing all the additions in the algorithm by additions modulo some suitable constant $M$. As long as $M$ is larger than the range of possible outputs, enough information is contained in the state to compute the outputs. Furthermore, any additions done in hardware are, in general, already additions modulo some power of two. So implementing this solution should not take much effort.

When we switch from infinite to finite data types during implementation we must not only be wary of round-off effects as discussed in Section 2.2.4 but
we must also take care that the state can be represented in a finite data type. The moving sum example demonstrates that a finite input alphabet and a finite output alphabet does not automatically mean that any state representation for such an algorithm is finite.

Another strategy that we discussed in this chapter is the finite memory span strategy for $FSM_*$ systems. This is a strategy based on the fact that $FSM_*$ systems are embarrassingly parallel. To obtain a scalable block processing algorithm for these systems we duplicate the computations such that each output is computed individually. This is a relatively easy strategy that works well.

Because the system is a member of the $FSM_\odot^*$ class, however, we can reuse some of the computations between the different outputs. This results in an efficient algorithm, one in which the complexity and throughput no longer depend on the window size.

All these block processing algorithms scale linearly. This means that their throughput is proportional to the amount of hardware used in implementing them. So they achieve our primary goal of linear scalability. To compare the algorithms with each other, however, we look at the complexity per output. The complexity per output is constant for algorithms that scale linearly, but those that have a lower complexity per output require less hardware to implement for a certain throughput than those with a higher complexity per output.

Two of the algorithms we have presented, namely the single stage infinite memory span strategy and the finite memory span strategy have a complexity per output that depends on the window size $W$. The two other algorithms, namely the two stage infinite memory span strategy and the associative window computation strategy have a complexity per output that is constant. In Figure 3.7 we have plotted, for various window sizes $W$, the complexity per output in terms of addition operations per output for each of the algorithms produced by the different strategies.

Figure 3.7 shows that the single stage infinite memory span strategy always results in the highest complexity for the moving sum system. The associative window computation strategy, on the other hand, always results in (one of the) lowest complexity algorithms. For large window sizes difference between the two stage infinite memory span strategy and the associative window computation strategy becomes zero, so in that respect there is little difference between these two strategies.

For systems other than the moving sum the graph in 3.7 will look very different. First of all, the complexity of the operations used to put the algorithm in the algebraic state update form will be different. Secondly, not all strategies can be applied to all systems. A system that does not belong to class $FSM_*$
Figure 3.7: Comparison of the algorithms. All these algorithms scale linearly.
cannot benefit from the finite memory span strategy, for example.

Similarly, it would seem that not all systems can benefit from the associative window computation strategy. In the remainder of this thesis we will show, however, how the look-ahead computations for any system can be rewritten to a window computation. This enables us to apply the low-complexity algorithms obtained by the associative window computation strategy to any system, including those of the $FSM_\infty$ class.
Chapter 4

Associative window computations

4.1 Introduction

In this chapter we discuss associative window computations in more detail, expand their definition, and examine their properties. In particular, we show that a look-ahead computation can be defined in terms of an associative window computation. This enables us to use any efficient algorithm for associative window computations to obtain an efficient algorithm for look-ahead computations.

Furthermore, we present the generic approach that allows us to actually design efficient algorithms for associative window computations. This approach consists of tiles, which represent intermediate results of the window computation, and tile-sets. This enables us to divide the window computations, recursively, into smaller window computations. This dividing of window computations into smaller window computations enables us to apply divide and conquer, and other strategies, to obtain efficient block processing algorithms, as demonstrated in Chapter 5.

In this chapter we also extend associative window computations on one-dimensional streams of data to window computations on multi-dimensional grids of data. This extension is useful because many image processing algorithms can be modelled as 2-dimensional window computations, or even 3-dimensional window computations for 3-D image processing.

Next to the transformation of a look-ahead computation into an associative
window computation, we also examine the reverse transformation: expressing an associative window computation in terms of a look-ahead computation. In Chapter 3 we have already shown that it is possible to implement an associative window computation using a look-head computation. In Section 4.4 we generalize this approach for associative window computations with certain properties.

The transformation from look-ahead computation to an associative window computation is the most important feature of this chapter, however. This transformation, combined with our approach to design efficient algorithms for associative window computations, enables us to define new algorithms, with better performance characteristics, for look-ahead computations.

4.2 Tiling

Each output of a window computation is the aggregation of some inputs. The inputs and the order in which the, possibly non-commutative, aggregation operator has to be applied to obtain a certain output are specified by a list. This list contains the indices of the inputs that have to be aggregated, and the order of the indices indicates the order in which the aggregation operator has to be applied. We refer to such a list as a tile. In this section we introduce tiles and tile-sets and discuss their properties. In Section 4.3 these tiles and tile-sets are used to define associative window computations.

4.2.1 Tiles

We call a list containing the indices of inputs a tile or window. Specifically, we call any list of indices a tile and use the term window only for tiles that contain a list of indices used for the computation of an output.

More formally, the inputs of the system are organized in a $D$-dimensional grid with $D > 0$ and coordinates from the set $\mathbb{Z}^D$. The data associated with a certain grid location is returned by function $\text{data}$:

$$\text{data} : \mathbb{Z}^D \rightarrow \mathbb{D}$$

(4.1)

Note that the input stream of a system is one-dimensional, while the grid used to organize the inputs is $D$-dimensional. Therefore there must be a mapping between the denumerable grid-coordinates and the indices of the input stream. This mapping is discussed in Section 4.3.

**Definition 4.2.1 (Tile).** A tile is an element of the set $\mathbb{T} = \mathbb{L}(\mathbb{Z}^D)$, where $\mathbb{L}(\mathbb{Z}^D)$ denotes the set of all finite lists over the grid locations $\mathbb{Z}^D$.  

We say that two tiles $S$ and $T$ have an equivalent shape, when there exists a translation vector $t \in \mathbb{Z}^D$ such that $T = S + t$ with tile $S + t$ defined as $(S + t)(i) = S(i) + t$. So, we do not use the term “shape” in the geometrical sense; tiles that differ by a rotation are not considered equivalently shaped and neither are tiles that have their elements permuted. Moreover, we allow tiles with shapes that are disconnected, i.e. shapes like $[(0, 0), (2, 2)]$.

In case we need to order a number of tiles, the tiles are sorted on their grid locations and each grid location is sorted on its coordinates. So we have, for example $[(0, 0), (1, 1)] < [(0, 0), (1, 2)]$.

In the rest of this thesis we often refer to tiles themselves, but also to the results obtained when the inputs specified by the tile are aggregated. Therefore we introduce the following shorthand to denote the aggregation of the inputs specified by the coordinates in a tile:

$$\llangle T \rrangle = (\circ g : g \in T : \text{data}(g)) \quad (4.2)$$

This shorthand allows us write the output of an $FSM^\oplus$ system as $\text{out}(i) = \llangle W_i \rrangle$, where $W_i = [i - W + 1, \ldots, i - 1, i]$.

From equation (4.2) and the associativity of the $\circ$ operator it follows that:

$$\llangle T_1 \oplus T_2 \rrangle = \llangle T_1 \rrangle \circ \llangle T_2 \rrangle \quad (4.3)$$

where $\oplus$ is the operator that concatenates two lists. We use this property to structure the computation of windows using tiles that represent intermediate results, i.e., sub-lists of the windows.

### 4.2.2 Tile-sets

In the previous section we have introduced tiles and windows, and their properties. Since the systems in which we are interested compute an infinite number of outputs, we are not interested single tiles or windows, but rather in sets of them.

Specifically, we are interested in sets of tiles with equivalent shapes, laid out in repetitive patterns, which reflects the simple computation performed by $FSM^\oplus$ systems. So we define the following function:

$$\text{tiles} : (T \times \mathbb{Z}^D \times \mathbb{Z}^D) \rightarrow \mathbb{P}(T)$$

$$\text{tiles}(S, f, p) = \{S + f + (g \cdot p) : g \in \mathbb{Z}^D\} \quad (4.4)$$

In this definition the dot-operator “$\cdot$” stands for coordinate-wise vector multiplication.
CHAPTER 4. ASSOCIATIVE WINDOW COMPUTATIONS

Figure 4.1: \texttt{tiles}(S, f, p) with \( S = [(0, 0), (1, 0), (0, 1)] \), \( f = (1, 2) \) and \( p = (4, 3) \)

Function \texttt{tiles} produces an infinite number of copies of tile \( S \) that are spread over the entire grid in a periodic fashion, as illustrated in Figure 4.1. We call \( S \) the shape, \( f \) the phase, and \( p \) the period of the resulting tile set.

**Definition 4.2.2 (Tile-set).** A tile-set \( T \) is a set of tiles that can be represented by a triple \((S_T, f_T, p_T)\) whose elements specify, respectively, the shape, phase and period of the tiles in the tile-set. The exact relation between the set representation and the triple representation is specified by the function \texttt{tiles} (4.4), which returns the set of tiles when the elements of the triple are used as its parameters. In the remainder of this thesis we interchange the representation of \( T \) as a triple and as a tile-set freely.

Note that there are multiple triples that represent the same tile-set, since \( \texttt{tiles}(S_T + k f_T, (1 - k) f_T, p_T) \) generates the same tile set for every \( k \in \mathbb{Z} \). Therefore we introduce the function \texttt{norm} that rewrites a triple to a unique canonical form. This canonical form describes the same tile set \((4.5)\), but its phase lies between zero and the period \((4.6)\). Also, the set of grid locations forming its shape are positioned in the positive hyper-quadrant \((4.7)\) in such a way that there is no tile with the same shape in the positive hyper-quadrant that lies closer to the origin \((4.8)\):

\[
\texttt{tiles} (\texttt{norm}(T)) = \texttt{tiles}(T) \quad (4.5)
\]
\[
0_D \leq f_{\texttt{norm}(T)} < p_{\texttt{norm}(T)} \quad (4.6)
\]
\[
S_{\texttt{norm}(T)} \cap \mathbb{N}^D = S_{\texttt{norm}(T)} \quad (4.7)
\]
\[
(\forall t : t \in \mathbb{N}^D \land t \neq 0_D : (S_{\texttt{norm}(T)} - t) \cap \mathbb{N}^D \neq (S_{\texttt{norm}(T)} - t)) \quad (4.8)
\]
4.3. ASSOCIATIVE WINDOW COMPUTATIONS

where \( \mathbf{0}_D \) is a vector of length \( D \) that contains only zeroes. Furthermore, relations like “\( \leq \)” and “\( < \)” use coordinate-wise comparison and they only hold for the entire vector when they hold for all of its elements. The triple used to generate the tile-set in Figure 4.1 is therefore in canonical form.

Due to this normalization practically all of the one-dimensional tiles we consider are lists containing the natural numbers from 0 up to and excluding the size of the tile\(^1\). Since many of our examples consider one-dimensional tiles we introduce the following shorthand to concisely express such shapes:

\[
\langle n \rangle = [g | g \in \mathbb{N} \land 0 \leq g < n]
\] (4.9)

So we write, for example, \( \langle 3 \rangle = [0, 1, 2] \) and \( \langle 7 \rangle = [0, 1, 2, 3, 4, 5, 6] \).

4.3 Associative window computations

In this section we redefine the notion of an associative window computation in terms of the tiles and tile-sets introduced in the previous section. This effectively extends window computations on a one-dimensional input stream to window computations on a multi-dimensional grid. Since a system that performs a window computation receives its inputs and produces its outputs in a certain order, we also present a mapping from the multi-dimensional grid to the one-dimensional streams. We also briefly discuss block processing algorithms based on multi-dimensional blocks.

4.3.1 Definition of an associative window computation

Using the tiles from the previous section we define associative window computations as:

**Definition 4.3.1** (Associative window computation). *An associative window computation is a computation parameterized by*

- a number \( D \in \mathbb{N}^+ \) that specifies the dimension of the grid,
- a tile-set \( W \) containing the so-called windows of finite size \( W \),
- and a semi-group \( (\mathbb{D}, \circ) \).

\(^1\)The only exceptions are disconnected tiles like, for example, \([0, 4]\).
Given a function \( \text{data} : \mathbb{Z}^D \rightarrow \mathbb{D} \) that specifies the data associated with the various grid locations, the computation results in \( [W] \) for every \( W \in \mathcal{W} \), i.e., it aggregates the data described by the set of windows using the associative binary operator \( \circ \). □

Systems from class \( F SM_\circ \), as defined in (1.10), adhere to this definition. The grid dimension for such systems is \( D = 1 \) and the set of windows is \( \mathcal{W} = ([W], 0, 1) \). Definition [4.3.1] however, also includes other systems. We therefore make the distinction between associative window computation systems, i.e. \( F SM_\circ \) systems, and systems that perform an associative window computation according to Definition [4.3.1]. Most of our examples concern window computations as performed by the \( F SM_\circ \) systems, but our algorithm design techniques also apply to systems of other classes that perform an associative window computation.

Some more examples of associative window computations according to Definition [4.3.1] are:

**The moving sum** with window size \( W \), as described in Chapter 3:

- The grid dimension is \( D = 1 \).
- The set of windows is \( \mathcal{W} = ([W], 0, 1) \).
- The semi-group is \((\mathbb{D}, \circ) = (\mathbb{N}, +)\).

**The 2-dimensional window sorter** with window size \( W = w^2 \):

- The grid dimension is \( D = 2 \).
- The shape of the windows is \( S_W = [(i, j) \mid 0 \leq i < w \land 0 \leq j < w] \).
- The set of windows is \( \mathcal{W} = (S_W, (0, 0), (1, 1)) \).
- The semi-group is \((\mathbb{D}, \circ) = (\mathbb{L}(D') \leq, +_\leq)\), where \( \mathbb{L}(D') \leq \) is the set of sorted lists, with the sorting order defined by the relation \( \leq \), over the alphabet \( D' \) and \( +_\leq \) is the operator that combines two sorted into a sorted list, i.e., the merge sort operator.

Note that in case of the two dimensional window sorter the inputs are lists of size one, while the outputs are lists of size \( W = w^2 \).

These examples can be used to construct many other systems. By using a simple post-processing step that divides the output of the moving sum computation by \( W \) we obtain a computation that results in the moving average. Similarly, window sorting can be used as a pre-processing step to obtain a number of statistics about a window. Window sorting also makes it possible to
4.3. ASSOCIATIVE WINDOW COMPUTATIONS

compute the median, or any other rank or quantile in a post-processing step. Furthermore, additional processing can be performed after the sorting to obtain the modulus, i.e. the most common value, or to obtain number of unique values in the window.

We now also introduce the notion of separable and non-separable window computations, akin to separable and non-separable filters \[7\]. We start with an example. Consider the 2-dimensional moving sum:

The 2-dimensional moving sum with window size \( W = w^2 \):

- The grid dimension is \( D = 2 \).
- The shape of the windows is \( S_W = [(i, j) \mid 0 \leq i < w \land 0 \leq j < w] \).
- The set of windows is \( W = (S_W, (0, 0), (1, 1)) \).
- The semi-group is \((\mathbb{D}, \odot) = (\mathbb{N}, +)\).

This computation is separable, because we can perform this computation using a 1-dimensional window and applying it \( D \) times. The results of the 2-dimensional moving sum can be obtained by first performing a moving sum with a one-dimensional window \( S_W = [(i, 0) \mid 0 \leq i < w] \) and subsequently perform another moving sum on the outputs of the first moving sum using the one-dimensional window \( S_W = [(0, j) \mid 0 \leq j < w] \).

More formally, an associative window computation is separable if there exist two associative binary operators \( \odot_1 \) and \( \odot_2 \), two shapes \( W_1 \) and \( W_2 \), and two vectors \( p_{W_1} \) and \( p_{W_2} \) such that:

\[
\begin{align*}
(\forall \mathbf{g}_1, \mathbf{g}_2 : \mathbf{g}_1 \in W_1 \land \mathbf{g}_2 \in W_2 : \mathbf{g}_1 \text{ and } \mathbf{g}_2 \text{ are orthogonal}) & \quad (4.10) \\
p_{W_1} \text{ and } p_{W_2} \text{ are orthogonal} & \quad (4.11) \\
(\forall \mathbf{g} : \mathbf{g} \in \mathbb{Z}^D : \langle S_W + f_W + (\mathbf{g} \cdot p_W) \rangle) = & \\
(\odot_1 \mathbf{g}_1 : \mathbf{g}_1 \in W_1 : (\odot_2 \mathbf{g}_2 : \mathbf{g}_2 \in W_2 : & \\
\text{data}((\mathbf{g} \cdot (p_{W_1} + p_{W_2})) + \mathbf{g}_1 + \mathbf{g}_2))) & \quad (4.12)
\end{align*}
\]

These equations specify that the associative window computation can be divided into two separate, orthogonal, window computations. When these equations hold we can perform the associative window computation with \( W = (S_W, f_W, p_W) \) and the aggregation operator \( \odot \) by first performing an associative window computation with \((W_2, 0_D, p_{W_2})\) and the aggregation operator \( \odot_2 \) and using the results of that computation as input for an associative window computation with \((W_1, 0_D, p_{W_1})\) with the aggregation operator \( \odot_1 \).
A separable window computation can therefore be split into two window computations. Since the results of the first window computation can be used multiple times in the second window computation, it is generally easier to obtain an efficient algorithm for a separable computation than for a non-separable one. Furthermore, the window computations into which a separable window computation can be separated are often separable themselves, allowing the original window computation to be split into, up to, \( D \) distinct window computations, one for each dimension. This makes separable window computations relatively easy to implement, and therefore we are mainly interested in the more challenging, non-separable window computations.

### 4.3.2 Mapping between grids and streams

Definition 4.3.1 defines associative window computations, but systems that perform an associative window computations consist of something more. These systems have to receive their inputs and produce their outputs in a certain order. In this section we discuss the mapping between the multi-dimensional grid in which the data is organized and the one-dimensional input and output streams of a system.

We introduce a mapping from function \( \text{data} \) on the input stream \( \text{in} \), and a mapping from the tiles in \( W \) to the output stream \( \text{out} \). This mapping is expressed by the following bijective functions:

\[
\text{inmap} : \mathbb{N} \rightarrow \mathbb{Z}^D \\
\text{outmap} : \mathbb{N} \rightarrow \mathbb{Z}^D
\]

These functions represent a traversal of the grid in which the data is organized. Generally the \( \text{inmap} \) and \( \text{outmap} \) represent the same traversal. We have different functions for each mapping, however, to be able to represent a shift of the input or output data. In many 2-dimensional filters, for example, the output obtained by aggregating a window of inputs is associated with the grid location at the center of the window, not with the grid location of the last input.

The mapping of the data represented by function \( \text{data} \) on the input stream is specified as:

\[
in(i) = \text{data}(\text{inmap}(i))
\]

The mapping of the outputs associated with the windows in \( W = (S_W, f_W, p_W) \) is based on the function \( \text{tiles} \) used to generate the tile-set \( W \). This mapping is specified as:

\[
\text{out}(i) = \langle S_W + f_W + (\text{outmap}(i) \cdot p_W) \rangle
\]
4.3. ASSOCIATIVE WINDOW COMPUTATIONS

Inputs are read only once by the system, so each input that is used to produce an output later on has to be stored as part of the state of the system. The two mapping functions therefore have an impact on the memory span of the system. To keep the memory span of finite, the number of steps between reading an input and producing the outputs that depend on it must be finite. For associative window computations with \( W > 1 \), however, it is not possible to find such traversals. This is a problem since an infinite memory span for a system that performs an associative window computation means that the system cannot be represented using a finite state.

Fortunately we are only interested in parts of the grid in practice, therefore we only need traversals of those limited parts of the grid. For one-dimensional grids we are, for example, only interested in the results obtained by aggregating the data with non-negative coordinates, and we use the following mapping functions:

\[
\text{inmap}(i) = i \\
\text{outmap}(i) = i - W + 1
\]

These mapping functions, together with an associative window computation with \( D = 1 \) and \( W = \langle(W), 0, 1 \rangle \), correspond to the \( FSM^c \) system as defined in (1.10). For higher dimensional grids we are generally only interested in the results obtained by aggregating the data from a subset of the grid bounded by hyper-bar. In such cases designing the walk corresponds to mapping a multi-dimensional array onto a linear memory and using, for example, a row-major ordering scheme to obtain the traversal.

The exact definitions of functions \text{inmap} and \text{outmap} affects the memory requirements of the algorithm. They have no effect on the time complexity of the computation, nor on its critical path, nor on any of the other performance measurements. Furthermore, all algorithms are affected equally by the order in which inputs are received and produced. In the remainder of this thesis we therefore do not focus on these functions, but focus on the actual computation instead.

To this end we use some short-hands to omit the mapping functions \text{inmap} and \text{outmap} from our notation. We overload function \text{in} so that \text{in}(\text{g}) for a grid location \text{g} denotes \text{data}(\text{g}). This allows us to switch between grid locations and stream indices when necessary. Furthermore, we use \( W_i = S_W + f_W + (\text{outmap}(i) \cdot p_W) \) to indicate the grid-locations on which an output is based, so that we can write \( \text{out}(i) = \langle W_i \rangle \).


4.4 From window to look-ahead computation

In section 3.2 we have demonstrated that the moving sum, an associative window computation, can be implemented using a look-ahead strategy. In this section we discuss this strategy for associative window computations in general and see what properties are needed to follow this strategy. It turns out that this strategy is only applicable to associative window computations with very specific properties.

The moving sum is an example of an associative window computation and a block processing algorithm can therefore be obtained by computing each output individually. In practice, however, the moving is not implemented by aggregating the inputs for each window separately. As shown in Section 3.2, it is possible to use fewer operations. This is accomplished by extending the semi-group, say \((\mathbb{D}, \circ) = (\mathbb{N}, +)\), to the group \((\mathbb{D}, \circ) = (\mathbb{Z}, +)\), i.e., by making sure that every element in \(\mathbb{D}\) has an inverse with respect to \(\circ\). The result from the previous window is then used to compute the new window by subtracting the input that no longer falls within the current window and adding the new input. This means that we just need two operations per output, regardless of the window size.

This approach can be generalized for associative window computation where the semi-group \((\mathbb{D}, \circ)\) is actually a group, or where it can be extended to become a group. In such cases we not only have the property of closure and associativity, but we also have an identity element \(1\circ\):

\[
(\exists 1\circ : 1\circ \in \mathbb{D} : (\forall a : a \in \mathbb{D} : 1\circ \circ a = a \circ 1\circ = a)) \tag{4.19}
\]

and each element of \(\mathbb{D}\) has an inverse:

\[
(\forall a : a \in \mathbb{D} : (\exists a^{-1} : a^{-1} \in \mathbb{D} : a^{-1} \circ a = 1\circ)) \tag{4.20}
\]

In case of an \(FSM^\circ\) system, where \(W = ([W], 0, 1)\), this means that the outputs are defined by the relation:

\[
\text{out}(i) = \text{in}(i - W)^{-1} \circ \text{out}(i - 1) \circ \text{in}(i) \tag{4.21}
\]

where we assume that \(\text{in}(i) = \text{out}(i) = 1\circ\) for \(i < 0\).

For associative window computations in general we have:

\[
\text{out}(i) = \langle \text{rev}(A_i) \rangle^{-1} \circ \text{out}(i - 1) \circ \langle B_i \rangle \tag{4.22}
\]

where function \(\text{rev}\) reverses a list and \(A_i\) and \(B_i\) are lists for which the following holds:

\[
W_{i-1} = A_i + W'_{i-1} \tag{4.23}
\]

\[
W_i = W'_{i-1} + B_i \tag{4.24}
\]
So $A_i$ holds the indices of inputs that have to be removed from the start of $W_{i-1}$, and $B_i$ holds the indices of inputs that have to be added to the end of $W_{i-1}$ to get $W_i$. In effect, $W'_{i-1}$ is the list containing the overlap between $W_{i-1}$ and $W_i$. Generally we try to choose the overlap $W'_{i-1}$ as large as possible to minimize the number of computations needed to obtain a new output.

This approach has some drawbacks. First of all it only applies to associative window computations where $(\mathbb{D}, \circ)$ is a group instead of a semi-group, or can be extended to become a group. Secondly, it requires the computation of the inverse, which may be a costly operation for some $\mathbb{D}$. Thirdly, the number of operations per output depends on the sizes of $A_i$ and $B_i$, which can be considerable for associative window computations with $D > 1$ and/or $p_W > 1$.

If we accept these drawbacks, however, we can write an algebraic state update equation for such a system, so it can benefit from a look-ahead computation:

\[
\begin{align*}
    s(0) &= 1_\circ \\
    s(i+1) &= s(i) \oplus b(i) \\
    b(i) &= \left( \frac{[\text{rev}(A_i)]^{-1}}{[B_i]} \right) \\
    \left( \begin{array}{c} a \\ b \end{array} \right) \oplus \left( \begin{array}{c} c \\ d \end{array} \right) &= \left( \begin{array}{c} c \circ a \\ b \circ d \end{array} \right) \\
    \text{out}(i) &= s(i+1)(0) \circ s(i+1)(1)
\end{align*}
\] (4.25)

(4.26)

(4.27)

(4.28)

(4.29)

These equations can be used to implement the associative window computation using a look-ahead strategy.

In cases where $(\mathbb{D}, \circ)$ is actually a commutative group, as it is for the moving sum, we obtain the following, considerably simpler, algebraic state update equation:

\[
\begin{align*}
    s(0) &= 1_\circ \\
    s(i+1) &= s(i) \circ b(i) \\
    b(i) &= \left( \frac{[A_i]^{-1}}{[B_i]} \right) \\
    \text{out}(i) &= s(i+1)
\end{align*}
\] (4.30)

(4.31)

(4.32)

(4.33)

For implementations based on the algebraic state update equations (4.26) or (4.31) the number of operations per output depends on the sizes of $A_i$ and $B_i$. For some window computations this number may exceed $W$, i.e., when there is no overlap between the windows of successive outputs. In such cases the associative window computation should be implemented without rewriting
it to the algebraic state update form, i.e., it should be implemented using the finite memory span strategy discussed in Section 3.3.

In cases where \( A_i \) and \( B_i \) are relatively small compared to \( W \) the look-ahead computation is an option. The drawback of a look-ahead computation is that even the types of look-ahead that scale linearly introduce some factor of overhead. This factor of overhead must be taken into account when a look-ahead based algorithm is compared to a finite memory span based algorithm. When the window size is smaller than the factor of overhead introduced by the look-ahead computation, it is more efficient, in terms of operations per output, to implement the system using the finite memory span strategy.

In summary, when \( (\mathbb{D}, \circ) \) is a group we can obtain an algebraic state update equation for an associative window computation. It is only effective, however, when it is relatively easy to determine an inverse, successive outputs share a considerable number of inputs and the window size \( W \) is large enough to warrant the overhead introduced by a look-ahead computation.

4.5 From look-ahead to window computation

Fettweis [23] described look-ahead computations using the algebraic state update equation (2.1). The exact representation of a state update function, however, is a secondary concern. The look-ahead computation is primarily specified as the function composition of several state update functions, regardless of the representation of these functions. Since function composition is an associative operator, we demonstrate in this section that it is possible to rewrite a look-ahead computation to an associative window computation.

Consider the recursive system described by:

\[
s(i + 1) = f_i(s(i)) \tag{4.34}
\]

where \( f_i : S \to S \) is the state update function that maps the state \( i \) onto the next state \( i+1 \). We make no demands on this function, which means that (4.34) can be used to describe any recursive system. We obtain, for example, the state transition function of a finite state machine with \( f_i(j) = \delta(j, \text{in}(i)) \).

The result of the look-ahead computation for this system is defined as:

\[
s(i + L) = (\circ j : 0 \leq j < L : f_{i+L-1-j})(s(i)) \tag{4.35}
\]

where \( \circ \) stands for function composition, i.e., \( (f \circ g)(x) = f(g(x)) \). Since function composition is associative, we can use an associative window computation to perform a look-ahead computation.
In the next two sections we discuss how Equation (4.35) can be turned into a associative window computation. In the section after that we briefly consider the problem of computing function compositions and discuss function representations.

4.5.1 Basic look-ahead computation

A look-ahead computation advances the state of the system by \( L \) steps. Using Equation (4.35) we distinguish two phases in the look-ahead computation. First there is the computation of the function composition, and then there is the application of this composite function to the old state to obtain a new state. So, to obtain a block processing algorithm for the system we can look at these two stages.

First we consider the latter stage of the algorithm, i.e., applying the function composition to an old state to obtain a new state. The block processing algorithm should produce a block of \( L \) new states based either on one preceding state (in case of linear and prefix look-ahead), or on a block of preceding states (the case of logarithmic look-ahead). In this section we discuss the latter case: the block processing algorithm produces a block of new states using a block of preceding states.

A new block of states is then computed by applying a block of composite functions to the block of preceding states. So, the algorithm only needs to compute the first \( L \) states in the conventional manner, after which it can use block processing to produce blocks of new states. Assuming, of course, that the algorithm is also capable of computing the composite function in the first stage of the algorithm.

So now we consider the first stage, i.e., computing the function compositions. This stage of the block processing algorithm produces blocks of function compositions. Since the function compositions are actually associative window computations, we just need a block processing algorithm for that computation:

**Theorem 4.5.1** (Look-ahead using an associative window computation). A look-ahead computation for a block size \( L \) can be performed by first computing the following associative window computation:

- The number of dimensions is \( D = 1 \)
- The windows are given by \( W = \text{tiles}((L),1,0) \)
- The function \( \text{in} \) returns the state transition functions \( f \) such that \( \text{in}(i) = f_i \).
• The semi-group \((\mathbb{D}, \odot)\) is such that \(\mathbb{D}\) is the set of all the state transition functions \(f\), and \(\odot\) is defined as \(a \odot b = b \circ a\).

Each output of this window computation is a composite function that is then used to advance a particular state by \(L\) steps, thereby producing the result of the look-ahead computation. □

Note that the window size \(W\) in Theorem 4.5.1 equals \(L\), which may be a problem for the scalability of the resulting algorithm. After all, a larger block size implies a larger window, which seems to imply an algorithm with a higher complexity. If the computation is implemented as an \(FSM_\ast\), i.e. based on the finite memory span strategy from Section 3.3 then the complexity is \(\Theta(W)\) aggregation operations per output, or \(\Theta(WL) = \Theta(L^2)\) operations per block. The finite memory span strategy therefore would result in an algorithm with super-linear scalability, where the costs grow quadratically with the throughput.

As we have announced in Section 3.4 however, we can do better. Section 4.4 shows that, under certain conditions, it is possible to obtain an algorithm with a throughput that is independent of the window size. In Chapter 5 we show that it is possible to obtain an algorithm with linear scalability and a throughput that is independent of the window size for any associative window computation adhering to Definition 4.3.1. Therefore, by combining the methods of Chapter 5 with Theorem 4.5.1 we can obtain a block processing algorithm for look-ahead computations with linear scalability for any \(FSM_\infty\) system.

4.5.2 Parallel prefix based look-ahead

As mentioned in the previous section, it is possible to design the look-ahead computation such that a new block of \(L\) states is calculated from a single old state. Two of the existing techniques for look-ahead computation use this approach, namely the linear look-ahead and the parallel prefix technique from Section 2.4 and 2.6 respectively. We can accomplish a similar result with an associative window computation.

Although both approaches, using a single old state and using a block of old states, result in the same states being computed, we present the alternative here for completeness. Using a single state has a minor advantage, namely that the algorithm requires less memory (it needs to store only one state instead of a block of states). The disadvantage of this approach that it complicates the algorithm. It also serves as an illustrative example of how a block prefix computation can be turned into a window computation, however, which is why we discuss it here.
So, we wish to compute $s(i + j)$ for $1 \leq j \leq L$ using $s(i)$ and $s(i + 1)$. The function compositions we need for this computation form a typical prefix computation, i.e., given a list of data $[f_i, f_{i+1}, f_{i+2}, \ldots, f_{i+L-1}]$ we need to compute the list $[f_i, f_i \circ f_{i+1}, f_i \circ f_{i+1} \circ f_{i+2}, \ldots, f_i \circ \ldots \circ f_{i+L-1}]$ using the associative operator $\circ$ as specified in Theorem 4.5.1.

This can be accomplished by extending $\mathbb{D}$ with two elements. First of all we need an identity element $1_\circ$ as in (4.19) and secondly we need a reset element $r$ such that:

$$ (\forall a : a \in \mathbb{D} : a \circ r = 1_\circ) \quad (4.36) $$

We use these elements to define the following associative window computation:

**Theorem 4.5.2** (Prefix based look-ahead using an associative window computation). A look-ahead computation using a prefix approach for a block size $L$ can be performed by first computing the following window computation:

- The number of dimensions is $D = 1$
- The windows are given by $W = \text{tiles}((L + 1), 1, 0)$
- The function $\text{in}$ returns the state transition functions $f$ interleaved with reset elements such that $\text{in}((L + 1)i + j) = f_i$ and $\text{in}((L + 1)i + L) = r$ for all $0 \leq j < L$.
- The semi-group $(\mathbb{D}, \circ)$ is such that $\mathbb{D}$ is the set of all the state transition functions $f$ and $1_\circ$ and $r$, and $\circ$ is defined as $a \circ b = b \circ a$ and (4.36).

The appropriate outputs of this window computation are composite functions that are then used to advance a particular state by $L$ steps, thereby producing the result of the look-ahead computation. □

The outputs of this window computation form the list $[r, f_0, f_0 \circ f_1, f_0 \circ f_1 \circ f_2, \ldots, f_0 \circ \ldots \circ f_{L-1}, 1_\circ, f_L, \ldots, f_L \circ \ldots \circ f_{2L-1}, 1_\circ, \ldots]$. By removing the initial $r$ element and all $1_\circ$ elements at position $i(L + 1)$ for all $i \in \mathbb{Z}^D$ the remaining list contains all the results needed to perform the look-ahead computation. When the remaining list is divided into sub-lists of length $L$ the resulting sub-lists form the blocks containing the results of the look-ahead computation.

Because the occurrences of the reset elements $r$ are known in advance (Theorem 4.5.2) it is possible to omit them. After all, the reset element is a constant with the added bonus that expressions containing that constant can be
(partially) evaluated at design time. So, using standard constant propagation techniques the algorithm can be simplified to the point that it no longer contains the reset element.

Despite these simplifications it is generally better to use the approach from the previous section, since the overhead introduced by this approach, i.e., identity element, reset element, a difference between block size and window size, is not worth the small saving in memory usage.

4.5.3 Function composition

The techniques described in the previous two sections both depend on the availability of a function composition operation. Function composition is generally not a simple operation and SIMD processors do not support them as basic operations. As long as a sequential algorithm is available to compute the composition of two functions, however, we can obtain an SIMD variant by replacing all the operations in that algorithm by operations on blocks. In this section we examine a few ways in which a function can be represented, and how a function composition can be computed.

Note the complexity of our algorithms is expressed in terms of the number of \( \odot \) operations, i.e., the number of function compositions. Therefore, the complexity of the \( \odot \) operations themselves indicate an overhead factor with which we should multiply the complexity of the final algorithm. This overhead factor does not influence the scalability of the algorithm, since it is independent of \( L \), but in some cases it is large enough to take into account. So, in this section we examine the complexity of the \( \odot \) operations themselves.

First of all we note that in practical situations the state transition functions that are composed by the \( \odot \) operation have a finite domain and co-domain. After all, these functions represent a part of the state update function of a system with a finite memory.

So, each function can be represented by a look-up table. Function composition of such functions is then accomplished by combining two look-up tables into a new look-up table. When the domain of the function has size \( N \), the look-up tables contain \( N \) entries. Combining two look-up tables is then a \( \Theta(N) \) operation, assuming that a look-up requires \( \Theta(1) \) time.

Alternatively a function can be represented by a \( N \times N \) matrix with elements from the set \( \{0, 1\} \). The value of 1 in a row \( i \) and a column \( j \) would then indicate that the function maps the element from the function’s domain represented by column \( j \) onto the element of the function’s co-domain represented by row \( i \). Note that this means that each column of such a matrix contains exactly one
element that is 1. Function composition can now be represented by multiplying two matrices representing the functions using the semi-ring \((\{0,1\}, \lor, \land)\), i.e., using the logical OR and logical AND instead of standard arithmetic addition and multiplication. Although matrix multiplication has a higher complexity \(\Theta(N^3)\) for a straightforward implementation) than table look-up, the use of the semi-ring \((\{0,1\}, \lor, \land)\) may result in a lower overall complexity, particularly for VLSI implementations.

Which of these two approaches results in the lowest complexity depends on the size of the state \(N\), and the hardware platform on which the function composition is implemented. Some SIMD processors, for example, do not support individual table look-up operations for each of their processing elements.

The drawback of both approaches is that their complexity depends on the size of the state \((N)\), which can be quite considerable for practical systems. So, although these approaches are capable of representing and composing all possible state transition functions, there may be more efficient ways of doing this.

State transition functions that can be represented by in the algebraic state update form (2.1) are very interesting in this sense:

\[
f_i(s(i)) = a(i) \otimes s(i) \oplus b(i)
\] (4.37)

Such a function can be represented by the single tuple \((a(i), b(i))\) and function composition can be computed using only three operations, as defined in (2.24):

\[
(a, b) \circ (c, d) = (a \otimes c, a \otimes d \oplus b)
\] (4.38)

Of course, using this representation means that the complexity of a function composition now depends on the complexity of the \(\otimes\) and \(\oplus\) operators. In many cases, however, the cost of these operators is relatively low. This approach can be readily used, for example, for linear systems and Viterbi decoders as suggested by Fettweis et al. in [23].

### 4.6 Conclusion

This chapter introduced definitions and notations which are used in subsequent chapters to design and implement algorithms that perform window computations. First of all we introduced tiles and tile-sets, which allow us to denote the results and intermediate results of a window computation. Secondly, we used this notation to expand our definition of associative window computations to
include more than the computations performed by $FSM_\circ$ systems, since we also include window computations on multi-dimensional data. Furthermore, we have explained the relation between the multi-dimensional grid on which the data is organized and the one-dimensional streams of our system.

A notation using tiles and tile-sets, similar to the one in this chapter, was presented by us in [40]. In contrast to the notation introduced in this chapter, the notation in [40] represents tiles using sets instead of lists and it uses the union operator instead of the concatenation operator from (4.3). We use a different notation here because the $\circ$-operation examined in [40] is commutative, which means that the order in which inputs are aggregated and hence the order of the indices of the input does not matter. In this thesis we make no assumptions on the commutativity of the $\circ$ operator in general. Specific instances of the operator may be commutative, like the sliced merge from Chapter 6, and in such cases we will exploit that property. In general, however, our algorithm design methods for associative window computations do not rely on the commutativity of the $\circ$ operator.

In this chapter we have also presented a method to write an algebraic state update equation for a system that performs an associative window computation. This method, however, only works if the semi-group involved in the computation is actually a group and generally it only leads to better algorithms for specific associative window computations.

