MASTER'S THESIS

Design of Optimal Buffers

by

A.M.C. Smets

Supervisors:  Drs. R.H. Mak
              Prof. Dr. Ir. C.H. van Berkel
              Dr. Ir. G. Zwaan

Eindhoven, August 2002
To my father
Preface

Overview

Chapter 1 gives a brief introduction of optimal buffers and shows a few examples of how buffers are put into practice.

In chapter 2 we will introduce buffers and their functional specification. We will present a set of basic components and several construction methods for composing buffers out of these basic components. Furthermore we will introduce some buffer families. Finally in this chapter we will define two static properties of a buffer design: the capacity and the i/o-distance, the latter being the shortest path through the buffer. This chapter forms the basis for the construction of buffers.

In chapter 3 we will show how to describe the communication behavior of a constructed system. Therefore we will introduce the formalism of a sequence function. We will define common performance parameters such as cycle time, latency and occupancy in terms of these sequence functions and show that they satisfy a simple relationship. Next we will derive some relations for these performance parameters between a constructed buffer using one of the construction methods and its subcomponents.

In chapter 4 we establish lower and upper bounds for the latency and occupancy in terms of the capacity and the i/o-distance. This in turn will raise the definition of a \((\kappa, \delta)\) – optimal buffer. Furthermore we will derive some restrictions for the construction methods such that, when obeyed, the application of the construction methods indeed yields a \((\kappa, \delta)\) - optimal buffer. Then we will introduce three classes of optimal buffers and present for each class of buffers an algorithm to generate that class of buffers. Also we will introduce buffer designs that are partial \((\kappa, \delta)\) – optimal.

In chapter 5 we will present advanced buffer design. We will extend the design of the basic components. First we will introduce basic components with two variables and then we will construct some buffers with these basic components. Next we will introduce basic components with a fan-in and fan-out of three and then present the cubic buffers.

In chapter 6 we will introduce two miscellaneous system designs. First we will introduce permutations and system designs. Then we will introduce blockreversers and show that some theorems will also hold for these systems.

Acknowledgements

It is a pleasure to express my gratitude to my employer drs. A. Dorrepaal, who made it possible for me to spend the time needed to write this thesis.

Above all I want to thank my wife Femke for her support and patience.

Finally, my special thanks goes to my supervisor drs. R.H. Mak, who spent so much time teaching me all he knows about buffer design.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Sequence functions for the basic component Split_{k1}</td>
<td>105</td>
</tr>
<tr>
<td>A.2 Sequence functions for the basic component Merge_{k1}</td>
<td>105</td>
</tr>
<tr>
<td>A.3 Early sequence functions for the basic component Split_{k1}</td>
<td>106</td>
</tr>
<tr>
<td>A.4 Late sequence functions for the basic component Split_{k1}</td>
<td>106</td>
</tr>
<tr>
<td>A.5 Early sequence functions for the basic component Merge_{k1}</td>
<td>107</td>
</tr>
<tr>
<td>A.6 Late sequence functions for the basic component Merge_{k1}</td>
<td>107</td>
</tr>
<tr>
<td><strong>B Sequence functions for the multi-wagging composite</strong></td>
<td><strong>108</strong></td>
</tr>
<tr>
<td>B.1 Early sequence functions for the multi-wagging composite</td>
<td>108</td>
</tr>
<tr>
<td>B.2 Late sequence functions for the multi-wagging composite</td>
<td>109</td>
</tr>
<tr>
<td>B.3 General sequence functions for the multi-wagging composite</td>
<td>110</td>
</tr>
</tbody>
</table>
1 Introduction to designing optimal buffers

Buffers are important components of systems containing producer and consumer processes that operate at different speeds. Often the overall performance of systems critically depends on the kind of buffers used. For instance the following devices need buffers to function properly.

**Compact Disc Player**

Unlike vinyl records that rotate at a constant speed (33 or 45rpm), the rotational speed of a CD varies such that the laser passes over the data at a constant rate of 1.4 meters/second. The mechanism for controlling speed is very simple. As the data is extracted from the disc, it is stored in a buffer memory before being passed on at the correct sampling rate to the decoding circuitry. If the disc is spinning too fast the memory fills up; if the disc is spinning too slowly the memory empties. A simple feedback system based on the memory capacity is used to control the spin-motor [NaOg].