Most importantly, we have shown that a look-ahead computation can be implemented using an associative window computation. For such computations the window size depends on $L$. This means that if the complexity of the window computation depends on the size of the window, the resulting look-ahead computation would not scale well. Fortunately it is possible to design an algorithm such that the complexity of the algorithm is independent of the window size, as discussed in the next chapter.
Chapter 5

Low-cost associative window computations

5.1 Introduction

In this chapter we consider the problem of designing efficient and scalable algorithms for associative window computations.

There is a straightforward design method that obtains algorithms with linear scalability for $FSM_*$ systems. This design method, called the finite memory span strategy in Section 3.3, also applies to associative window computation systems, since they are a subclass of those systems, i.e., $FSM_\circ \subset FSM_*$. The finite memory span strategy relies on the property that the window computations are independent. A block processing algorithm is obtained by first designing an algorithm for a single window computation and then executing this algorithm in parallel $L$ times. Block processing algorithms designed according to the finite memory span strategy have a throughput that is proportional to $L$ and require an amount of hardware that is also proportional to $L$, which means that they scale linearly.

By applying the finite memory span strategy to associative window computations we obtain algorithms with a complexity of $\Theta(WL)$ operations per block. The algorithms scale linearly, but there is a overhead factor of $W$, which may be considerable in some cases. The dependence between window size and block size is especially a problem for cases in which a look-ahead computation is performed by using a window computation, as explained in Section 4.4. In that
case the block size $L$ and the window size $W$ are coupled by $L \in O(WI)$, and the linear scalability is destroyed.

Therefore we introduce design methods for window computation algorithms in this chapter that result in algorithms with a complexity per output that depends neither on $L$ nor on $W$. This effectively results in an algorithm that has linear scalability no matter whether the block size $L$ and window size $W$ are coupled or not.

Furthermore, the algorithms designed in this way are such that the overhead for $FSM_{\odot}$ systems is a small constant, i.e., less than 4. This means that the number of $\odot$ operations required per output is less than 4, regardless of the window size $W$ or the block size $L$. This small overhead makes these algorithms an interesting alternative for existing algorithms.

## 5.2 Calculation graph

In this section we introduce the notion of a calculation graph to capture the computation specified by a window computation.

The calculation graph is basically a dependence graph \cite{65} with extra annotation that is specific for associative window computations.

**Definition 5.2.1 (Calculation graph).** A calculation graph $G = (V, E)$ is a directed, acyclic graph, with a set of nodes nodes $V \subseteq T$ and a list of edges given by the list $E \in \mathbb{L}(V \times V)$. Furthermore, a calculation graph adheres to the restrictions described by (5.2), (5.3) and (5.5).

The set of nodes of a calculation graph $V$ contains tiles representing all the (intermediate) results produced during a computation. A node $T \in V$ represents the result obtained by aggregating the inputs whose indices are in the tile, i.e. it represents $(T)$ as defined in (4.2). The edges of the graph $E$ describe the dependencies between the nodes. An edge from $T_1$ to $T$ denotes that $(T_1)$ is used to compute $(T)$ using a $\odot$-operation. In this manner a calculation graph describes an associative window computation.

An example calculation graph is depicted in Figure 5.1. This calculation graph describes the computation proposed in Section 3.4, where the windows, i.e. outputs, (out) are the aggregation of 3 inputs, and each window is computed using intermediate results (in) that are the aggregation of two inputs (in).

There are two reasons that we represent the edges of the calculation graph as a list. Firstly, the calculation is a multi-graph; the result associated with a tile $T = [0, 0]$, for example, is constructed from $T_1 = [0]$ like so $(T) = (T_1) \odot (T_1)$. 
In this case there are two edges in the calculation graph from node $T_1$ to node $T$. By representing $E$ as a list, instead of a set, we are able to represent these edges without introducing multi-sets. Secondly, we need an ordering on the edges of the graph so that the order of the operands for the non-commutative $\odot$ operation is clear. We have in Figure 5.1 for example, the node $T = [0, 1, 2]$ and this node has two incoming edges, one from $T_0 = [0]$ and one from $T_1 = [1, 2]$. We obtain $\langle T \rangle$ by evaluating $\langle T_0 \rangle \odot \langle T_1 \rangle$ and not by evaluating $\langle T_1 \rangle \odot \langle T_0 \rangle$. By representing $E$ as a list we are able to represent the order of the operands by the order in which the edges appear in the list.

Not every graph $G$ with $V \subseteq T$ and $E \in L(V \times V)$ is a calculation graph, there are two restrictions. First of all, we restrict the indegree of any node to at most two. By limiting the calculation graph to an indegree of 2, we enforce the specification of the evaluation order of the $\odot$-operations, and thus remove any ambiguity caused by the associativity of the $\odot$ operator. Formally there is the function $\text{din}$ that returns the indegree of node $T$:

$$\text{din} : V \rightarrow \mathbb{N}$$

$$\text{din}(T) = \# \{ T_0 | (T_0, T) \in E \}$$ (5.1)

and we have the restrictions that:

$$(\forall T : T \in V \wedge \#T > 1 : \text{din}(T) = 2)$$ (5.2)

$$(\forall T : T \in V \wedge \#T = 1 : \text{din}(T) = 0)$$ (5.3)
CHAPTER 5. LOW-COST ASSOCIATIVE WINDOW COMP.

These equations enforce that each tile with a size larger than one is constructed using exactly one $\circ$-operation. Furthermore, they specify that only tiles of size larger than one have a construction method and tiles of size one, i.e. the inputs of the system, have no construction method.

The second restriction on the calculation graph is that it must be possible to actually compute the result represented by a node $T$ from the results represented by the nodes that have edges to $T$. We do not allow, for example, an edge from $T_0 = [2]$ to $T = [0, 1]$, since $\langle T \rangle$ cannot be computed from $\langle T_0 \rangle$. To this end we introduce the partial functions $\text{pred}0(T)$ and $\text{pred}1(T)$ that return respectively the first and the second node that have an edge leading to $T$, i.e., these partial functions return the predecessors of a node. Formally:

$$\text{pred}0 : V \rightarrow V \text{ and } \text{pred}1 : V \rightarrow V \text{ such that }$$
$$\left( \exists i, j : (\text{pred}0(T), T) = E(i) \land (\text{pred}1(T), T) = E(j) : i < j \right)$$  \hspace{1cm} (5.4)

Note that the order is determined by the indices $i$ and $j$ in the list of edges. The second restriction is formalized using these functions as:

$$\forall T : T \in V \land \#T > 1 : \langle T \rangle = \langle \text{pred}0(T) \rangle \circ \langle \text{pred}1(T) \rangle$$  \hspace{1cm} (5.5)

So the result associated with any tile ($T$) of size larger than 1 can be computed by applying the $\circ$ operation to the results associated with the tiles $\text{pred}0(T)$ and $\text{pred}1(T)$.

Note that (5.5) implies that (4.3) must also hold and that every tile is therefore concatenation of its constructing tiles. Since the concatenation of tiles results in larger tiles, (5.5) also implies that the graph is acyclic: edges only connect tiles to larger tiles.

The calculation graph, like a dependence graph, describes the dependencies between the $\circ$-computations. It can therefore be used, like a dependence graph, to derive various implementations. To implement a window computation with the set of windows $W$, the graph must describe the entire window computation. So the set of nodes of the calculation graph must contain all the windows:

$$W \subseteq V$$  \hspace{1cm} (5.6)

Note that this is not a restriction on calculation graphs, but rather a property that is satisfied by graphs that represent an entire window computation.

The introduction of the calculation graph allows us to compute the costs, i.e. complexity, of the computation. Each node, with a size larger than one, of the calculation graph represents a result obtained by the application of one
⊙-operation. Since the calculation graph is infinite, we amortize the cost over the number of outputs, or windows. This results in the cost, or complexity, per output. For the calculation graph in Figure 5.1 we have one operation to construct each output (out) from an input (in) and an intermediate result (f), and each intermediate result (f) requires one operation to construct, but we only need one intermediate result per 2 outputs, resulting in a total of 1.5 operations per output.

To formalize the cost we require the hypercube function:

\[
\text{hypercube}(n, D) = \{ g \mid g \in \mathbb{Z}^D \land -n1_D \leq g \leq n1_D \}
\]  

(5.7)

where \(1_D\) represents the unit vector, i.e., a vector of length \(D\) which contains only ones. The function hypercube therefore produces a set containing \((2n+1)^D\) grid locations centered around the origin, i.e., a hypercube in \(D\) dimensions.

We express the complexity, in terms of operations per output, of the computation represented by a calculation graph as \(C(G)\):

\[
C(G) = \lim_{k \to \infty} \frac{\#(\{T \mid T \in V \land \#T > 1\} \cap L(\text{hypercube}(k, D)))}{\#(W \cap L(\text{hypercube}(k, D)))}
\]  

(5.8)

Note that \(W, V\) and \(E\) are infinite, therefore we take the limit using \(k\) as the size of the hypercube containing all the grid locations used to form the tiles that we consider. So \(C(G)\) effectively gives us the cost in ⊙-operations per output, or “opo” for short. To get the complexity \(C\) in terms of number of operations per block we multiply \(C(G)\) with \(L\).

The problem of designing an algorithm for a window computation is now: given the specification of a window computation find a calculation graph \(G\) that satisfies (5.6) and that adheres to the restrictions (5.2), (5.3) and (5.5). Furthermore, the cost \(C(G)\) should be as small as possible.

### 5.3 Generator graph

A calculation graph is an infinite graph that represents the window computation on an infinite stream. The system performing the window computation, however, consists of an algorithm implemented either in hardware or software, and it is a finite object. The gap between infinite calculation graphs and finite systems is bridged by so-called generator graphs, which we discuss in this section.

A generator graph is a finite representation of a system that performs a window computation. This graph is used to exploit the regularities present in
the window computation. On the one hand the generator graph, as its name suggests, can be used to generate a calculation graph. On the other hand, a generator graph can serve as the starting point for a concrete filter implementation.

In this section generator graphs are defined and it is shown how calculation graphs can be generated from them. We also explain the composition techniques used in Sections 5.4 and 5.5 to construct generator graphs for specific window computations. Deriving block processing algorithms from generator graphs is the subject of Chapter 7.

5.3.1 Preliminaries

In this section we discuss the basic properties of generator graphs.

A calculation graph contains infinitely many tiles, but the windows all have the same, finite, shape. There are only a finite number of ways in which a tile with a certain finite shape can be constructed. Therefore the entire calculation graph contains a finite number of construction methods. This suggests that one can condense the calculation graph such that only the different construction methods are retained.

Consider all tiles of a specific shape in a calculation graph, for example the graph from Figure 5.1. We group these tiles according to their construction method. Groups or sets of tiles can be represented using the function tiles, using a triple consisting of a shape, a phase and a period. The nodes of size two in Figure 5.1, for example, can be represented using the triple \((\langle 2 \rangle, 1, 2)\). Since the number of construction methods is finite, the calculation graph must consist of repeating patterns and we need a finite number of triples. Specifically, for Figure 5.1 we can represent all tiles using the triples \((\langle 3 \rangle, 0, 1)\), \((\langle 2 \rangle, 1, 2)\), and \((\langle 1 \rangle, 0, 1)\).

These triples, or tile-sets, form the nodes of our generator graphs. We use these triples in their canonical form, see Section 4.2.2, as the nodes of our generator graph \(G_g = (V_g, E_g)\). Each of these nodes represents a tile-set whose tiles are constructed in the same manner. So, the node set of a generator graph \(V_g\) is a subset of \((\mathbb{T} \times \mathbb{Z}^D)\), with each node representing a tile-set.

There is a surjective function \(\text{map}\) that maps the tiles of the calculation graph onto the tile-sets of the generator graph such that each tile is mapped to the tile-set that contains it:

\[
\text{map} : V \rightarrow V_g \\
(T \in \mathbb{T}) \equiv (\text{map}(T) = T)
\]
Note that this function implies that the tile-sets of the generator graph are disjoint, i.e., there are no tiles that are mapped onto multiple tile-sets. This is what we expect, since the tile-sets are used to group tiles that have the same construction method and a tile can have only one construction method. Furthermore, the tile-sets in $V_g$ form a partition of the tiles in the set $V$.

The construction methods of the tiles from a particular tile-set are represented by the edges of the generator graph. This means that the edges of the generator graph mimic those of the calculation graph. The edges of the generator graph are represented by a list $E_g \in \mathbb{L}(V_g \times V_g)$, and we have the following relation between the edges of the calculation graph and the generator graph:

$$((T_0, T) \in E) \implies ((\text{map}(T_0), \text{map}(T)) \in E_g) \quad (5.10)$$

The function $\text{map}$ therefore specifies a homomorphism from the calculation graph to the generator graph.

Equation (5.10) specifies that the construction methods from the calculation graph are represented in the construction methods from the generator graph. We do have the additional requirement, however, that tiles that are grouped in the same tile-set have the same construction method. Therefore we have the additional restriction that each tile-set, containing tiles of with sizes larger than one, has only two incoming edges. Essentially we lift (5.2) and (5.3) to generator graphs:

$$\forall T : T \in V_g \land \#S_T > 1 : \text{din}_g(T) = 2) \quad (5.11)$$

$$\forall T : T \in V_g \land \#S_T = 1 : \text{din}_g(T) = 0) \quad (5.12)$$

Where function $\text{din}_g$ returns the indegree of a node in the generator graph, i.e., function $\text{din}$ (5.1) for calculation graphs lifted to generator graphs:

$$\text{din}_g(T) = \#\{T_0 \mid (T_0, T) \in E_g\} \quad (5.13)$$

Equation (5.10) enforces that the construction methods of all tiles in the calculation graph are represented in the construction methods of the tile-sets in the generator graph. Equation (5.11) enforces that each tile-set, with tiles larger than one, in the generator graph has only one construction method, therefore tiles from the same tile-set share the same construction method.

In summary we have:

**Definition 5.3.1 (Generator graph).** A generator graph is a finite representation a window computation. The nodes of a generator graph $G_g = (V_g, E_g)$ are
Figure 5.2: Example generator graphs for (a) $W = (\langle 3 \rangle, 0, 1)$, corresponding to Figure 5.1, and for (b) $W = (\langle 6 \rangle, 0, 1)$

given by set $V_g \subseteq (T \times \mathbb{Z}^D \times \mathbb{Z}^D)$ and they represent tile-sets whose tiles are constructed in a similar manner. The construction methods of the tiles from a certain tile-set are described by the edges of the generator graph, given by the list $E_g \in \mathbb{L}(V_g \times V_g)$. A generator graph is derived from a calculation graph using a partition of the nodes of the calculation graph represented by the function $\text{map}$, such that it adheres to (5.9), (5.10), (5.11) and (5.12).

Figure 5.2(a) shows one possible generator graph that corresponds to the calculation graph from Figure 5.1. Other generator graphs are obtained by choosing a different partitioning of $V$, and therefore a different function $\text{map}$, to get a different set of nodes for the generator graph. There exist, therefore, multiple generator graphs that represent the same calculation graph. The gen-
erator graph in Figure 5.3(a) for example, corresponds to the same calculation graph as Figure 5.2(a).

Figure 5.2(b) shows another example of a generator graph. We have not drawn the associated calculation graph, however, since it would become cluttered with edges. Figure 5.2(b) shows that a tile $T$ from the tile-set $(\langle 4 \rangle, 1, 2)$ is formed by taking two tiles $T_1$ and $T_2$ from the same tile-set $(\langle 2 \rangle, 1, 2)$. Note that the generator graph is therefore a multi-graph, just like the calculation graph, since nodes represent tile-sets and two tiles from the same tile-set might be required to construct a tile. Furthermore, the generator graph is acyclic like the calculation graph, since tiles can only be formed from smaller tiles.

### 5.3.2 Canonical generator graphs

In this section we briefly discuss the relation between calculation graphs and generator graphs and introduce the canonical form for generator graphs.

There are multiple generator graphs that correspond to the same calculation graph. Generator graphs are intended to use as a basis for our algorithms and implementations. So, it is not surprising that a calculation graph, which represents a computation, can be described by different generator graphs, each corresponding to a different algorithm.

Conversely, however, a generator graph must correspond to only one calculation graph. This ensures that all algorithms derived from the same generator graph correspond to the same computation. Equation (5.9) allows us to reconstruct the nodes of a calculation graph for any given generator graph. The edges of the generator graph specify the construction methods of tile-sets, but mapping these on construction methods for tiles is not trivial. Equation (5.10) specifies that for every tile from a tile-set with an incoming edge in the generator graph there is a corresponding edge in the calculation graph. The equation does not describe, however, the source of the edge in the calculation graph, only that the source tile comes from a certain tile-set.

Consider a tile-set $T \in V_g$, with incoming edges from $T_1$ and $T_2$. The calculation graph contains a tile $T \in T$, and we want to know which tiles $T_1 \in T_1$ and $T_2 \in T_2$ are used to construct $T$. There are at most two combinations of tiles $T_1$ and $T_2$ that can form the tile $T$. There are only two combinations because the first grid location of the tile, $g = T(0)$, is found in one of the two tiles, i.e $T_1(0) = g$ or $T_2(0) = g$. There is at most one tile starting with $g$ in each tile-set, because these tile-sets are generated using the tiles function (4.2). So, we have $T = T_1 + T_2$, $T = T_2 + T_1$, or, possibly, both. In the latter case we can derive two different calculation graphs from the generator graph.
A generator graph therefore contains enough information to derive a calculation graph, but there may be multiple possible calculation graphs. To remove this ambiguity we need to make the choice between two options during the derivation of the calculation graph in a consistent manner. So, when both \( T = T_1 + T_2 \) and \( T = T_2 + T_1 \) are possible we choose the option where the tile with the lowest rank is used as the first operand\(^1\). This allows us to derive exactly one calculation graph from each generator graph, leading to the following equivalence relation on generator graphs:

**Definition 5.3.2 (Equivalence of graphs).** Two generator graphs \( G_{g1} \) and \( G_{g2} \) are considered equivalent when the calculation graphs derived from them, i.e. \( G_1 \) and \( G_2 \), are equivalent. These two calculation graphs \( G_1 = (V_1, E_1) \) and \( G_2 = (V_2, E_2) \) are equivalent when:

- They have the same set of nodes: \( V_1 = V_2 \).
- Each node has the same construction method, i.e. each node has the same predecessors and these predecessors have the same order: \( \text{pred0}_1 = \text{pred0}_2 \land \text{pred1}_1 = \text{pred1}_2 \). Where the predecessor functions with subscript \( 1 \) are associated with \( G_1 \) and the predecessor functions with subscript \( 2 \) are associated with \( G_2 \).

The derivation of a calculation graph from a generator graph becomes easier when the order of the operands is encoded in the ordering of the edges of the generator graph, just like we did for the calculation graph. This allows us to derive the ordering of the edges of the calculation graph directly, without having to check for, and decide between, multiple possibilities. Formally, the ordering on the edges of the generator graph allows us to introduce the functions \( \text{pred0}_g \) and \( \text{pred1}_g \), essentially the functions \( \text{pred0} \) and \( \text{pred1} \) (5.4) for calculation graphs lifted to generator graphs:

\[
\text{pred0}_g : V_g \rightarrow V_g \quad \text{and} \quad \text{pred1}_g : V_g \rightarrow V_g \quad \text{such that} \\
(\exists i, j : (\text{pred0}_g(T), T) = E_g(i) \land (\text{pred1}_g(T), T) = E_g(j) : i < j) \tag{5.14}
\]

So, when we derive a calculation graph from a generator graph we would like to do it in such a manner that the ordering of the edges in the calculation graph and the generator graph correspond with each other, i.e.:

\[
\text{pred0}(T) \in \text{pred0}_g(\text{map}(T)) \\
\text{pred1}(T) \in \text{pred1}_g(\text{map}(T)) \tag{5.15}
\]

\(^1\)Section 4.2.1 describes the ordering relation on tiles.
It is possible that the calculation graph derived from a generator graph does not adhere to property (5.15), however. The example in Figure 5.2(b) shows such a case. Consider the tile-set ($\langle 5 \rangle, 0, 1$): the tiles starting with an even coordinate, like $T = [0, 1, 2, 3, 4]$ from ($\langle 5 \rangle, 0, 1$), must be constructed using $T = T_0 + T_1$ with $T_0 = [0] \in (\langle 1 \rangle, 0, 1)$ and $T_1 = [1, 2, 3, 4] \in (\langle 4 \rangle, 1, 2)$. The tiles starting with an odd coordinate, like $T = [1, 2, 3, 4, 5]$ from ($\langle 5 \rangle, 0, 1$), must be constructed using $T = T_0 + T_1$ with $T_0 = [1, 2, 3, 4] \in (\langle 4 \rangle, 1, 2)$ and $T_1 = [5] \in (\langle 1 \rangle, 0, 1)$. So, the ordering $\text{pred}_g((\langle 5 \rangle, 0, 1)) = (\langle 1 \rangle, 0, 1)$ would satisfy (5.15) for the tiles from ($\langle 5 \rangle, 0, 1$) that start with an even coordinate, but not for the tiles that start with an odd coordinate. Conversely, the ordering $\text{pred}_g((\langle 5 \rangle, 0, 1)) = (\langle 4 \rangle, 1, 2)$ would satisfy (5.15) for the tiles from ($\langle 5 \rangle, 0, 1$) that start with an odd coordinate, but not for the tiles that start with an even coordinate.

Since different generator graphs can be used to derive the same calculation graph, however, we can always find a generator graph that generates the same calculation graph and also satisfies the property (5.15). From the generator graphs depicted in Figure 5.3, for example, we can derive the same calculation graph as from the ones in Figure 5.2, but unlike Figure 5.2 these graphs adhere to (5.15). We call the the generator graphs of Figure 5.3 the canonical forms of the generator graphs from 5.2.

**Definition 5.3.3 (Canonical generator graph).** The canonical form of a generator graph is an equivalent generator graph, but the canonical generator graph has the following properties:

- all the nodes have the same period, which we call the meta-period $m$.
- this meta-period $m$ is as small as possible.

The canonical form of a generator graph adheres to (5.15), since the period of the tile-sets used to construct a new tile-set and the period of the new tile-set itself are the same. Furthermore, the canonical form makes it easier to compare generator graphs, to see if they generate the same calculation graph. The meta-period also plays an important role in actually implementing the computation, as discussed in Chapter 7.

To put a generator graph in the canonical form we introduce two transformations:

**Definition 5.3.4 (Generator graph reduction).** A generator graph $G_{g1}$ is reduced to a generator graph $G_{g2}$ by replacing a set of nodes $\{T_0, \ldots, T_{N-1}\}$ from $G_{g1}$ by a single node $T$ in $G_{g2}$, where the following conditions hold:
Figure 5.3: Canonical forms of the generator graphs from (a) Figure 5.2(a) and (b) Figure 5.2(b) both with a meta-period of $m = 2$ and with, respectively, 3 nodes with 2 incoming edges and 6 nodes with 2 incoming edges, resulting in, respectively, a cost of $3 \times 2 = 6$ and $6 \times 2 = 12$ operations per output.
5.3. GENERATOR GRAPH

1. \{T_0, \ldots, T_{N-1}\} partitions T

2. The tile-sets T_0, \ldots, T_{N-1} and T have the same construction method, if they need one: \((\#S_T > 1) \implies (\forall i : 0 \leq i < N : \text{pred}_{0,g_2}(T_i) = \text{pred}_{0,g_1}(T))\)

Definition 5.3.5 (Generator graph expansion). A generator graph \(G_{g_1}\) is expanded to a generator graph \(G_{g_2}\) by replacing a single node T from a generator graph \(G_{g_1}\) with a set of nodes \(\{T_0, \ldots, T_{N-1}\}\) in graph \(G_{g_2}\), where the following conditions hold:

1. \(\{T_0, \ldots, T_{N-1}\}\) partitions T

2. The tile-sets T_0, \ldots, T_{N-1} and T have the same construction method, if they need one: \(\text{pred}_{0,g_2}(T_i) = \text{pred}_{0,g_1}(T) \wedge \text{pred}_{1,g_2}(T_i) = \text{pred}_{1,g_1}(T)\)

Theorem 5.3.1. When a generator graph \(G_{g_2}\) is obtained by either reducing or expanding a generator graph \(G_{g_1}\), these two generator graphs are equivalent.

To obtain the canonical form of a generator graph we use these transformations. The generator graph is first reduced such that all the nodes have minimum periods. This is done by repeatedly replacing sets of nodes by a single node using the reduction transformation from Definition 5.3.4. This reduction process stops when there are no more sets of nodes that can be replaced. The generator graphs from Figure 5.2, for example, are the reduced versions of the graphs from Figure 5.3.

After a generator graph has been reduced the meta-period \(m\) is computed. This meta-period is the least common multiple of all the periods of the nodes. The canonical form of the generator graph is then obtained by using the expansion transformation from Definition 5.3.5. This is done by duplicating each node T such that a total of \(\Pi(\frac{m}{p_T})\) nodes are obtained, where \(\Pi(v)\) denotes the product of all the elements of the vector \(v\). These duplicates of the original node are created in such a manner that each one has a period of \(m\). The generator graphs from Figure 5.3, for example, are the canonical forms of the graphs from Figure 5.2.

The canonical form of the generator graph makes easier to compare two generator graphs and it is also used to compute the complexity of algorithms based on it.
CHAPTER 5. LOW-COST ASSOCIATIVE WINDOW COMP.

The cost of a calculation graph generated by a generator graph can be calculated without actually generating the calculation graph itself. Take Figure 5.3(b) for example; node \( (2, 1, 2) \) has incoming edges from \( (1, 0, 2) \) and \( (1, 1, 2) \), which means that it represents the combining of two tiles of size 1 and that this operation occurs with a period of 2. Because the period of these operations is 2 and the period of the windows is \( p_W = 1 \), we know that this operation can be amortized over 2 windows. Hence the contribution of this merging operation to the total is \( \frac{1}{2} \) operations per output (opo).

Each node of the generator graph with tiles larger than one represents one \( \odot \) operation with a period of \( m \). Therefore the cost in opo for a generator graph in canonical form is expressed by the equation:

\[
C(G_g) = \# \{ T \mid T \in V_g \land \# S_T > 1 \} \prod \left( \frac{p_W}{m} \right) \tag{5.16}
\]

where \( \prod(v) \) denotes the product of all the elements of vector \( v \). The generator graph in Figure 5.3(a) for example, contains 3 tile-sets with tiles larger than one. The period of the windows \( p_W = 1 \) and the meta-period is \( m = 2 \), so \( \prod(p_W/m) = \frac{1}{2} \). Therefore the cost of the generator graph is \( \frac{3}{2} = 1.5 \) operations per output (opo). Similarly, the generator graph in Figure 5.3(b) contains 6 tile-sets with tiles larger than one and has the same period for its windows and the same meta-period as Figure 5.3(a) resulting in a cost of \( \frac{6}{2} = 3 \) opo.

Note that, besides not having to actually generate the calculation graph to calculate the cost, the canonical generator graph has another advantage; we are able to avoid taking a limit as required by definition (5.8).

5.3.3 Generator graph composition

Section 5.3 describes, in detail, several heuristic algorithms for constructing generator graphs for a given window computation. These algorithms are all based on the same construction method, described in Section 5.4, which is parameterized with subroutines. This construction method is based on composing generator graphs, which we discuss in this section.

The algorithms for constructing a generator graph start with an empty generator graph \( (\emptyset, []) \) and add nodes and edges until a full generator graph is formed. It is not necessary that the generator graph is constructed in canonical form, since we can always turn it into a canonical form by the procedure described in Section 5.3.2.

To construct a generator graph we take a generator graph, for example the empty graph, and add nodes and edges to it. The nodes and edges added in a
single step are generator sub-graphs. These generator sub-graphs adhere to all the properties of Definition 5.3.1 except for, possibly, (5.11). They adhere to a weaker version of (5.11) instead, namely:

\[(\forall T : T \in V_g \land \#S_T > 1 : \text{din}_g(T) \in \{0, 2\})\]  \hspace{1cm} (5.17)

This equation specifies that the construction method of a node in a generator sub-graph is either described completely, i.e. by 2 edges, or not at all, i.e. the case with no edges. Note that this is similar to an induced sub-graph, but we also require that a generator sub-graph contains either the entire construction method of a tile-set, or no construction method at all. Most of generator sub-graphs used in the construction of a generator graph consisting of a single node and its construction method.

The function \text{merge} merges a generator graph \(G_{g1}\) with a generator sub-graph \(G_{g2}\) into a generator graph, when the following holds:

\[(\forall T : T \in V_{g2} \land \#S_T > 1 : (\text{din}_{g1}(T) = 2) \lor (\text{din}_{g2}(T) = 2))\]  \hspace{1cm} (5.18)

This means that the construction method of every node with tiles larger than one in the generator sub-graph \(G_{g2}\) is specified in at least one of the two graphs.

In some cases the generator graph reduction and expansion transformations can be used to modify \(G_{g1}\) and \(G_{g2}\) such that (5.18) holds. For example in cases where two tile-sets \(T_1, T_2 \in V_{g1}\) have different construction methods and form a partition of \(T \in V_{g2}\). Expansion is then used on the generator sub-graph\(^2\) \(G_{g2}\) such that \(T\) is partitioned into \(T_1\) and \(T_2\).

Once the precondition (5.18) is satisfied, the merge function merges a generator graph \(G_{g1}\) and a generator sub-graph \(G_{g2}\) into a generator graph in the following manner:

\[\text{merge}(G_{g1}, G_{g2}) = (V_{g1} \cup V_{g2}, E_{g1} + [(T_1, T) \mid (T_1, T) \in E_{g2} \land T \notin V_{g1}])\]  \hspace{1cm} (5.19)

This function merges the nodes of the two graphs, keeps the edges of graph \(G_{g1}\) and adds all edges from \(G_{g2}\) leading to nodes not in \(G_{g1}\). In other words, the construction methods, i.e. edges, from \(G_{g2}\) are copied only if there is no construction method to be copied from \(G_{g1}\). Together with the precondition (5.18) this ensures that the resulting generator graph adheres to the properties specified in Definition 5.3.1. An example of the application of the \text{merge} function is shown in Figure 5.4.

\(^2\)Theorem 5.3.1 also applies to a generator sub-graphs, because we can always embed a sub-graph in a generator graph, apply the transformation and extract the appropriate sub-graph from the result.
CHAPTER 5. LOW-COST ASSOCIATIVE WINDOW COMP.

(a) Generator graph $G_{g1}$
(b) Generator sub-graph $G_{g2}$
(c) $\text{merge}(G_{g1}, G_{g2})$

Figure 5.4: The $\text{merge}$ function applied to (a) and (b) results in (c)
Figure 5.5: Example application of the $\text{add}$ function, with $G_{g1}$ from Figure 5.4(a) and $G_{g2} = \{((5), 0, 2), (((1), 0, 2), (5), 0, 2)), (((4), 1, 2), (5), 0, 2))\}$. 
CHAPTER 5. LOW-COST ASSOCIATIVE WINDOW COMP.

Often the construction process of a generator graph requires the addition of a single node and its construction method. To add a single node and its incoming edges to a generator graph the function \texttt{add} is used, which returns the updated graph, as shown in Figure 5.5. Formally, \texttt{add} is defined as:

\[
\text{add}(G_g, (T, T_1, T_2)) = \text{merge}(\text{prepmerge}(G_g, ([\{T\}, ([T_1, T], (T_1, T)])))
\]

(5.20)

where the function \texttt{prepmerge} takes two generator graphs and returns modified versions of them, using generator graph transformations to ensure that (5.18) holds. In the remainder of this thesis we use the function \texttt{merge*} to denote the combination of the functions \texttt{merge} and \texttt{prepmerge}.

Several properties must hold to ensure that the graph returned by function \texttt{add} is valid. Firstly, the construction of the tiles in \(T\) from the tiles in \(T_1\) and \(T_2\) must be possible. Secondly, the construction method of \(T_1\) and \(T_2\) must be in the generator graph \(G_g\), otherwise the \texttt{prepmerge} cannot return generator graph that adheres to (5.18). Thirdly, all the triples have to be in canonical form. In summary, the graph containing the nodes \(T, T_1\) and \(T_2\) and edges from the latter two nodes to the first, must be a valid generator sub-graph.

More formally, the function \texttt{add}(G_g, (T, T_1, T_2)) (5.20) results in a generator graph if the following properties hold:

1. \(G_g\) is a generator graph
2. \(T = \text{norm}(T)\), i.e., the node \(T\) is in canonical form.
3. \((T \notin V \land \#S_T > 1 \implies (T_1 \subseteq V \land T_1 = \text{norm}(T_1) \land T_2 \subseteq V \land T_2 = \text{norm}(T_2)))\), i.e. when the nodes \(T_1\) and \(T_2\) are used in the construction of \(T\), they are in canonical form and part of the generator graph.
4. \((T \notin V \land \#S_T > 1 \implies (T \subseteq \{T_1 \oplus T_2 \mid (T_1 \in T_1 \land T_2 \in T_2) \lor (T_2 \in T_1 \land T_1 \in T_2)\}), \) i.e. when the nodes \(T_1\) and \(T_2\) are used in the construction of \(T\), it must be possible to actually construct the tiles from \(T\) by combining tiles from \(T_1\) and \(T_2\).

5.4 Generator graph construction

The composition techniques discussed in the previous section are used in the construction of generator graphs for specific associative window computations. In this section we discuss such a construction procedure and indicate how it can
be used to design heuristic algorithms. Furthermore, we discuss the auxiliary functions used to define the subroutines for our construction method.

Generator graphs can be constructed by using the function add successively, starting with the empty generator graph \((\emptyset, [])\), and making sure that the parameters adhere to the properties listed in Section 5.3.3. This results in a so-called bottom-up approach, where we start by adding tiles of small sizes and work towards larger tiles until we obtain a generator graph that contains all the windows \(W\) of the window computation.

We can also use the add in a top-down approach, where we start by adding \(W\) and work toward smaller tiles until we reach tiles of size 1. For this approach Property 3 from Section 5.3.3 forms a problem; the larger tiles need smaller tiles in their construction, but these smaller tiles have not been added to the generator graph yet. We can still follow the top-down approach, however, by delaying the adding of nodes to the generator graph until their composing nodes have been added. We discuss this top-down approach in more detail here, because the algorithms in Section 5.5 are based on it.

The top-down approach is based on two functions. The first one is a function that specifies how tiles from an existing tile-set should be split into two sub-tilers from (possibly different) tile-sets and which (possibly other) function specifies the construction method of these new tile-sets. Defining this function is the creative step in designing heuristic algorithms based on the top-down approach. This function therefore forms the core of the heuristic algorithm and we call it the heuristic function. The second function used by this approach is the function that recursively calls the heuristic function and adds the results to the generator graph.

We distinguish two separate top-down approaches, based on the heuristic function. In the first approach the heuristic functions simply specifies the construction method of a tile-set. We call this approach the basic top-down approach. In the second approach the heuristic function first partitions a given tile-set and specifies, possibly different, construction methods for each part. We call this approach the partitioning top-down approach. We discuss these two top-down approaches separately in the following two sections.

### 5.4.1 Basic top-down approach

In this section we discuss the basic top-down approach, using a basic heuristic function.

The method of construction of a tile set is defined by the heuristic function, say \(f\). The function differs for each algorithm, but all functions return the
same kind of tuple: \( f(T) = (T_1, f_1, T_2, f_2) \), where \( T_1 = \text{norm}(T_1) \) and \( T_2 = \text{norm}(T_2) \). The function effectively specifies that the tiles in tile set \( T \) are to be constructed from tiles in \( T_1 \) and \( T_2 \). It also specifies that the tile-sets \( T_1 \) and \( T_2 \) should be constructed in the manner defined by the heuristic functions \( f_1 \) and \( f_2 \) respectively. For basic algorithms we have \( f = f_1 = f_2 \), but more sophisticated algorithms switch heuristics at a certain point.

Although this heuristic function differs per algorithm, there are enough similarities between algorithms to introduce the so-called split function. This function is used to split a tile at a specific point into two sub-tiles:

\[
\text{split}(T, c, f_1, f_2) = (\text{cut}(T, c), f_1, \text{rem}(T, c), f_2) \quad (5.21)
\]

The result returned by function split adheres to the format for the heuristic function and is often used as a basis for such a function.

To split a tile into two sub-tiles the function split uses the functions cut and rem, which respectively return the first part of a tile and the remainder of the tile. More formally:

\[
\text{cut}(T, c) = \text{norm}((S_T[0..c], f_T, p_T)) \quad (5.22)
\]
\[
\text{rem}(T, c) = \text{norm}((S_T[c..#S_T], f_T, p_T)) \quad (5.23)
\]

where \( S_T[a..b] \) denotes the sublist of list \( S_T \), starting at index \( a \) up to but excluding index \( b \). The functions cut\((T, c)\) and rem\((T, c)\) therefore return normalized triples based on \( T \) by splitting the shape of the tile, i.e. the list of grid locations, at position \( c \).

The construction method in Figure 5.5(a) for \((\langle 5 \rangle, 0, 2)\), for example, is specified by \( \text{split}((\langle 5 \rangle, 0, 2), 1, f_1, f_2) = ((\langle 1 \rangle, 0, 2), f_1, (\langle 4 \rangle, 1, 2), f_2) \).

The other major part of the strategy for top-down algorithms is the function that recursively calls the heuristic function until the construction method of every tile has been specified and the generator graph \( G_g = (V_g, E_g) \) has been constructed. This function is called TD:

\[
\text{TD}(G_g, T, f) =
\begin{cases}
G_g & T \subseteq V \\
\text{add}(G_g, (T, \ldots)) & T \notin V \land \#S_T = 1 \\
\text{add}(\text{TD}(\text{TD}(G_g, T_1, f_1), T_2, f_2), (T, T_1, T_2)) & T \notin V \land \#S_T > 1
\end{cases}
\]

where \((T_1, f_1, T_2, f_2) = f(T) \) \quad (5.24)

This function returns an updated generator graph by taking a generator graph \( G_g \) and adding the node \( T \) to it (we assume that \( T = \text{norm}(T) \)), using the construction method specified by the heuristic function \( f \). The first and second case
5.4. GENERATOR GRAPH CONSTRUCTION

of (5.24) express the situations in which no further constructions are necessary, i.e. the $T$ does not have to be constructed, because construction methods are already in the graph$^3$ or $T$ contains tiles of size one. In the third case the triple $T$ is not only added as a node, but the function also recurses to add the nodes used in the construction of the new node.

This approach can therefore be summarized as divide, reuse and conquer, since we divide the problem of constructing tiles into the problem of constructing smaller tiles, and we reuse any construction methods already in place.

5.4.2 Partitioning top-down approach

It is also possible to define a heuristic function that partitions the tile-set $T$ before constructing it. We call such functions partitioning heuristic functions, and the top-down approach based on it is called the partitioning top-down approach.

A partitioning heuristic function does not return a single tuple describing the construction of a tile-set, but it returns a list describing the construction of all the parts of the tile-set. The elements of the list returned by a partitioning heuristic function are tuples of the form:

$$f(T)(i) = (T_0, (T_1, f_1, T_2, f_2))$$

for $0 \leq i < \#f(T)$ \hspace{1cm} (5.25)

Such a tuple effectively specifies that the tile-set $T_0$, which is a sub-set of $T$, is to be constructed from the two tile-sets $T_1$ and $T_2$. Furthermore, the tile-sets $T_1$ and $T_2$ themselves should be constructed according to the partitioning heuristic functions $f_1$ and $f_2$ respectively.

To obtain a valid generator graph the constructions specified by such a function must be valid. Additionally, there is the requirement that the list returned by the function covers the entire tile-set $T$, i.e., for any partitioning heuristic function $f$ the set $\{T_0 | (T_0, \cdot) \in f(T)\}$ partitions the set $T$.

The partitioning top-down approach that uses these partitioning heuristic functions to construct generator graphs is defined by the function $TDP$. This function is similar to the $TD$ function, but it is defined as:

$$TDP(G_g, T, f) =$$

$$\begin{cases} 
G_g & T \subseteq V \\
\text{add}(G_g, (T, \cdot, \cdot)) & T \notin V \land \#S_T = 1 \\
TDPr(G_g, T, f, 0) & T \notin V \land \#S_T > 1 
\end{cases}$$

\hspace{1cm} (5.26)

$^3$Note that we check $V$, not $V_g$. We do this because $(T \subseteq V) \not\equiv (T \in V_g)$, since $V_g$ can contain a partition of $T$ instead of $T$ itself.
with

\[
TDP_{Pr}(G_g, T, f, n) =
\begin{cases} 
G_g & n \geq \#f(T) \\
TDP_{Pr}(\text{add}(TDP(G_g, T_1, f_1), T_2, f_2), (T_0, T_1, T_2)), T, f, n + 1) & \text{otherwise}
\end{cases}
\]

where \((T_0, (T_1, f_1, T_2, f_2)) = f(T)(n)\) (5.27)

The function \(TDP\) uses the function \(TDP_{Pr}\) to add all the nodes into which \(T\) is partitioned by the partitioning heuristic function \(f\), and to recursively add the tile-sets from which \(T\) is constructed.

### 5.4.3 Auxiliary functions

The most important functions used to describe the algorithms in the next section have been covered in the previous section. There are, however, three more auxiliary functions which we use:

\[
pow2(a) = (\exists i : i \in \mathbb{N} : 2^i = a)
\]

\[
msb_1(a) = (\max i : 2^i < a : 2^{i+1})
\]

\[
lsb_1(a) = (\max i : 2^i \mid a : 2^{i+1})
\]

These functions respectively return: a boolean value that indicates whether \(a\) is a power of two and the power of two represented by the most-significant 1-bit of \(a\) and the least-significant 1-bit of \(a\).

### 5.5 Heuristic algorithms

In the previous sections of this chapter the problem of designing an efficient algorithm for a window computation has been reduced to that of finding a generator graph with a low cost. Finding an optimal generator graph, i.e. one with the lowest possible cost, is possible for smaller window sizes, but for larger problems the number of possibilities explode. Therefore we resort to heuristic algorithms, of which several are introduced in this section. The algorithms we present are intended for 1-dimensional associative window computations, but we also show how they can be generalized to support multi-dimensional computations.

We will examine the performance of the algorithms in terms of the cost function, i.e., in operations per output. We will also be interested in some other
measures, namely the meta-period and the depth of the graph. For the exact influence of these measures on the system performing the window computation we refer to Chapter 7 but a summary is provided here.

The meta-period influences the block size $L$, which has to be a multiple of the meta-period $m$ divided by the period of the windows $p_W$. A heuristic that results in a low meta-period therefore leaves us with more choices for the block size and allows for more precise fine-tuning of the throughput.

The depth of the graph is the length of the longest path between any two nodes in the generator graph. Since the generator graph is acyclic, and the tile size increases with every node on a path, the longest path will be at most $W - 1$. This depth influences the latency of any VLSI algorithm based on that particular generator graph, but it has little influence on an SIMD algorithm.

In each of the next four sections we discuss heuristic algorithms that produce generator graphs. The cost, meta-period and depth of these generator graphs are discussed in general terms, but they are also demonstrated using a running example. This running example is a window computation with $W = (\langle7\rangle, 0, 1)$.

### 5.5.1 Divide and conquer

In this section we present the heuristic algorithm obtained by applying a divide and conquer strategy. This heuristic adds tiles to the generator graph in a top-down fashion, and each tile is constructed from tiles that are, roughly, half its size.

The formal definition of the generator graph obtained by the divide and conquer heuristic is:

$$G_g = \text{TD}((\emptyset, []), \text{norm}(W), \text{dac})$$

where the heuristic function $\text{dac}$ is specified by:

$$\text{dac}(T) = \text{split}(T, \lceil \frac{\#S_T}{2} \rceil, \text{dac}, \text{dac})$$

The generator graph produced by this heuristic are in canonical form, since all tiles will have the same period, namely $m = p_W$. The cost and depth of the resulting generator graphs is $\Theta(\log W)$.

The result of applying this heuristic to a window computation with a window size of 7 is depicted in Figure 5.6(a). Note that the $\text{dac}$ heuristic divides tiles into tiles half the original size, this leads to the tile sizes 7, 4, 3, 2 and 1 in the generator graph.
Figure 5.6: Generator graphs produced by the dac-heuristic

Figure 5.6(b) depicts the generator graph obtained by this heuristic when a window size of 8 is used. Note that despite the larger window size the cost of the generator graph is less for the $W = 8$ computation. The divide and conquer heuristic obtains a low cost generator graph for $W = 8$, and for any window size that is a power of two, because in such cases it divides every tile into two parts of equal size.

Note that the generator graph in Figure 5.6(b) can be expanded, using Theorem 5.3.1 in such a way that it resembles the circuit from Figure 2.9(a). After expanding the graph such that all nodes have a period of 8, it resembles the circuit used for a logarithmic look-ahead computation for $L = 8$. This is no coincidence, implementing a system that performs a look-ahead computation based on the generator graph of Figure 5.6(b) as described in Chapter 7 results in a
system that performs the look-ahead computation in the manner described by
the logarithmic look-ahead approach of Section 2.5. In fact, any window com-
putation used to perform a look-ahead computation, as described in Section 4.5.1,
results in the same algorithm as the logarithmic look-ahead computation from
Section 2.5 when the divide and conquer heuristic is used.

5.5.2 Maximum reuse

The cost of a generator graph depends directly on the number of nodes in the
graph. So, to obtain a generator graph with a low cost it is important to reuse
any node as much as possible. The idea is that the reusing of a node means
that we do not have to introduce an other node and thus increase the cost of
the graph. The maximum reuse heuristic, which we present in this section, is
based on this idea.

The maximum reuse heuristic constructs a tile-set by partitioning it into
two tile-sets and constructing both of them from three other tile-sets such that
they both have one of those three tile-sets in common. This common tile-set is
therefore reused in the construction of two other tile-sets, resulting in a low-cost
generator graph. The common tile-set is as large as possible, hence it is called
the maximum reuse heuristic.

To determine the common tile-set we must be able to determine the overlap
between two tile-sets. To do this we have to determine the overlap between
individual tiles. To this end we introduce the following functions that determine
the set of respectively all prefixes and all postfixes of a given list \( T \):

\[
\begin{align*}
\text{prefix}(T) &= \{ T[0..i] \mid 0 \leq i < \#T \} \\
\text{postfix}(T) &= \{ T[i..\#T] \mid 0 < i \leq \#T \}
\end{align*}
\]

(5.33) (5.34)

The overlap between two tiles is determined by the intersection of their pre-
and postfix sets. Since we will be partitioning tile-sets, we are interested in
the overlap between tiles from the same tile-set. Therefore we introduce the
function \( \text{lovl}(T) \), which determines the largest overlap between two consecutive
tiles from the same tile-set \( T \):

\[
\text{lovl}(T) \in \text{postfix}(S_T) \cap \text{prefix}(S_T + p_T) \\
(\forall T: T \in \text{postfix}(S_T) \cap \text{prefix}(S_T + p_T) : \#T \leq \#\text{lovl}(T))
\]

(5.35)

Note that by consecutive tiles we mean tiles that have the same shape but are
shifted by the period of the tile-set.
The generator graph is constructed by the maximum reuse heuristic using:

\[ G_g = \text{TDP}((\emptyset, []), \text{norm}(W), \text{mr}) \]  

(5.36)

where the partitioning heuristic function \( \text{mr} \) is defined by:

\[
\text{mr}(T) = \begin{cases} 
\text{lrv}(T, \text{lrvl}(T)) & \text{if } \text{lrvl}(T) \neq [] \\
[(T, \text{split}(T, \text{msb}(\#S_T), \text{mr}, \text{mr}))] & \text{if } (\text{lrvl}(T) = []) \land \neg \text{pow2}(\#S_T) \\
[(T, \text{split}(T, \lfloor \frac{\#S_T}{2} \rfloor, \text{mr}, \text{mr}))] & \text{if } (\text{lrvl}(T) = []) \land \text{pow2}(\#S_T) 
\end{cases}
\]  

(5.37)

The first case of (5.37) describes the construction of tile-sets in which tiles overlap. These tile-sets are partitioned into two parts such that the parts have twice the period of the original tile-set. The construction method of the parts is such that they share one tile-set \( (T_0) \), determined by their overlap, resulting in a total of three new tile-sets to construct the two parts of the original tile-set. This construction is specified by function \( \text{lrv} \):

\[
\text{lrv}(T, T_0) = [(\text{norm}((S_T, f_T, 2p_T)), (\text{cut}(T', \#S_T - \#T_0), \text{mr}, T_0, \text{mr})), \\
(\text{norm}((S_T, f_T + p_T, 2p_T)), (T_0, \text{mr}, \text{rem}(T', \#T_0), \text{mr}))] \\
\text{where } T_0 = \text{norm}((T_0, f_T + p_T, 2p_T)) \\
\text{and } T' = (S_T, f_T, 2p_T)
\]  

(5.38)

The last two cases of (5.37) describe the construction of tile-sets without overlap. The construction of those tile-sets is based on the function \( \text{split} \) described earlier. The third case specifies the same construct method as the divide and conquer heuristic. The third case is used for non-overlapping tiles with a size that is a power of two, since we have seen in the previous section that the divide and conquer heuristic works well in such cases. The second case specifies the splitting of tiles using the \( \text{msb} \) function. This results in at least one tile with a size that is a power of two, on which we subsequently can apply the divide and conquer heuristic.

The generator graph depicted in Figure 5.7 is obtained when this heuristic is applied to a window computation with \( W = (\langle 7 \rangle, 0, 1) \). The cost of the computation represented by that generator graph is 2.25 operations per output. This is an improvement of almost 44% over the 4 opo required by the divide and conquer heuristic.

Analyzing the maximum reuse heuristic for any possible \( W \) is complicated. Since many window computations are of the form \( W = (\langle W \rangle, 0, 1) \) for \( W \in \mathbb{N}^+ \), however, we analyze the heuristic for those window computations.
Figure 5.7: The canonical generator graph produced by the $mr$ heuristic for $W = (7, 0, 1)$, with a meta-period $m = 4$ and 9 nodes with 2 incoming edges, resulting in a cost of $\frac{9}{4} = 2.25$ operations per output.
For such window computations we can sketch the operation of the heuristic in the following way:

**Phase one** Tiles with an overlap are split into their overlapping parts and a rest. The period of the constructing tiles is twice that of the original tile, while the size of the constructing tiles is less than that of the original. This process repeats, but at a certain point there are no tiles with overlap left because their period is larger than their size. This phase corresponds to the first case of (5.37), and the tile-sets whose construction method is determined in this phase have been indicated in Figure 5.7.

**Phase two** Tiles without overlap and a size that is not a power of two are split into a tile whose size is a power of two and a rest. Eventually this rest is also a tile with a size that is of a power of two, at the very least the rest will be a tile with a size of one. This phase corresponds to the second case of (5.37), this phase is skipped in Figure 5.7 since tiles without overlap and a size that is not a power of two do not occur.

**Phase three** Tiles without overlap and a size that is a power of two are split into two smaller tiles, whose size are again a power of two. This last process repeats until tiles of size one are reached. This phase corresponds to the third case of (5.37), and the tile-sets whose construction method is determined in this phase have been indicated in Figure 5.7.

Note that tiles created in a later phase do not have overlapping parts because of our assumption that the original windows have a shape that is \( S_W = \langle W \rangle \).

The first phase of the algorithm is the only phase in which tile-sets with new periods are added to the generator graph. All tile-sets added in the other phases have the same period as the tile-sets they construct. Therefore the meta-period of the generator graph is determined solely in the first phase of the heuristic. The meta-period \( m \) of the generator graph produced by the algorithm is \( m = 2^m \) where \( m = \lceil \log_2(W + 1) - 1 \rceil \).

This \( m \) is also used to express the cost of the tiles introduced by phase one of the heuristic. The first phase of the heuristic creates a tree, of sorts, in the generator graph. The the windows form the leaves of this tree and the last overlapping part forms the root. In Figure 5.7 this root node is \((4, 3, 4)\). This tree has a depth of \( m \), so it contains \( 2^{m+1} - 1 \) nodes. The construction of each node costs 1 operation, with the possible exception of the root node if it happens to be of size 1. If we exclude the root node, the contribution to the total cost per output of the first phase of the heuristic is \( \frac{2^{m+1} - 2}{2^m} = \frac{2^{m+1} - 2}{2^m} = 2 - \frac{2}{m} \) opo.
The construction of the root node, which contains tiles of size $W + 1 - m$, and the non-overlapping parts is handled in the other two phases.

The last two phases may produce the same tile-sets, allowing reuse between the two phases. The exact way in which the tile-sets are reused is hard to analyze, but it is possible to say something about them. The additional depth introduced by each phase is the logarithm of the size of the tiles, therefore the total depth of the generator graph produced by the maximum reuse heuristic is $\Theta(\log W)$.

All the tiles introduced in phase one of the algorithm as the “rest” tiles have a size that is a power of two and thus they are constructed using phase three. The size of the tiles of these tile-sets range from 1 up to, but excluding the meta-period. The periods of each tile-set is equal to the size of its tiles. Such a “rest” tile can be constructed from two such “rest” tiles that are half the size. For each size there are, possibly, two such tile-sets constructed. One of them has a phase that is equal to it’s size minus 1 and the other one has a phase that is equal to $W$ modulo the size of the tiles. If the phases of these tile-sets are the same, then the tile-sets are the same and thus only one tile-set needs to be constructed.

So the cost introduced in phase three of the algorithm for the “rest” tiles is:

$$1 - \frac{2}{m} + \begin{cases} \frac{1}{2^w} - \frac{1}{2^m-1} & \text{if } w < m \\ 0 & \text{if } w \geq m \end{cases}$$

opo, where $w$ is the largest $w$ for which $W \mod 2^w = 2^w - 1$ holds.

This leaves only the cost of the root node from phase one. This node is constructed in phase two or three, depending on its size. Whichever phase is used, the construction will look like a tree of depth $\Theta(\log(W - m))$. Usually some sharing between nodes from the other phases can be realized. Even without any sharing the total cost introduced by this tree will never be more than $O(\frac{W-m}{m})$ opo. Since $\frac{1}{2}W < m \leq W$, this means that the tree never introduces more than a constant cost, i.e. $O(\frac{W-m}{m}) = O(1)$.

This brings the total cost of a window computation, according to this heuristic, to:

$$3 - \frac{4}{m} + O(1) + \begin{cases} \frac{1}{2^w} - \frac{2}{m} & \text{if } w < m \\ 0 & \text{if } w \geq m \end{cases}$$

(5.40)

We have verified this cost experimentally for window sizes up to 250 and in those cases the cost of the maximum reuse heuristic is never more than 4 opo. Furthermore, (5.40) is exact for window sizes that are a power of two, or
CHAPTER 5. LOW-COST ASSOCIATIVE WINDOW COMP.

a power of two minus one. That is, the $O(1)$ term equals zero for those window sizes, since the root node from phase one has a cost of 0 in such cases. So, for window sizes that are a power of two the cost approaches 4 opo, and for window sizes that are a power of two minus one the cost approaches 3 opo.

5.5.3 van Herk-Gil-Werman algorithm

Our maximum reuse heuristic is not the only way to obtain a constant cost per output. In [36] van Herk and in [29] Gil and Werman develop an algorithm that results in a constant cost per output, regardless of the window size. In this section we describe their algorithm, formulate it in terms of our top-down strategy and discuss its performance.

The algorithm is based on the assumption that $W = (\langle W \rangle, 0, 1)$, i.e., that the window is solid and one-dimensional, and that the period is 1. It can be generalized to multiple dimensions by the methods described in Section 5.5.5.

The van Herk-Gil-Werman algorithm is based on the following equations:

$$
\begin{align*}
g(i) &= \begin{cases} 
\text{in}(i) & \text{if } i \mod W = 0 \\
g(i - 1) \odot \text{in}(i) & \text{otherwise}
\end{cases} \\
h(i) &= \begin{cases} 
\text{1}_\circ & \text{if } i \mod W = 0 \\
\text{in}(i) \odot h(i + 1) & \text{otherwise}
\end{cases} \\
\text{out}(i) &= h(i - W + 1) \odot g(i)
\end{align*}
$$

(5.41)

where $1_\circ$ is the identity of the $\odot$ operator.

We have adapted these equations from the ones in the original articles so that they also hold for even window sizes $W$, and so that the indexing of the output stream conforms to our I/O relation for associative window computation systems, i.e., such that an output only depends on inputs with a same or lower index. The computation specified by the equations remains the same, however.

According to [36], equation (5.41) should contain the case $h(i) = \text{in}(i)$ when $i \mod W = W - 1$ instead of $h(i) = i$ when $i \mod W = 0$. This would lead, however, to an algorithm that aggregates some inputs multiple times. This is alright for the idempotent $\odot$ operators discussed in [36], but not in the general case. Therefore we used the equations presented in [29] instead.

A generator graph based on this algorithm is constructed by:

$$G_g = \text{TDP}((\emptyset, []), \text{norm}(W), hgw)$$

(5.42)
where the partitioning heuristic function $h_{gw}$ is defined by:

$$h_{gw}(T) = \text{split1}((S_T, 0, W)) \oplus \left[ (T', \text{split}(T', W - i, \text{split1}, \text{splitL})) \mid T' = (S_T, i, W) \land 1 \leq i < W \right]$$ (5.43)

with

$$\text{split1}(T) = [(T, \text{split}(T, 1, \text{split1}, \text{split1}))]$$ (5.44)

$$\text{splitL}(T) = [(T, \text{split}(T, \#S_T - 1, \text{splitL}, \text{splitL}))]$$ (5.45)

Note that (5.41) is relatively simple compared to the notation needed to put the heuristic in our notation. Our notation, on the other hand, is capable of handling multiple heuristics in a consistent manner, and can handle multiple dimensions and window periods other than one.

The generator graph produced by the van Herk-Gil-Wereman heuristic for $W = (\langle 7 \rangle, 0, 1)$ is shown in Figure 5.8. We have marked the nodes whose construction method is determined by the $h$ and $g$ functions from (5.41), or respectively the $\text{split1}$ and $\text{splitL}$ functions from (5.44) and (5.45), accordingly.

This heuristic is relatively easy to analyze, compared to our maximum reuse heuristic. First of all, the meta-period is $m = W$. Secondly, the depth of the generator graph is $W - 1$, since tile-sets of every size, from 1 to $W$ are constructed in the generator graph. Thirdly, and finally, the generator graph contains $W + 2(W - 2)$ nodes with tiles larger than 1, namely $W$ tile-sets with tiles of size $W$, and $W - 2$ tile-sets from both the $h$ and $g$-chains. The cost of the generator graph is therefore $3 - \frac{4}{W}$ opo.

The cost approaches 3 operations per output, which is an improvement over the 4 operations per output of our maximum reuse heuristic. On the other hand, the meta-period and the depth of the generator graph are worse than those of the maximum reuse heuristic.

5.5.4 Harrington’s algorithm

In [34] Harrington patents a “method and apparatus for fast computation of associative operations over fixed size regions of a digital image”. In this section we formulate this algorithm in terms of our top-down strategy and analyze the generator graphs it produces.

The approach in [34] is generalized to multiple dimensions, but this generalization is based on the assumption that the $\odot$ operator is idempotent. Generalizing this method into multiple dimensions for non-idempotent operators can be done as described in Section 5.5.5.
Figure 5.8: The canonical generator graph produced by the \textit{hgw} heuristic for $W = ((7), 0, 1)$, with a meta-period of $m = 7$ and 17 nodes with two incoming edges, resulting in a cost of $\frac{17}{7} = 2.43$ operations per output.
Although the van Herk-Gil-Werman algorithm is not mentioned, it seems that Harrington's method is based on their algorithm, with a few improvements. The equations on which the algorithm is based resemble (5.41):

\[
\begin{align*}
    g(i) &= \begin{cases} 
    1 \odot & \text{if } i \mod (W + 1) = W \\
    g(i-1) \odot \text{in}(i) & \text{otherwise}
    \end{cases} \\
    h(i) &= \begin{cases} 
    1 \odot & \text{if } i \mod (W + 1) = 0 \\
    \text{in}(i) \odot h(i+1) & \text{otherwise}
    \end{cases} \\
    \text{out}(i) &= h(i-W+1) \odot g(i)
\end{align*}
\] (5.46)

where \(1 \odot\) is the identity element of the \(\odot\) operator.

We adapt his method to our notation in the following way:

\[
G_g = \text{TDP}((\emptyset, []), \text{norm}(W), \text{ht})
\] (5.47)

where the partitioning heuristic function \(\text{ht}\) is defined by:

\[
\text{ht}(T) = \text{split1}((S_T, 0, W + 1)) + \left( [T', \text{split}(T', W - i, \text{split1}, \text{splitL})] | T' = (S_T, i, W + 1) \land 1 \leq i < W \right) + \text{splitL}((S_T, W, W + 1))
\] (5.48)

with \(\text{split1}\) and \(\text{splitL}\) as defined in (5.44) and (5.45) respectively.

An example of a generator graph produced by this heuristic is shown in Figure 5.9. We have marked the nodes whose construction method is determined by the \(h\) and \(g\) functions from (5.46), or respectively the \(\text{split1}\) and \(\text{splitL}\) functions from (5.44) and (5.45), accordingly.

Like the van Herk-Gil-Werman algorithm this heuristic is relatively easy to analyze. First of all, the meta-period is \(m = W + 1\). Secondly, all tiles, except some of the windows, are formed by combining a smaller tile and a tile of size one. This means that the depth of the generator graph is \(W - 1\). Thirdly, each chain contains \(W - 2\) tile-sets. So, in each meta-period \(2(W - 2)\) tiles of sizes ranging from 2 to \(W - 1\) are constructed, incurring a cost of \(2W - 4\) \(\odot\) operations per \(W + 1\) windows. The \(W + 1\) windows are constructed using \(W + 1\) operations. So the total number of operations per meta-period is \(3W - 3\), which results in a cost of \(\frac{3W-3}{W+1}\) operations per \(W+1\) windows. This means that this heuristic results in a generator graph with a cost that is less than 3.

Both the van Herk-Gil-Werman algorithm and Harrington’s algorithm therefore result in a cost that approaches, but is never more than, 3 operations per
Figure 5.9: The canonical generator graph produced by the \texttt{ht} heuristic for $W = (\langle 7 \rangle, 0, 1)$, with a meta-period of $m = 8$ and 18 nodes with 2 incoming edges, resulting in a cost of $\frac{18}{8} = 2.25$ operations per output.
output. Examining the cost in detail shows that the cost of Harrington’s algorithm is lower for all $W > 2$. For the trivial case of $W = 2$ the costs are the same. The depth of both solutions is the same, but there is a small difference in the meta-period. Harrington’s approach results in a meta-period that is one larger than that of the van Herk-Gil-Werman algorithm.

5.5.5 Other heuristics

So far we have presented four heuristics, but many others are possible. We have presented some in [40], and the top-down strategy discussed in Section 5.4 can be used to design many more. Generally it suffices to design a heuristic for a one dimensional window computation, since the method discussed in this section can then be used to generalize it to handle multiple dimensions.

This generalization method is only applicable for a certain class of problems, namely the class of problems whose windows have the shape of a hyper-bar, i.e $S_W = \text{hyperbarlist}(s)$ for some $s \in \mathbb{Z}^D$, with

$$\text{hyperbarlist}(s) = \begin{cases} [j | 0 \leq j < s] & \text{if } s = [s] \\ (+ : 0 \leq j < s : \text{hyperbarlist}(s')) & \text{if } s = [s] + [s'] \end{cases} \quad (5.49)$$

For window computations that adhere to this property the generalized algorithm works as follows: We first use the one-dimensional algorithm to produce a solution for one-dimensional windows of size $s(0)$. This solution shows us how we can construct those windows from tiles of size 1. We then take this solution and extend all the generated tiles into another dimension. All the tiles get a size of $s(1)$ in this dimension. The construction methods we have for all the tiles are still valid, except for the tiles of size $1 \times s(1)$. We never generated a construction method for them because they used to be of size 1 in the one-dimensional problem.

So now we execute the algorithm again, but take the tiles with size $1 \times s(1)$ as the new windows and generate a way to constructing these tiles. Since these tiles only have a size larger than one in the second dimension, they can be considered one dimensional in practice and the algorithm can be executed normally on them.

We then repeat this process as often as necessary to get a solution in all the dimensions.
5.6 Conclusion

In this chapter we have presented a strategy, consisting of calculation graphs and generator graphs, that is suitable for representing the problem of designing associative window computation algorithms. We have also presented the concept of a heuristic function that can be used, together with the bottom-up approach, to represent heuristic algorithms that solve this design problem. Furthermore, we have applied this theory and presented four heuristic algorithms for this design problem and analyzed the solutions they provide.

The divide and conquer is the most basic of the heuristics we considered. For window computations, however, there are better heuristics. These are the maximum reuse heuristic, the van Herk-Gil-Werman heuristic and Harrington’s heuristic. Those heuristics result in a constant cost per output, regardless of the window size for windows of consecutive inputs.

The van Herk-Gil-Werman and Harrington’s heuristic apply only to window computations where the windows consist of consecutive inputs, but they produce algorithms with a very low complexity for those computations. The maximum reuse, which applies to every window computation, produces algorithms with a slightly higher complexity, but these algorithms have some other advantages. Both the meta-period and the depth of the maximum reuse heuristic are smaller than those of the other two heuristics. The smaller meta-period means that the block size of algorithms based on this heuristic can be tuned more accurately. The smaller depth of the means that the latency of the VLSI implementation can be reduced. Furthermore, the maximum reuse heuristic has the advantage that it does not require strided memory access operations when it is used to implement a look-ahead window computation on an SIMD processor. These advantages are discussed in more detail in for Chapter 7.

All three heuristics, however, show that the number of required operations per output does not grow with the size of the windows for windows of consecutive inputs. In fact, the number of operations required per output for those computations is not more than 4. This makes it possible to perform window computations at an extremely low-cost.
Chapter 6

Rank order filtering

6.1 Introduction

The heuristics described in the previous chapter result in low-complexity algorithms for associative window computations. In this chapter we apply these heuristics to rank order filters and we show that the examination of this specific system in more detail allows us to exploit its properties, resulting in even better algorithms.

Rank order filters are filters that perform an associative window computation. The rank order filter assumes some ordering relation on the inputs, which means that each input in a window has a certain rank. A rank order filter selects from each window the input with a specific rank \( r \) as output. Some results

Figure 6.1: Erosion and dilation filtering
produced by rank order filters are shown in Figure 6.1 and 6.2. These figures are obtained by sorting a rectangular window of pixels on their brightness. The rank order filter selects either the brightest ($r = W - 1$) pixel as output (erosion filtering), the darkest ($r = 0$) pixel (dilation filtering), or the pixel with the middle ($r = W/2$) rank (median filtering).

In addition to the standard rank order filters that select a single rank as output, we also consider rank order filters that produce a range of ranks $r \in [l..u]$. This allows us to model the window sorter, which selects the full range of ranks, and to model other interesting intermediate cases.

Parts of this chapter were published in [39].

6.2 Rank order filters

In this section we give a formal definition of rank order filters (ROFs).

A rank order filter assumes an ordering relation $\preceq$ on the domain of the inputs $\mathbb{D}$. When $T$ is a list of indices in the input stream $\mathit{in}$, i.e. a tile, we use the notation $\mathit{T}$ to denote the sorted list of data elements. So, the resulting list of inputs is a permutation of the inputs indicated by the indices in $T$, and the resulting list adheres to the following property:

$$(\forall i, j : 0 \leq i < j < \#T : T(i) \preceq T(j))$$

(6.1)

Using this notation we define a rank order filter as:

**Definition 6.2.1 (Rank order filter (ROF)).** A rank order filter performs a window computation parameterized by:

- A number of dimensions $D \in \mathbb{N}^+$
6.2. RANK ORDER FILTERS

- A set of windows $W = (S_W, f_W, p_W)$
- Two natural numbers $l$ and $u$ such that $0 \leq l \leq u < W$

Given an arbitrary input stream $in$, the rank order filter determines for each $W \in W$ the sorted list $\overline{W}[l..u]$, i.e., the elements of $W$ with ranks ranging from $l$ up to and including $u$.

Most rank order filters are designed for a specific value or range of values for the parameters $D, W, l$ and $u$. A photo processing application, for example, only needs 2-dimensional rank order filters ($D = 2$). In most cases we also have $l = u$, i.e., rank order filters that select one specific rank. We have chosen this particular definition, however, because it also includes so-called window sorters ($l = 0$ and $u = W - 1$) and other intermediate ranges of ranks which are useful in the design of rank order filters.

Some examples of rank order filters are:

**The 1-D median filter** with a kernel of size $2K + 1$:

- The number of dimensions is $D = 1$
- The windows themselves are given by $W = tiles(S_W, 1, 0)$, where the shape of the windows is $S_W = hypercube(K, D)$.
- We are interested in the item with rank $K$, hence $l = K = u$

Note that the window is usually called a kernel in filter terminology.

**The 2-D median filter** with kernel size $(2K + 1) \times (2K + 1)$:

- The number of dimensions is $D = 2$
- The windows themselves are given by $W = tiles(S_W, (1,1), (0,0))$, where the shape of the windows is $S_W = hypercube(K, D)$
- We are interested in the item with rank $2K^2 + 2K$, hence $l = 2K^2 + K = u$

Since each output is based on a finite window of inputs, rank order filters belong to class $FSM_*$. We show, however, that rank order filters also belong to class $FSM_*$ by choosing the correct aggregation operator. This aggregation operator is specific for each ROF, as discussed in the next section.
6.3 Sliced merge operator

Although ROFs belong to class $FSM_\ast$, and can therefore be implemented with linear scalability, we can obtain even more efficient algorithms if the ROFs belong to class $FSM^{\oplus}_\ast$. In that case we would apply the algorithms from Chapter 5 and require only a few aggregation operations per output.

In Chapter 4 we have shown that window sorters ($l = 0$ and $u = W - 1$) belong to class $FSM^{\oplus}_\ast$. Sorting each window of inputs is accomplished by choosing the merge-sort operation, which merges two sorted lists into a single sorted list, as the associative aggregation operator.

We could realize a rank order filter by first implementing a window sorter and subsequently selecting the range of ranks that is required by the rank order filter. For a rank order filter that selects only the element with the minimum or maximum rank, however, such an algorithm is overly complex. Selecting the minimum or maximum rank is accomplished by using the minimum or maximum operator as the aggregation operator. Such a minimum or maximum operator has a lower complexity than a merge sort operator, so this would reduce the complexity of the resulting algorithm considerably.

So, there is an associative aggregation operator for window sorters, minimum and maximum rank order filters. This indicates that there is an aggregation operator for each range of ranks, with a complexity ranging from a single operation for the lowest or highest rank to a full merge sort for the full range of ranks. In this section we explore this aggregation operator, which we have named the sliced merge operator, and in the next section we discuss its implementation.

For window sorters the aggregation operation is a merge sort operation:

$$\text{mergesort}(T_1, T_2) = T_1 \oplus T_2 \quad (6.2)$$

Note that the data will be reordered according to the ordering relation, so we have the property that:

$$T_1 \oplus T_2 = T_2 \oplus T_1 \quad (6.3)$$

This means that the $\text{mergesort}$ operation is not only associative, like the $\oplus$ operator, but also commutative.

As a first step in designing our aggregation operator for rank order filters we note that the rank order filter produces a range of ranks, instead of fully sorting the entire window. Therefore, we generalize (6.2) to:

$$T[l_0..u_0] = \text{mergesort}(T_1[l_1..u_1], T_2[l_2..u_2])[l_3..u_3] \quad (6.4)$$
where $\overline{T} = T_1 + T_2$. This equality does not hold for every $l_i$ and $u_i$ for $i = 0, 1, 2, 3$, however. An example of a merge operation where the equality holds is depicted in Figure 6.3.

To examine the relation between the parameters we introduce the partial function “rnk” to denote the rank of $\text{in}(g)$ in $\overline{T}$, for $g \in T$. More formally:

$$\text{rnk} : (\mathbb{Z}^+ \times \overline{T}) \rightarrow \mathbb{N}$$

$$\text{rnk}(g, T) = \min i : 0 \leq i < \#T \land \overline{T}(i) = \text{data}(g) : i$$

(6.5)

Observe that for any element $g \in T_i$ (for $i = 1, 2$) it holds that $\text{rnk}(g, T) \geq \text{rnk}(g, T_i)$. In other words, adding elements to a set can only increase the rank of an element already in the set, but it cannot decrease it.

Because $T$ contains $\#T - \#T_i$ elements more than $T_i$, we know that $\text{rnk}(g, T_i) \leq \text{rnk}(g, T) \leq \text{rnk}(g, T_i) + (\#T - \#T_i)$ for $i = 1, 2$. So the interval $[l_i..u_i]$, for $i = 1, 2$, must be such that it contains at least all elements $g \in T_i$ with $\text{rnk}(g, T_i) \leq u_0$ and $\text{rnk}(g, T_i) + (\#T - \#T_i) \geq l_0$, i.e., at least all the elements $g$ of $T_i$ for which $l_0 \leq \text{rnk}(g, T) \leq u_0$ might hold. This is expressed by:

$$0 \leq l_i \leq ((l_0 - \#T + \#T_i) \max 0)$$

$$u_i \min (\#T_i - 1) \leq u_i < \#T_i$$

(6.6)

Furthermore, the indices $l_3$ and $u_3$ in the list resulting from the merge operation on the sliced lists $T_1[l_1..u_1]$ and $T_2[l_2..u_2]$ have to be translated to the
indices $l_0$ and $u_0$ of the full list $T$. Note that the merge operation results in a list that is a sub-list of $T$. The elements from $T$ that are missing from the sub-list are elements that, as a consequence of (6.6), have a rank in $T$ that is not within the interval $[l_0..u_0]$. Also, there are $l_1 + l_2$ elements in $T$ that have a rank lower than $l_0$, and that are not in the sub-list. Therefore an index $r$ in the sub-list corresponds to a rank $r + l_1 + l_2$ in the list $T$, though only for $l_0 \leq r + l_1 + l_2 \leq u_0$. Therefore the translation between the indices in the final tile and the sub-list adhere to the following relation:

$$
\begin{align*}
l_3 &= l_0 - (l_1 + l_2) \\
u_3 &= u_0 - (l_1 + l_2)
\end{align*}
$$

(6.7)

A merge operation where the parameters adheres to (6.6) and (6.7) is called a valid merge operation.

From these properties we can draw some interesting conclusions. First of all, let us return to our original problem of calculating for a single window $W \in W$ a range of ranks $W[l..u]$. When we write such a calculation as a series of nested merge operations on a number of tiles $T_j$ for some $j$, we can use (6.6) to calculate the ranges for the lower and upper bounds for each of the tiles $T_j$. It turns out that these bounds depend only on $\#T_j, W, l$ and $u$. So we introduce functions that calculate the highest valid lower bound, and the lowest valid upper bound for a given ROF:

$$
\begin{align*}
\text{low} &: T \rightarrow \mathbb{N} \\
\text{low}(T) &= (l - W + \#T) \max 0 \\
\text{up} &: T \rightarrow \mathbb{N} \\
\text{up}(T) &= u \min(\#T - 1)
\end{align*}
$$

(6.8)

Note that $\text{low}(W_i) = (l - W + W) \max 0 = l$ and $\text{up}(W_i) = u \min(W - 1) = u$ for any $W_i \in W$.

Using these functions to compute $l_i$ and $u_i$ for $i = 1, 2$ guarantees the validity of the merge operation, w.r.t. (6.6), and at the same time minimizes the size of the interval $[\text{low}(T) .. \text{up}(T)]$. A small interval means little information to compute and to keep track of, which benefits the performance. So, once $T_1$ and $T_2$ have been chosen, the best choice for $l_1, u_1, l_2$ and $u_2$ is given by (6.8). Therefore we will be using these bounds throughout this chapter and demand that the aggregation of the inputs specified by a tile contains only that information that is relevant to the outputs of the system, i.e. we demand:

$$
\langle T \rangle = T[\text{low}(T) .. \text{up}(T)]
$$

(6.9)
Also, since (6.7) uniquely determines \(l_3\) and \(u_3\), we omit them in our notation of valid merge operations. Therefore we introduce the sliced merge operator \(\bowtie\), so that we can write (6.4) as:

\[
\langle T \rangle = \langle T_1 \rangle \bowtie \langle T_2 \rangle
\] (6.10)

Note that the sliced merge operator is associative and commutative, as can be expected from a generalization of the standard merge operation.

The output of a ROF now adheres to:

\[
\text{out}(i) = \langle W_i \rangle = (\bowtie g : g \in W_i : \langle g \rangle)
\] (6.11)

Note that \(\langle \{g\} \rangle = \text{in}(g)\), which means that ROFs belong to the \(FSM^*\) class of systems, since their outputs are aggregations of inputs using an associative aggregation operator: \(\bowtie\).

Our sliced merge operator ensures that the aggregated data is a sorted list, and that the sorted list contains only those elements that can influence the result of aggregating the final window. Note that this is what makes the rank order filter interesting: the size of the sorted list varies with the size of the tile and the range of ranks selected by the filter. Since the size of the sorted list varies among the tiles, it is no surprise that the cost of merging tiles also varies among different tiles. We examine this cost in the next section.

### 6.4 Implementation of the sliced merge

The aggregation operator used for the rank order filters is the sliced merge. This operation is basically a merge sort operation that is applied to lists of which the size has been minimized, with the additional property that only a slice of the resulting list is returned as a result. In this section we examine the complexity of this sliced merge operation and how it can be implemented.