**Fax**

A fax cannot directly print the data that it is receiving. The stream of data that the fax receives has to be buffered before it can be printed.

**High Definition Television HDTV**

Nowadays televisions are for sale with a display frequency of 100 Hz, that is 100 screenshots per seconds. The information that the television receives from the broadcasting network contains only 50 screenshots per seconds. The television first buffers the input data. Next the data is sent two times to the screen.

As already mentioned the overall performance of systems critically depends on the kind of buffers are used. Therefore it is essential that we are able to design buffers with the right performance characteristics. We will study buffers that are composed of a small set of basic components. We will show that a simple calculus of two operators suffices to show the functional (partial) correctness of various buffer designs.

Furthermore, we use the formalism of a sequence function both as a means to establish absence of deadlock (total correctness) and more importantly as a vehicle for performance analysis. We define common performance parameters such as cycle time, latency and occupancy in terms of these sequence functions, and show that they satisfy a simple relationship. This in turn enables us to establish lower and upper bounds for the latency and occupancy in terms of two static properties of a buffer design, viz. the capacity and the i/o-distance; the latter being the minimum length of any path through the buffer. However not every buffer meets its optimal bounds. This raises the next question:

When buffers are created using a fixed set of basic components and a fixed set of construction methods for given capacity and i/o-distance, do these buffers meet their optimal bounds?
2 Building components and construction methods

In this thesis we will discuss the design of optimal buffers. We will design these buffers using a fixed set of basic components. Using parallel composition we will then construct systems of basic components. Since parallel composition is a very general composition method, the class of systems thus obtained contains not only buffers, but many other systems as well. Therefore we also introduce a number of restricted construction methods aimed specially at the design of buffers.

In section 2.1 we will start with the definition of some basic components. For constructing larger components we need some construction methods, those we will define in later sections. But first we start with the introduction of a simple calculus of two operators. The calculus suffices to show the functional (partial) correctness of the buffer designs described in this thesis. The calculus is taken from [Mak].

In terms of functionality buffers are extremely simple. Their output stream is a copy of their input stream. Advanced buffer designs, however, are characterised by the fact that there exist multiple paths from the input port to the output port. So in general the stream of values that passes between interconnected internal components of such a buffer is a subset of the input stream. In order to establish the functional correctness of the buffer as a whole it is therefore necessary to specify how the internal streams are divided and recombined. For this purpose we introduce two families of postfix operators that select a substream of a given stream.

For integers $0 \leq k < l$ and for $A = (++:0 \leq i: A(i))$ an infinite stream of values we define

$$A(k : l) = (++:0 \leq i \land i \mod l = k : A(i))$$

$$A(k : l) = (+,+i : 0 \leq i \land i \mod l \neq k : A(i))$$

These postfix operators are called the take and the drop operator respectively. The stream of values $A(k : l)$ expresses a stream where every $k$-th out of $l$ values of stream $A$ is taken, and $A(k : l)$ expresses a stream where every $k$-th out of $l$ values of stream $A$ is dropped.

For any pair of operators $P_1$ and $P_2$ we define their product by

$$A(P_1P_2) = A(P_1)P_2$$

For the take and drop operators the following rules hold

$$(k : l)(p : q) = (k+p : lq)$$

**take–take rule**

$$(k : q+1)(p : q) \begin{cases} if \ p < k & \rightarrow (p : q+1) \\ k \leq p \leq q & \rightarrow (p+1 : q+1) \\ fi \end{cases}$$

**drop–take rule**

Note that $0 : 2 = (1 : 2)$

**complement rule**
2.1 Basic components

In this chapter we will discuss the functionality of systems. In the next chapter we will present the communication behavior of a system.

2.1.1 Buf

The simplest basic component is the one-place buffer, called Buf. When we denote the stream of values communicated along its input port a and output port b respectively by capitals A and B, then these values satisfy the buffer specification B=A(0:1)=A. Figure 2.1.1 shows a diagram for this component.

![Figure 2.1.1 Buf](image)

2.1.2 Split

Apart from components that merely store a value we also need components to divide and recombine streams. Component Split_{k,l} (see figure 2.1.2) divides a stream into two substreams. Its functionality is specified by C = A(k : l) and D = A(k : l), 0 ≤ k < l.