Although the methods from Chapter 5 indicate that an ROF algorithm requires less than 4 sliced merge operations per output, this does not give us an accurate indication of the complexity of a rank order filter based on the same method. The number of operations required per sliced merge operation highly depends on the size of the tiles and the data being merged. This means that the heuristics from Chapter 5 result in very different complexities for ROFs when measured in the basic operations used to implement the sliced merge.

There are many sorting algorithms which can be used to implement a sliced merge, but for practical reasons we will resort to so-called oblivious sorting al-
algorithms. Oblivious sorting algorithms always perform the same number of operations in the same order, regardless of the data that has to be sorted, i.e. they are data-independent [45]. The well-known quick-sort algorithm, for example, is not oblivious, whereas another well-known sorting algorithm, namely merge sort, is oblivious. We choose oblivious sorting algorithms because they are data-independent and therefore have a single chain of execution with a constant complexity, whereas there is a large spread in the complexity and number of execution chains of non-oblivious algorithms. This makes oblivious sorting algorithms suited for implementation on SIMD processors and implementation in VLSI. For SIMD processors the single chain of execution enables us to execute several sorting operations in parallel, one on each of the PEs of an SIMD processor. Furthermore, when the operation is implemented in VLSI there is no overhead in the form of control logic.

The elementary operation of oblivious sorting is a so-called compare and swap (C&S) operation. This operation effectively sorts a list of two inputs by comparing them and swapping them if necessary. The same result can be obtained by using a minimum and a maximum operator (the so-called extrema operations). Sorting is then accomplished by making the first element of the resulting list the minimum of the two inputs and the last element of the resulting list the maximum of the two inputs.

In this chapter we use the number of extrema operations to measure the complexity of a ROF. The extrema operations allow for a more fine-grained determination of the cost and a more fine-grained optimization than C&S operations. If, for example, we only need one of the outputs of a C&S operation, we still need a single C&S operation. Whenever two extrema operations are used to implement each C&S operation, however, we can eliminate the ones that produce unnecessary outputs.

A sorting algorithm based on these extrema operations is represented by a sorting network. The best known algorithm for merging two sorted lists is Batcher’s merging network [45]. Batcher’s merging networks are discussed in Section 6.4.1. These networks are designed to merge two lists into a new list, whereas the sliced merge operation requires only a slice of the new list. This allows for some optimization, which we discuss in Section 6.4.2.

### 6.4.1 The cost of merging networks

We choose to implement each merge operation using an instance of Batcher’s merging networks, because those networks are the most efficient, known, networks for merging two sorted lists [45]. Also, there is a simple formula to
6.4. IMPLEMENTATION OF THE SLICED MERGE

calculate the number of extrema operations used in such a network. In this section we briefly explain both the structure of Batcher’s merging networks and the costs, in terms of extrema operations, induced by this structure.

Our approach differs from most of the literature, where a combination of the best known sorting networks and Batcher’s merging networks is used to implement ROFs. Sorting by recursively using Batcher’s merging networks is not optimal, so most authors assume a combination of the best known sorting networks and Batcher’s merging networks provides better results. In Section 6.7 however, we show that we improve upon the results existing in the literature, even though we only use Batcher’s merging networks.

A formula for the number of compare and swap operations in the network can be found in [45]. We, however, measure the cost of our computation in extrema operations. The cost function in [45] measures cost in C&S operations, so we adapt this cost function to produce its result in extrema operations. Also, since each tile $T$ represents only an interval $[\text{low}(T)..\text{up}(T)]$ of the resulting list, we do not need the entire merging network. We only need that part of the network that has an effect on the desired outputs, and we can prune the network to get rid of the rest of the extrema operations.

Pruning is a relatively simple optimization that can be done automatically at design time. In the case of a software implementation it is called “dead-code elimination” and in circuit design it is called “unused logic elimination”, which is performed automatically by the compiler or synthesizer respectively. So it is not a new optimization, but in this section we introduce the first cost function for such networks. This cost function allows us to evaluate our algorithm designs prior to implementation, which is a new feature.

The function $\text{PB}(s_1, s_2, l_0, u_0)$ for $0 \leq l_0 \leq u_0 < s_1 + s_2$ returns the number of extrema operations required by a pruned version of Batcher’s merging network which takes sorted lists, say $T_1$ and $T_2$, of size $s_1 = \#T_1$ and $s_2 = \#T_2$ respectively and produces the interval $[l_0..u_0]$ of the sorted list $T_1 \cup T_2$. This function $\text{PB}$ satisfies the following recurrence relation:

$$\text{PB}(s_1, s_2, l_0, u_0) =$$

$$\begin{cases} 
0 & \text{if } (s_1 = 0) \lor (s_2 = 0) \lor e \\
(u_0 \min 1) - (l_0 \max 0) + 1 & \text{if } (s_1 = s_2 = 1) \land \neg e \\
\text{PB}(\lceil \frac{s_1}{2} \rceil, \lceil \frac{s_2}{2} \rceil, \lceil \frac{l_0}{2} \rceil, \lceil \frac{u_0}{2} \rceil) + \\
\text{PB}(\lfloor \frac{s_1}{2} \rfloor, \lfloor \frac{s_2}{2} \rfloor, \lceil \frac{l_0}{2} \rceil, \lfloor \frac{u_0}{2} \rfloor - 1) + \\
(u_0 \min 2 \lfloor \frac{s_1 + s_2 - 1}{2} \rfloor) - (l_0 \max 1) + 1 & \text{otherwise}
\end{cases}$$

where $e \equiv (u_0 = -1) \lor (l_0 = s_1 + s_2)$

(6.12)
This recurrence relation has a larger domain than $0 \leq l_0 \leq u_0 < s_1 + s_2$. We expanded the domain to $l_0 \leq u_0 \wedge -1 \leq u_0 \wedge l_0 \leq s_1 + s_2$ to simplify the relation’s notation. The extra range of values introduced for $l_0$ and $u_0$ does not affect the outcome of the function, since we have that $PB(s_1, s_2, l_0, u_0) = PB(s_1, s_2, l_0 \text{ max } 0, u_0 \text{ min } (s_1 + s_2 - 1))$.

The recurrence relation is a modification of the cost function for Batcher’s merging networks without pruning, as given in [45]. The first case of (6.12) is explained as follows: if $s_1 = 0, s_2 = 0$ or if $e$ is true, then there is no merging necessary, since either one of the lists is empty ($s_1 = 0$ or $s_2 = 0$), or we need an empty interval from the resulting list ($e$ is true).

The second case occurs when two lists of length 1 are merged. So, if only one member of the resulting list is needed ($l_0 = u_0$) then we need 1 extrema operation to determine it. If both members of the resulting list are needed ($l_0 < u_0$) then we need 2 extrema operations.

The third case is the most complicated, but Figure 6.5 and the example in Figure 6.3 might help in understanding. In these figures a merging network is depicted by a box. The numbers in the upper left ($s_1$) and upper right corner ($s_2$) denote the size of the lists to be merged. The box contains a merging network that accomplishes that operation. This merging network consists of three parts, the first two are recursively merging the even and odd sequences, which have length $\lceil \frac{s_1}{2} \rceil + \lceil \frac{s_2}{2} \rceil$ and $\lfloor \frac{s_1}{2} \rfloor + \lfloor \frac{s_2}{2} \rfloor$ respectively. In the figure this recursion is depicted by the nesting of two boxes in the main box, which corresponds to
Figure 6.5: Batcher’s merging network for a 4-5 merge
the two recursive calls in equation (6.12).

Note that the recursion ends when the lists are size one, in which case the merging of the lists is accomplished by the network shown in Figure 6.4(c) where each extrema operation is represented by a black dot, and a pair of black dots represents a compare and swap operation [45]. The recursion also ends when one or both of the lists are empty, in which case the box contains no extrema operations, only connections from inputs to outputs, if any.

The third part of the merging network consist of a network that merges the results of the other two (odd and even) parts. In the figures this part has been marked by a dashed box. Its structure is such that it pairwise compares and swaps elements from the even and the odd list, starting from element 1 from the even list and element 0 from the odd list. This is illustrated in Figure 6.5 by the small gray numbers, which indicate the indices of the elements in the sorted lists.

This pair-merging network allows us to derive the lower and upper bounds of the ranks for the recursive calls; from the even sequence we need $\lceil \frac{l_0}{2} \rceil$ and $\lceil \frac{u_0}{2} \rceil$ as lower and upper bound respectively. This, and the structure of the pair-merging network implies that we need $\lceil \frac{l_0}{2} \rceil - 1$ through $\lceil \frac{u_0}{2} \rceil - 1$ from the odd sequence.

The cost of the pair-merging network is expressed by the last term of the third case of (6.12). This term looks daunting, but we can understand it by noting that every element passing through the pair-merging network takes part in one extrema operation, except for the element with index 0 and possibly the element with the highest index. So, we count the number of elements that pass through the pair-merging network to get the number of extrema operations. The first element that passes through the pair-merging network and participates in an extrema operation is $l_0 \max 1$. The last element that participates is $u_0$, unless $u_0$ is equal to the highest possible index, i.e $s_1 + s_2 - 1$, and $u_0$ is odd. Hence when pruning takes place the last element that participates in an extrema operation is the element with index $u_0 \min 2\lceil \frac{s_1 + s_2 - 1}{2} \rceil$.

6.4.2 Mirroring

Since Batcher’s merging networks are the best merging networks known, we might expect that their pruned versions are also the best. This, however, is not the case. In this section we examine an optimization that produces more efficient merging networks when pruning is involved.

Batcher’s networks are not the best when pruned because, when the range of desired ranks is not symmetric around the center of the window, the amount
6.4. IMPLEMENTATION OF THE SLICED MERGE

of pruning depends on the order in which the extrema operators are arranged in the network. As an example let $\mathbf{T}_1$ be a list of size $\#\mathbf{T}_1 = 1$ and let $\mathbf{T}_2$ be a list of size $\#\mathbf{T}_2 = 2$ and let $\mathbf{T} = \mathbf{T}_1 \uplus \mathbf{T}_2$ be a list of size 3, as shown in Figure 6.6. When $\mathbf{T}[0..2]$, i.e. every rank, is required Batcher’s merging network requires 4 extrema operations, as shown in Figure 6.6(a). When $\mathbf{T}[1..1]$, i.e. the median, is required a pruned version of the network containing 2 extrema operations suffices, as shown in Figure 6.6(c). When $\mathbf{T}[2..2]$, i.e. the maximum, is required a pruned version of the network also containing 2 extrema operations suffices, as shown in Figure 6.6(b). Based on symmetry we would expect a similar result for the minimum, however, when $\mathbf{T}[0..0]$, i.e. the minimum, is required a pruned version of the network containing only 1 extrema operation suffices, as shown in Figure 6.6(d).

This example suggests that in case of pruning we can come up with another...
automatic optimization which involves varying the layout of the network by varying the sorting order. We either sort in the standard ascending order, or we switch to a descending order if that is less costly. Changing the sorting order amounts to taking the mirror image of a sorting network. Hence we refer to this optimization as “mirroring”.

This mirroring optimization leads to a cost function that is almost identical to (6.12), but at each point in the recursion we now consider whether the pruned version of the original, or the pruned version of the mirror network, minimizes the cost:

\[
\begin{align*}
\text{MPB}(s_1, s_2, l_0, u_0) &= \text{MPB}'(s_1, s_2, l_0, u_0) \\
\text{MPB}'(s_1, s_2, l_0, u_0) &= \min \text{MPB}'(s_1, s_2, s_1 + s_2 - 1 - u_0, s_1 + s_2 - 1 - l_0) \\
&\begin{cases} 
0 & (s_1 = 0) \lor (s_2 = 0) \lor e \\
(u_0 \text{ min } 1) - (l_0 \text{ max } 0) + 1 & (s_1 = s_2 = 1) \land \neg e \\
\text{MPB}(\lceil \frac{s_1}{2} \rceil, \lceil \frac{s_2}{2} \rceil, \lceil \frac{l_0}{2} \rceil, \lceil \frac{u_0}{2} \rceil) + \\
\text{MPB}(\lfloor \frac{s_1}{2} \rfloor, \lfloor \frac{s_2}{2} \rfloor, \lceil \frac{l_0}{2} \rceil - 1, \lfloor \frac{u_0}{2} \rfloor - 1) + & \text{otherwise} \\
(u_0 \text{ min } (2 \lfloor \frac{s_1 + s_2 - 1}{2} \rfloor)) - (l_0 \text{ max } 1) + 1 \\
\end{cases}
\end{align*}
\]

(6.13)

This new optimization method can also be applied to networks other than pruned versions of Batcher’s merging networks. Optimal sorting networks that are pruned can also be mirrored prior to the pruning operation to see if this leads to a decrease in cost. In at least one case, namely in the solution presented in [46], this leads to an improvement.

We are now in a position to define the cost function MC that expresses the cost of merging tiles from tile-sets \(T_1\) and \(T_2\) into a new tile from tile-set \(T\). Using function MPB (6.13) and functions \(\text{low}\) and \(\text{up}\) as defined in (6.8) we define MC as:

\[
\begin{align*}
\text{MC}(T_1, T_2, T) &= \text{MPB}(\text{up}(S_{T_1}) - \text{low}(S_{T_1}) + 1, \\
&\text{up}(S_{T_2}) - \text{low}(S_{T_2}) + 1, \\
&\text{low}(S_T) - \text{low}(S_{T_1}) - \text{low}(S_{T_2}), \\
&\text{up}(S_T) - \text{low}(S_{T_1}) - \text{low}(S_{T_2})) \\
&\text{(6.14)}
\end{align*}
\]

We use this function to refine the complexity computation of a rank order filter. Instead of expressing the complexity in terms of number of \(\odot\) operations per output, we compute the complexity of each \(\odot\) operation, i.e. \(\odot\) operation, in terms of extrema operations using (6.14), and express the complexity in terms
of extrema operations per output. Therefore the cost function of a generator graph for ROFs is defined as:

\[ C_{ROF}(G_g) = \left( +T : T \in V_g : MC(\text{pred}_{0_g}(T), \text{pred}_{1_g}(T), T) \right) \Pi(\frac{pW}{m}) \]  

where \( \Pi(v) \) denotes the product of all the elements of vector \( v \). This function returns the cost of the generator graph in terms of extrema operations per output, which we abbreviate as epo from here on.

6.5 The ORM heuristic

The heuristic algorithms we used to construct generator graphs in Chapter 5 are designed to minimize the number of aggregation operations. The cost of the final algorithm, however, also depends on the cost of the aggregation operator, and in case of ROFs this cost depends on \( W, l, u \) and the size of the tiles involved in the aggregation operation. Therefore we can obtain better results by taking these parameters into account. This is what the “Overlap Recursive Mirroring” heuristics do, which we discuss in this section.

6.5.1 Informal description

In this section we explain the “ORM” heuristic using some simple examples. The actual specification of the algorithm is presented in Section 6.5.2

Like the heuristics presented in Chapter 5 the ORM heuristic starts with an empty graph and the tile-set \( W \). It then recursively constructs the generator graph, while decomposing the tile-set \( W \) into tile-sets with increasingly small tiles. The decomposition of the tile-sets is done in one of the following two ways.

Option 1: Overlap. If \( r+1 \) consecutive tiles from the current tile-set share a common part, called an \( r \)-overlap, this indicates a common sub-calculation that can be re-used \( r \) times. Therefore overlaps are good candidates for intermediate tiles.

Figure 6.7 illustrates the 1-overlap between every two consecutive tiles from the tile-set \( T = (\langle 7 \rangle, 0, 2) \). A new tile-set \( T_o = (\langle 5 \rangle, 2, 4) \) is constructed that contains tiles in the shape of this overlap and two new tile-sets \( T_{s1} = (\langle 2 \rangle, 0, 4) \) and \( T_{s2} = (\langle 2 \rangle, 3, 4) \) are constructed to contain the remaining parts. Furthermore, instead of adding \( T \) to the graph the two tile-sets \( T_1 = (\langle 7 \rangle, 0, 4) \) and \( T_2 = (\langle 7 \rangle, 2, 4) \) (which partition the set \( T \)) are added. The tiles in \( T_1 \) are constructed from tiles in \( T_o \) and \( T_{s1} \) and the tiles in \( T_2 \) are constructed from tiles...
CHAPTER 6. RANK ORDER FILTERING

Figure 6.7: \( \mathbf{T}_1, \mathbf{T}_2 \in (\langle 7 \rangle, 0, 2) \) share the overlap \( \mathbf{T}_o \in (\langle 5 \rangle, 2, 4) \)

in \( \mathbf{T}_o \) and \( \mathbf{T}_{s2} \). The new tile-sets \((\mathbf{T}_o, \mathbf{T}_{s1} \) and \( \mathbf{T}_{s2} \)) are constructed by applying the algorithm recursively.

In this example we only considered overlap between pairs of consecutive tiles, but overlap between three, or even more consecutive tiles may occur and in case of multi-dimensional problems the overlap may vary per dimension. The overlapping part of multiple tiles is smaller than the overlapping part of two tiles, but can be reused more often. The 2-overlap between the tiles from \( \mathbf{T} = (\langle 7 \rangle, 0, 2) \), for example, is formed by the tile-set \( \mathbf{T}_{o'} = (\langle 3 \rangle, 4, 6) \). The tiles form \( \mathbf{T}_{o'} \) are smaller than the ones from \( \mathbf{T}_o \), but they can be used to construct 3 different tiles from \( \mathbf{T} \), as opposed to only 2. This multiple overlap creates a difficult trade-off; either a small (cheap) part is reused many times, or a large (expensive) part is reused a few times. To make this trade-off our algorithm considers all possibilities (including the option of ignoring overlap) and selects the one that results in the graph with the lowest cost.

Option 2: No overlap. This option is used to construct tile-sets in which overlap does not occur, or is ignored. This option consists of two variants:

Option 2a: The algorithm searches the generator graph accumulated thus far for the tile-set containing the tiles that have the largest overlap with the tiles from the current tile-set. The tile-set found in this manner is reused and a new tile-set is constructed for the remaining non-overlapping parts by applying the algorithm recursively.

Option 2b: Let \( n \) be the largest power of two that divides the size of the tiles. The algorithm splits the tiles in the current tile-set into two tile-sets; one containing tiles of size \( n \) and one containing the remaining parts. So a 1-dimensional tile of size 14 is split into a tile of size 2 and tile of size 12. A multidimensional tile is split in this manner along the dimension in which it is largest. The two resulting tile-sets are then constructed by applying the algorithm recursively.

The choice between option 2a and 2b is made by estimating the (relative) cost of both options. The estimated cost for step 2a equals the size of the
original tile minus the size of the available tile. The estimated cost for step 2b equals the size of the largest of the two parts into which the tile would be split. Note that these estimates give no indication of the actual cost. They only serve to choose between the two variants.

Note that the recursion in this algorithm always ends because the size of the tiles in the tile-sets decreases with each recursive step.

The algorithm as described here has a very large running time when implemented in practice, so we make the following modifications. First we limit the number of overlaps considered in option 1 by introducing an upper limit $R$ such that only $r$-overlaps with $r \leq R$ are considered. Second, we only consider tiles that have a simple shape, i.e. that of a $D$-dimensional hyper-bar. So, when the tile is not a hyper-bar we ignore option 1 and only use option 2. Furthermore, option 2b is altered such that, if the tile is not a hyper-bar, we split the tile into the largest hyper-bar contained within it and the remaining part.

These modifications keep the computation time of the ORM implementation under control. They do not affect the outcome of the algorithm much. Firstly, because we will be considering windows in the shape of a hyper-bar in our examples and comparisons to existing algorithms. Secondly, because we sampled the results of the algorithm for large $R$ and the results do not improve much for $R > 4$, if they improve at all.

Because of its importance we attach the value of $R$ as a subscript to the name of the algorithm. So ORM$_1$ is the ORM algorithm with $R = 1$, ORM$_2$ is the algorithm with $R = 2$, etc. The parameter $R$ influences the complexity of the heuristic, and its effects on the results of the algorithm are discussed in Section 6.6.

### 6.5.2 Algorithm specification

In this section we specify the ORM heuristic described in the previous section in a more formal manner. We first handle the preliminaries in Section 6.5.2.1 and present the base of the algorithm in Section 6.5.2.2 after which the two decomposition options are discussed in Section 6.5.2.3 and 6.5.2.4 respectively.

Note that the aggregation operator used in ROFs is commutative. This means that the order of the elements in a tile or shape is no longer significant. We can therefore treat tiles as bags$^1$ instead of lists for ROFs. For this reason we use the “$\cap$”, “$\setminus$” and “$\subseteq$” operators on lists in this section. These operations treat the lists as bags and they are defined more formally in Appendix A.1.4.

---

$^1$Bags are sets which can contain multiple instances of the same element.
6.5.2.1 Auxiliary functions

In this section we introduce auxiliary functions used in the formal description of the ORM algorithm.

The first function that we need is function \( \text{ovl}(T) \), which returns the set of possible manners in which tiles from \( T \) can be overlapped with tiles from the same tile-set, including the possibility that there is no overlap. In other words it returns all the possible overlaps that the algorithm considers for option 1 from Section 6.5.1, plus the possibility of using option 2 in case of no overlap.

\[
\text{ovl} : \ (T \times \mathbb{Z}^D \times \mathbb{Z}^D) \rightarrow \mathcal{P}(\mathbb{Z}^D)
\]

\[
\text{ovl}(T) = \begin{cases} 
\{ g | g \in \text{hypercube}(R \cdot 1_D, D) \land g \geq 0_D \land (S_T \cap (g \cdot p_T + S_T)) \neq \emptyset \} & \text{if } S_T \text{ is a hyper-bar} \\
\{ 0_D \} & \text{otherwise}
\end{cases}
\]  

(6.16)

Each element of the returned set is a vector. This vector does not represent a coordinate on the grid, but a way in which the tiles overlap with other tiles from the same tile-set. So \((0,0)\) means no overlapping, \((1,0)\) means that a tile has a 1-overlap in the first dimension, but that there is no overlap in the second dimension, \((2,1)\) means that a tile has a 2-overlap in the first dimension and a 1-overlap in the second dimension, etc. This function is used to generate the set of overlaps considered by the algorithm.

We only consider overlap when \( S_T \) is a hyper-bar, we do this because this makes it possible to implement function \( \text{ovl} \) in a fast manner in practice. Whenever the tile is not a hyper-bar, we only consider the overlaps represented by the set \( \{0_D\} \), i.e., only option 2 from Section 6.5.1 is used.

Furthermore, the parameter \( R \) is used in function \( \text{ovl} \) to control the number of possibilities, thereby keeping the complexity of the ORM heuristic under control. So, for \( R = 0 \) the algorithm only considers constructing the tile without making use of any overlap. For \( R = 1 \) it also considers constructing a tile by using the 1-overlap. For \( R = 2 \) it considers all the possibilities of \( R = 1 \) and the possibility of using the 2-overlap. And so forth for larger values of \( R \).

To describe the ORM algorithm we need some additional auxiliary functions. There is the \text{hyperbar}, which returns a set of coordinates that fall inside, or on the edge, of a hyper-bar with the dimensions specified by the vector \( s \):

\[
\text{hyperbar} : \ \mathbb{Z}^D \rightarrow \mathcal{P}(\mathbb{Z}^D)
\]

\[
\text{hyperbar}(s) = \{ g | 0 \leq g \leq s \}
\]  

(6.17)
Furthermore, there are functions that deal with the overlapping parts of tiles:

\[
\begin{align*}
os : \left( (T \times \mathbb{Z}^D \times \mathbb{Z}^D) \times \mathbb{Z}^D \right) & \rightarrow T \\
\text{o}\mathsf{s}(T, o) & = \left( \cap g : g \in \text{hyperbar}(o) : S_T + g \cdot p_T \right) \\
op : \left( (T \times \mathbb{Z}^D \times \mathbb{Z}^D) \times \mathbb{Z}^D \right) & \rightarrow \mathbb{Z}^D \\
op(T, o) & = (o + 1_D) \cdot p_T \\
ot : \left( (T \times \mathbb{Z}^D \times \mathbb{Z}^D) \times \mathbb{Z}^D \right) & \rightarrow \mathbb{Z}^D \\
\text{o}\mathsf{t}(T, o) & = \text{norm}(\text{o}\mathsf{s}(T, o), f_T, \text{o}\mathsf{p}(T, o))
\end{align*}
\]  

(6.18) (6.19) (6.20)

In these functions the parameter \( o \) is an element of the set produced by function \( \text{ovl} \) that indicates the manner of overlap under consideration. The function \( \text{o}\mathsf{s} \) returns the shape of the overlapping part of the tiles, \( \text{o}\mathsf{p} \) the period and \( \text{o}\mathsf{t} \) returns the normalized triple representing the tile-set of overlapping parts.

### 6.5.2.2 Algorithm base

The generator graph created by the ORM heuristic is defined as follows:

\[
G_g = \text{orm}(\emptyset, [], \{\text{norm}(W)\})
\]  

(6.21)

where function \( \text{orm}(G_g, T) \) extends the given generator graph \( G_g \) such that the resulting generator graph contains all the tile-sets in the set \( T \).

The function \( \text{orm} \) distinguishes three cases. First, there is the degenerate case in which no tile-sets have to be added:

\[
\text{orm}(G_g, \emptyset) = G_g
\]  

(6.22)

Secondly, we consider the case in which more than two tile-sets have to be added. In this case the ORM algorithm adds the tile-sets individually to the generator graph, one by one, starting with the smallest tiles:

\[
\text{orm}(G_g, T) = \text{orm}(\text{orm}(G_g, \{T\}), T \setminus \{T\})
\]  

with \( \#T > 1 \) and \( T \in T \) such that \( \#S_T \) is minimal  

(6.23)

These first two cases are used later on in the recursive part of the algorithm. Thirdly, there is the case in which exactly one tile-set has to be added. In that case the function \( \text{orm} \) tries several possible overlaps between tiles from this tile-set and selects a generator graph with the lowest cost in the following manner:

\[
\text{orm}(G_g, \{T\}) \in \{\text{ov}(G_g, T, o) | o \in \text{ovl}(T)\} \text{ such that } \mathcal{C}_{\text{ROF}}(\text{orm}(G_g, \{T\})) = (\min o : o \in \text{ovl}(T) : \mathcal{C}_{\text{ROF}}(\text{ov}(G_g, T, o)))
\]  

(6.24)
The function \texttt{orm} therefore selects the generator graph with the lowest cost produced by function \texttt{ov}. This function \texttt{ov} produces a generator graph by using the given overlapping pattern \texttt{o} for the tile \texttt{T}.

\[
\text{ov}(G_g, T, o) = \begin{cases} 
\text{ovt}(G_g, T, o) & o \neq 0_D \\
\text{nov}(G_g, T) & o = 0_D 
\end{cases}
\]

The generator graph in which overlap for tiles in \texttt{T} is used is generated by \texttt{ovt}, and the generator graph in which no overlap is used is generated by \texttt{nov}. Note that these two functions correspond to option 1 and 2 from Section \ref{sec:overlap} respectively. We describe these two functions in a formal manner in the following two sections.

### 6.5.2.3 Using overlap

In this section we describe option 1 from Section \ref{sec:overlap} in a formal manner. That is, we describe how the ORM heuristic decomposes a tile-set into smaller tile-sets based on a certain number of overlaps.

Consider the tile-set \texttt{T} = ((4), 0, 1) where we consider the overlap indicated by \texttt{o} = 2, i.e., the 2-overlap between three consecutive tiles from \texttt{T}. In such a case we use the overlap ((2), 2, 3) and we would need ((2), 0, 3), ([0, 3], 1, 3) and ([2, 4], 3) to construct the tiles from \texttt{T}. This construction is done by the function \texttt{ovt}, which is based on the following functions:

\[
\begin{align*}
\text{ctile}(T, o, g) &= \text{norm}(S_T + g \cdot p_T, \text{op}(T, o)) \\
\text{rtile}(T, o, g) &= \text{norm}((S_T + g \cdot p_T) \setminus \text{os}(T, o), f_T + g \cdot p_T, \text{op}(T, o))
\end{align*}
\]

The functions \texttt{ctile} and \texttt{rtile} return the tile-sets describing, respectively, \texttt{T} and its non-overlapping parts at a phase indicated by \texttt{g} for a specific overlapping pattern \texttt{o}.

These functions are used by function \texttt{cmeth} to produce a list of construction methods. The construction methods are such that \texttt{T} at every phase indicated by \texttt{g}, as specified by \texttt{ctile}, is constructed by using the overlapping part, specified by \texttt{ot}, and a non-overlapping part, specified by \texttt{rtile}. These construction methods are subsequently be added to the generator graph using the \texttt{addlist} function:

\[
\text{cmeth}(T, o) = [(\text{ctile}(T, o, g), \text{ot}(T, o), \text{rtile}(T, o, g)) | g \in \text{hyperbar}(o)]
\]
addlist\((G_g, l)\) = \[
\begin{align*}
  & G_g & \text{if } l = [] \\
  & \text{addlist}(\text{add}(G_g, (T, T_1, T_2)), l') & \text{if } l = (T, T_1, T_2) + l'
\end{align*}
\]  

(6.29)

Function \text{overt} specifies the adding of these construction methods to the generator sub-graph, and the recursive calls to \text{orm} for the constructing tiles:

\[
\text{overt}(G_g, T, o) = \text{addlist}(\text{orm}(G_g, \{\text{rtile}(T, o, g) \mid g \in \text{hyperbar}(o)\}) \cup \{\text{ot}(T, o)\}), \text{cmeth}(T, o))
\]

(6.30)

6.5.2.4 Using no overlap

The previous section describes how the heuristic algorithm constructs tiles with overlap using function \text{overt}. In this section we describe function \text{nov}, which is used when the algorithm decides to consider the possibility of constructing a tile without overlap (either as one of the possibilities, or because the tiles simply do not overlap). Function \text{nov} corresponds to option 2 described in Section [5.5.1]

Option 2 itself consists of two options, named 2a and 2b. The first one, option 2a, searches in the tiles generated so far for any tile that overlaps with the tile that is to be constructed. At the very least it finds \([D_1, 0_D, 1_D]\), but perhaps it can find something bigger. If the tile it finds is of “reasonable” size, it is used in the construction. Otherwise the second one, option 2b, is used and the tile to be constructed is created by splitting it into two parts. This splitting is done in one of two ways: if the tile is not a hyper-bar then the largest possible hyper-bar is found and split from the tile. If the tile is a hyper-bar we take the dimension in which it is largest, say of size \(x\), and split the tile along that dimension in a tile of size \(\text{lsb}_1(x)\) and \(x - \text{lsb}_1(x)\).

The algorithm decides what a “reasonable” size is by estimating the amount of work that still has to be done after the available tile has been reused. It does this by taking the size of the tile and subtracting the size of the available tile. It then estimates the amount of work that has to be done if the tile is split. It does this by taking the size of the largest of the two components in which the tile should be split. The idea is that the smaller of the two parts is reused in the larger part, so that no extra costs are incurred for the smaller part. These two estimates are compared and the option with the lowest estimate is used. Note that these estimates give no indication of the actual cost, they only serve to choose between the two options.
CHAPTER 6. RANK ORDER FILTERING

So, the construction of tiles without reusing the overlap is done as follows:

\[
\text{nov}(G_g, T) = \text{add} \left( \text{orm}(G_g, \{T_1, T_2\}), (T, T_1, T_2) \right)
\]
where \((T_1, T_2) = \text{ch}(V_g, T)\) \hspace{1cm} (6.31)

The function \text{nov} adds the construction method specified by the function \text{ch} to the generator graph and calls \text{orm} to add the composing tile sets to the generator graph. The function \text{ch} makes the choice between either reusing available tiles, or by splitting the tile without reuse:

\[
\text{ch}(V_g, T) = \begin{cases} 
\text{reu}(V_g, T) & \#S_T - \#S_{T_2} \leq \#S_{T_3} \max \#S_{T_4} \\
\text{spl}(T) & \text{otherwise}
\end{cases}
\]
where \((T_1, T_2) = \text{reu}(V_g, T) \land (T_3, T_4) = \text{spl}(T)\) \hspace{1cm} (6.32)

Splitting depends on whether the shape is a hyper-bar or not:

\[
\text{spl}(T) = \begin{cases} 
\text{spl}_1(T) & S_T \text{ is a hyperbar} \\
\text{spl}_2(T) & \text{otherwise}
\end{cases}
\]

(6.33)

If the shape happens to be a hyper-bar we split it along the largest dimension:

\[
\text{spl}_1(T) = \left( \text{norm}(S_{T_1}, f_T, p_T), \text{norm}(S_{T_2}, f_T, p_T) \right) \text{ such that}
\]
\[
\begin{align*}
S_T &= \text{hyperbarlist}(q) \\
S_{T_1} &= \text{hyperbarlist}(r) \\
S_{T_2} &= S_T \setminus S_{T_1} \\
r(e) &= \text{lsb}_1(q(e)) \\
(\forall d : 0 \leq d < D \land d \neq e : r(d) = q(d)) \\
(\forall d : 0 \leq d < D : q(d) \leq q(e))
\end{align*}
\]

(6.34)

Otherwise we just extract the largest hyper-bar from the shape:

\[
\text{spl}_2(T) = \left( \text{norm}(S_{T_1}, f_T, p_T), \text{norm}(S_{T_2}, f_T, p_T) \right) \text{ such that}
\]
\[
\begin{align*}
S_{T_2} &= S_T \setminus S_{T_1} \\
S_{T_1} &= \text{the largest hyper-bar in } S_T
\end{align*}
\]

(6.35)

The function \text{reu} returns, given an original tile, a triple containing an already
existing tile that can be reused and the rest of the original tile.

\[ \text{reu}(V_g, T) = \left( T_1, \text{norm}( (S_T + f_T) \setminus (S_{T_1} + g \cdot p_{T_1} + f_{T_1}), 0, p_T) \right) \text{ such that} \]
\[ g \in \mathbb{Z}^D \]
\[ S = \{ T_2 | T_2 \in V_g \land (p_T \ mod \ p_{T_2} = 0) \land \]
\[ (S_{T_2} + g \cdot p_{T_2} + f_{T_2}) \subseteq (S_T + f_T) \} \]
\[ T_1 \in S \text{ such that } \# S_{T_1} \text{ is maximal} \] (6.36)

where the set from which \( T_1 \) is selected is the set containing all the triples describing tile set of which all the tiles are a subset of a tile described by \( T \). Furthermore, this set is constructed in such a way that the tiles occur at a regular position in \( T \), which is expressed by the variable \( g \).

### 6.5.3 Examples of generated graphs

In this section we present the generator graph constructed by our heuristic algorithm for \( W = ((9), 0, 1) \) and compare its costs with those of the generator graphs obtained by the other heuristics from Chapter 5.

The ORM\(_4\) algorithm generates the same generator graph shown in Figure 6.8 for both window sorters and median filters. This generator requires 2.5 merge operations per output. This is better than the 5 and 2.56 required by respectively the divide and conquer and the van Herk-Gil-Werman algorithm, equal to the 2.5 required by the maximum reuse algorithm and worse than the 2.4 required by Harrington’s algorithm.

In the end, however, the ORM\(_4\) heuristic produces an algorithm with a lower complexity, because its cost in terms of extrema operations per output is only 29 for a window sorter. This is a an improvement over all four algorithms, since a window sorter based on the divide and conquer algorithm, the maximum reuse algorithm, Harrington’s algorithm or the van Herk-Gil-Werman algorithm results in a complexity of respectively 46, 30.5, 31.6 and 33.3 extrema operations per output.

For median filtering the cost of the generator graph is 10 extrema operations per output. In contrast, a median filter based on the divide and conquer algorithm, maximum reuse algorithm, Harrington’s algorithm or the van Herk-Gil-Werman algorithm results in respectively 30, 11, 12 and 13.11 extrema operations per output.
Figure 6.8: The canonical generator graph produced by the ORM$_R$ heuristic for $1 \leq R \leq 4$ and $W = \text{tiles}((9), 0, 1)$, at a cost of 29 epo for a window sorter ($l = 0, u = 8$), and 10 epo for a median filter ($l = 3 = u$).
6.6 Results

In this section we define the three filter classes that we use to evaluate the ORM heuristic and examine the impact of the parameter $R$. These classes are also used in the next section to compare the results of the ORM heuristic with designs from the literature and the results of other heuristics. These three classes are therefore chosen such that they are representative for filters commonly found in literature and provide sufficient variety to illustrate the differences between the heuristic algorithms.

Although our heuristics could easily generate generator graphs for ROFs with an arbitrary number of dimensions, we restrict our attention to one- and two-dimensional cases, and in the latter case we only consider square windows. Although for each window size there are many rank order filters that select a single rank and many more that select an interval of ranks, we restricted our attention to the following three classes:

**Dilation filters** This rank order filter is interesting because the minimum is the least costly single rank that we could wish to obtain. Note that the dilation filter, on grounds of symmetry, costs the same as the erosion filter.

**Median filters** This rank order filter is interesting because the median is the most costly single rank that we could wish to obtain.

**Window sorters** Since a window sorter obtains every rank, this class gives us an upper bound for any filter which has to select one or more ranks.

Note that all results for the dilation filter correspond directly to the results of the heuristic algorithm for other associative window systems. After all, the dilation filter is a associative window computation where the aggregation operation is the minimum operator. In other words the cost of a dilation filter in extrema operations per output is the same as the cost in aggregation operations per output.

In figures 6.9, 6.10 and 6.11 the cost in terms of extrema operations per output have been plotted for respectively dilation filters, median filters and window sorters. Each figure contains plots for both one-dimensional and two-dimensional filters generated by the ORM algorithm for $1 \leq R \leq 4$, with the higher values of $R$ depicted by darker shades of grey. In many cases the ORM algorithms produce the same design, regardless of the parameter $R$, in that case the plot associated with the highest value of $R$ obscures the plots associated with the lower values of $R$. The data used to plot these graphs can be found in
Figure 6.9: Dilation filtering using the ORM heuristic
Figure 6.10: Median filtering using the ORM heuristic
Figure 6.11: Window sorting using the ORM heuristic
Appendix A of our technical report [40], and it was generated using the tools in Appendix B.1 of this thesis.

Figure 6.9 shows us that the cost for dilation filtering is asymptotic. One dimensional dilation filtering costs no more than 4 extrema operations per output, and for two dimensional filtering not more than 8 extrema operations per output are needed. Furthermore, Figure 6.11 shows that the cost of window sorters designed by the ORM algorithm grows linearly with the size of the windows.

For median filtering the cost of the design grows logarithmically with the size of the windows, as shown in Figure 6.10. This complexity is worse than the best known complexities for median filtering [44, 81] because we resort to oblivious sorting. Our designs, however, have three major advantages over these other algorithms precisely because we rely on oblivious sorting. First of all our designs allow for scalable parallelism; the throughput of the filter scales linearly with the amount of parallelism. Secondly, the constant factor for our designs is very low, which means that our designs perform better for practical (i.e. relatively small) window sizes. Thirdly, our designs can be implemented both in VLSI and on SIMD processors with very little overhead, because there is only one chain of execution in oblivious sorting methods.

As we can see from these figures, and especially from Figure 6.11, an increase in $R$ generally results in a lower cost generator graph. This decrease in cost is the result of the algorithm taking more generator graphs into consideration when $R$ is large, i.e., it has a larger search space when $R$ is large. Figure 6.11 also shows that increasing the parameter $R$ from 1 to 2 reduces the cost of the designs produced by the ORM algorithm. Increasing the parameter from 3 to 4 however, does not yield the same cost decrease, since the data points for $R = 4$ actually obscure those for $R = 3$.

A higher $R$ can also lead to a design with a higher cost, however. In Figure 6.10(b) we see that the ORM$_1$ algorithm sometimes produces better results than the ORM$_4$ algorithm. Although a larger value for $R$ increases the size of the search space, the larger search space is not necessarily a super-set of the search space covered for a lower $R$.

Consider the following example: the function orm selects the best generator graphs generated by function ov. For $R = 1$ the best graph it encounters is $G_1$. For $R = 2$ it also encounters $G_1$, but also the slightly better $G_2$. So the ORM$_1$ algorithm selects $G_1$ and the ORM$_2$ algorithm selects $G_2$. If this happens to be an intermediate step in the generation of the final graph, however, then there are still some tiles that have to be added to the selected graph. The cost of adding tiles depends on the chosen graph, since tiles already in the chosen graph might be reused. It is possible that some of the tiles which are to be added can be
constructed with more reuse in $G_1$ than in $G_2$. The net effect of this is that the final generator graph produced by the $R = 1$ algorithm can be less costly than the $R = 2$ algorithm, because the $R = 1$ algorithm based its solution on the (at first sight slightly more costly) $G_1$ graph.

The ORM$_1$ algorithm can generate, for two-dimensional median filtering, designs with a cost that is up to 2% lower than the designs produced by the ORM$_4$ algorithm. In general, however, the ORM$_4$ algorithm produces the designs with the lowest costs, so we use $R = 4$ to compare the ORM algorithm to the other design methods in the next section.

6.7 Comparison

In this section we compare the results found by our ORM$_4$ algorithm to the results found by the algorithms from Chapter 5 and the results available in the literature.

6.7.1 Dilation filter

In this section we examine the first class of filters, namely the dilation filters. The costs of these filters when designed by the ORM$_4$ algorithm and the algorithms from Chapter 5 is plotted in Figure 6.12.

The results for the dilation filter correspond directly to the results of the heuristic algorithm for other associative window systems. So, based on our analysis of the heuristic algorithms in Chapter 5, it is no surprise that for the one-dimensional dilation filter all the results in Figure 6.12 except of the divide and conquer algorithm, fall in the range of 3 to 4 extrema operations per output. For two-dimensional filters more or less the same holds; the cost of the divide and conquer algorithm increases steadily with the size of the windows, while the costs of the other algorithms never exceed 8 epo.

For dilation filters our ORM heuristic does not do worse than the maximum reuse heuristic, but neither does it improve upon Harrington’s algorithm [34] or the van Herk-Gil-Werman algorithm [36, 29]. The ORM algorithm, however, was not designed for dilation filtering specifically, but for rank order filters in general. In the next two sections we show that this pays off for the other filter classes.
Figure 6.12: Dilation filtering using ORM$_4$ and the heuristics from Chapter 5.
6.7.2 Median filter

In this section we survey the second class of problems: the median filters. The costs of these filters when designed by the ORM algorithm and the algorithms from Chapter 5 is plotted in Figure 6.12.

The ORM algorithm is not the best algorithm for dilation filtering, but it was designed for rank order filters in general and in Figure 6.13 we see that this pays off. The divide and conquer algorithm provides the most costly designs. After that come Harrington’s algorithm and the van Herk-Gil-Werman algorithm, whose designs are so similar that their plots partially obscure each other in the figure. Furthermore, the cost of the designs produced by these two algorithms grows linearly with the size of the windows. The maximum reuse algorithm and the ORM algorithm result in the lowest costs, and the cost of their designs grows logarithmically with the size of the windows.

Though the ORM algorithm results in the lowest costs, the difference between the maximum reuse algorithm and the ORM algorithm can only be seen in the two dimensional plot. So, although the maximum reuse algorithm was not designed for ROFs in particular, it still performs well.

Note that Figure 6.13(b) also contains the results obtained by Kolte et al. [46], who focus specifically on designs that can be implemented on SIMD processors. Only for windows of size $W = 3 \times 3 = 9$, however, do their results improve upon the divide and conquer heuristic, but not the ORM heuristic. For all other window sizes the divide and conquer heuristic and the ORM heuristic produce designs with lower costs.

Other authors have also presented median filter designs in their papers [11, 12, 55], though not for such a large range of window sizes as in Figure 6.13. Therefore we have plotted a zoomed-in version of Figure 6.13 in Figure 6.14 and included their results there. Note that we have omitted the designs of the van Herk-Gil-Werman and Harrington’s algorithm. We did this to avoid clutter and because these algorithms produce neither the best results, like the maximum reuse and ORM algorithm, nor the worst results, like the divide and conquer algorithm.

To plot Figure 6.14 we had to convert the cost in C&S operations used in [11, 12, 55] to the cost in extrema operations. This conversion is done using the rule that 1 C&S operation equals 2 extrema operations. For median filters this conversion is not entirely fair; it might be possible to prune some of those extrema operations, but this can only be done when the design is known. The exact designs were not presented in [11, 12, 55], however, which is why we used this simple conversion rule.
Figure 6.13: Median filtering using ORM\textsubscript{4} and the heuristics from Chapter 5
(a) 1-dimensional

(b) 2-dimensional

Figure 6.14: Median filtering using various designs
6.7. COMPARISON

In their article about one-dimensional window sorters [55], Lucke and Parhi also make an excursion to the median filter for $W = 5$. Their approach is to specify the median filter as a stack filter in a max-min structure and then applying their method for implementing stack filters. As shown in Figure 6.14(a), however, the resulting solution is not very efficient. In fact, it is less costly to implement a window sorter according to their own method (see Figure 6.16(a) in the next section) than it is to implement a median filter.

Chakrabarti focuses entirely on 1-dimensional median filters in [11] and achieves better results. The costs of Chakrabarti’s designs could be lowered by pruning excess extrema operations. Although we are not sure how many extrema operations could be pruned, we know that it can never be more than 50%, namely in the case that all the C&S operations of the algorithm can actually be replaced by a single extrema operation. Even if the maximum of 50% of all the extrema operations could be pruned however, the cost of Chakrabarti’s designs would still be higher than that of the ORM4 heuristics for all $W$, except for $W = 3$. For $W = 3$, however, we are certain that 3 epo is the minimum, and that pruning Chakrabarti’s design will not lead to 2.5 epo.

In a later publication Chakrabarti and Wang [12] also consider 2-dimensional median filtering and, again, obtain good results, as shown in Figure 6.14(b). The ORM4 heuristic, however, produces the best results.

6.7.3 Window sorters

In this section we survey the third and final class of problems: the window sorters. The costs of these filters when designed by the ORM4 algorithm and the algorithms from Chapter 5 is plotted in Figure 6.15.

Figure 6.15 shows that the performance of the various heuristic algorithms, relative to each other, is similar to their relative performance for the median filters. The divide and conquer algorithm results in the highest costs, followed by Harrington’s algorithm and the van Herk-Gil-Werman algorithm with intermediate costs. The lowest costs are obtained by the maximum reuse algorithm and the ORM algorithm, with the ORM algorithm beating the maximum reuse algorithm.

Although the costs of Harrington’s algorithm and the van Herk-Gil-Werman algorithm appear to be linear in the size of the window, a quadratic curve provides a better fit. Only the maximum reuse algorithm and the ORM algorithm have a cost that grows linearly with the size of the window for window sorters.

Note that Figure 6.15(b) also contains the results obtained by Kolte et al. [46]. Although the focus of Kolte et al. lies on finding implementations
Figure 6.15: Window sorting using ORM$_4$ and the heuristics from Chapter 5
Figure 6.16: Window sorting using various designs
that are easy to implement on an SIMD processor, they nevertheless manage to get comparable results for smaller window sizes. The difference becomes larger, however, when the window sizes increase, and they do not improve upon any of the heuristics presented in Chapter 5 nor upon the ORM \(_4\) algorithm.

Lucke and Parhi also examine window sorters, which they call merge sorters (because of the merging used), in the first part of their paper [55]. The design method of Lucke and Parhi is quite similar to our ORM algorithm, in that it exploits overlap between successive windows. Instead of working with infinite input streams, however, they optimize for a fixed number of overlapping windows, which they call the block size. For all block sizes they considered the results shown in Figure 6.16 are the best results reported by them.

Note that we have omitted the designs of the van Herk-Gil-Werman and Harrington’s algorithm in Figure 6.16. We did this to avoid clutter and because these algorithms produce neither the best results, like the maximum reuse and ORM \(_4\) algorithm, nor the worst results, like the divide and conquer algorithm.

In one specific instance in Figure 6.16(a), indeed in anything we found in the literature, there is a better solution than the one found by the ORM algorithm. The 1-dimensional window sorter for \(W = 6\) by Lucke and Parhi [55] requires only 15 epo, compared to the 16 epo required by the solution of the ORM algorithm. Unfortunately we cannot explain this anomaly, since we have been unable to reconstruct this design of Lucke and Parhi from the algorithm described in their article. In all other cases their results come close to those of the ORM \(_4\) algorithm, but do not improve upon it.

### 6.8 Conclusion

In this chapter we examined a family of class \(FSM^\odot\) systems known as rank order filters. We introduced the associative aggregation operator used by these rank order filters and named it the sliced merge operator. The sliced merge operator takes two lists and produces a new list, but the size of the new list is not trivial to determine. Furthermore, the complexity of the operator depends on the sizes of the lists, which makes the rank order filters and interesting set of systems to study.

After examining the complexity and implementation of the sliced merge operation we presented a heuristic that takes these into account when constructing generator graphs. This heuristic, which we named the ORM heuristic, results in generator graphs with lower costs than any of the previous heuristics, which were not designed with rank order filters in mind. Furthermore, we have com-
pared the results of the ORM heuristic with design methods from the literature and we have shown that our approach results in algorithms with the lowest cost.

The maximum reuse heuristic, which has not been optimized for rank order filters, also performs well for the classes of rank order filters we examined. The maximum reuse heuristic is not the best heuristic for any class of algorithms so far, i.e., not for the associative window computations in general nor for median filters or window sorters. However, it seems to obtain results that are quite close to those obtained by the best heuristic for each particular class. This is especially interesting because the maximum reuse heuristic is also the only heuristic whose generator graphs can always implemented on an SIMD without requiring strided memory access, as discusses in the next chapter.
Chapter 7

Implementation

7.1 Introduction

In the previous chapters several algorithms for stream processing systems have been presented. In this chapter we discuss their implementation in VLSI and, primarily, on SIMD processors. We discuss the properties of SIMD processors in Section 7.2 and choose one processor, namely the NXP Embedded Vector Processor (EVP), to implement our examples on. In Section 7.3 we discuss the implementation of the pipeline structures from Section 2.3. These pipelines enable us to implement linear look-ahead and block post-computation techniques on a vector processor. We also discuss the implementation of algorithms described by the generator graphs introduced in Chapter 5 on SIMD processors. The general approach for generator graphs is discussed in Section 7.4 and in Section 7.5 we consider the special case of implementing a look-ahead computation based on a generator graph.

There exist vectorizing compilers which are capable, to some extend, of mapping a sequential program onto an SIMD processor. These compilers, however, perform relatively simple optimizations and the advanced algorithm transformations based on look-ahead or generator graphs are beyond them. None of them is able to derive the IIR or ROF implementations presented in this chapter by compiling a simple sequential program based directly on the system’s specification.
7.2 SIMD Processors

In this section we give a brief overview of the properties of SIMD processors. We discuss the properties that most SIMD processors have in common, or should have if they are to be used for block processing algorithms that scale linearly. We also show that, for scalability, the data manipulation operations of the processor are not as important as the data communication operations. That is, for a scalable implementation on an SIMD processor we need the same data manipulation operations used in a sequential implementation of the algorithm. The scalable implementation on an SIMD processor, however, requires some special data communication or transportation operations. Therefore we also discuss the various ways in which the processing elements (PEs) of an SIMD processor can communicate with the memory, and with each other.

The processors we consider are also known as vector processors or array processors. They are called vector processors or array processors because they can perform element-wise operations on vectors or arrays of length \( P \), where \( P \) depends on the processor in question. The Single Instruction Multiple Data (SIMD) processor is named such because all the operations it performs on the vector elements are the same, i.e., a single instruction is executed on the multiple elements of the vector. The SIMD processors that we consider are capable of performing these \( P \) operations in constant \( \Theta(1) \) time because they consist of \( P \) processing elements (PEs) operating in parallel.

Although the terms vector processor and array processor are used interchangeably nowadays, Gurd \cite{32} originally distinguished them as two separate classes. Older super computers, like the STAR-100 \cite{15} and CRAY-1 \cite{18}, belong to the vector processing class because they execute the vector operations on a single, extensively pipelined, PE. Although these machines are fast, the execution time of still depends on the size of the vector. These machines are unsuited for our purpose of obtaining scalable algorithms, since an increase in vector size results in a proportional increase in execution time. Therefore we will not consider them any further in this chapter and focus instead on machines that execute their vector operations using \( P \) processing elements simultaneously, i.e., machines belonging to the array processor class according to \cite{32}. Furthermore, we will use the terms vector processor and SIMD processor interchangeably.

In the next few sections we discuss the common and distinguishing properties of these processors, focussing mainly on the operations the processors can perform in \( \Theta(1) \) time. This is by no means a full taxonomy of SIMD processors like those found in \cite{20,32,35,74,79}, but it was inspired by the work of these authors. The main difference is that we are only interested in the operations
supported by the processor, not in the hardware architecture that makes these operations possible. Furthermore, we are only interested in the operations that are useful in the implementation of the algorithms presented in the previous chapters.

### 7.2.1 Common properties

In this section we discuss the properties that all SIMD processors have in common. These properties are, first, that there is a host processor, and that the SIMD processor’s $P$ processing elements (PEs), numbered from 0 to $P-1$, perform the operation broadcast by the host processor simultaneously. Secondly, that the SIMD processor is capable of applying the standard logical and arithmetic operations on vectors in an element-wise fashion. Thirdly, that each PE of the processor is capable of I/O so that it can read from (a part of) the input stream, and write to (a part of) the output stream. In this section we discuss the consequences of the second property and we also explain why the third property is necessary for scalable block processing algorithms.

The second property, that the SIMD processor is capable of applying the standard logical and arithmetic operations on vectors in an element-wise fashion, implies that the processor also supports the so-called masking of PEs. An SIMD processor with the masking feature is able to selectively turn its PEs on or off for a particular operation. So, instead of calculating $r = f(x)$ such a processor is capable of computing:

$$r = \text{mask} \cdot f(x) + (1 - \text{mask}) \cdot r$$  \hspace{1cm} (7.1)

where \text{mask} is a vector containing booleans, where each boolean is represented by a zero (for false) or a one (for true). The EVP$_{16}$ [3, 4], for example, supports this masking feature in hardware, but a vector processor that does not have this hardware support can still perform a masked operation in $\Theta(1)$ time by implementing (7.1) using the standard arithmetic operations it supports.

Many of the operations supported on one machine but not on another can be expressed using a small number of operations that are supported on that other machine. Another example is the multiply-accumulate (MAC) operation. A processor that does not support the MAC operation can simply implement it using a multiply and an addition operation. Or, conversely, a processor that does not support a multiply operation, like the AltiVec, can implemented it by using a MAC operation that accumulates with zero [33].

Therefore, for our purposes, it is not the computational operations, or data manipulation operations, that distinguish one vector processor from the other.
These operations certainly have an effect on the exact number of operations needed to implement an algorithm, but they do not influence the scalability of the algorithm. The computational operations supported intrinsically on one processor can be emulated by a constant number of standard arithmetic or logical operations on another processor. So, rather than the computational operations, it is the communication or data transport operations that distinguish one SIMD processor from the other. It is the way in which data is communicated between the different PEs of the processor and the memory of the system that distinguishes them. This is the subject of the next two sections, namely the communication between PEs and memory and the communication between the PEs themselves.

We already want to note here, however, that there is one communication operation that all SIMD processors should support: every single PE must be capable of I/O, i.e., reading inputs from the input stream and writing outputs to the output stream. The PEs must be capable of I/O because we are interested in algorithms whose throughput scales with the amount of hardware, i.e., the number of PEs. So an SIMD architecture should have I/O capabilities that scale with the number of PEs, since a high throughput algorithm on a low bandwidth system is impossible. The Kestrel processor [6], for example, has $P = 512$ processing elements, but I/O is handled by the host processor which can only communicate the data to two of the 512 PEs. The Kestrel processor is used in [6] to search for a DNA string in a database. The processor scales with the problem size, i.e., a larger number of PEs allows it to search for longer DNA strings in the same amount of time. Because not all the PEs support I/O operations, however, the processor cannot search DNA patterns of the same length any faster when the number of PEs is increased, since there is no way to feed the database any faster into the SIMD processor.

### 7.2.2 Memory access

In this section we take a look at the memory access operations of an SIMD processor. We make no distinction between read and write operations, but simply handle them all as memory access operations. We distinguish three of such operations that allow us to divide the SIMD processors into three hierarchical classes. We also discuss the strided memory access, which is useful for many of the algorithms presented in this thesis.

All the PEs of an SIMD processors are connected, at least conceptually, to a global memory. We distinguish the different SIMD processors by the way in which they can access this memory. A processor in which the PEs are, in reality,
7.2. SIMD PROCESSORS

each connected to a different memory bank is, for example, limited in the way
in can access the “global” memory of the system. In such a system each of the
$P$ PEs is capable of accessing only a $P$-th fraction of the memory.

The first class of systems that we consider is one where each PE is, at least
conceptually, connected to its own private memory. Furthermore, the memory
access operations on this system are such that each PE must use the same
address to access its private memory. This is the M1 class of processors and it
contains processors like the EVP$_{16}$ [4], or those that support the AltiVec [33]
and MMX [69] instruction sets. The SIMD processor is restricted to the M1
class memory access operation because these SIMD instructions are generally
extensions to a scalar processor and by limiting the processor to these memory
access operations there has to be only one address bus to a memory with a
width of $P$ words.

We map the private memories of the PEs of a M1 class processor onto our
global memory by multiplying the address in the private memory by $P$ and
adding the number of the PE. So, in terms of our global memory the M1 class
SIMD processor can only access consecutive memory locations from the global
memory. Furthermore, these memory accesses must be aligned, i.e., the global
memory can only be accessed such that PE 0 accesses a memory location with
an address such that that address modulo $P$ is 0. There is also such a thing as
unaligned memory access, where the memory locations must still be consecutive,
but there is no restriction on the global memory address used in PE 0. Such
an unaligned memory access can be emulated by two aligned memory access
operations and a few of the shift operations discussed in the next section.

The memory access operations of the M1 class are not what one might expect
from an SIMD processor, since a memory address is, generally, not considered
to be part of an instruction, but rather it is considered to be data. So an SIMD
processor should be able to handle multiple memory addresses so that each
PE can access its own memory bank at a different location. Processors that
support this kind of memory access fall into the second class, labeled M2, like
the MasPar MP-1 [5].

For the M2 class we use the same mapping as for the M1 class to map
the private memories of the PEs onto the global memory. The PEs of an M2
class processor can access memory locations in the global memory of which the
address modulo $P$ equals the number of the PE.

The M2 class of processors is interesting because these processors can ex-
ecute MIMD programs with a constant, through considerable, overhead [33].
The drawback of multiple memory address buses makes these processors more
expensive and hence less common in practice.
The third and final class of SIMD processors is the M3 class. Processors from this class are, at least conceptually, connected to a global memory and the PEs are capable of accessing any memory location. These type of memory access operations are also known as scatter (for writing) and gather (for reading) operations. This class corresponds to the classical PRAM model, with the restriction that all the processors execute the same program. To support such memory accesses in practice, however, a large interconnection network is needed between the PEs and collisions in the data transport become a problem. These problems can be solved, indeed there is an effort to implement a PRAM model on a chip [82], but scaling such a processor is not straightforward. So, since there are no SIMD processors that support these memory operations, we regard the M3 class as one that contains only theoretical processors.

Most existing SIMD processors currently fall into the M1 class, therefore we will use this class for our examples in the rest of this chapter. The other two classes are certainly interesting, but there are few representatives of the M2 class and the none of the M3 class.

This covers the classes into which we divide the SIMD processors based on their memory access operations. Note that each class can emulate all the memory access operations supported by its predecessor, but not the other way around, forming an hierarchical order on the classes.

Note that we have not mentioned strided memory access. We did this because strided memory access can fit into different classes to some extent. Recall that a strided memory access operation is a memory access operation where the distance between the memory addresses accessed by subsequent PEs is equal. This distance is called a stride. The M3 class of processors support strided memory accesses fully, but the M2 class is also able to support some strided memory accesses. The M2 class of processors support strided memory access operations on the global memory, as long as the stride modulo $P$ is 1 and the memory address for the first PE modulo $P$ equals 0. If the processor also supports unaligned memory access it supports strided memory access with any address for the first PE, as long as the stride modulo $P$ is 1. If the processor has a completely connected topology, discussed in the next section, then it supports strided memory access with strides that are relatively prime with $P$. The reason for this is that when the stride is relatively prime with $P$, there are no collisions in the memory access operations and the completely connected topology enables the communication of the data between the PEs. Technically, the M1 class of processors also supports strided memory access operations, though only an extremely limited set of them, namely those where the stride is 1.
7.2. SIMD PROCESSORS

7.2.3 Topology

All SIMD taxonomies concern themselves with the topology of the PEs, i.e., the way in which they are connected to each other. These connections determine the possible ways in which data can be communicated between the different PEs. We, however, are not concerned with the physical connection between the PEs, but in the \( \Theta(1) \) communication operations that the topology allows. Neither are we interested in a full taxonomy. In this section we only discuss the topologies that provide operations that support the implementation of block processing algorithms with linear scalability.

There are many possible topologies, such as linear, ring, perfect shuffle, grid, cube, complete, etc. The communication operations that these topologies support are characterized by the allowed pattern vectors \( \mathbf{p} \) for the communication operation that constructs a new vector \( \mathbf{n} \) by taking values from a source vector \( \mathbf{v} \) according to:

\[
    n(i) = v(p(i)) \quad \text{for} \quad 0 \leq i < P
\]

(7.2)

Note that this description allows not only permutations, but all kinds of copying and reorganizing of the vector elements, since an element of the source vector can be copied multiple times. In this respect the memory access operations and the topology overlap a bit. A system belonging to the M3 class can emulate all the operations specified by (7.2) in \( \Theta(1) \) time, but a system that supports (7.2) cannot emulate all M3 memory access operations in \( \Theta(1) \) time.

In the next few sections we introduce several topologies based on the pattern vectors supported by the SIMD processor. After introducing these topologies we compare them and briefly consider other topologies.

7.2.3.1 Completely connected topology

An SIMD processors that supports the communication operation (7.2) for any \( \mathbf{p} \) in \( \Theta(1) \) times is said to have a completely connected topology. The reason for this is that, to support this operation in \( \Theta(1) \) time, each PE of the processor has to be connected to every other PE, essentially forming a complete graph with \( P \) nodes. The operation using an arbitrary pattern vector \( \mathbf{p} \) is called a permute [33], shuffle [4], or swizzle [78] operation. Since we use the EVP\(_{16}\) as our example processor we use the EVP terminology and call it a shuffle operation.

To completely connect all PEs with each other physically, the system needs \( \Theta(P^2) \) connections, which does not scale well. Despite this drawback there are several SIMD processors that support it, since it is such a useful topology.
Many other SIMD processors, however, support only a small number of specific pattern vectors \( \mathbf{p} \). We discuss these patterns in the next sections.

### 7.2.3.2 Linear and Ring topologies

Two common and similar topologies are the linear and ring topologies, which we discuss in this section.

A processor that supports the pattern vectors:

\[
\mathbf{p}_{\text{shift}(k)}(i) = i - k
\]

(7.3)

for \( k = 1 \) and \( k = -1 \) is said to have a linear topology. Such a processor supports the shift operation over a distance of \( k = 1 \), and there is generally functionality to set some default value for the original vector elements \( \mathbf{v}(i) \) with \( i < 0 \) or \( i \geq P \).

A processor that supports the pattern vectors:

\[
\mathbf{p}_{\text{rot}(k)}(i) = i - k \mod P
\]

(7.4)

for \( k = 1 \) and \( k = -1 \) is said to have a ring topology. Such a processor supports the rotation of a vector over a distance of \( k = 1 \). Note that a processor with a ring topology can emulate shift operations in \( \Theta(1) \) by using a rotation operation and a masked assignment.

Recall that we are only interested in the operations supported by the processor in \( \Theta(1) \) times. Processors that support a shift operation can also support the rotate operation by inserting the element that falls out of the vector at one back into the vector at the other end. On the Kestrel processor, for example, there is no physical connection between the first and last PE, but by using a few extra operations on the array controller a rotate operation can be emulated in \( \Theta(1) \) time [6].

Topologies that allow the rotation or shifting of a vector over a distance of more than one, i.e. \( k > 1 \), allow an SIMD processor to perform unaligned memory accesses in \( \Theta(1) \) time. The number of connections in such a topology, however, corresponds to \( \Theta(P^2) \), which means that the processor might as well have a completely connected topology.

### 7.2.3.3 Perfect shuffle topology

The perfect shuffle is a permutation that is very useful in parallel processing, as shown by Stone [76].
The perfect shuffle permutes the elements of a vector according to the following pattern:

\[
p_{ps}(i) = \begin{cases} 
2i & \text{for } 0 \leq i \leq \frac{1}{2} P \\
2i + 1 - P & \text{for } \frac{1}{2} P \leq i < P 
\end{cases}
\]  

(7.5)

As we did for the ring topologies we assume that an SIMD processor with the perfect shuffle topology also supports its own inverse, which is specified by the pattern vector:

\[
p^{-1}_{ps}(i) = \begin{cases} 
\frac{i}{2} & \text{for } i \text{ mod } 2 = 0 \\
\frac{i-1}{2} + \frac{P}{2} & \text{for } i \text{ mod } 2 = 1 
\end{cases}
\]  

(7.6)

Note that the perfect shuffle essentially rotates the address bits of the vector elements. That is, vector element \( i \) with \( i = i_{p-1}2^{p-1} + i_{p-2}2^{p-2} + \ldots + i_12^1 + i_02^0 \) of the source vector is mapped onto the element \( p_{ps}(i) = i_{p-2}2^{p-1} + i_{p-3}2^{p-2} + \ldots + i_02^1 + i_{p-1}2^0 \). The inverse of the perfect shuffle does the same, but rotates in the other direction.

In terms of physical connections, the PEs of an SIMD processor with a perfect shuffle topology are each connected to only 2 other PEs. Furthermore, the PEs are partitioned into sets, called necklaces, and the PEs in each set are connected in a ring topology [52]. This makes this topology scalable in practice, but the partitioning means that there can never be communication between the different necklaces. Therefore the perfect shuffle topology is generally combined with another topology, which we discuss in the next section.

### 7.2.3.4 Perfect shuffle nearest neighbor topology

In this thesis we are particularly interested in the perfect shuffle nearest neighbor (PSNN) topology. We use this PSNN topology in Section 7.5 to avoid strided memory access operations, allowing us to implement a scalable look-ahead computation on a M1 class SIMD processor with a PSNN topology.

The PEs in this topology are connected according to the perfect shuffle pattern and to their nearest neighbors, making this topology a combination of the ring topology and the perfect shuffle topology [31]. An SIMD processor with this topology therefore supports, next to (7.4) for \( k = 1 \) and \( k = -1 \), also (7.5) and (7.6).

The PSNN topology is not only useful because of the permutation operations that it supports, it is also useful because it can emulate other permutation operations using a small number of perfect shuffles, rotations and their inverses. In Section 7.5 for example, we need rotation operations over a distance of \( 2^j \)
and $-2^j$ for $0 \leq j < p$ with $P = 2^p$. We note that the perfect shuffle rotates the address bits of the vector, as explained in Section 7.2.3.3 and that the rotate adds one to or subtracts one from the address bits of the vector. This means that to rotate over a distance of $(-)2^j$ we simply need to rotate the address bits until bit $j$ is in the least significant position, perform a rotation operation and rotate the address bits back. More formally:

$$P_{rot(2^j)} = (P_{ps})^j \circ P_{rot(1)} \circ (P_{ps})^{-j}$$ (7.7)

$$P_{rot(-2^j)} = (P_{ps})^j \circ P_{rot(-1)} \circ (P_{ps})^{-j}$$ (7.8)

Of course, we can also rotate in the opposite direction first:

$$P_{rot(2^j)} = (P_{ps}^{-1})^{p-j} \circ P_{rot(1)} \circ (P_{ps})^{p-j}$$ (7.9)

$$P_{rot(-2^j)} = (P_{ps}^{-1})^{p-j} \circ P_{rot(-1)} \circ (P_{ps})^{p-j}$$ (7.10)

This means that a rotate over a distance of $2^j$ or $-2^j$ can be emulated on an SIMD processor with a PSNN topology using 1 rotate operation and either $j$ perfect shuffles and $j$ inverse perfect shuffles, or $(p-j)$ perfect shuffles and $(p-j)$ inverse perfect shuffles, whichever is less.

In a similar manner the PSNN topology can emulate the rotate operation over any distance $d$ with $d < 2^j$ or $d \mod 2^{p-j} = 0$ using $\Theta(j)$ operations. Furthermore, the PSNN topology, together with masked operations, supports all the permutations of the so-called shuffle-exchange network, and therefore it supports many common permutations in $O(p) = O(\log P)$ operations [51], and any other permutation in $O(\sqrt{P})$ operations [50].

On an SIMD processor with the PSNN topology each PE is connected to only 4 other PEs, which limits the number of wires, making this topology suitable for scaling in this respect. Obtaining a layout for the PEs such that the length of the wires and other cost measures are minimal is, however, not trivial and this problem has been studied by in [52], where this topology is referred to as shuffle-shift.

### 7.2.3.5 Other topologies

In this thesis we are only interested in the topologies discussed in the previous section. The completely connected topology covers all possible pattern vectors, and any special pattern vectors we need are covered by the perfect shuffle and ring topologies. So, although other topologies may allow for some specialized pattern vectors, we do not need those patterns.
Consider a grid topology consisting of a grid of $N \times M$ PEs all connected to their nearest neighbor. The $N$-th order IIR filter can be implemented on a ring topology with $M$ PEs, as discussed in Section 7.3.2. This ring topology implementation can also be adapted to run a grid of PEs. Each of the PEs in the ring topology implementation has to perform many vector operations on vectors of length $N$. So, on a $N \times M$ grid topology these vector operations on vectors of length $N$ could be executed efficiently. This grid topology with $N$ times more PEs would therefore increase the throughput of the implementation by, at most, a factor of $N$. However, because the implementation on the ring topology scales linearly, the easiest way to increase the throughput by a factor of $N$, is to simply take the implementation on the ring topology and increase the number of processors in the ring by a factor of $N$.

Our implementations scale linearly. So, although the throughput of our implementations could be improved by other topologies, the easiest and most certain way to increase their throughput is therefore to increase the number of PEs and use the same topology. We therefore restrict ourselves to the topologies discussed in the previous section.

7.2.3.6 Comparison of topologies

So, we have the linear, ring, perfect shuffle, PSNN and completely connected topology. Except for the perfect shuffle these topologies form a hierarchical ordering on the topologies of SIMD processors.

The completely connected topology can emulate any communication operation provided by the other topologies. The PSNN can emulate any communication operation provided by the other topologies, excluding the completely connected topology, in $\Theta(1)$ operations. Furthermore, the ring topology can emulate a linear topology in $\Theta(1)$ operations. The perfect shuffle topology is the only topology that cannot emulate any of the other topologies.

In the remainder of this thesis we are mainly interested three topologies: ring, PSNN and completely connected. The ring topology is interesting, since this topology is supported by virtually all SIMD processors in practice. The completely connected topology is also interesting because it is so versatile, and it is actually supported on some SIMD processors in practice, like the EVP$\_16$ [3] and the AltiVec [33]. We are interested in the PSNN topology because it has a small number of connections, but these are enough to implement a look-ahead computation without also requiring strided memory access operations.
7.2.4 Example: Embedded Vector Processor

The Embedded Vector Processor (EVP\textsubscript{16}) is an SIMD processor designed for embedded applications \cite{4}. It belongs to the M1 class and has a completely connected topology. In the remainder of this chapter we implement our examples on this processor, so we discuss it in more detail in this section.

Any SIMD processor of the M1 class with a completely connected topology is able to run the programs presented in the remainder of this chapter. The EVP\textsubscript{16}, however, has some features that allow us to obtain a very low cycle count for these programs.

The EVP\textsubscript{16} supports Very Long Instruction Word (VLIW) parallelism. This means that it can perform multiple operations simultaneously, as long as those operations require different functional units. The functional units for the vector operations that we will use are a load/store unit, an ALU unit, a MAC unit and a shuffle unit. Furthermore, the EVP\textsubscript{16} also has functional units to handle scalar operations, like the load/store, ALU and a MAC unit. This allows us to execute many operations simultaneously, thus reducing the clock cycle count.

The EVP\textsubscript{16} has many more features that allow us to save some clock cycles. It supports zero-overhead looping, allowing the repetitive execution of a number of instructions without spending clock cycles on updating the loop counter or checking its value. The shuffle unit also has built-in common patterns and some registers to hold custom patterns so that no time has to be spend constructing them or loading them from memory. The common patterns include those that shift or rotate a vector over any distance, and the patterns for the FFT butterflies.

The vector load/store unit of the EVP\textsubscript{16} also supports unaligned memory accesses, though this also occupies the shuffle unit. This effectively combines two memory access operations and two shift operations into a single memory access operations that occupies only two functional units of the vector processor.

Some other interesting features of the EVP\textsubscript{16} are its intra-vector unit and its code generation unit. We will not use these in the programs that follow, but it is worth mentioning them. The intra-vector unit supports vector reduction operations which aggregate the vector elements using a simple operation like the addition operator or the logical AND operator. The code generation unit is capable of generating 16 new code chips for a variety of codes (CRC, UMTS, CDMA-2000, GPS, etc.).

The EVP\textsubscript{16} is also interesting to us because it allows us to scale our program. The EVP\textsubscript{16} operates on vectors of 256 bits which are divided into $P = 16$ words of 16-bits. The processor also supports operations on $P = 8$ words of 32-
7.3. Implementing pipelines

In this section we discuss the implementation of a pipeline containing identical components on an SIMD processor. Pipeline structures are common all manner of circuits, but we have also seen a number of scalable block processing algorithms that rely on pipelines in Chapter 2. Particularly the block post-computation approach and the linear look-ahead computation make use of a pipelined series of components. In this section we first show how such a pipeline can be mapped onto a vector processor and then apply it by implementing an IIR filter based on the linear look-ahead approach onto the EVP.

7.3.1 Approach

Implementing a pipelined structure with identical components on an SIMD processor is relatively straightforward. The SIMD implementation is basically the emulation of the hardware components on the SIMD processor.

The algorithms in Chapter 2 start out with an unpipelined structure containing identical components, as shown in Figure 7.1(a). This structure allows the processing of $P$ inputs and outputs simultaneously. However, the critical path runs through $P$ components and thus depends on the block size. To reduce the critical path the structure is pipelined and we obtain Figure 7.1(b). The pipelining reduces the length of the critical path to a constant, since it now runs through only one component.

The pipelined structure in Figure 7.1(b) is implemented by mapping each component to a PE of the SIMD processor. Since all the components are identical, but operate on different data, this construction is ideal for mapping onto an SIMD processor. A sequential program for the function $f$ is executed on each PE by replacing all operations in the program by vector operations. In order to do this the processor needs a linear topology, so that it is able to shift the vector $w$, whose elements are produced at the end of each $f$ computation, to the next PE. Furthermore, the processor must support strided memory access operations to read a vector that contains the necessary in and to write the computed out.
The pseudo-code for the implementation is shown in Listing 7.1. Note that the code in Listing 3.1 in Chapter 3 was derived from the code in Listing 7.1.

The strided memory access operations are needed because PE \( j \) requires the input \( \text{in}(i - j)(j) = \text{in}((i - j)P + j) \) and produces the output \( \text{out}(i - j)(j) = \text{out}((i - j)P + j) \). So, assuming that the inputs and outputs occupy adjacent memory locations, the processor needs to support a strided memory access operation with a stride of \( -P + 1 \). This stride modulo \( P \) is 1, therefore an SIMD processor belonging to the M2 class supports this kind of operation.

It is possible that the number of components in the pipeline \( (L) \) does not match with the number of PEs \( (P) \) of the processor. In such cases the pipeline can be divided into sequences of \( P \) components and each sequence can be emulated in turn. Alternatively, if the number of components is a multiple of \( P \), the number of pipeline stages can be reduced so that each PE emulates \( I \) components. This last options results in \( L = IP \) and it requires unaligned strided memory access with a stride of \( -P + I \). We examine this option in our example in the next section.

### 7.3.2 Example: Infinite impulse response filter

In this section we summarize our paper [38], by implementing a infinite impulse response filter on the EVP processor, using the linear look-ahead approach.
\textbf{7.3. IMPLEMENTING PIPELINES} \hspace{1cm} 201

\begin{verbatim}
var P, i, wp : scalar;
input_vec, w_vec, output_vec : vector;
{[\text{P} = P \land [i] \in \text{N}}
{\text{Inv}_0 : [w\_vec](p) = w_p(i - p)}
\text{do true} \rightarrow
\text{input\_vec} := \text{strided\_read(IN,P*i,1-P)};
{[\text{input\_vec}(p) = \text{in}(Pi + p(1 - P))}
\text{w\_vec, output\_vec} := \text{f(input\_vec, w\_vec)};
\text{strided\_write(OUT,P*i,1-P,output\_vec)};
{[\text{w\_vec}(p) = w_{p+1}(i - p) \land \text{output\_vec} = \text{out}(Pi + p(1 - P))}
\text{w\_vec[j], wp} := \text{shift(w\_vec[j],-1,w_0(i + 1)), w\_vec[P-1]};
{[\text{w\_vec}(p) = w_p(i + 1 - p) \land \text{wp} = w_P(i - P + 1)}
i := i + 1;
\text{od}
\end{verbatim}

Listing 7.1: SIMD pseudo-code for the pipeline structure from Figure 7.1(b)
CHAPTER 7. IMPLEMENTATION

We base our approach on the incremental block state architecture by Parhi and Messerschmitt [67]. This architecture is based on a pipeline and, as explained in the previous section, we need strided memory access operations to implement such a pipeline on an SIMD processor. Since the EVP_{16} does not support strided memory accesses, we will be working with a modified, theoretical version of the EVP that does.