![Figure 2.1.2 Split_{k,l}](image)

2.1.3 Merge

The output streams of a Split_{k,l} can be recombined into the original stream by means of a Merge_{k,l} component (see figure 2.3), possibly after being led through buffers. The functionality of a Merge_{k,l} component is specified by E = B(k : l) and F = B(k : l), 0 ≤ k < l.

![Figure 2.1.3 Merge_{k,l}](image)
2.2 Parallel composition

Now that we have defined the basic components we would like to construct larger components. These components can be obtained through parallel composition. As an example consider the component in figure 2.2.1 that consists of four basic components: two split components and two merge components.

![Diagram showing parallel composition with two split components and two merge components.]

Figure 2.2.1 Odd-Even Shuffle

These basic components are connected by four internal channels e, f, g, and h. Each channel connects precisely one input port of one basic component with precisely one output port of one other basic component. In the diagram of figure 2.2.1 these channels are represented by arrows, which indicate the direction of the data flow. In addition to its name, each channel in the figure apart from the input channels carries a second label. This is the postfix operator that yields the substream of values communicated along that channel.

Apart from using diagrams to define systems of basic components we can also define them using a program text. For instance, the Odd-Even shuffle system of figure 2.2.1 is given by:

```
Odd-Even Shuffle = proc (in a, b; out c, d)
  [[ chan e, f, g, h
    | Split0₂(a, e, f)
    | Split₁₂(b, g, h)
    | Merge₀₂(e, g, c)
    | Merge₁₂(f, h, d)
  ]]
```

where the program header `proc (in a, b; out c, d)` denotes that component Odd-Even Shuffle has two input ports a, b and two output ports c, d. The body i.e. the part of the program text between the bracket pair "[[ " and "]"], reveals the internal structure. It starts with a channel declaration `chan e, f, g, h` that indicates that component Odd-Even Shuffle has four internal communication channels e, f, g and h. The command `Split₀₂(a, e, f) || Split₁₂(b, g, h) || Merge₀₂(e, g, c) || Merge₁₂(f, h, d)` expresses that the Odd-Even Shuffle is the parallel composition (denoted by the operator "||") of two instances of a split component and two instances of a merge component. The connectivity pattern is indicated by the way in which the formal port names are instantiated with actual channel names. Consider the basic component `Split₀₂(a, e, f)` that has input port a and internal channels e and f. This basic component is connected with basic component `Merge₀₂(e, g, c)` by channel e and with basic component `Merge₁₂(f, h, d)` by channel f. Here only the structure specification is given. In next chapter we specify the communication behavior of a system.

Occasionally it will convenient to use a slightly different notation for a component definition, in which the block only contains one component. For example Buf equals `proc (in a; out c₀) [ Buf (a, c₀) ]`. 
2.3 Construction methods

So far we have defined three basic components and a general construction method: parallel composition through which we are able to construct a large set of systems. Next we will be interested in the subclass of all buffers, called \textit{BUF}. A buffer is defined by:

\textbf{Definition 2.2.1 Buffer}

A buffer is a component with a single input port and a single output port with the functionality that the output stream is a copy of the input stream.

We conclude this section with the construction of two special components that will turn out to be very convenient for the construction of some classes of buffers: the multi-way splitter and the multi-way merger.

The multi-way splitter, also called \( n \)-way splitter, divides a stream into \( n \) equal substreams. Next we show how to construct the \( n \)-way splitter using several split components.

\textbf{Definition 2.2.2 Multi-way splitter}

The family \( \{MS_n\}_{n=1}^\infty \) of \( n \)-way splitters is defined by

\[
\begin{align*}
MS_1 &= \text{Buf} \\
MS_{n+1} &= \text{proc (in a; out c_0, ..., c_n)} \\
& \quad \parallel \text{chan d} \\
& \quad \parallel \text{Split}_{n-1}(a, c_0, d) \\
& \quad \parallel MS_n(d, c_{n+1}, ..., c_n) \\
& \quad \}, 1 \leq n
\end{align*}
\]

Figure 2.2.2 shows a diagram for this component.

\begin{center}
\begin{tikzpicture}
\node[draw, rounded corners] {Split\textsubscript{\(n-1\)}} (split) at (0,0) [fill=white]
\node[draw, rounded corners] {MS\textsubscript{n-1}(d, c_{n-2}, ..., c_0)} (ms) at (3,0) [fill=white]
\draw [->] (split) -- node[below] {\(c_{n-1} \downarrow (n-1:n)\)} (ms)
\draw [->] (split) -- node[above] {\(a\rightarrow\)} (0,1)
\draw [->] (ms) -- node[below] {\(c_0 \downarrow (n-2:n)\)} (3,1)
\draw [->] (ms) -- node[above] {\(d\rightarrow (n-1:n)\)} (3,1)
\draw [->] (ms) -- node[above] {\(c_1 \downarrow (1:n)\)} (5,1)
\draw [->] (ms) -- node[above] {\(c_n \downarrow (n-2:n)\)} (7,1)
\draw [->] (ms) -- node[above] {\(0:n\)} (9,1)
\end{tikzpicture}
\end{center}

\textit{Figure 2.2.2 Multi-way Splitter \( MS_n \)}

The output streams of a \( n \)-way splitter can be recombined into the original stream by means of a multi-way merger, also called \( n \)-way merger. Next we show how to construct the \( n \)-way merger using several merge components.

\textbf{Definition 2.2.3 Multi-way merger}

Next the family \( \{MM_n\}_{n=1}^\infty \) of \( n \)-way mergers is defined by

\[
\begin{align*}
MM_1 &= \text{Buf} \\
MM_{n+1} &= \text{proc (in f_0, ..., f_n; out b)} \\
& \quad \parallel \text{chan e} \\
& \quad \parallel MM_n(f_0, ..., f_{n-1}, e) \\
& \quad \parallel \text{Merge}_{n-1}(e, f_n, b) \\
& \quad \}, 1 \leq n
\end{align*}
\]

Figure 2.2.3 shows a diagram for this component.
2.3 Construction methods

As indicated in the previous section we will use some special construction methods to create buffers. We have defined three basic components Buf, Split and Merge, and two composite components the multi-way splitter and the multi-way merger. For building buffers out of these components we will define three construction methods: serial composition, wagging and multi-wagging.

2.3.1 Serial composition

Definition 2.3.1 Serial composition

For any pair of components X and Y, each with one single input port and one single output port, we define the serial composite SER(X,Y) by

\[
SER(X,Y) = \text{proc} \ (\text{in} \ a; \text{out} \ b) \ [\ [ \text{chan} \ c \ | \ X(a,c) || Y(c,b) ] ]
\]

![Figure 2.3.1 SER(X,Y)](image)

Theorem 2.3.2

Let X and Y be buffers. Then the serial composition SER(X,Y) is also a buffer

Proof

Let a be the input port of X, b the output port of Y and c the channel connecting X and Y. Moreover let A, B and C be the corresponding streams of values. Since X is a buffer we have C = A (0 : 1). Since Y is a buffer we have B = C (0 : 1) = C. Hence we obtain B = C = A (0 : 1).

2.3.2 Wagging

To create components that have multiple paths from their inputs to their outputs, we define a technique, named wagging, that creates these paths.

Definition 2.3.3 Wagging

For any pair of components X and Y, each with one single input port and one single output port, we define the wagging composite \( WAG_{k,l}(X,Y) \) by
\[ \text{WAG}_{k,l} (X,Y) = \text{proc (in a; out b)} \]
\[
\| \text{chan c,d,e,f} \\
\| \text{Split}_{k,l} (a, c, d) \\
\| X(c, e) \\
\| Y(d, f) \\
\| \text{Merge}_{k,l} (e, f, b) \\
\| , 0 \leq k < l
\]

Figure 2.3.2 shows a diagram for this construction.

\[\text{Figure 2.3.2 Wagging WAG}_{k,l} (X,Y)\]

**Theorem 2.3.4**

Let \(X\) and \(Y\) be buffers. Then the wagging composite \(\text{WAG}_{k,l} (X,Y)\) is also a buffer.