The incremental block state architecture is based on the state space form of the IIR filter, discussed in Section 3.3 for the moving sum. The state space form of an \( N \)-th order linear filter like an IIR filter consists of two equations:

\[
\begin{align*}
s(i + 1) &= A s(i) + b \text{in}(i) \\
\text{out}(i) &= c^T s(i) + d \text{in}(i)
\end{align*}
\]

(7.11) (7.12)

where \( A \) is a \( N \times N \) matrix, \( b \) and \( c \) are vectors of length \( N \) and \( d \) is a scalar.

As explained in Section 3.2.1 there are many combinations of \( A, b, c \) and \( d \) that describe the same filter. This allows us to choose the representation such that \( A \) is in block diagonal form by splitting the filter into parallel first and second order sections [48] and then combining those into a single filter [54]. This means that matrix \( A \), and any matrix \( A^i \) for \( i \in \mathbb{N} \), has the form depicted in Figure 7.2. We use \( N_2 \) to denote the number of second order sections, i.e., the number of blocks of size \( 2 \times 2 \) and \( N_1 \) to denote the number of first order sections, i.e., the number of blocks of size \( 1 \times 1 \) in matrix \( A \). Note that \( N = 2N_2 + N_1 \).

The linear look-ahead computation for an IIR filter in state-space form is
7.3. IMPLEMENTING PIPELINES

represented using the following matrices:

\[
\begin{align*}
A(L) & = A^L \\
B(L) & = (A^L b \quad A^{L-1} b \ldots b) \\
C(L) & = \begin{pmatrix}
c^T \\
\vdots \\
c^T A^{L-1} \\
c^T A^L
\end{pmatrix} \\
D(L)_{i,j} & = \begin{cases} 
0 & \text{if } i < j \\
d & \text{if } i = j \\
c^T A^{i-j-1} b & \text{if } i > j
\end{cases}
\end{align*}
\]

where matrix \( D(L) \) is a \( L \times L \) matrix. The sizes of the other matrices follow from their definitions: matrix \( A(L) \) is \( N \times N \), \( B(L) \) is \( N \times L \) and \( C(L) \) is \( L \times N \).

Using these equations the block processing algorithm for the filter is represented by the equations:

\[
\begin{align*}
s((i+1)L) & = A(L)s(iL) + B(L)\text{in}(i) \\
\text{out}(i) & = C(L)s(i) + D(L)\text{in}(i)
\end{align*}
\]

When these equations are implemented directly we obtain the block state architecture \[54, 72\]. However this implementation does not scale, since the number of elements in matrix \( D \) grows quadratically with the block size.

The trick to obtaining a scalable block processing algorithm, as explained in Chapter 2 and [67], is to compute the outputs in an incremental fashion. So we use the incremental block state architecture, as depicted in Figure 7.3(a).

Note that \( \text{in}_I \) is used to denote the input stream that has been divided into blocks of size \( I \), instead of into blocks of size \( L \). Furthermore, the inputs of the matrix-vector multiplication components (Figure 7.3(c)) contain the zero vector, unless Figure 7.3(a) indicates otherwise.

The incremental block state architecture can be pipelined as shown in Figure 7.3(b). Although the first pipeline stage differs from the others we can still map this pipeline onto an SIMD processor. First of all the \( A(L) \) matrix is different from all the \( A(I) \) matrices in the other stages. However, all these matrices share the same size and block diagonal structure, so all the SIMD processor has to do is execute the exact same matrix-vector multiplication on each PE, but with the first PE using a different matrix. The \( B(L) \) forms a bigger challenge, since its size differs from the \( B(I) \) matrices.
CHAPTER 7. IMPLEMENTATION

Figure 7.3: Incremental block state architecture for $L = 4I$
7.3. IMPLEMENTING PIPELINES

<table>
<thead>
<tr>
<th>Subcomputation</th>
<th>Number of instructions</th>
<th>Memory access</th>
<th>MAC</th>
<th>Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication with $A^{(L)}$ and $A^{(I)}$</td>
<td></td>
<td>$4N_2 + N_1$</td>
<td>$4N_2 + N_1$</td>
<td>$N$</td>
</tr>
<tr>
<td>Multiplication with $B^{(L)}$</td>
<td></td>
<td>$IN + I$</td>
<td>$IN$</td>
<td>$N$</td>
</tr>
<tr>
<td>Multiplication with $B^{(I)}$</td>
<td></td>
<td>$IN + I$</td>
<td>$IN$</td>
<td>$I$</td>
</tr>
<tr>
<td>Multiplication with $C^{(I)}$</td>
<td></td>
<td>$IN$</td>
<td>$IN$</td>
<td>$0$</td>
</tr>
<tr>
<td>Multiplication with $D^{(I)}$</td>
<td></td>
<td>$I(I+1)$</td>
<td>$\frac{1}{2}I(I+1)$</td>
<td>$0$</td>
</tr>
<tr>
<td>Store result</td>
<td></td>
<td>$1$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

Table 7.1: Instruction count for the incremental block-state implementation

We solve this by computing $B(L)\text{in}(i)$ separately and by replacing the $B(L)$ in Figure 7.3(b) as shown in Figure 7.3(d). After this replacement all pipeline stages perform a multiplication with $B(I)$. The only difference is that the top input of the matrix-vector multiplication component for $B(I)$ is the zero vector for all pipeline stages, except the first.

The separate computation of $B(L)\text{in}(i)$ can be done using another pipeline structure, using the approach described in Section 7.3.1. Alternatively we can make use of the vector reduction operations of the EVP to implement the computation. We only consider the pipeline structure, however, since we have shown in [38] that the difference in terms of throughput between the two versions is small and in most cases the pipelined computation has a better throughput.

When we implement this design onto an SIMD processor according to the method discussed in Section 7.3.1 we obtain the instruction count shown in Table 7.1 per iteration. The number of pipeline stages is $P = \frac{L}{I}$, so by choosing $I = 1$ we obtain an implementation with $P = L$ and an instruction count of $4N_2 + 10N + 7$. However, another choice for $I$ results in a different block size, namely $L = IP$ and a different total instruction count per iteration:

$$\frac{3}{2}I^2 + 6IN + \frac{9}{2}I + 4N + 4N_2 + 1$$  \quad (7.19)

The throughput of the implementation is in the order of $L$ divided by this instruction count. The optimum throughput should therefore be obtained by choosing $I$ such that:

$$I = \sqrt{\frac{2 + 8N + 8N_2}{3}}$$  \quad (7.20)

This theoretical optimum, however, does not correspond to the practical optimum, since we have not taken everything into account, most notably we did not


Table 7.2: Performance of the IIR program on an EVP equipped with strided memory access

<table>
<thead>
<tr>
<th>( N = 2, N_2 = 1 )</th>
<th>( N = 4, N_2 = 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I = 1 )</td>
<td>( I = 1 )</td>
</tr>
<tr>
<td>( I = 2 )</td>
<td>( I = 2 )</td>
</tr>
<tr>
<td>( I = 3 )</td>
<td>( I = 3 )</td>
</tr>
<tr>
<td>( I = 4 )</td>
<td>( I = 4 )</td>
</tr>
<tr>
<td>( L )</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>64</td>
</tr>
<tr>
<td>( T )</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>1.45</td>
</tr>
<tr>
<td></td>
<td>1.30</td>
</tr>
<tr>
<td></td>
<td>0.57</td>
</tr>
<tr>
<td>( S )</td>
<td>6.67</td>
</tr>
<tr>
<td></td>
<td>7.27</td>
</tr>
<tr>
<td></td>
<td>6.49</td>
</tr>
<tr>
<td></td>
<td>2.86</td>
</tr>
<tr>
<td></td>
<td>2.57</td>
</tr>
<tr>
<td></td>
<td>2.97</td>
</tr>
<tr>
<td></td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td>2.98</td>
</tr>
</tbody>
</table>

Consider the VLIW parallelism of the EVP processor.

We therefore implemented the incremental block-state architecture on a theoretical EVP equipped with strided memory access operations, resulting in Table 7.2. The row marked “S” contains the speedup of our implementation compared to a theoretical SISD processor that requires \( 2N + 1 \) clock cycles per output, namely \( 2N + 1 \) MAC operations to compute the output. Ideally the speedup would equal \( P = 16 \), unfortunately this is not the case since the incremental block state architecture introduces some overhead. Most notably, the architecture consists of two pipelines and each state is computed (almost) once in each pipeline, thereby reducing the maximum speedup by 50%. Furthermore, the EVP suffers from register pressure, which means that as more registers are needed the compiler is forced to insert extra memory access operations to swap registers to memory. This is especially clear for the second order filter (\( N = 2 \)): for smaller \( I \) there are enough registers available but for \( I = 4 \) the compiler is forced to swap registers to memory.

7.3.3 Discussion

The previous two sections show how a pipeline structure can be mapped onto an SIMD processor that supports strided memory access. In particular the program requires strided memory access with a stride of \(-P + I\), where the starting addresses in global memory modulo \( P \) is at least 0 and less than \( I \).

So, if we choose \( I = 1 \) the algorithm can be mapped onto any M2 class processor. Furthermore, if the stride \(-P + I\) is relatively prime with \( P \) the algorithm can also be mapped onto a M2 class processor, provided the processor also has a completely connected topology: the stride and the number of PEs are relatively prime, so every strided memory access operations accesses exactly one memory bank per PE, and after such a memory access the completely connected
topology is used to route each data item to or from these memory banks. In all other cases we need a processor that truly supports strided memory access, i.e., an M3 class processor.

A pipeline of identical components is a common part of the existing scalable block processing algorithms, like the block post-computation approach and the linear look-ahead approach. Therefore it is important to be able to implement them on an SIMD processor. In this section we have shown that by choosing \( I = 1 \), and therefore \( L = P \), we can implement the pipeline on any M2 class processor. Unfortunately processors of the M1 class are more common than those of the M2 class.

We have also shown that the number of PEs \( P \) does not necessarily have to match up with the block size \( L \). It is also possible to choose a block size that is a multiple of the number of PEs, i.e., \( L = IP \). For the pipeline program this leads to extra requirements on the SIMD processor, but disconnecting the block size from the number of PEs can also be the key to solving other implementation problems, as shown in the remainder of this chapter.

## 7.4 Implementing generator graphs

In this section we describe the implementation in VLSI and on an SIMD processor of an algorithm represented by a generator graph, but we cover the general implementation method only. When an algorithm for look-ahead computation is based on a generator graph there are some specific issues which we discuss in Section 7.5. As an example of the implementation method we implement a few 2-dimensional median filters on the EVP\textsubscript{16} processor.

### 7.4.1 Approach

In this section we first examine the implementation of a generator graph as a VLSI circuit, as we have done in [39]. After that we examine a method of mapping this VLSI implementation on an SIMD processor, as we described in [40].

#### 7.4.1.1 VLSI

To implement an algorithm described by a generator graph as a VLSI circuit we first construct a component that has the same structure internally as the
CHAPTER 7. IMPLEMENTATION

generator graph. One such component corresponds to a block processing algorithm, and multiple of such components can be connected together to form a block processing algorithm with a larger block size.

Figure 7.4 shows an abstract component based on some generator graph. As the arrows in the figure indicate, this component is reads the inputs of the system, produces the outputs of the system and is connected to some buffers. The buffer on the top stores tiles that have previously been calculated, but are still needed. The buffer on the bottom is updated by the component so that it stores those tiles that are needed in a next iteration or component of the block processing algorithm.

The internal structure of a component like the one in Figure 7.4 is derived from a generator graph. Figure 7.5(b) shows, for example, the internal structure of the component that has been derived from the canonical generator graph in Figure 7.5(a). This derivation can be done automatically with the tools presented in Appendix B.

As an example we discuss the derivation of Figure 7.5(b) from Figure 7.5(a) here. First the tiles of each node are named, as shown in Figure 7.5(a). These names are used to obtain the following set of equations:

\[
\begin{align*}
\langle B_{2i+1} \rangle &= \langle A_{2i+1} \rangle \odot \langle A_{2i+2} \rangle \\
\langle W_{2i} \rangle &= \langle A_{2i} \rangle \odot \langle B_{2i+1} \rangle \\
\langle W_{2i+1} \rangle &= \langle B_{2i+1} \rangle \odot \langle A_{2(i+1)+1} \rangle
\end{align*}
\]

Each equation corresponds to a node with tiles larger than 1, and the right hand side of an equation is determined by the incoming edges of that node. Note that the tile-sets containing tiles of the same shape are represented by the same letter.
7.4. IMPLEMENTING GENERATOR GRAPHS

(a) The generator graph

(b) The component based on the generator graph

Figure 7.5: A VLSI component for $W = ((3), 1, 0)$
and that the subscripts of the letters are of the form $a(i + b) + c$, where $a$ equals the meta-period and $c$ the phase of the tile-set. On the left-hand side of the equations we have $b = 0$ and on the right-hand side the value of $b$ depends on the way the tiles fit together to form the tile on the left hand side. Each equation corresponds to a $\odot$ operator in Figure 7.5(b), and the incoming connections of the operator are specified by the equation. This process of deriving a component from a generator graph takes some work, but can be done automatically.

Once the internal structure of the component from Figure 7.4 has been designed, it is used to design a block processing algorithm. It can be used in an implementation that calculates $I$ outputs in parallel, where $I$ is the meta-period ($m$) divided by the period of the windows ($p_w$). Such an implementation is obtained by connecting the component to a buffer, as depicted in Figure 7.6(a).

Figure 7.6: A block processing circuit based on a generator graph
7.4. IMPLEMENTING GENERATOR GRAPHS

<table>
<thead>
<tr>
<th>VLSI (unpipelined)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency $L$</td>
</tr>
<tr>
<td>Memory $M$</td>
</tr>
<tr>
<td>Complexity $C$</td>
</tr>
<tr>
<td>Iteration period $I$</td>
</tr>
<tr>
<td>Throughput $T$</td>
</tr>
<tr>
<td>Scalability</td>
</tr>
<tr>
<td>Remarks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VLSI (pipelined)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency $L$</td>
</tr>
<tr>
<td>Memory $M$</td>
</tr>
<tr>
<td>Complexity $C$</td>
</tr>
<tr>
<td>Iteration period $I$</td>
</tr>
<tr>
<td>Throughput $T$</td>
</tr>
<tr>
<td>Scalability</td>
</tr>
<tr>
<td>Remarks</td>
</tr>
</tbody>
</table>

Table 7.3: Characteristics of a generator graph based algorithm

To obtain a VLSI circuit that calculates $L = IP$ windows in parallel we use $P$ components and connect them in a similar manner. For $P = 4$ this has been done in Figure 7.6(b). Any number of components can be connected in this way to increase the block size $L$ of the circuit.

This leads to a VLSI circuit with the characteristics shown in Table 7.3. The characteristics of the unpipedined implementation are relatively straightforward. It takes one iteration to compute a block of outputs, therefore the latency is one block. The only memory used by the implementation is the buffer shown in Figure 7.6. Its exact size depends on the generator graph, but it is in the same order as $W$. The complexity of the algorithm is $P$ times the complexity of a single component, and from Chapter 5 we know that the complexity of the best generator graphs require $\Theta(1)$ operations per output. Therefore the complexity of the entire algorithm is $\Theta(P I) = \Theta(L)$ operations per block. The iteration period of the circuit depends on the depth of the circuit. This depth corresponds to the depth of the generator graph, and since the best generator graphs have a depth of $\Theta(\log W)$, this is also the iteration period of the algorithm. All this leads to a throughput of $\Theta(\frac{L}{\log W})$. Since the amount of hardware is linear in $L$, this means that the algorithm has linear scalability. Its only drawback is that we cannot choose the block size freely, but that it must be a multiple of $I = \frac{m}{P_N}$. 
The second part of Table 7.3 shows the characteristics obtained by pipelining the circuit. Pipelining is possible because the circuit is essentially a feed forward network, despite the apparent cycle in Figure 7.6. The smaller tiles are fed into the rest of the circuit to calculate larger tiles, which allows us to pipeline it. The pipelining requires a number of cut-sets that is $\Theta(\log W)$, i.e. equal to the depth of the generator graph, and each cut-set intersects with at most $L$ data-paths. This pipelining results in the characteristics shown in the second part of Table 7.3. In this case we obtain a throughput that is $T = O(L)$, i.e., not only does the algorithm scale linearly, but also its throughput no longer depends on the window size $W$.

### 7.4.1.2 SIMD

The VLSI implementation form the previous section also serves as a basis for implementations on an SIMD processor. An SIMD processor has more restrictions on possible operations than hardware does, but it is possible to emulate $P$ of the VLSI components on an SIMD processor with $P$ processing elements. In this section we discuss this approach.

The implementation on an SIMD processor starts by first writing a sequential program that emulates a single VLSI component like the one in Figure 7.4. The sequential program is then turned into an SIMD program by, basically, replacing all the operations by vector operations, resulting an algorithm that functions like the VLSI circuit in Figure 7.6. The conversion from an sequential program to SIMD program is tedious, but once the basic $\odot$ operation has been specified the rest of the procedure is fully automated, using the tools described in Appendix B.

In Figure 7.6 the components communicate tiles from one component to the next. For an SIMD program this corresponds to a shift operation, where data elements are shifted from one PE to the next. Note that during such a shift operation the data received by PE 0 has to come from a buffer and the element send by PE $P - 1$ has to be placed in that buffer.

The reading of the inputs and the writing of the outputs requires some attention. All the tile-sets in the generator graph have a period equal to the meta-period, so the implementation requires the inputs and produces the outputs of the system with that same period. Consider our example; the sequential program for the component from Figure 7.5 computes, say, first the even and then the odd window. The SIMD version of the program then computes first $P$ even windows and then $P$ odd windows. The same reasoning holds for the reading of the inputs. We therefore require strided memory accesses with a stride
equal to the meta-period to implement our algorithm, as shown in Listing 7.2.

This program results in a system with the characteristics shown in the first part of Table 7.4. The performance characteristics are obtained in the same manner as those for the VLSI algorithm. The SIMD algorithm requires more memory because it not only needs the buffer, but also vector registers to store intermediate results. The exact memory usage will depend on the generator graph used, but in it is bounded by $O(W + L)$. The iteration period of the algorithm is $\Theta(I)$, since the $\Theta(L)$ operations per block are mapped onto $P$ PEs and $\Theta(I)$ time slots ($L = IP$). This results in a throughput of $\Theta(P)$ and therefore in an algorithm with linear scalability.

This algorithm, however, requires strided memory access. Since not many SIMD processors support strided memory access, we present another solution to this problem. This solution avoids strided memory access operations, provided that the processor is capable of the shuffle operation.

Instead of using strided memory accesses the inputs are read in consecutive order and we use several (masked) shuffling operations to distribute them to the correct PEs. This process is depicted in Figure 7.7 for an SIMD processor.

<table>
<thead>
<tr>
<th>Table 7.4: Characteristics of a generator graph based algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SIMD (strided memory access)</strong></td>
</tr>
<tr>
<td>Nr. of PEs</td>
</tr>
<tr>
<td>Latency</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>Complexity</td>
</tr>
<tr>
<td>Iteration period</td>
</tr>
<tr>
<td>Throughput</td>
</tr>
<tr>
<td>Scalability</td>
</tr>
<tr>
<td>Remarks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>SIMD (no strided memory access)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr. of PEs</td>
</tr>
<tr>
<td>Latency</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>Complexity</td>
</tr>
<tr>
<td>Iteration period</td>
</tr>
<tr>
<td>Throughput</td>
</tr>
<tr>
<td>Scalability</td>
</tr>
<tr>
<td>Remarks</td>
</tr>
</tbody>
</table>
var P, i : scalar;
  buf0, buf1 : scalar;
  a2_vec, a3_vec, b_vec : vector;
  t0_vec, t1_vec : vector;
  out0_vec, out1_vec : vector;

{[P] = P ∧ [i] ∈ N}
{Inv0 : [buf0] = (A2i)}
{Inv1 : [buf1] = (A2i+1)}

do true →
a2_vec := strided_read (IN , 2*i+2 , 2);
{[a2_vec](p) = in(2i + 2 + 2p) = (A2(i+p)+2)}
a3_vec := strided_read (IN , 2*i+3 , 2);
{[a3_vec](p) = in(2i + 3 + 2p) = (A2(i+p)+3)}
t1_vec, buf1 := shift (a3_vec , -1 , buf1) , a3_vec [P -1];
{[t1_vec](p) = (A2(i+p)+1) ∧ [buf1] = (A2(i+p)+1)}
b_vec := t1_vec ⊙ a2_vec;
{[b_vec](p) = B2(i+p)+1 = A2(i+p)+1 ⊙ A2(i+p)+2}
t0_vec, buf0 := shift (a2_vec , -1 , buf0) , a2_vec [P -1];
{[t0_vec](p) = A2(i+p) ∧ [buf0] = A2(i+p)}
out0_vec := t0_vec ⊙ b_vec;
strided_write (OUT , 2*i , 2 , out0_vec);
{[out0_vec](p) = A2(i+p) ⊙ B2(i+p)+1 = W2(i+p) = out(2(i + p))}
out1_vec := b_vec ⊙ a3_vec;
strided_write (OUT , 2*i+1 , 2 , out1_vec);
{[out1_vec](p) = B2(i+p)+1 ◦ A2(i+p)+3=W2(i+p)+1=out(2(i+p)+1)}
i := i + P;

Listing 7.2: SIMD pseudo-code for the example from Figure 7.5
7.4. IMPLEMENTING GENERATOR GRAPHS

Figure 7.7: Shuffling consecutive inputs for $P = 4$ and a $I = 2$

with 4 PEs. The top of the figure shows the inputs as they read from memory, at the bottom it shows how two vectors have been created, both containing the inputs with a period of two, but the left-most vector with a phase of 0, and the right-most vector with a phase of 1. The numbers at the arrows indicate the number of the shuffle operation in which the movement of the input to the destination vector is accomplished. The figure shows that 4 shuffle operations are needed. An inverted shuffling pattern can be used to get the outputs back into a consecutive order.

Note that the number of shuffling operations depends only on $I$, not on the number of processing elements ($P$). The program constructs $I$ vectors and each vector requires $I$ shuffle operations to construct. Therefore, although the iteration period of the SIMD program now contains a factor $I^2$, it is still independent of $P$ and we end up with a the throughput of $T \in \Theta(\frac{F}{P})$ as shown in the second part of Table 7.4.

7.4.2 Example: Rank order filtering

In this section we discuss the implementation of a generator graph on the EVP$_{16}$ processor. Specifically, we discuss our implementation of a rank order filter, which can also be found in our technical report [40].

Before we present the EVP$_{16}$ implementation, however, we first note that there are two options for implementing a compare and swap (C&S) operator on the EVP$_{16}$.

A C&S operator requires less hardware than two separate extrema operators when implemented in VLSI. Therefore hardware can be saved by replacing all
pairs of extrema operators involved in a C&S operation by a single C&S operator when implementing the design as a VLSI circuit. In a similar manner an SIMD processor can be capable of extrema operations, but also of C&S operations. If this is the case the C&S operations should be used in all places in the program where a pair of extrema operations would otherwise be used to implement a C&S operation.

The EVP_16 instruction set does not contain C&S operations, but it does allow us to program a C&S operation in different ways. Two options are shown in Table 7.5. Option 1 implements a C&S operation by using extrema operations, while option two compares the two values and uses the result as a mask for some assignment operations.

Option 1 requires less instructions, and by using dead code elimination any unused extrema operations can be automatically removed from the program. Option 2, however, might be faster than simple extrema operations since the EVP_16 processor also provides VLIW parallelism.

If we look at the VLIW parallelism of the EVP_16 processor in detail, we see that there is only one unit that supports extrema operations, while assignment or move operations are supported on 4 units. This means that the alternative implementation of the C&S operation, i.e. option 2 from Table 7.5 is possible and might even be an improvement in some cases.

Option 1 would take two clock cycles (one for the first two assignments, and one for the final assignment). Option 2 would also take two clock cycles (one for the compare and move operation and the other for the two remaining masked move operations). So there seems to be no gain. When using option 2, however, we can start a new C&S operation in each clock cycle. Assuming, of course, that the second of the subsequent C&S operations does not require the outputs of the first as input. This is something to take into account.

Another thing we must take into account is that the compiler also needs to

---

1 VALU
2 VLSU, VMAC, VALU and VSHU
7.4. IMPLEMENTING GENERATOR GRAPHS

<table>
<thead>
<tr>
<th>Window:</th>
<th>3 × 3</th>
<th>5 × 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock cycles per:</td>
<td>iteration</td>
<td>output</td>
</tr>
<tr>
<td>Using only option 1</td>
<td>84</td>
<td>1.31</td>
</tr>
<tr>
<td>Using only option 2</td>
<td>77</td>
<td>1.20</td>
</tr>
<tr>
<td>After optimization</td>
<td>71</td>
<td>1.11</td>
</tr>
</tbody>
</table>

Table 7.6: Clock cycles for median filtering on the EVP

schedule other operations, like the memory accesses and the shuffling of vectors. These operations can only be performed on the same units that are used by option 2. So producing the most efficient program code is an entirely new optimization problem.

We have not examined this problem extensively, because with an algorithm with (for example) \( N \) C&S operations, and two options for implementing each C&S operation, results in \( 2^N \) possible implementations. It is impractical to examine all these possibilities and select the best one.

To find a good program we use two steps. The first step is to find all the C&S operations of which one of the extrema operations can be pruned. For these C&S operations the best implementation is option 1, since that implementation reduces to a single extrema operation and thus requires only one clock cycle on the ALU unit.

After this first step there may still be C&S operations left for which we can choose between the two alternative implementations. So in the second step we use a simple heuristic algorithm to find good choices for these remaining C&S operations. The choices for these operations can be represented as a bit vector. We start by using the bit vector representing that all C&S operations are implemented as option 2. We then examine the neighborhood of the current configuration for a new configuration that is better, i.e., one that results in a lower iteration period. The neighborhood of a configuration consists of all configurations with a Hamming distance of one to the current configuration, i.e., all the configurations that have a different option for a single one of the C&S operations. The best configuration in this neighborhood is taken and we apply the same algorithm again and again, until there is no more improvement. This optimization was performed using the tools described in Appendix B.1 and the performance of the resulting implementations is shown in Table 7.6. Solving this optimization problem with a more sophisticated algorithm, however, may improve these results.

Table 7.6 presents the information on two 2-dimensional median filters im-
implemented on the EVP$_{16}$ processor. The implementations produce $L = IP$ outputs per iteration. The number of windows per component happens to be $I = 4$ for both problems. Since the processor has $P = 16$ PEs, the number of clock cycles per output is the number of clock cycles per iteration divided by $L = IP = 64$. The rows of the table indicate the number of clock cycles for different implementation strategies. The row marked “Using only option 1” and “Using only option 2” shows the results when step 2 of our optimization is skipped and all remaining C&S operations are implemented using option 1 or option 2 respectively. The row marked “After optimization” shows the results when the neighborhood search algorithm is used to optimize the implementation.

Note that the $3 \times 3$ and $5 \times 5$ median filters require 1.15 and 6.6 clock cycles per output respectively on the AltiVec, according to [46]. Our implementation on the EVP$_{16}$ with 1.11 and 3.77 clock cycles respectively is an improvement on those numbers, especially for the latter filter.

Table 7.6 also shows that median filters can be implemented very efficiently. There are algorithms that have a lower asymptotic complexity than our solutions [44, 81], but our approach using oblivious sorting allows for easy parallelism. For practical window sizes this parallelism compensates small differences in complexity and provides better overall performance.

### 7.4.3 Discussion

In the previous sections we have shown that a generator graph can be used to implement a scalable block processing algorithm on an SIMD processor. The drawbacks are that the SIMD processor needs to support either strided memory access or have a completely connected topology.

The EVP$_{16}$ does not support strided memory access, but with its completely connected topology it produces good results, as shown in Table 7.6. These results will improve significantly if the EVP$_{16}$ is ever extended with strided memory access capabilities, because multiple shuffle operations could then be replaced by a single strided memory access operation.

The stride used in the strided memory access operations is $I$, which we have chosen to be $m_{P_W}$. We could also choose $I$ to be a multiple of $m_{P_W}$, effectively emulating multiple of the VLSI components on each PE of the SIMD processor. In this way we might be able to get an $I$ that is 1 modulo $P$, so that we can run our algorithm on a M2 class processor. This is far from a general solution, however, since it only works for specific systems. The period of the windows ($p_W$) is system specific, and the generator graph depends on both the system
parameters and the algorithm used to construct it.

So, in general, we need an SIMD processor that supports the shuffle operation, i.e., one that has a completely connected topology or an SIMD processor that supports strided memory access. In the next section we show, however, how we can implement a scalable block processing algorithm, using only the PSNN shuffle patterns, for systems where $W$ is a power of two and the maximum reuse algorithm has been used to construct a generator graph. Furthermore, these shuffle patterns are used in such a manner that they do not increase the iteration period by $I^2$, but only by $I$.

## 7.5 Look-ahead computations

The previous section demonstrated how a generator graph can be turned into an implementation. Section 4.5.1 shows us that a look-ahead computation can be implemented by designing an algorithm for an associative window computation with $W = (\langle L \rangle, 0, 1)$. So it is only logical to combine the two to obtain an implementation of a look-ahead computation. This works for the VLSI algorithms, but we encounter some difficulties SIMD algorithm, which we address in this section.

The problem is that the window size, which we assumed to be a constant part of the system specification, is now linked to the block size. This has repercussions for the algorithm, since the look-ahead computation based on $W = (\langle L \rangle, 0, 1)$ results in $I = \frac{m}{P_W} = m$ and $m \in O(W)$. In fact, all the algorithms we considered that construct a generator graph with a constant cost per output have $m \in \Theta(W)$. For the SIMD implementation presented in the previous section we have $L = IP$, which leads to $L \in \Theta(L)P$. This is a problem because this leaves only a single choice for $P$. This destroys the scalability of the algorithm, since we can no longer use more hardware, i.e. an SIMD processor with a larger $P$, to increase the throughput.

In this section we show how this problem can be solved. First we solve it using strided memory access and then we show how it can be solved using the shuffle operation. Furthermore, we demonstrate the approach using the shuffle operation by implementing an IIR filter on the EVP$_{16}$ processor.

### 7.5.1 Approach using strided memory access

The look-ahead computation, where the block size and the window size are linked, can be implemented on an SIMD processor using strided memory access.
In this section we demonstrate this approach.

We split the look-ahead computation from Section 4.5.1 into two block processing algorithms connected in series. We run these algorithms on the same SIMD processor, but use a different block size for each algorithm. The first algorithm performs the window computation with \( W = (\langle L' \rangle, 0, 1) \) and has a block size of \( L \). The second algorithm has a block size of \( L' \) and uses the results of the first algorithm to simultaneously update \( L' \) states to obtain \( L' \) new states.

Ideally we have \( L = L' \), so that no buffering has to take place between the two block processing algorithms. We have already demonstrated, however, that this is not an option since it destroys the scalability of the first algorithm.

So we focus on the block size \( L' \) of the second algorithm. The algorithm is not very complicated and the most logical choice for the block size is \( L' = P \), resulting in a system with an iteration period of \( I \in \Theta(1) \). That is, the second block processing algorithm maintains \( P \) states and updates them in parallel in each iteration, requiring \( \Theta(1) \) time.

This construction of the second algorithm means that the first algorithm has to perform a window computation with \( W = (\langle P \rangle, 0, 1) \). Therefore we get \( I = \frac{m}{P_W} = m \in \Theta(P) \) and a block size of \( L \in \Theta(P) P \in \Theta(P^2) \).

We then run these two algorithms on the same SIMD processor and buffer the outputs of the first algorithm in a buffer of size \( \Theta(P^2) \). One iteration of the first algorithm is then followed by \( \Theta(P) \) iterations of the second algorithm.

The generator graphs produced by the van Herk-Gil-Werman algorithm and the maximum reuse algorithm are well suited for this construction. This is because these algorithms produce a generator graph with \( m = P \), provided that \( W = P \) is a power of two\(^3\). Therefore the block size of the first algorithm is \( L = P^2 \), the buffer between the algorithm is exactly \( L \) and one iteration of the first algorithm is followed by exactly \( P \) iterations of the second algorithm.

The combination of the two algorithms based on a generator graph with \( m = W = P \) therefore results in algorithms with the characteristics shown in Table 7.7. Note that the table basically contains the characteristics from Table 7.4 with \( I = P \). This accounts for the characteristics of both the first and the second algorithm, since the second algorithm does not contribute much to the performance characteristics in the \( \Theta \) notation. The latency introduced by the second algorithm is one block of outputs, i.e., \( L \) outputs. It only stores a block of \( L \) states, which is the only contribution it makes to the memory requirements. The extra complexity per output is 1 single state update, i.e.,

\(^3\)For the maximum reuse algorithm that is. The van Herk-Gil-Werman algorithm always produces generator graphs with \( m = W \)
### Table 7.7: Characteristics of a generator graph based look-ahead computation

<table>
<thead>
<tr>
<th></th>
<th>SIMD (strided memory access)</th>
<th>SIMD (no strided memory access)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr. of PEs</td>
<td>$P$</td>
<td>$P$</td>
</tr>
<tr>
<td>Latency</td>
<td>$\mathcal{L}$ $\Theta(L)$</td>
<td>$\mathcal{L}$ $\Theta(L)$</td>
</tr>
<tr>
<td>Memory</td>
<td>$\mathcal{M}$ $\mathcal{O}(L)$</td>
<td>$\mathcal{M}$ $\mathcal{O}(L)$</td>
</tr>
<tr>
<td>Complexity</td>
<td>$\mathcal{C}$ $\Theta(L)$</td>
<td>$\mathcal{C}$ $\Theta(PL)$</td>
</tr>
<tr>
<td>Iteration period</td>
<td>$\mathcal{T}$ $\Theta(P)$</td>
<td>$\mathcal{T}$ $\Theta(P^2)$</td>
</tr>
<tr>
<td>Throughput</td>
<td>$\mathcal{T}$ $\Theta(P)$</td>
<td>$\mathcal{T}$ $\Theta(1)$</td>
</tr>
<tr>
<td>Scalability</td>
<td>linear</td>
<td>none</td>
</tr>
<tr>
<td>Remarks</td>
<td>Requires strided memory access</td>
<td>Requires the shuffle operation</td>
</tr>
</tbody>
</table>
\(\Theta(1)\). Furthermore, the second algorithm has an iteration period of \(\Theta(1)\) which it needs to execute \(P\) times per iteration of the combined algorithm, therefore its contribution to the total iteration time is \(\Theta(P)\).

The stride needed in the strided memory access operations is equal to the meta-period of the generator graph. In this respect the generator graphs of the van Herk-Gil-Werman and the maximum reuse algorithm are not well suited for this implementation. The meta-period of those graphs is \(m = P\), which means that each strided memory access operation accesses the memory bank of the same PE \(P\) times. The algorithm has to make \(P\) of such accesses in each iteration, i.e., the memory bank of each PE gets a turn. These memory accesses actually correspond to reading, column by column, a \(P \times P\) matrix stored in row-major order in memory. So, instead of strided memory access we could also use a \(P \times P\) matrix transposition operation followed by ordinary memory accesses. Unfortunately, matrix transposition is not a \(\Theta(1)\) operation, except on a M3 class SIMD processor. Therefore we require a M3 class SIMD processor to implement this algorithm.

In this respect the generator graphs produced by Harrington’s algorithm are more suited for this implementation. Harrington’s algorithm results in a meta-period and increment of \(m = I = P + 1\). So, the stride used in the strided memory access operations is relatively prime with \(P\), which means that we can implement the algorithm a M2 class processor. The drawback of this is that the block size \(L = IP = P^2 + P\), which means that the buffering between the two sub-algorithms becomes more complicated.

So, the algorithm presented here is a good solution for SIMD processors that support strided memory access operations or \(P \times P\) matrix transpositions, i.e., SIMD processors of the M3 class. For other SIMD processors, however, we need to refine the algorithm before we can get an implementation. For such SIMD processors the approach presented here is only the first step towards implementing a scalable look-ahead computation. The other steps are discussed in the next section.

### 7.5.2 Approach using shuffle operations

The solution from the previous section, i.e., splitting the algorithms into two parts, is a first step towards implementing a look-ahead computation on an SIMD processor that does not support strided memory accesses, but does support shuffle operations. In Section 7.4 we used shuffle operations to, basically, emulate the strided memory access. In the previous section, however, we have shown that this destroys the scalability of the algorithm for look-ahead compu-
7.5. **LOOK-AHEAD COMPUTATIONS**

In this section we show how we can actually reduce the number of shuffle operations to the point that the scalability of the algorithm is restored.

The method we present here would work best when the number of processing elements $P$ of the SIMD processor is a power of two minus one, i.e., $P = 2^p - 1$ for some $p \in \mathbb{N}^+$. Practically all SIMD processors in existence, however, have a number of PEs that is exactly a power of two, i.e., $P = 2^p$ for some $p \in \mathbb{N}^+$. Therefore we adapt our method to this latter case.

Just like in the last section we design an algorithm for $L = P^2$, but this time the algorithm consists of three sub-algorithms. The first sub-algorithm performs a look-ahead computation using an associative window computation with $W = \langle(P - 1), 0, 1 \rangle$. The second sub-algorithm uses the results of the first to obtain the results of the look-ahead computation with $W = \langle(P), 0, 1 \rangle$. So the second algorithm takes the result of the first algorithm, which are aggregations over a window of size $P - 1$ and uses the aggregation operator $\circ$ to add an extra input to obtain the aggregation over a window of size $P$. The third and last sub-algorithm then performs the actual look-ahead computation by updating $P$ old states, using the results of the second sub-algorithm, to $P$ new states.

Note that the complexity per output for the second and third sub-algorithm is $\Theta(1)$, that the iteration time of both is $\Theta(P)$ per block of $L = P^2$ outputs and that neither of them requires shuffle nor strided memory access operations. Therefore, to obtain a scalable algorithm, we only need to design the first sub-algorithm in such a manner that it has a complexity per output of $\Theta(1)$ and an iteration time of $\Theta(P)$. We have to do this in such a manner that the algorithm uses no strided memory accesses, but we can make use of shuffle operations.