**Proof**

Let \(a\) be the input port of the basic split component. Let \(b\) be the output port of the basic merge component. Let \(c\) be the channel connecting the split component and \(X\). Let \(e\) be the channel connecting \(X\) and the merge component. Let \(d\) be the channel connecting the split component and \(Y\). Let \(f\) be the channel connecting \(Y\) and the merge component. Moreover let \(A, B, C, D, E\) and \(F\) be the corresponding streams of values. The split component divides the substream \(A\) into two substreams \(C\) and \(E\). The substreams are specified by \(C = A (k : l)\) and \(D = A(k : l)\). Since \(X\) and \(Y\) are buffers we have \(E = C (0 : 1)\) and \(F = D (0 : 1)\). So for substreams \(E\) and \(F\) we obtain \(E = C (0 : 1) = C = A (k : l)\) and \(F = D (0 : 1) = D = A (k : l)\). The merge component recombines the two substreams \(E\) and \(F\) into stream \(B\). Hence the substream \(B\) is specified by \(B = A (0 : 1)\).

The wagging composition divides the input stream in two substreams. In addition we also occasionally like to divide the substreams in more than two substreams. Therefore we define a composition technique named multi-wagging where the input stream is divided into \(n\) substreams.
2.3.3 Multi-Wagging

Definition 2.3.5 Multi-Wagging

For any set of components \(\{X_i\}_{i=0}^{n-1}\) we define the multi-wagging composite \(MW_n(X_0, \ldots, X_{n-1})\) by

\[
MW_n(X_0, \ldots, X_{n-1}) = \text{proc} (\text{in} \ a; \text{out} \ b)
\]

\[
[ \text{chan} \ c_0, \ldots, c_n
\]

\[
| \text{MS}_n (a, c_{n-1}, \ldots, c_0)
\]

\[
| X_{n-1} (c_{n-1}, f_{n-1}) \| \ldots \| X_0 (c_0, f_0)
\]

\[
| \text{MM}_n (f_{n-1}, \ldots, f_0, b)
\]

Figure 2.3.3 shows a diagram for this construction.

![Diagram of Multi-Wagging](image)

Figure 2.3.3 Multi-wagging \(MW_n(X_0, \ldots, X_{n-1})\)

Theorem 2.3.6

Let \(X_0, \ldots, X_{n-1}\) be buffers. Then the multi-wagging composite \(MW_n(X_0, \ldots, X_{n-1})\) is also a buffer.

Proof

Let \(a\) be the input port of the multi-way splitter and \(b\) the output port of the multi-way merger. Let \(c_0, \ldots, c_{n-1}\) the channels connecting the multi-way splitter and the buffers \(X_0, \ldots, X_{n-1}\) and \(f_0, \ldots, f_{n-1}\) the channels connecting the buffers \(X_0, \ldots, X_{n-1}\) and the multi-way merger. Moreover let \(A, C_0, \ldots, C_{n-1}, F_0, \ldots, F_{n-1}\) and \(B\) the corresponding streams of values. The multi-way splitter divides the substream \(A\) into \(n\) substreams \(C_0, \ldots, C_{n-1}\). These substreams are specified by \(C_i = A(i : n), 0 \leq i < n\). Since \(X_0, \ldots, X_{n-1}\) are buffers we have \(F_i = C_i(0 : 1)\). So for substreams \(F_0, \ldots, F_{n-1}\) we obtain \(F_i = C_i(0 : 1) = C_i = A(i : n), 0 \leq i < n\). The multi-way merger recombines the \(n\) substreams \(F_0, \ldots, F_{n-1}\) into stream \(B\). So the substream \(B\) is specified by \(B = A(0:1)\).

The definition of these three construction methods raises the following question: When starting with the basic component \(Buf\) using the three construction methods, are we able to construct the set \(BUF\) of all buffers?
The answer is no, which can be seen from the following counter example. Consider figure 2.3.4 where a diagram is given of a Split-Merge buffer that is constructed using the wagging construction. However the wagging composition is taken of the "empty" components X and Y.

![Figure 2.3.4 A Split-Merge Buffer](image)

So if we add the empty buffer to the collection of buffers that can be used for construction, can we construct all possible buffers? Again, we have to answer this question negative. Consider figure 2.3.5 where a diagram is given of a component that is composed using two Odd-Even Shuffles. A split component can divide a input stream into two substreams that can be taken as the two input streams of the Double Odd-Even Shuffle component. Using a merge component the two output streams of the Double Odd-Even Shuffle can be recombined into one output stream. So we have a buffer that is constructed using two basic components Split and Merge, and the Double Odd-Even Shuffle component. However, there is no method to define the constructed buffer as a wagging composition.

![Figure 2.3.5 A Double Odd-Even Shuffle](image)

We have shown that not all buffers can be created using the construction methods. In section 2.8 we shall look at some of these buffer classes in more detail.

### 2.4 Buffer families

In the preceding sections we have given the basic components and construction methods. We have also identified the set of all buffers $BUF$. In this section we define some families of buffers, each a subset of the class of all buffers $BUF$. These families are nested and allow increasingly more complicated buffer designs. Therefore one may hope by going from one family to the next to obtain buffers with increasing performance. In chapter 4 we will define our optimality criterion and we will identify the optimal buffers within each of these families.

First we define the family of basic components $\{ C_i \}_{i=1}^{10} = \{ \text{Buf} \}$ by

$$ C_1 = \{ \text{Buf} \} $$

$$ C_{i+1} = C_i \cup \{ \text{Split}_{i,k} \mid 0 \leq k < i \} \cup \{ \text{Merge}_{i,k} \mid 0 \leq k < i \} $$
Example:

\[ C_1 = \{ \text{Buf} \} \]
\[ C_2 = C_1 \cup \{ \text{Split}_{0.2} \} \cup \{ \text{Merge}_{0.2} \} \cup \{ \text{Split}_{1.2} \} \cup \{ \text{Merge}_{1.2} \} \]
\[ C_3 = C_2 \cup \{ \text{Split}_{0.3} \} \cup \{ \text{Merge}_{0.3} \} \cup \{ \text{Split}_{1.3} \} \cup \{ \text{Merge}_{1.3} \} \cup \{ \text{Split}_{2.3} \} \cup \{ \text{Merge}_{2.3} \} \]

Furthermore, we define the family of construction methods \( \{ M_i \}_{i=1}^{\infty} \) by

\[ M_i = \{ \text{SER} \} \]
\[ M_i = M_{i-1} \cup \{ \text{WAG}_{k,i} | 0 \leq k < i \} \cup \{ \text{MW}_i | 2 \leq i \}, 1 \leq i \]

Example:

\[ M_1 = \{ \text{SER} \} \]
\[ M_2 = M_1 \cup \{ \text{WAG}_{0.2} \} \cup \{ \text{WAG}_{1.2} \} \cup \{ \text{MW}_2 \} \]
\[ M_3 = M_2 \cup \{ \text{WAG}_{0.3} \} \cup \{ \text{WAG}_{1.3} \} \cup \{ \text{WAG}_{2.3} \} \cup \{ \text{MW}_3 \} \]

Note that in the set of construction methods \( M_2 \) the construction method multi-wagging \( \text{MW}_2 \) the same construction method yields as \( \text{WAG}_{2.2} \).

Let \( B_i \) be the set of all buffers that can be constructed using the set of basic components \( C_i \) and using the set of construction methods \( M_i \) to be precise:

The set of buffers \( B_i \) is the smallest set such that

1. \( \text{Buf} \in B_i \)
2. If \( X_0, X_1 \in B_i \) then \( \text{SER}(X_0, X_1) \in B_i \)
3. If \( X_0, X_1 \in B_i \) and \( 0 \leq k < l \leq i \) then \( \text{WAG}_{k,i}(X_0, X_1) \in B_i \)
4. If \( X_0, X_1, \ldots, X_{n-1} \in B_i \) and \( n \leq i \) then \( \text{MW}_n(X_0, X_1, \ldots, X_{n-1}) \in B_i \)

As a consequence \( (\forall i: 1 \leq i: B_i \subseteq B_{i+1}) \).

Define \( BC \) as the union of all set of buffers of the family of \( B_i \) from \( i=1 \) to \( \infty \):

\[ BC = \bigcup_{i=1}^{\infty} B_i \]

### 2.5 Standard Buffers

On the one hand we have defined the set \( \text{BUF} \) of all buffers, on the other hand we have seen that the composition methods, when applied to buffers, again yield a buffer. In this section we will use these construction methods to specify special subclasses of \( \text{BUF} \).

We start with the family of linear buffers.