We accomplish this by basing our algorithm on the generator graph produced by the maximum reuse algorithm of Section 5.5.2. This algorithm produces, for $W = P - 1 = 2^p - 1$, generator graphs with the following properties. First of all the meta-period of the generator graph is $m = \frac{1}{2} P$. So, since $I$ has to be a multiple of $\frac{m}{p_{uv}} = \frac{1}{2} P$, as explained in Section 7.4.3 we choose $I = 2 \cdot \frac{1}{2} P$ to obtain a block size $L = IP = P^2$. Secondly, the complexity is $3 - \frac{4}{m} = 3 - \frac{8}{P} \in \Theta(1)$ operations per output, which satisfies our complexity requirement. This means that our algorithm obtains an iteration period of $\Theta(P)$ provided that the number of shuffle operations per iteration is at most $O(P)$.

The reason we chose the maximum reuse algorithm to construct the generator graph is that it produces generator graph with a very regular pattern for $W = 2^p - 1$. We describe the generator graphs produced by the maximum reuse

---

4which is the function composition operator in case of look-ahead computations.
heuristic using the functions $\text{even}$ and $\text{odd}$ that take sub-sets of tile-sets:

\[
\begin{align*}
\text{even} & : (T \times \mathbb{Z}^D \times \mathbb{Z}^D) \rightarrow (T \times \mathbb{Z}^D \times \mathbb{Z}^D) \\
\text{even}(\langle s \rangle, f, p) & = (\langle s \rangle, f, 2p) \\
\text{odd} & : (T \times \mathbb{Z}^D \times \mathbb{Z}^D) \rightarrow (T \times \mathbb{Z}^D \times \mathbb{Z}^D) \\
\text{odd}(\langle s \rangle, f, p) & = (\langle s \rangle, f + p, 2p)
\end{align*}
\] (7.21) (7.22)

and the tile-sets $A_i = (\langle 2^i \rangle, 2i - 1, 2i)$ and $B_i = (\langle 2p - 2^i \rangle, 2i - 1, 2i)$. The generator graphs produced by the maximum reuse heuristic are then defined as:

\[
\begin{align*}
A_i & = \text{odd}(A_{i-1}) \odot \text{even}(A_{i-1}) \quad \text{for } 1 \leq i < p \\
\text{even}(B_i) & = \text{even}(A_i) \odot B_{i+1} \quad \text{for } 0 \leq i < p - 1 \\
\text{odd}(B_i) & = B_{i+1} \odot \text{odd}(A_i) \quad \text{for } 0 \leq i < p - 1
\end{align*}
\] (7.23)

where we slightly abuse the notation $T = T_1 \odot T_2$ to indicate that the predecessors, i.e. constructing tile-sets, of $T$ are $T_1$ and $T_2$.

Note that $A_{p-1} = B_{p-1}$ and that $B_0 = W = (\langle P - 1 \rangle, 0, 1)$. Furthermore, we have that $(\forall i : 0 \leq i < p : \{T \mid T \in V_g \land \#S_T = 2^i \} \text{ partitions } A_i)$ and $(\forall i : 0 \leq i < p : \{T \mid T \in V_g \land \#S_T = 2p - 2^i \} \text{ partitions } B_i)$. Since this covers all the nodes in the generator graph, this means that the equations above describe the entire graph and therefore the entire first sub-algorithm.

We implement the first sub-algorithm based on these equations. To describe the algorithm in more detail we introduce a notation that allows us to describe a vector containing a finite number of tiles. Such a vector is described by a tuple of the form $(S_T, f_T, p_T, n)$, which defines a vector like so:

\[
(S_T, f_T, p_T, n) = [\langle S_T + f_T + (i \cdot p_T) \rangle | 0 \leq i < n]
\] (7.24)

So $S_T, f_T, p_T$ and $n$ describe, respectively, the shape, phase and period of the tiles in the vector and $n$ describes the the length of the vector. Note that the phase now also indicates the first tile of the vector. We use this notation in our pseudo-code to annotate our program.

Listing 7.3 shows the SIMD pseudo-code used to construct some of the tiles from tile-set $A_j$. At first glance this procedure requires only three vector operations and a scalar operation, but we must keep in mind that the vectors in Listing 7.3 have a length that is longer than $P$. This is denoted by the type $\text{vector*}$, instead of the ordinary type $\text{vector}$. The length of these $\text{vector*}$ vectors is a multiple of $P$ and therefore the vector operations in the pseudo-code represent multiple vector operations. The $\text{shuffle}$ operation used to divide the
7.5. LOOK-AHEAD COMPUTATIONS

Listing 7.3: SIMD pseudo-code for the construction of part of $A_j$ in iteration $i$ of the algorithm
A_vec vector into its even and odd parts, therefore represents multiple operations. The dividing of the vector d0_vec+d1_vec of length 2P into the even even_vec and odd_vec parts using only the shift and inverse perfect shuffle is done as follows:

\[
\begin{align*}
\text{even_vec} & := \text{shuffle}(\text{d0_vec, invPS}); \\
t_{\text{vec}} & := \text{shuffle}(\text{shift(\text{d1_vec}, -1,0), invPS}); \\
\text{even_vec} & := \text{masked_move}(t_{\text{vec}}, \text{mask}); \\
\text{odd_vec} & := \text{shuffle}(\text{shift(\text{d0_vec}, 1,0), invPS}); \\
t_{\text{vec}} & := \text{shuffle}(\text{d1_vec, invPS}); \\
\text{odd_vec} & := \text{masked_move}(t_{\text{vec}}, \text{mask});
\end{align*}
\]

where mask is a masking vector containing \(\frac{1}{2}P\) false values followed by \(\frac{1}{2}P\) true values. This operation is repeated \(P2^{-(j+1)}\) times, once for each pair of vectors used to represent \(A_vec\) in Listing 7.3. In a similar manner the shift and \(\odot\) operations in Listing 7.3 represent \(P2^{-(j+1)}\) shift and \(\odot\) operations on vectors of length \(P\). This means that the construction of part of the tile-set \(A_j\) in iteration \(i\) requires \(\Theta(P2^{-j})\) vector operations.

Listing 7.4 shows the SIMD pseudo-code for constructing part of the tile-set \(B_{j-1}\) in iteration \(i\). The vector operations in this Listing also correspond to multiple vector operations. So, to construct \(B_{j-1}\) we need \(L2^{-(j-1)}\) \(\odot\) operations, which corresponds to \(P2^{-(j-1)}\) vector operations. The shuffle operation used to merge the newevenB_vec and newoddB_vec into newB_vec is in fact the inverse of the shuffle operation used in Listing 7.3 and therefore requires the repetition of a few shuffle and shift operations \(P2^{-j}\) times. This results in a total, so far, of \(\Theta(P2^{-j})\) vector operations, but we have not taken the shift operation into account. This operation corresponds to \(P2^{-j}\) shift operations over a distance of \(-2^{p-j}+1\). As shown in Section 7.2.3.4 a rotate operation over a distance of \(-2^{p-j}\) can be emulated on a vector processor with a PSNN topology using \(2(p-j)\) or \(2j\) (inverse) perfect shuffle operations, whichever is less, and one rotate operation. We then require an additional rotate operation to bring the total distance to \(-2^{p-j}+1\). Using masked assignments we then convert the resulting rotated vector to a shifted vector. So, the construction of \(B_{j-1}\) requires \(\Theta(P2^{-j}) + P2^{-j+1}(p-j \min j)\) vector operations.

These two procedures for computing the tiles from \(A_j\) and \(B_j\) are combined in the pseudo-code of Listing 7.5 to form the first sub-algorithm. By summing the operation counts of the ConstructA and ConstructB procedures we derive that this sub-algorithm requires \((+j : 0 \le j < p-1 : \Theta(P2^{-j})) + (+j : 1 \le j < p : \Theta(P2^{-j}) + P2^{-j+1}(p-j \min j))\) vector operations, which equals \(\Theta(P) + (+j : 1 \le j < p : P2^{-j+1}(p-j \min j)) = \Theta(P) + (+j : 0 \le j < p : 2^{-(j+1)}(p-j \min j)) = \Theta(P) + (+j : 0 \le j < p : 2^{-(j+1)}(p-j \min j))\).
7.5. LOOK-AHEAD COMPUTATIONS

proc ConstructB (newB_vec, evenA_vec, oddA_vec, B_vec, buf_vec : vector*)
{ Pre : \[B_vec\] = \((2^p - 2^j), Li - 2^p + 2^j + 1, 2^j, L2^{-j}\) }
{ { evenA_vec } = \((2^{j-1}), Li - 2^{j-1} + 1, 2^j, L2^{-j}\) }
{ { oddA_vec } = \((2^{j-1}), Li + 1, 2^j, L2^{-j}\) }
{ { buf_vec } = \((2^{j-1}), Li - 2^p + 2^{j-1} + 1, 2^j, 2p^{-j}\) }
{ Post : \[newB_vec\] = \((2^p - 2^{j-1}), Li - 2^p + 2^{j-1} + 1, 2^j, L2^{-j-1}\) }
{ { buf_vec } = \((2^{j-1}), Li + 1, 2^j, L2^{-j}\) }
{ PS contains the perfect shuffle pattern }

var newevenB_vec, newoddB_vec, shevenA_vec : vector*;

newoddB_vec := B_vec ⊕ oddA_vec;
{ [newoddB_vec] = \((2^p - 2^{j-1}), Li - 2^p + 2^j + 1, 2^j, L2^{-j}\) }

shevenA_vec := shift (evenA_vec, \(-2^{p-j} + 1, buf_vec\));
{ [shevenA_vec] = \((2^{j-1}), Li - 2^p + 2^{j-1} + 1, 2^j, L2^{-j}\) }

buf_vec := evenA_vec [\(L2^{-j} - 2^{p-j} .. L2^{-j} - 1\)];
{ [buf_vec] = \((2^{j-1}), Li - 2^p + 2^{j-1} + 1, 2^j, L2^{-j}\) }

newevenB_vec := shevenA_vec ⊕ B_vec;
{ [newevenB_vec] = \((2^p - 2^{j-1}), Li - 2^p + 2^{j-1} + 1, 2^j, L2^{-j}\) }

newB_vec := shuffle (newevenB_vec + newoddB_vec, PS);
{ [newB_vec] = \((2^p - 2^{j-1}), Li - 2^p + 2^{j-1} + 1, 2^j, L2^{-j-1}\) }

Listing 7.4: SIMD pseudo-code for the construction of part of \(B_{j-1}\) in iteration \(i\) of the algorithm
\begin{lstlisting}[language=plaintext]
var A_vec : array [0..p) of vector*;
evenA_vec : array [0..p-1) of vector*;
oddA_vec : array [0..p-1) of vector*;
B_vec : array [0..p) of vector*;
buf : array [0..p-1) of scalar;
buf_vec : array [0..p-1) of vector*;
i,j,p,L : scalar;

\{ \[L\] = L ∧ P = 2^p ∧ \[1\] ∈ N \}
\{ Inv_0 : \[buf\](j) = (\langle 2^j \rangle, Li - 2^{j+1} + 1, 2^{j+1} + 1, 2^{j+1}) \text{ for } 0 \leq j < (p - 1) \}
\{ Inv_1 : \[buf_vec\](j-1) = (\langle 2^{j-1} \rangle, Li - 2^p + 2^{j-1} + 1, 2^j, 2^{p-j}) \text{ for } 1 \leq j < p \}

do true →
  A_vec[0] := read(IN,i*L);
  j:=0;
  do (j < (p-1))
    ConstructA(A_vec[j+1], evenA_vec[j],
                oddA_vec[j], A_vec[j], buf[j]);
    j:=j+1;
  od
  B_vec[p-1]:= A_vec[p-1];
  do (j > 0)
    ConstructB(B_vec[j-1], evenA_vec[j-1],
                oddA_vec[j-1], B_vec[j], buf_vec[j-1]);
    j:=j-1;
  od
  write(BUF1, i*L, B_vec[0]);
  i:=i+1;
  od
\end{lstlisting}

Listing 7.5: SIMD pseudo-code for the first sub-algorithm
7.5. LOOK-AHEAD COMPUTATIONS

\[ \frac{1}{2}p - 1 : (p + \frac{3}{2}jp)2^{\frac{1}{2}p - j + 1} + p^2 - 2p = \Theta(P) - 2p(6 - 5 \cdot 2^{\frac{1}{2}p} + p) = \Theta(P) \] vector operations. This means that algorithm scales linearly, since it requires \( \Theta(P) \) vector operations and it produces \( L = P^2 \) outputs per iteration, resulting in a throughput of \( \Theta(P) \).

Compared to the first sub-algorithm the two other sub-algorithms are relatively straightforward. They are shown in Listing 7.6 and 7.7 respectively. Note that one iteration of the first sub-algorithm is followed by \( P \) iterations of the second and third sub-algorithms. The performance characteristics of the algorithm obtained by combining these three sub-algorithms are summarized in Table 7.8.

The approach discussed in this section therefore results in a block processing algorithm that performs a look-ahead computation. More importantly, this algorithm has linear scalability and requires that the SIMD processor supports the shuffle operation, instead of strided memory accesses. Furthermore, it does not require that the SIMD processor a completely connected topology, but it only requires a PSNN topology.

Listing 7.6: SIMD pseudo-code for the second sub-algorithm

```
var input_vec : vector;
output_vec : vector;
t_vec : vector;
i : scalar;
P : scalar;
{[P] = P ∧ [1] ∈ \mathbb{N}}
do true →
    input_vec:=read(IN, (i-1)*P+1);
    {[[input_vec] = ((1),Pi-P+1,1,P)}
t_vec:=read(BUF1, i*P);
    {[[t_vec] = ((P-1),P(i-1) + 2,1,P)}
output_vec:=input_vec ⊕ t_vec;
    {[[output_vec] = ((P),P(i-1) + 1,1,P)}
write(BUF2, i*P, output_vec);
i:=i+1;
```

```
\begin{verbatim}
var input_vec : vector;
    state_vec : vector;
    output_vec : vector;
i,P : scalar;

\{ [P] = P \land [i] \in N \}
\{ Inv_0 : \text{[state_vec]}(p) = s(P(i-1)+p+1) \}

\textbf{do} true \rightarrow
\begin{align*}
\text{input_vec} &= \text{read}(\text{BUF2},i*P); \\
\text{[input_vec]} &= ([P],P(i-1)+1,1,P) \\
\text{state_vec} &= \text{input_vec}(\text{state_vec}); \\
\text{Inv}_0(i) &= i+1 \\
\text{output_vec} &= \text{output_function}(\text{state_vec}); \\
\text{write}(\text{OUT},i*P+1,\text{output_vec}); \\
i &= i+1;
\end{align*}
\textbf{od}
\end{verbatim}

Listing 7.7: SIMD pseudo-code for the third sub-algorithm

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{SIMD program} & & \\
\hline
\textbf{Nr. of PEs} & \textit{P} & \textit{L} = \textit{P}^2 \\
\textbf{Latency} & \mathcal{L} & \Theta(L) \text{ outputs} \\
\textbf{Memory} & \mathcal{M} & \Theta(L) \text{ registers} \\
\textbf{Complexity} & \mathcal{C} & \Theta(L) \text{ operations/block} \\
\textbf{Iteration period} & \mathcal{I} & \Theta(P) \text{ time units} \\
\textbf{Throughput} & \mathcal{T} & \Theta(P) \text{ outputs/time unit} \\
\textbf{Scalability} & & \text{linear} \\
\textbf{Remarks} & & \text{Requires the shuffle operation} \\
\hline
\end{tabular}
\caption{Characteristics of a maximum reuse based look-ahead computation}
\end{table}
7.5. **LOOK-AHEAD COMPUTATIONS**

### 7.5.3 Example: Infinite impulse response filter

In this section we discuss the implementation of an infinite impulse response filter on the EVP\textsubscript{16} processor. We have already treated this example in \[7.3.2\] but there we needed a fictional version of the EVP processor that supported strided memory access operations. In this section we implement the filter using the EVP\textsubscript{16} as it is, using shuffle operations instead of strided memory access operations.

In the previous section we have discussed the algorithm and its pseudo-code. We can map it directly on the EVP\textsubscript{16}, but the EVP\textsubscript{16} has a completely connected topology, while we assumed a PSNN topology in the previous section. Therefore we can make some improvements by combining (inverse) perfect shuffle and shift operations into a single shuffle operation. This has no effect on the scalability of the algorithm, but it does decrease the iteration time of the algorithm.

Something else that influences the iteration time of the algorithm is the way in which the perfect shuffle operation and its inverse are implemented. These operations are used to split a vector represented by multiple vector registers, e.g. \texttt{d0\_vec} and \texttt{d1\_vec}, in its even and odd elements, e.g. \texttt{even\_vec} and \texttt{odd\_vec}. In the previous section we have shown how this operation can be implemented on an SIMD processor with a PSNN topology. On the EVP\textsubscript{16}, with its completely connected topology, we can implement it using only four (masked) shuffle operations:

\[
\texttt{even\_vec} := \text{shuffle}\ (\texttt{d0\_vec}, \text{iPS}) \; ;
\]
\[
\texttt{even\_vec} := \text{masked\_shuffle}\ (\texttt{d1\_vec}, \text{iPS\_roth}, \text{lasth\_mask}) \; ;
\]
\[
\texttt{odd\_vec} := \text{shuffle}\ (\texttt{d1\_vec}, \text{iPS}) \; ;
\]
\[
\texttt{odd\_vec} := \text{masked\_shuffle}\ (\texttt{d0\_vec}, \text{iPS\_roth}, \text{firsth\_mask}) ;
\]

where \texttt{lasth\_mask} is a mask that contains false in the first \(\frac{1}{2}P\) positions and true in the other positions, \texttt{firsth\_mask} is the inverse of \texttt{lasth\_mask}. Furthermore, the pattern \texttt{iPS} is the inverse perfect shuffle and the pattern \texttt{iPS\_roth} is a combination of the inverse perfect shuffle and a rotation over \(\frac{1}{2}P\). This implementation requires 4 shuffle operations, but there are other implementations possible which require less shuffle operations.

A second option is:

\[
\texttt{even\_vec} := \texttt{d0\_vec} ;
\]
\[
\texttt{odd\_vec} := \text{rotate}\ (\texttt{d1\_vec}, 1) ;
\]
\[
\texttt{even\_vec, odd\_vec} := \text{masked\_move}\ (\texttt{odd\_vec}, \text{odd\_mask}) , \\
\text{masked\_move}\ (\texttt{even\_vec}, \text{odd\_mask}) ;
\]
\[
\texttt{even\_vec} := \text{shuffle}\ (\texttt{even\_vec}, \text{PO}) ;
\]
TABLE 7.9: Performance characteristics of a second order IIR filter on the EVP16 for $L = 256$

<table>
<thead>
<tr>
<th>Option</th>
<th>Iteration period</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option 1</td>
<td>736 clock cycles</td>
</tr>
<tr>
<td>Option 2</td>
<td>790 clock cycles</td>
</tr>
<tr>
<td>Option 3</td>
<td>830 clock cycles</td>
</tr>
</tbody>
</table>

odd_vec := shuffle(odd_vec,P1);

where odd_mask is a mask that contains true on the odd positions and false on the even positions. This program first ensures that the appropriate values are moved to even_vec and odd_vec, after which the patterns P0 and P1 put these values in the right order. This implementation requires more operations, but it requires only three operations that occupy the VSHU unit of the EVP16, namely the rotate and shuffle operations. The other operations are assignment operations, which can be executed on a number of other units of the EVP16.

A third option is:

even_vec := shuffle(d0_vec,iPS);
odd_vec := shuffle(d1_vec,invPSroth)
even_vec,odd_vec := masked_move(odd_vec,lasth_mask),
                     masked_move(even_vec,lasth_mask);
odd_vec := rotate(odd_vec, P/2);

where lasth_mask is a mask that contains false in the first $\frac{1}{2}P$ positions and true in the other positions. Furthermore, iPS and iPSroth contain, respectively, the inverse perfect shuffle pattern and the inverse perfect shuffle pattern combined with a rotation over $\frac{1}{2}P$. This implementation also requires only three operations that occupy the VSHU unit of the EVP16 and it contains less assignment operations than the last option.

We used the tool described in Appendix B.2 to consider all these options. As shown in Table 7.9, option 1 results in the lowest iteration period on the EVP16. Therefore we assume that option 1, based on 4 shuffle operations, is also the most efficient way to implement the reverse operations, i.e., the merging of the even and odd elements into a single vector.

Implementing a second order IIR filter on the EVP16 with $P = 16$ results in an algorithm that produces $P^2 = 256$ outputs per iteration. The implementation therefore requires $\frac{736}{256} = 2.88$ clock cycles per output. So the speedup is not very large, a scalar DSP processor would require $2N + 1 = 5$ clock cycles.
for a single output, resulting in a speedup of 1.74. The EVP\textsubscript{16} implementation, however, can scale and doubling the number of PEs will double the throughput.

7.5.4 Discussion

In the previous sections we have shown that it is possible to implement a look-ahead computation on an SIMD processor. This implementation is based on the generator graph implementation from Section 7.4, resulting in an implementation for SIMD processors that support strided memory access and an implementation for SIMD processors that support the shuffle operation.

The algorithm for processors that support shuffle operations requires only the shuffle patterns supported by the PSNN topology. This means that the algorithm is also suited for SIMD processors that do not have a completely connected topology, but it is also suited for processors with the, more limited, PSNN topology.

7.6 Conclusion

In this chapter we have demonstrated that all the methods described in the previous chapters can be implemented on an SIMD processor. In particular we have examined their implementation on the NXP EVP\textsubscript{16}, and found efficient implementations that scale well.

We have also examined the properties needed by SIMD processors to implement these algorithms. Particularly the strided memory access operations and the shuffle operations are necessary for these algorithms. This is unfortunate since the strided memory access operation is not found on common SIMD processors and the shuffle operation does not scale well for processors with more PEs.

We are able to reduce the requirements on these operations, however. The linear look-ahead computation requires strided memory access to be implemented on an SIMD processor, but the stride is such that it can be emulated on a M2 class processor. The look-ahead computation based on a generator graph needs either strided memory access or shuffle operations. The variant using strided memory access can, again, be emulated on a M2 class processor. The variant based on the shuffle operations requires only M1 class processor with a PSNN topology, i.e., a topology consisting of a ring and a perfect shuffle.

The only problem are the algorithms based on generator graphs in general, like the rank order filters for example. For these algorithms we either need
strided memory access or shuffle operations where the exact stride or pattern depends on \( W \) and the generator graph. In particular cases it is possible to reduce the requirements, like we have done for the look-ahead computation, but in general one of these two operations is needed to implement the algorithm such that it scales linearly.
Chapter 8

Conclusion

In this thesis we have provided an overview of existing methods, and developed new methods, to obtain high-throughput stream processing systems using relatively slow hardware components. These methods are such that the amount of hardware required to implement a stream processing system is proportional to the throughput, i.e., the methods result in systems that scale linearly.

We are interested in these methods, because we not only wish to benefit from the growing speed of hardware, but also from the growing number of hardware components that can be put on a single chip. Furthermore, the linear scalability property enables us to trade, at fixed ratios, between the clock frequency, the amount hardware and the throughput of the system. Therefore, we can increase the throughput by increasing the clock frequency, the amount of hardware, or a combination of the two. We can also optimize other performance characteristics indirectly. The same throughput can be maintained, for example, while increasing the amount of hardware and decreasing the clock frequency. This leads to a decrease in the required energy per output, since the decrease in clock frequency also enables us to supply a lower the voltage to the hardware.

We achieve linear scalability by means of block processing. A block processing algorithm does not produce one output per iteration, but it produces, in parallel, a block of outputs per iteration. This means that the throughput of the block processing algorithm equals the size of the blocks divided by the iteration period. Block processing algorithms can therefore benefit both from faster hardware, since this reduces the iteration period, and from more hardware, since this allows for a larger block size. Not every block processing algorithm scales linearly, but it is possible to design block processing algorithms where the re-
lation between the amount of hardware and the throughput, which depends on the block size and iteration period, is such that the resulting algorithm does scale linearly.

Designing a block processing algorithm that scales linearly is only the first stage in realizing a system, however. The second stage is the implementation of this block processing algorithm on an actual hardware platform. In the first stage we only concern ourselves with the computation that has to be performed, regardless of the operations supported by the hardware platform. Only in the second stage do we concern ourselves with mapping the algorithm on the operations supported by the hardware platform.

Existing methods for obtaining a high-throughput stream processing system, as presented in Chapter 2, do not follow this separation of concerns. These methods are based on the assumption that the algorithm will be implemented in dedicated hardware. We are also interested in implementation on SIMD processors, however, since there is renewed interest in using these processors for stream processing [4, 47]. We are able to adapt these existing, VLSI-based methods such that they result in a program for an SIMD processor. This adaptation has some drawbacks, however; in every case the resulting SIMD program either does not scale linearly, or it requires strided memory access operations, which are unavailable on common SIMD processors [38]. There is, therefore, room for improving the existing design methods, which we have done with the research discussed in this thesis.

We have identified several classes of the stream processing systems in Chapter 1. Although our results can be applied to every class of stream processing system, we focus primarily on the class of associative window computations in this thesis. Associative window computations are computations where each output is computed by aggregating a finite number of inputs using an associative aggregation operator. An example of such a system, namely the moving sum, is discussed in Chapter 3. Each output of a moving sum computation is the sum of \( W \) consecutive inputs. In the case of the moving sum the associative aggregation operator is therefore the addition operator. Other examples of associative window computation systems are the rank order filters discussed in Chapter 6, though it is not immediately obvious that every rank order filter belongs to the class of associative window computation systems.

Associative window computations where the outputs are the aggregation of \( W \) consecutive inputs have an interesting property. It has been shown that the number of aggregation operations needed to compute each output has a constant upper bound, regardless of the window size \( W \) [29, 36, 34]. Additionally, this bound is relatively low; no more than three aggregation operators are needed to
compute a single output. Such a low complexity is obtained by reusing intermediate results in the computation. By reusing an intermediate result to compute multiple outputs, the complexity of that intermediate result is amortized over those outputs.

We associate each output and intermediate result produced during an associative window computation with a tile, as described in Chapter 4. Each tile covers a number of inputs and is associated with the result obtained by aggregating those inputs. Tiles that cover one input are therefore associated with the inputs of the computation, and there are tiles covering $W$ inputs that are associated with the outputs of the computation. An aggregation operation corresponds to the combining of two tiles into a larger tile, where the larger then covers the inputs that are covered by both of the smaller tiles. To efficiently compute the outputs of an associative window computation we therefore have to combine tiles, starting with the tiles associated with the inputs, into the tiles associated with the outputs while keeping the number of combination operations, i.e. aggregation operations, as small as possible.

Note that tiles can come in all sorts of shapes and sizes and are not limited to covering consecutive inputs from a one-dimensional input stream. Tiles also enable us to represent two-dimensional associative window computations like rank order filters on images, and they enable us to represent windows that have an irregular shape or hole in them, like those used in [73] for example.

Next to tiles we have also introduced generator graphs. These generator graphs capture the repetitive nature of an associative window computation and enable us to represent algorithms for associative window computations in a concise, and graphical manner. Furthermore we have introduced a canonical form for generator graphs. The canonical form of a generator graph facilitates the derivation of the algorithm’s complexity, and it makes it relatively straightforward to implement the algorithm in VLSI and on an SIMD processor such that the implementation scales linearly, as discussed in Chapter 7.

We have used tiling to specify heuristic algorithms, in Chapter 5, that produce generator graphs for any associative window computation. Two of these algorithms are based on the van Herk-Gil-Werman algorithm [29, 36], and Harrington’s algorithm [34]. We have also introduced two new algorithms, one is based on a divide and conquer strategy, while the other is based on a maximum reuse strategy. The latter algorithm has a slightly higher complexity for windows of consecutive inputs than the existing algorithms [29, 36, 34], namely 4 instead of 3 operations per output, but it has some interesting properties. Unlike the van Herk-Gil-Werman algorithm and Harrington’s algorithm it can produce algorithms for windows of any shape, not just to windows of consecu-
tive inputs. Furthermore, it produces the best, or very close to the best, results for a variety of associative window computations, as shown in Figures 6.12 and 6.13. Also, the generator graphs produced by the maximum reuse heuristic often allow for a more fine-grained selection of the block size, they lead to a lower latency when the algorithm is implemented in VLSI, and they require only a limited set of permutation operations, i.e. perfect shuffles and rotates, and no strided memory access operations when implemented on an SIMD processor.

Although our efforts are aimed at associative window computations, our design methods also apply to other classes of stream processing systems. Scalable block processing algorithms for stream processing systems are, with the exception of those based on the block post-computation approach, based on look-ahead computations. We have shown that look-ahead computations are in fact window computations (Theorem 4.5.1). The inputs of a look-ahead computation are the state update functions that advance the state of the stream processing system by a single step, and the aggregation operator is the function composition operator. The outputs of a look-ahead computation are therefore state update functions that advance the state of the stream processing system by $W$ steps. Since at most 4 aggregation operations are required to compute each of these outputs, regardless of the window size $W$, we set $W$ equal to the block size and thereby design a block processing algorithm for look-ahead computations that scales linearly. Moreover, by using the maximum reuse strategy to design the block processing algorithm, we can implement the algorithm on an SIMD processor and require only a limited, and common, set of instructions.

The problem of developing high-throughput stream processing systems using slow hardware components is therefore solved, at least theoretically. To verify our methods and examine their practical performance we have applied them to simple stream processing systems, like IIR filters and rank order filters, and implemented them on the NXP EVP, an SIMD processor with $P = 16$ processing elements. In the next two paragraphs we compare the theoretical results for these two system to their practical implementation.

The class of rank order filters (ROFs) contains the erosion, dilation and median filters, and also, according to our definition (Definition 6.2.1), any filter that selects a range of ranks from a window, like window sorters. It is not immediately obvious that all ROFs perform an associative window computation. As we have shown in Chapter 6 however, each output is the result of the aggregation of a number of inputs using an intricate aggregation operator. We have named this aggregation operator the sliced merge operation. The complexity of the sliced merge depends on the size of the tiles that are aggregated and the range of ranks selected by the filter. This varying complexity makes it challenging to
apply our methods to the ROFs. The heuristic algorithms from Chapter 5 have been designed for an aggregation operator with constant complexity. So, to take the varying complexity of the sliced merge operator into account we designed a new heuristic algorithm to produce generator graphs specifically for ROFs. This new heuristic, based on an oblivious implementation of the sliced merge operation, produces ROF algorithms with very low complexity. In fact, with the exception of [55] for \( W = 6 \), it produces ROFs with the fewest operations per output compared to the literature. These algorithms do not only perform well in theory, but also in practice. The algorithms have been implemented on the EVP SIMD processor and these implementations obtain a throughput, see Table 7.6 that exceeded the throughput of other SIMD implementations [46].

We have also examined IIR filters to verify and benchmark the implementation of look-ahead computations using our methods. There are various ways of implementing IIR filters in VLSI [66, 67], but no implementations that scale linearly existed on SIMD processors. We introduced the first implementation with linear scalability in [38], also discussed in Section 7.3.2 For second order IIR filters this implementation resulted in a speedup of 7.27 for \( P = 16 \) processing elements, see Table 7.2. This means that the SIMD implementation exceeds the speed of a scalar implementation for \( P \geq 3 \). The implementation relies, however, on the availability of strided memory access operations, which are not available on most SIMD processors like the EVP. Therefore we have also implemented a second order IIR filter using our maximum reuse strategy, resulting in a SIMD program that requires only commonly available operations. This program on the EVP results in a speedup of 1.74 for \( P = 16 \) processing elements, see Table 7.9. This means that the SIMD implementation exceeds the scalar implementation only for \( P \geq 10 \). So, although the SIMD implementation results in a larger throughput than a scalar implementation, it requires either a relatively large \( P \) or strided memory access before any significant speedup is realized.

Our methods have been applied successfully to associative window computations in general, like the rank order filters. Applying them to look-ahead computations is more involved, since it requires the efficient representation of the state update functions and an efficient way of composing these functions. In Section 2.2 it is shown that it is always possible to find a representation, but the efficiency may be extremely low. So, theoretically it is always possible to find a scalable block processing algorithm, but in practice the complexity per output may be extremely large. Further research into the representation of state update functions and their composition is therefore useful for improving the complexity and throughput of our algorithms.
Another way to improve the performance is to take our algorithms into account during the design of new SIMD processors. Strided memory access is a very useful feature that would enable many of the strategies leading to implementations that scale linearly. Although full support of strided memory access is hard to realize some of the more limited forms of strided memory access are also useful, as discussed in Sections 7.2.2 and 7.3.1. Alternatively, a $P \times P$ matrix transposition operation would come in handy, as discussed in Section 7.5.1. Future SIMD processors might support these operations, but for now SIMD processors that allow arbitrary permutation of vectors, like the EVP, are sufficient to run our algorithms. In fact, the SIMD processor does not have to support all permutations, we have shown that the rotate and perfect shuffle permutations, and their inverses, are also sufficient. Therefore any stream processing system can be implemented such that it scales linearly on an SIMD processor that supports these four permutations.
Appendix A

Notation

This appendix contains an overview of the terms and notations used throughout this thesis. The first section describes the mathematical notation, while the second section describes the notation used for the SIMD pseudo-code.

A.1 Mathematical notation

In this section we describe the mathematical notation used in this thesis.

A.1.1 Types

Different fonts are used to make it easier to remember the type of value contained in a variables. In this section we briefly describe the different types and the fonts used to represent them.

For simple values like booleans, integers or reals a lower case letter in the standard math font is used, i.e., $a$. Any letter can be used, but dummy variables generally come from the set $\{a, i, j, n\}$.

There are two exceptions to the use of this font. The first one concerns variables describing the performance characteristics of algorithms, those are typeset in a calligraphic font using an upper case letter, i.e., $A$. The different performance characteristics are summarized in Table A.1. The second exception concerns important constants and design parameters, those are typeset in the standard math font using an upper case letter, i.e., $A$. These constants are summarized in Table A.2.
Table A.1: Performance characteristics

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>Latency</td>
<td>outputs</td>
</tr>
<tr>
<td>$M$</td>
<td>Memory</td>
<td>registers</td>
</tr>
<tr>
<td>$C$</td>
<td>Complexity</td>
<td>operations per block</td>
</tr>
<tr>
<td>$T$</td>
<td>Iteration period</td>
<td>time units</td>
</tr>
<tr>
<td>$T$</td>
<td>Throughput</td>
<td>outputs per time unit</td>
</tr>
</tbody>
</table>

Table A.2: Important constants and design parameters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>Block size of a block processing algorithm.</td>
</tr>
<tr>
<td>$P$</td>
<td>Number of processing elements of a SIMD processor.</td>
</tr>
<tr>
<td>$I$</td>
<td>Increment, the ratio between block size and number of PEs ($L = IP$).</td>
</tr>
<tr>
<td>$N$</td>
<td>Some measure of problem size, i.e number of states of an FSM or the order of a filter.</td>
</tr>
<tr>
<td>$W$</td>
<td>Window size and memory span of a system.</td>
</tr>
</tbody>
</table>

Upper case letters in the standard math font not found in Table A.2 indicate a set of values, i.e. $A = \{a_0, a_1, \ldots\}$. The way we use sets is discussed in more detail in Appendix A.1.3.

For variables that contain multiple values in a certain order, like vectors and lists, we use a bold font and a lower case letter, i.e. $a = (a_0, a_1, \ldots)^T$ and $a = [a_0, a_1, \ldots]$. For grid locations, i.e. a vector containing coordinates, the letter $g$ is generally used as dummy variable. We discuss lists and vectors in more detail in Appendix A.1.4.

A matrix (a vector containing vectors) and a tile (a list of grid locations) are also type-set in a bold font, but a capital letter is used. To denote tiles we generally use the letter $T$ and any other letter indicates a matrix. The transpose of a matrix $A$ is denoted using $A^T$.

Tile-sets, which are infinite sets of tiles that can also be represented by a tuple, are represented by upper case letters in a sans serif font, i.e., $A = (S_A, f_A, p_A) = \text{tiles}(S_A, f_A, p_A)$. Again, for tile-sets we usually use the letter $T$ in favor of any other. Tiles and tile-sets are discussed in more detail in Section 4.2.

Functions are typeset in a type-writer font, i.e. $a(i) = \ldots$, or in a bold
A.1. MATHEMATICAL NOTATION

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Sigma)</td>
<td>Input alphabet.</td>
</tr>
<tr>
<td>(\Gamma)</td>
<td>Output alphabet.</td>
</tr>
<tr>
<td>(S)</td>
<td>Set of possible states.</td>
</tr>
<tr>
<td>(\text{in})</td>
<td>The input stream. (\text{in} : \mathbb{N} \rightarrow \Sigma)</td>
</tr>
<tr>
<td>(\text{out})</td>
<td>The output stream. (\text{out} : \mathbb{N} \rightarrow \Gamma)</td>
</tr>
<tr>
<td>(s)</td>
<td>The states. (s : \mathbb{N} \rightarrow S)</td>
</tr>
<tr>
<td>(s_0)</td>
<td>Initial state, i.e., (s(0) = s_0).</td>
</tr>
<tr>
<td>(\delta)</td>
<td>State update function, i.e., (s(i + 1) = \delta(s(i), \text{in}(i))).</td>
</tr>
<tr>
<td>(\omega)</td>
<td>Output function, i.e., (\text{out}(i) = \omega(s(i))).</td>
</tr>
</tbody>
</table>

Table A.3: Symbols used to describe finite state machines (FSMs)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\mathcal{T})</td>
<td>The set containing every possible tile.</td>
</tr>
<tr>
<td>(\mathcal{D})</td>
<td>The data-set for class (FSM^\circ) systems, i.e., (\Sigma = \Gamma = \mathcal{D}).</td>
</tr>
<tr>
<td>(m)</td>
<td>The meta-period of a generator graph.</td>
</tr>
<tr>
<td>(-)</td>
<td>A variable whose value is not of interest to us.</td>
</tr>
</tbody>
</table>

Table A.4: Other symbols

typewriter font if they return a vector or list, i.e., \(a(i) = \ldots\). A function \(f\), or any other function denotes with a single lower case letter, is used only in the section of the thesis where it is defined. Other functions are used throughout the thesis and their definitions can be found by using the index on page 271.

Note that the symbols we use to describe finite state machines, summarized in Table A.3, contain a few exceptions to our notation. Finite state machines (FSMs) are generally represented using Greek letters, and we have followed this convention. This means, however, that the sets are Greek capital letters instead of roman capital letters. Also, we use Greek letters to represent the functions associated with an FSM, but these are not set in a type writer font. Finite state machines and their notation are discussed in more detail in Section 1.2.1.

Finally, table A.4 summarizes those symbols from this thesis that, for some reason, deviate from the notational conventions discussed in this chapter. The importance of the set \(\mathcal{T}\) is the reason it has been put in a blackboard font, like \(\mathbb{N}\) and \(\mathbb{Z}\), instead of a normal upper case letter. For the same reason we use a blackboard font to represent the data set \(\mathcal{D}\). The meta-period of a generator...
graph \( \mathbf{m} \) can either be a single value, for one-dimensional systems, or a vector for multi-dimensional systems. For consistency, however, we opted to always use our vector notation for the meta-period. The symbol \( \_ \) is used as a variable whose value is not of interest to us. We pass it, for example, as one of the parameters of a function when the value does not influence the return value of the function and writing the value in full would take up much space. We use \( f(x, \_) \) for example, when we know \( x > 0 \) and \( f \) is defined as:

\[
f(x, y) = \begin{cases} 
x & \text{if } x > 0 \\
y & \text{if } x \leq 0 
\end{cases}
\]  

(A.1)

So, because of this special role of \( \_ \), we choose a symbol to represent it, instead of a letter.

### A.1.2 Eindhoven Notation

We use the so-called Eindhoven notation in this thesis [28] to write down expressions. These expressions have the following form:

\[
(Ox : D(x) : T(x))
\]  

(A.2)

where \( O \) is a binary operator, \( x \) declares the dummy variables, \( D(x) \) is a predicate defining the domain of the dummy variables and \( T(x) \) a term based on the dummy variables.

To evaluate \( A.2 \) all values of the dummy variables \( x \), for which the domain predicate \( D(x) \) is true, are used to evaluate the term \( T(x) \), after which these terms are aggregated using the operation \( O \). Note that the standard Eindhoven notation assumes that \( O \) is associative, commutative and has an identity element.

We also allow operators that do not have an identity element, in which case we take care that the domain is not empty. Furthermore, we allow commutative operators, in which case the domain \( D(x) \) not only specifies the domain of the dummy variables but also the order in which the domain is traversed. The order of traversal is specified in one of two manners. One manner is by using an expression like \( a \leq i < b \), where \( i \) is the dummy variable. In such cases the traversal starts with \( i = a \) and subsequently higher values of \( i \) up until \( i = b - 1 \). The other manner is by using an expression like \( i \in l \), which specifies that \( i \) is an element of the list \( l \). In such cases the traversal happens in the order specified by the list \( l \).
A.1. MATHEMATICAL NOTATION

A.1.3 Sets

We use the standard notations for sets. The operations on sets are the intersection $\cap$, union $\cup$ and difference $\setminus$. The empty set is denoted by $\emptyset$ and the size of a set $A$ is denoted by $\#A$. The relation $\in$ is used to describe that an element is in a set, i.e. $a \in \{a, b, c\}$.

We also use the following two short-hands to define sets:

\[
\{T(x) \mid D(x)\} = (\cup x : D(x) : \{T(x)\})
\]

(A.3)

\[
\{x : D(x) : T(x)\} = (\cup x : D(x) : \{T(x)\})
\]

(A.4)

A partition $P$ of a set $S$ is a set containing sub-sets of $S$ such that the following holds:

\[
(\cup S' : S' \in P : S') = S
\]

(A.5)

\[
(\forall S_1, S_2 : S_1 \in P \land S_2 \in P \land S_1 \neq S_2 : (S_1 \cap S_2) = \emptyset)
\]

(A.6)

Furthermore, we use $\mathcal{P}(A)$ and $\mathcal{L}(A)$ to denote, respectively, the set containing all subsets of $A$, i.e. the power-set of $A$, and the set containing all finite lists over the alphabet $A$.

A.1.4 Lists

We also use lists, the notation for this is almost the same as for sets. The main difference is that we use square brackets instead of curly braces. So, the empty list is denoted by $[]$ and $l = [a, b, c]$ defines the list $l$. We use this list $l$ as our running example in this section.

We use $\in$ to denote the membership of an element, so we have $a \in l$. The size of a list, just like the size of a set, is denoted using $\#l$, for our running example we therefore have $\#l = 3$.

Lists impose an ordering on their members. To obtain the element at a particular index we use the list as a function like so: $l(0) = a$, $l(1) = b$, etc. Furthermore, we can take slices of a list, i.e., sublists of a list by giving a range of indices. So we have $l[1..2] = [b, c]$. Note that the indices start at 0. The values of the list are undefined for negative indices and indices equal to or larger than the size of the list.

The most important operator on lists, that we use, is the concatenation operation $\oplus$, which combines two lists into a new list by concatenating them:

\[
(a \oplus b)(i) = \begin{cases} 
  a(i) & \text{if } i < \#a \\
  b(i - \#a) & \text{if } i \geq \#a 
\end{cases}
\]

(A.7)
This operator allows us to write the short-hands for defining lists, similar to the short-hands for sets, as:

\[ T(x) \mid D(x) = \{ x \in D(x) : [T(x)] \} \quad (A.8) \]

\[ x : D(x) : T(x) = \{ x \in D(x) : [T(x)] \} \quad (A.9) \]

There are some functions that operate on lists. First there is the function \( \text{set} \), which turns a list into a set:

\[
\text{set} : \mathbb{L}(A) \rightarrow \mathbb{P}(A) \text{ such that } (l \in \text{set}(l)) \equiv (l \in l) \quad (A.10)
\]

Second, we have permutations. Permutations are functions, denoted by \( \pi \), that take a list and return that same list with its elements rearranged. More formally we have that for every permutation \( \pi \) there is a bijective function \( f \) such that:

\[
\pi(l)(i) = l(f(i)) \text{ for } 0 \leq i < \#l \quad (A.11)
\]

In Section 6.5.2 we use some set operations on lists, namely “\( \cap \)”, “\( \setminus \)” and “\( \subseteq \)”. The reason we use the set operations is that, in that particular case, we are not interested in the exact ordering of the elements in the list, but only in the number of occurrences of a particular value. Therefore these operations are defined as:

\[ l_1 \subseteq l_2 \equiv (\exists \pi : \pi \text{ is a permutation : } \pi(l_2)[0..\#l_1] = l_1) \quad (A.12) \]

The intersection of two lists \( l_1 \) and \( l_2 \) is defined as:

\[ l_1 \cap l_2 \text{ is the longest list such that } (l_1 \cap l_2) \subseteq l_1 \land (l_1 \cap l_2) \subseteq l_2 \quad (A.13) \]

Furthermore the difference is defined as:

\[
\begin{align*}
\text{l}_1 \setminus {} & = l_1 \\
\text{l}_1 \setminus [l] + \text{l}_2 ' = \left\{ \begin{array}{ll}
\text{l}_1 \setminus \text{l}_2 ' & \text{if } l \notin l_1 \\
(l_1[0..i] + l_1[i + 1..\#l_1]) \setminus \text{l}_2 ' & \text{if } l \in l_1
\end{array} \right.
\end{align*}
\quad (A.14)
\]

Any other operation applied to lists, i.e. an operation that is not listed in this section, is applied element-wise to the elements of the lists to obtain a new list. For the addition operator, for example, we have:

\[ [1, 2, 3] + [4, 5, 6] = [5, 7, 9] \quad (A.15) \]
and for the multiplication operation, which we denote with \( \cdot \) for clarity, we have:

\[
[1, 2, 3] \cdot [4, 5, 6] = [4, 10, 18]
\]

Note that the lists are of the same length in such cases.

### A.1.5 Groups, semi-groups and semi-rings

We use the terms group, semi-group and semi-ring a few times in this thesis. This section contains the definition of these concepts.

A group and a semi-group are denoted using a set and a binary operator \((\mathbb{D}, \otimes)\). A semi-group has the following properties:

**Closure:** The set \(\mathbb{D}\) is closed under the binary operator \(\otimes\), i.e. \(\otimes : \mathbb{D} \times \mathbb{D} \to \mathbb{D}\).

**Associativity:** The operator \(\otimes\) is associative, i.e., \((\forall a, b, c : a, b, c \in \mathbb{D} : a \otimes (b \otimes c) = (a \otimes b) \otimes c)\).

A group has the following properties in addition to those of a semi-group:

**Identity element:** There is an identity element \(1_{\otimes}\) in \(\mathbb{D}\) such that \((\forall a : a \in \mathbb{D} : 1_{\otimes} \otimes a = a \otimes 1_{\otimes} = a)\).

**Inverse element:** Each element of \(\mathbb{D}\) has an inverse element, i.e. \((\forall a : a \in \mathbb{D} : (\exists a^{-1} : a^{-1} \in \mathbb{D} : a \otimes a^{-1} = a^{-1} \otimes a = 1_{\otimes}))\).

A semi-ring is denoted using a set and two binary operators \((\mathbb{D}, \oplus, \otimes)\). A semi-ring has the following properties:

- \((\mathbb{D}, \oplus)\) is a semi-group.
- \((\mathbb{D}, \otimes)\) is a semi-group.
- \(\oplus\) is commutative, i.e., \((\forall a, b : a, b \in \mathbb{D} : a \oplus b = b \oplus a)\).
- \(\otimes\) distributes over \(\oplus\), i.e., \((\forall a, b, c : a, b, c \in \mathbb{D} : a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c))\).

### A.1.6 Asymptotic notation

Throughout the thesis we use asymptotic notation to denote the order of the performance characteristics of our algorithms. This notation, using \(O\), \(\Omega\) and \(\Theta\), was taken from \cite{17}, and we summarize it in this section.
The letters $O$, $\Omega$ and $\Theta$ indicate, respectively, an lower asymptotic bound, an upper asymptotic bound and a tight asymptotic bound respectively. Formally:

\[
O( g(n) ) = \{ f(n) \mid (\exists c, n_0 : c, n_0 \in \mathbb{N}^+ : \\
(\forall n : n \geq n_0 : 0 \leq f(n) \leq c \ g(n)) ) \} 
\]  \hspace{1cm} (A.17)

\[
\Omega( g(n) ) = \{ f(n) \mid (\exists c, n_0 : c, n_0 \in \mathbb{N}^+ : \\
(\forall n : n \geq n_0 : 0 \leq c \ g(n) \leq f(n)) ) \} 
\]  \hspace{1cm} (A.18)

\[
\Theta( g(n) ) = O( g(n) ) \cap \Omega( g(n) ) 
\]  \hspace{1cm} (A.19)

This notation allows us to write, for example:

\[
n^2 + n + 1 \in O(n^3) 
\]  \hspace{1cm} (A.20)

\[
n^2 + n + 1 \in \Omega(n) 
\]  \hspace{1cm} (A.21)

\[
n^2 + n + 1 \in \Theta(n^2) 
\]  \hspace{1cm} (A.22)

### A.2 SIMD Pseudo-code

In this section we discuss the pseudo code we use in this thesis to describe our SIMD programs. The pseudo-code is based on the guarded command language (GCL) \cite{19}, which we have extended with SIMD operations.

We first discuss the use of variables and types in our pseudo-code, followed by GCL statements that we use, and we finish this appendix by discussing the extensions we made to the GCL to represent our SIMD programs.

#### A.2.1 Variables and types

Variables in our pseudo-code have a type and are declared using the `var` keyword. This keyword is followed by a comma-separated list of names, followed by a colon and then the type of the variables in the list. The line is terminated with a semi-colon, after which the program starts or another line of variable declarations follow. So we write, for example:

```plaintext
var a, b, c : scalar;
    v_vec : vector;
```

To gap the bridge between the program variables and mathematical concepts we use an abstraction function. The abstraction function maps the value of a program variable on a mathematical value. So, if the variable `a` contains the value 3, we have \([ a ] = 3\).
A.2. SIMD PSEUDO-CODE

The two main types that we use in our programs are scalar and vector, which are mapped onto a scalar and a vector of length $P$ by the abstraction function respectively. Furthermore, we use the convention that the name of a variable of the type vector ends with \_vec.

In addition to the normal vector type we also have the type vector*, which represents a vector with a length that is a multiple of $P$. Such a vector is used in same manner as the regular vector type. The difference is that it eases notation for describing vector operations on vectors of a length that is larger than $P$. The drawback is that vector operations on a variable of type vector* require multiple vector operations to implement. Therefore computing the iteration period of an SIMD program becomes a little more complicated than just counting the number of vector operations.

We also use variables of the type array, which consists of a sequence of variables of another type. The length of this sequence and the type of the variables in the sequence is defined using array[0..N) of X, where N denotes the length of the sequence and X the type. The abstraction function maps arrays of scalars onto vectors of the same length as the array. Arrays of vectors are mapped onto matrices in a similar manner, where each vector is mapped onto a row of the resulting matrix. So, in those cases an index in the array corresponds to the index of a row and an index in the vector corresponds to the index of a column. To refer to a particular value in an array, we enclose the index between square brackets in our pseudo-code, i.e., a[i].

A.2.2 Statements

In this section we describe the GCL statements that we use in our pseudo-code. More importantly, however, we specify the way in which we describe the effect of these statements.

We annotate our programs and specify the effect that our statements have on the state of the system by using Hoare triples \[37\]. A Hoare triple looks like this:

\[
\{ P_0 \} \\
S \\
\{ P_1 \}
\]

Where $P_0$ is a called a pre-condition, $S$ is a statement and $P_1$ is called a post-condition. The pre- and post-conditions are predicates on the state of the system. The semantics are that if statement $S$ is executed when the state of the
system satisfies $P_0$, then the state of the system after the execution of the statement satisfies $P_1$.

Using Hoare triples we can specify the assignment operation as:

$$\{P_0([a] := [E])\}$$

$$a := E$$

$$\{P_0\}$$

This triple specifies that if predicate $P_0$ holds after the value of expression $E$ is assigned to variable $a$, then, before the assignment, predicate $P_0$ with all instances of $[a]$ replaced by $[E]$ must have held\(^1\). For example:

$$\{[a] > 0\}$$

$$a := a + 3$$

$$\{[a] > 3\}$$

Note that the abstraction function maps any literals and operators on their mathematical counterparts, therefore $[a+3] = [a] + 3$.

We also allow the multiple assignment statement, i.e., the assignment of multiple values to multiple variables simultaneously. So we have for example:

$$\{[a] > 0\}$$

$$a, b := a + 3, a + 1;$$

$$\{([a] > 3) \land [b] > 1\}$$

Note that since the expressions are evaluated simultaneously, the update of variable $a$ has no effect on the update of variable $b$.

Since SIMD processors are ill-suited for branching programs we do not use the `if` statement in our code. The only other statement that we use from the GCL is the `do` statement, which is used to loop through a series of statements like so:

$$\{Inv\}$$

$$\textbf{do } G \textbf{ } \rightarrow$$

$$\{Inv \land [G]\}$$

$$S$$

$$\{Inv\}$$

$$\textbf{od}$$

$$\{Inv \land \neg [G]\}$$

where $G$ is a called “guard”, i.e. an expression that evaluates to true or false, $Inv$ is called a loop invariant and $S$ is a statements. If the guard evaluates to true

\(^1\)Or any other predicate implying this pre-condition.
A.2. SIMD PSEUDO-CODE

the SIMD processor executes statement S repeatedly until the guard evaluates to false.

Statements can be concatenated to form a new statement, using the semi-colon (;). So we can write:

\{
P_0 \}
S_1 ;
\{
P_1 \}
S_2
\{
P_2 \}

Note that the post-condition of \( S_1 \) has to imply the pre-condition of \( S_2 \) for this annotation to be valid.

In practice some of the annotation is omitted. Generally, the predicates in the annotation only describe the state changes effected by a statement. Any parts of the state not mentioned in the annotation is assumed to remain the same. We write, for example:

\{
[x] > 0 \}
x := x + 2;
\{
[x] > 2 \}
y := x + 3;
\{
y > 5 \}

instead of:

\{
[x] > 0 \}
x := x + 2;
\{
[x] > 2 \}
y := x + 3;
\{ [x] > 2 \land [y] > 5 \}

A.2.3 Extensions

We have added a few extensions to the guarded command language, to be able to represent SIMD operations. We discuss these extensions in this appendix.

First of all, any arithmetic operations on variables of type vector are evaluated in an element-wise fashion. Any scalar variables used in such an expression are converted into a vector by replicating the scalar value \( P \) times.

Secondly, we introduce the keyword proc to declare procedures using the call-by-reference semantics. So the procedure that swaps the value of two scalar variables is declared as:
\textbf{proc} \texttt{swap} (x,y: scalar) \\
\{ \\
\quad \{[x] = x \wedge [y] = y\} \\
\quad x,y:=y,x; \\
\quad \{[x] = y \wedge [y] = x\} \\
\} \\

And it can be used like so: \\
\{[a] = a \wedge [b] = b\} \\
\texttt{swap} (a,b); \\
\{[a] = b \wedge [b] = a\} \\

The most important extensions of the GCL, however, are those used to describe SIMD operations. We first discuss the memory access operations. These operations model the reading from and writing to the memory of the SIMD processor. By modeling the memory as an very large array named \texttt{MEM} we obtain the following specifications for the various memory access operations.

\{\lbrack BASE\rbrack + \lbrack offset\rbrack \geq 0\} \\
\texttt{a_vec} := \texttt{read} (BASE, offset) \\
\{\lbrack a_vec\rbrack (p) = \texttt{MEM}(\lbrack BASE\rbrack + \lbrack offset\rbrack + p) \text{ for } 0 \leq p < P\} \\

\{\lbrack BASE\rbrack + \lbrack offset\rbrack \geq 0\} \\
\texttt{write} (BASE, offset, a_vec) \\
\{\texttt{MEM}(\lbrack BASE\rbrack + \lbrack offset\rbrack + p) = \lbrack a_vec\rbrack (p) \text{ for } 0 \leq p < P\} \\

\{(\forall p : 0 \leq p < P : \lbrack BASE\rbrack + \lbrack offset\rbrack + p[\texttt{stride}] \geq 0\} \\
\texttt{a_vec} := \texttt{strided_read} (BASE, offset, \texttt{stride}) \\
\{\lbrack a_vec\rbrack (p) = \texttt{MEM}(\lbrack BASE\rbrack + \lbrack offset\rbrack + p[\texttt{stride}]) \text{ for } 0 \leq p < P\} \\

\{(\forall p : 0 \leq p < P : \lbrack BASE\rbrack + \lbrack offset\rbrack + p[\texttt{stride}] \geq 0\} \\
\texttt{strided_write} (BASE, offset, \texttt{stride}, a_vec) \\
\{\texttt{MEM}(\lbrack BASE\rbrack + \lbrack offset\rbrack + p[\texttt{stride}]) = \lbrack a_vec\rbrack (p) \text{ for } 0 \leq p < P\} \\

The \texttt{BASE} parameter of the memory access operations is used to distinguish between different base addresses in the memory. We assume that the SIMD processor uses memory mapped I/O, and the constants \texttt{IN} and \texttt{OUT} represent the base addresses of the of the input and output stream respectively.

We also introduced a operation that generates a mask by comparing values of two vectors:
A.2. SIMD PSEUDO-CODE

\[
\text{mask_vec := compare_gte(a_vec, b_vec)}
\]
{\([\text{mask_vec}]_p = [\text{a_vec}]_p \geq [\text{b_vec}]_p \text{ for } 0 \leq p < P\) }

Most of the other operations we introduced are used to rearrange the elements of a vector.

\[
\text{a_vec := rotate(b_vec, dist)}
\]
{\([\text{a_vec}]_p = [\text{b_vec}]_{(p + [\text{dist}]) \mod P} \text{ for } 0 \leq p < P\) }

\[
\text{a_vec := shift(b_vec, dist, newval)}
\]
{\([\text{a_vec}]_p = [\text{b_vec}]_{(p + [\text{dist}])} \text{ for } 0 \leq p < [\text{dist}] < P\) }  
{\([\text{a_vec}]_p = [\text{newval}] \text{ for } p < -[\text{dist}] \lor p \geq P - [\text{dist}]\) }

\[
\text{a_vec := shuffle(b_vec, pattern_vec)}
\]
{\([\text{a_vec}]_p = [\text{b_vec}]([\text{pattern_vec}]_p) \text{ for } 0 \leq p < P\) }

Furthermore, we introduced masked versions of some statements. In a masked version not all of the results are written to a variable:

\[
\text{a_vec := masked_move(b_vec, mask_vec)}
\]
{\([\text{a_vec}]_p = [\text{b_vec}]_p \text{ for } [\text{mask_vec}]_p \land 0 \leq p < P\) }  
{\([\text{a_vec}]_p = [\text{a_vec}]_p \text{ for } \neg [\text{mask_vec}]_p \land 0 \leq p < P\) }

\[
\text{a_vec := masked_shuffle(b_vec, pattern_vec, mask_vec)}
\]
{\([\text{a_vec}]_p = [\text{b_vec}]([\text{pattern_vec}]_p) \text{ for } [\text{mask_vec}]_p \land 0 \leq p < P\) }  
{\([\text{a_vec}]_p = [\text{a_vec}]_p \text{ for } \neg [\text{mask_vec}]_p \land 0 \leq p < P\) }
Appendix B

Tools

This appendix contains a brief overview of the tools used to generate the results presented in this thesis. These tools, written in Java, are stored in the SAN repository of the Technische Universiteit Eindhoven.

B.1 OWS tools

The Oblivious Window Sorting (OWS) tools are used to construct generator graphs and translate them into EVP-C code, i.e., code that can be compiled to run on the EVP\textsubscript{16} processor.

To construct generator graphs two things are needed. First of all we need the associative window computation (Definition 4.3.1) and a cost function for the merging of tiles, this is provided by the ProblemDescription class. Secondly, we need an heuristic algorithm to construct the generator graph. These heuristic algorithms are described by a TilingStrategy class, and the package tiling.strategies contains the code for every heuristic described in Chapter 5, Chapter 6 and in [40].

The OWSTable tool compares the different heuristics for a specific associative window computation and a range of window sizes. For every window size and heuristic it computes, and prints, the cost in operations per output. The results of this tool were used to generate the data in Appendix A of our technical report [40], on which the graphs and tables in Chapter 6 are based.

The OWSDesign tool produces a generator graph for a specific associative window computation using a specific heuristic. This generator graph can be
saved to a file and processed further by the other tools.

The generator graph can be depicted by the ShowTileSet tool. This tool can produce a text output that is readable by humans, or the LaTeX output used to produce the tables in Appendix B of our technical report [40].

The generator graphs produced by the heuristic algorithms are not necessarily in canonical form. To convert them to canonical form there is the NormalizeTileSet tool.

Once a generator graph is in canonical form the GenerateCode tools can be used to produce EVP-C code. This tool can currently generate code for one- and two-dimensional rank order filters only. It is possible to extend it with more dimensions by providing a header file containing functions to load and store the multi-dimensional data. Furthermore, by modifying or replacing the Merging-Variable class the tool can generate code for other aggregation operations than the sliced merge operator.

The EVP-C code produced by these tools was compiled using the standard EVP compiler, resulting in the performance shown in Table 7.6. The optimization described in Section 7.4.2 is performed by the "optimizecas" script, included with these tools.

### B.2 WinLA tool

The WinLA tool is used to generate EVP-C programs for stream processing systems.

The GenerateCode program produces the EVP-C code. This code is based on the algorithm obtained by modeling a look-ahead computation as an associative window computation and applying the maximum reuse heuristic, as described in Section 7.5.2.

A stream processing system is described by the SystemSpec class. This class describes the number of registers needed to store the state of the system, the number of registers needed to store a state update function, code used to combine state update functions, code to obtain state update functions based on the inputs, and code to obtain the outputs from the state of the system. The tool can be used to generate EVP-C code for any stream processing system described in this manner.

A second order IIR filter is described by the IIR2 class included in the tool. The code produced by the tool when the IIR2 class is used as the system specification was compiled using the standard EVP compiler and produced the results in Table 7.9.
Summary

Scalable Block Processing Algorithms

Tremendous amounts of data are processed by relatively small and cheap processors nowadays. Consider the radio waves arriving at a cell phone’s antenna and the light intensities on a DVD player’s lens. These are digitized into a stream of bits, which is then filtered, demodulated and decoded into an clear audio or video stream fit for the human senses.

My research is about these so-called stream processing systems, i.e., systems that perform relatively simple computations on a large stream of data. Research into these systems is necessary because their data rates have been growing at a faster rate than the speed of our processors. So, if this trend continues we will face a shortage of processing power.

The solution to this problem is parallelism. By using multiple processing elements it is possible to achieve the same data rate, or throughput, as one very fast processor. Ideally the throughput obtained in this manner is proportional to the number of processing elements, i.e., the throughput scales linearly with the amount of hardware used. Linear scalability makes it possible to avoid the shortage of processing power; it makes it possible to achieve a higher throughput by simply using more, instead of faster, hardware.

Is it possible to achieve linear scalability in every case? Amdahl’s law, or the law of diminishing returns, seems to indicate that linear scalability is not possible for systems in which even a small fraction of the computation is sequential. This seems to indicate that it is impossible to obtain linear scalability for every stream processing system, since many of them contain some kind of sequential computation, consider recursive filters or Viterbi decoders for example. It turns out, however, that there is actually no theoretical limit on the throughput of any stream processing system.
It is possible to obtain any throughput by using block processing algorithms. Block processing algorithms process, as their name implies, blocks of data instead of single data elements. A block processing algorithm achieves linear scalability by balancing, in the correct manner, the block size, the computation time per block and the number of processing elements needed for implementation. Existing and new methods for devising such algorithms are described in this thesis. The few existing methods with linear scalability produce algorithms that have to be implemented as dedicated hardware. The new ones, however, are also suited for implementation on SIMD processors.

In this thesis I present the theory and design methods required to design a block processing algorithm with linear scalability for any stream processing system. Additionally, I show how these methods can benefit the implementation of various specific systems like the moving sum, recursive filters and rank order filters. All of these can be implemented on a SIMD processor, like the NXP Embedded Vector Processor (EVP), such that the throughput scales linearly with the number of processing elements.
Samenvatting

Enorme hoeveelheden gegevens worden tegenwoordig verwerkt door vrij kleine en goedkope processoren. Neem bijvoorbeeld de radiogolven die binnenkomen bij de antenne van een mobiele telefoon en de lichtintensiteit op de lens van een speler DVD. Deze worden gedigitaliseerd naar een stroom van enen en nullen, vervolgens gefilterd, gedemoduleerd en uiteindelijk gedecodeerd naar een heldere, en voor mensen geschikte, stroom van geluid of video.

Mijn onderzoek gaat over deze zogenaamde stream processing systemen, d.w.z. systemen die vrij eenvoudige berekeningen op een grote stroom van gegevens uitvoeren. Het onderzoek naar deze systemen is noodzakelijk omdat hun verwerkingsnauwten momenteel sneller groeien dan de snelheden van onze processoren. Als deze tendens zich voortzet hebben we straks dus een tekort aan verwerkingscapaciteit.

De oplossing voor dit probleem is parallellisme. Door meerdere processoren te gebruiken is het mogelijk om dezelfde doorvoersnelheid, of throughput, te bereiken als met één zeer snelle processor. Idealiter is de throughput die we op deze manier bereiken rechtmeterd met het aantal processoren, ofwel de throughput schaalt lineair m.b.t. de hoeveelheid hardware die we gebruiken. Lineaire schaalbaarheid maakt het mogelijk om het tekort aan verwerkingscapaciteit te vermijden; het maakt het mogelijk om een hogere throughput te bereiken door simpelweg meer, in plaats van snellere, hardware te gebruiken.

Is lineaire schaalbaarheid echter wel mogelijk? De wet van Amdahl, of de wet van afnemende meeropbrengst, wijst erop dat lineaire schaalbaarheid niet mogelijk is in systemen waarin zelfs maar een kleine fractie van de berekening sequentieel is. Dit lijkt erop te wijzen dat lineaire schaalbaarheid niet mogelijk is voor ieder stream processing systeem, aangezien veel van deze systemen een sequentiële berekening bevatten, zoals bijvoorbeeld het geval is in recursieve filters en Viterbi decoders. Het blijkt echter dat er geen theoretische grens zit op de throughput van stream processing systemen.
Het is mogelijk om een willekeurige throughput te realiseren door zogenaamde block processing algoritmen te gebruiken. Block processing algoritmen verwerken, zoals hun naam al aangeeft, blokken van gegevens in plaats van individuele gegevens. Een block processing algoritme wordt lineaire schaalbaar door, op de juiste manier, te balanceren tussen de blok grootte, de verwerkingsstijd per blok en de hoeveelheid hardware die nodig is om het algoritme te implementeren. In dit proefschrift beschrijf ik zowel bestaande als nieuwe methodes voor het ontwerpen van zulke algoritmen. De weinige bestaande methodes resulteren in algoritmen die alleen gerealiseerd kunnen worden in dedicated hardware. De nieuwe methodes zijn echter ook geschikt voor implementatie op SIMD processoren.

In mijn proefschrift presenteer ik de theorie en ontwerpmethodes die nodig zijn om een block processing algoritme met lineaire schaalbaarheid te realiseren voor ieder willekeurig stream processing systeem. Daarnaast laat ik zien hoe deze methodes bijdragen aan de implementatie van diverse specifieke systemen zoals de moving sum, recursieve filters en rangorde filters. Al deze systemen kunnen op een SIMD processor, zoals de NXP Embedded Vector Processor (EVP) worden geïmplementeerd, zodanig dat de throughput lineair schaalt met het aantal processor elementen.
Curriculum Vitae

Martijn van der Horst was born on the 21st of January 1980 in Boxmeer, the Netherlands. In 1998 he received his Atheneum diploma from the Commanderij college in Gemert. He studied computer science at the Eindhoven University of Technology and received his Master’s degree in 2003 with his master thesis “Optimal route planning for car navigation systems” under the supervision of Johan Lukkien and Jacques Verriet at Siemens VDO. After receiving his Master’s degree he started as a PhD student in the System Architecture and Networking group under the supervision of his promoter Kees van Berkel and his supervisor Rudolf Mak.
Bibliography


Index

+, 245
$n$, 97
$\langle T \rangle$, 95
-, 244
$0_D$, 97
$1_D$, 115

add, 128
addlist, 167
algebraic state update function, see SUF, algebraic
algorithm, 10
   block processing, 5
array processor, 188
associative window computation, 9
   97
block diagonal form, 32, 62, 202
block post-computation, 51
block pre-computation, 51
block prefix computation, 106
block processing, 11
block processing algorithm, 5
calculation graph, 112
ch, 168
cmeth, 166
compare_gte, 253
complexity, 13
ctile, 166
cut, 130
data, 94
dependence graph, 112
din, 113
din_g, 117
do, 250
epo, 161
even, 224
EVP, 189, 191, 197, 198
extrema operations, 154
filter
dilation, 148, 171
erosion, 148
median, 148, 171
finite impulse response filter, 9
finite state machine, see FSM
finite state transducer, 7
FIR filter, 9
FSM, 7
   class FSM, 8
class $FSM_\infty$, 8
class $FSM_*$, 8
class $FSM_\odot^*$, 9

generator graph, 115
   canonical form, 121
generator sub-graph, 125
Gildier’s law, 3
GOPS, 4

heuristic function, 129
   basic, 129
   partitioning, 131
hyperbar, 164
hyperbarlist, 145
hypercube, 115

IIR filter, see infinite impulse response filter

in, 7
   in, 11
incremental block computation, 44
infinite impulse response filter, 9
inmap, 100
inter-iteration registers, 13
intra-iteration register, 20
intra-iteration registers, 13
iteration period, 17

$L()$, 245
latency, 12
law of diminishing returns, 25
look-ahead
   computation, 29
   linear, 40
   logarithmic, 44
   parallel prefix, 48
   window computation, 105

prefix based, 107
lovl, 135
lovt, 136
low, 152
msb, 132

MAC, 189
map, 116
masked_move, 253
masked_shuffle, 253
masking, 189
matrix-vector multiplication
   see MVM, 63
max, 8
maximum reuse, 135
MC, 160
memory access, 190
   aligned, 191
   strided, 22
   unaligned, 78, 191
merge
   valid, 152
merge, 125
merge*, 128
mergesort, 150
meta-period, 121
min, 63
Moore machine, 7
Moore’s law, 4
MOPS, 4
MPB, 160
msb, 132
multiply-accumulate, 189
MVM, 63

non-separable, 99
norm, 96
nov, 168

oblivious sorting, 218
INDEX

odd, 224
op, 165
operation
masked, 189
opo, 25, 115
orm, 165
os, 165
ot, 165
out, 7
out, 11
outmap, 100
ov, 166
overlap, 103, 135, 161
ovl, 164
ovt, 167

P(), 245
PB, 155
PE, 21, 188
pipelining, 38
postfix, 135
pow2, 132
PRAM, 192
pred0, 114, 120
pred0_g, 120
pred1, 114, 120
pred1_g, 120
prefix, 135
premerge, 128
processing element, see PE
PSNN, see topology, PSNN

r-overlap, 161
rank order filter, 148
read, 252
recursive system, 9
registers
inter-iteration, 13
intra-iteration, 13

rem, 130
reu, 169
rnk, 151
ROF, 148
rotate, 253
rtile, 166

s, 7
scalability
linear, 5, 24
super-linear, 25
select, 14
semi-group, 9, 30
separable, 99
set, 246
shape, 95
shift, 253
shuffle, 22
shuffle, 253
shuffle-shift, 196
SIMD, 18
SIMD processor, 21
M1 class, 191
M2 class, 191
M3 class, 192
spl, 168
split, 130
state space form, 61, 85
stream processing systems, 6
strength reduction, 84
strided_read, 252
strided_write, 252
SUF, 14
algebraic, 30, 54
basic, 14
system
stream processing, 6
T, 94
taxonomy, 188
TD, 130
TDP, 131
TDPr, 132
tile, 94
  ordering, 95
tiles, 95
topology
  completely connected, 193
  linear, 194
  other, 196
  perfect shuffle, 194
  PSNN, 195
  ring, 194
up, 152

vector processor, 188
VLIW, 198
VLSI, 19

window, 94
window sorter, 148, 171
workload, 3
write, 252
Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02


S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04


N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09


J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14


Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02


S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

S.V. Nedea. Analysis and Simulations of Catalytic Reactions. Faculty of Mathematics and Computer Science, TU/e. 2003-06


H.P. Benz. Casual Multimedia Process Annotation – CoMPAs. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08


M.H. ter Beek. Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. The $\lambda$ Abroad – A Functional Approach to Software Components. Faculty of Mathematics and Computer Science, UU. 2003-11


G.I. Jojgov. Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. Theory of Molecular Computing – Splicing and Membrane systems. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. Models of Tree Translation. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. Data Synchronization and Browsing for Home Environments. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05


L. Cruz-Filipe. Constructive Real Analysis: a Type-Theoretical Formalization and Applications. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications. Faculty of Technology Management, TU/e. 2004-08

N. Goga. Control and Selection Techniques for the Automated Testing of Reactive Systems. Faculty of Mathematics and Computer Science, TU/e. 2004-09


J. Pang. Formal Verification of Distributed Systems. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. Evolutionary Agent-Based Economics. Faculty of Technology Management, TU/e. 2004-15
E.O. Dijk. Indoor Ultrasonic Position Estimation Using a Single Base Station. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. On Distributed Verification and Verified Distribution. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17


P.J.L. Cuijpers. Hybrid Process Algebra. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. Supervisory Machine Control by Predictive-Reactive Scheduling. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Abrahám. An Assertional Proof System for Multithreaded Java -Theory and Tool Support-. Faculty of Mathematics and Natural Sciences, UL. 2004-01

R. Ruimerman. Modeling and Remodeling in Bone Tissue. Faculty of Biomedical Engineering, TU/e. 2004-02

C.N. Chong. Experiments in Rights Control - Expression and Enforcement. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. Design and Verification of Lock-free Parallel Algorithms. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. Specification and Analysis of Internet Applications. Faculty of Mathematics and Computer Science, TU/e. 2005-05


I. Kurtev. Adaptability of Model Transformations. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. Computational Aspects of Treewidth - Lower Bounds and Network Reliability. Faculty of Science, UU. 2005-09

O. Tveretina. Decision Procedures for Equality Logic with Uninterpreted Functions. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. Evolution of Finite Populations in Dynamic Environments. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. Data Mining using Genetic Programming: Classification and Symbolic Regression. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. Top Quality Type Error Messages. Faculty of Science, UU. 2005-13

G.F. Frehse. Compositional Verification of Hybrid Systems using Simulation Relations. Faculty of Science, Mathematics and Computer Science, RU. 2005-14


T. Gelsema. Effective Models for the Structure of pi-Calculus Processes with Replication. Faculty of Mathematics and Natural Sciences, UL. 2005-17
P. Zoeteweij. Composing Constraint Solvers. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18


M. Valero Espada. Modal Abstraction and Replication of Processes with Data. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. Stepping through Haskell. Faculty of Science, UU. 2005-21


E. Dolstra. The Purely Functional Software Deployment Model. Faculty of Science, UU. 2006-01


P.R.A. Verbaan. The Computational Complexity of Evolving Systems. Faculty of Science, UU. 2006-03


M. Kyas. Verifying OCL Specifications of UML Models: Tool Support and Compositionality. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. Model Checking Timed Automata - Techniques and Applications. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. Böhm-Like Trees for Rewriting. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. On JML: topics in tool-assisted verification of JML programs. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. Towards Hybrid Molecular Simulations. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. Mining Structured Data. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. Separation and Adaptation of Concerns in a Shared Data Space. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. Reconciling Nondeterministic and Probabilistic Choices. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. Verification techniques for Extensions of Equality Logic. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. Constructive formal methods and protocol standardization. Faculty of Mathematics and Computer Science, TU/e. 2006-14


M.E. Warnier. Language Based Security for Java and JML. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. At Home In Service Discovery. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. Expressivity of Timed Automata Models. Faculty of Science, Mathematics and Computer Science, RU. 2006-18
L.C.M. van Gool. Formalising Interface Specifications. Faculty of Mathematics and Computer Science, TU/e. 2006-19


J.V. Guillen Scholten. Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. Flexible Heterogeneous Software Systems. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. A run-time reconfigurable Network-on-Chip for streaming DSP applications. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems. Faculty of Mathematics and Computing Sciences, RUG. 2007-03


M.W.A. Streppel. Multifunctional Geometric Data Structures. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Tréka. Silent Steps in Transition Systems and Markov Chains. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. Searching in encrypted data. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. Putting types to good use. Faculty of Science, Mathematics and Computer Science, RU. 2007-10


R. Boumen. Integration and Test plans for Complex Manufacturing Systems. Faculty of Mechanical Engineering, TU/e. 2007-12


C.F.J. Lange. Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. Component-based Configuration, Integration and Delivery. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-15

B.S. Graaf. Model-Driven Evolution of Software Architectures. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijsen. Logical Calculi for Reasoning with Binding. Faculty of Mathematics and Computer Science, TU/e. 2007-17

M. A. Abam. *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19


A.L. de Groot. *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02


N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multidisciplinary Systems.* Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates.* Faculty of Science, UU. 2008-06


I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines.* Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras.* Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit.* Faculty of Mathematics and Computer Science, TU/e. 2008-10


M. Farshi. *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12


E.M. Bortnik. *Formal Methods in Support of SMC Design.* Faculty of Mechanical Engineering, TU/e. 2008-16


M.G. van der Horst. *Scalable Block Processing Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2008-18