#### 2.5.1 Linear Buffers

**Definition 2.5.1 Linear buffer**

The family of linear buffers \( \{ \text{LBuf}_n \}_{n=1}^{\infty} \) is defined by
2.5 Standard buffers

\[\text{LBuf}_n = \text{Buf} \]
\[\text{LBuf}_{n+1} = \text{SER(Buf, LBuf}_n), 1 \leq n\]

Figure 2.5.1 shows a diagram for the linear buffer LBuf

![Figure 2.5.1 Linear buffer LBuf](image)

The construction of the linear buffer reveals that there is only one path through the buffer.

2.5.2 Binary Tree buffers

**Definition 2.5.2 Binary - Tree buffer**

The family of binary - tree buffers \(\{\text{BBuf}_n\}_{n=1}^{\infty}\) is defined by

\[\text{BBuf}_1 = \text{Buf}\]
\[\text{BBuf}_{n+1} = \text{WAG}_{0,2}(\text{BBuf}_n, \text{BBuf}_n), 1 \leq n\]
\[\text{proc (in a; out b)}\]
\[\text{[ chan c,d,e,f} \]
\[\text{ Split}_{0,2}(a,c,d) \| \text{BBuf}_n(c,e) \| \text{BBuf}_n(d,f) \| \text{Merge}_{0,2}(e,f,b)\]
\[\text{]}\], \(1 \leq n\)

Figure 2.5.2 shows a diagram for the binary buffer BBuf

![Figure 2.5.2 Binary - Tree BBuf](image)

The construction of the binary tree buffer BBuf reveals that the number of paths through the buffer equals \(2^{n-1}\), because all split and merge components have fan-out respectively fan-in 2. A binary tree buffer BBuf is constructed using \(2^{n-1}\) Buf, \(2^{n-1}-1\) split and \(2^{n-1}-1\) merge components.

Finally, we also define the family of rectangular buffers.
2.5.3 Rectangular buffers

**Definition 2.5.3 Rectangular buffer**

The family of rectangular buffers \( \{ \text{RBuf}_{m,n} \}_{m=2}^{\infty}, n=1 \to \infty \) is defined by

\[
\text{RBuf}_{m,n} = \text{proc (in a; out b)} \\
\langle \text{chan } c_0,\ldots, c_{n-1}, f_0,\ldots, f_{n-1} \\
\text{MS}_n(a,c_0,\ldots,c_n) \\
\text{|| LBuf}_{m2}(c, n-1, f_{n-1}) \text{ || } \ldots \text{ || LBuf}_{m2}(c_0, f_0) \\
\text{|| MM}_n(f_{n-1},\ldots,f_0,b) \\
\rangle, 2 \leq m, 1 \leq n
\]

Note that \( \text{RBuf}_{m,n} = MW_n(LBuf_{m=2}, \ldots, LBuf_{m=2}) \)

Figure 2.5.3 shows a diagram for the rectangular buffer \( \text{RBuf}_{m,n} \).

![Diagram of rectangular buffer](image)

*Figure 2.5.3 Rectangular buffer \( \text{RBuf}_{m,n} \)*

2.5.4 Square buffers

A special case of a rectangular buffer is a buffer where \( m = n \). For obvious reasons these buffers are called square buffers.

**Definition 2.5.4 Square buffer**

The family of square buffers \( \{ \text{SBuf}_n \}_{n=2}^{\infty} \) is defined by

\[
\text{SBuf}_n = \text{RBuf}_{0,n}, 2 \leq n
\]

2.6 The capacity of a buffer

The capacity of a buffer is the sum of the number of variables of each component. Every basic component has exactly one variable. As a consequence the capacity of a buffer consisting of its basic components equals the number of basic components, the capacity is additive. In the sequel capacity will be denoted with the Greek letter \( \kappa \).
2.6.1 capacity and serial composition

Theorem 2.6.1 The capacity of a serial composite

Let $X$, $Y$ be buffers of capacity $k_X$, $k_Y$. Then the serial composite $SER(X,Y)$ is a buffer with capacity equal to the sum of the capacities of the buffers $X$ and $Y$.

$$k_{SER(X,Y)} = k_X + k_Y$$

Proof

The number of basic components in the serial composite $SER(X,Y)$ equals the sum of basic components of buffers $X$ and $Y$.

2.6.2 capacity and wagging

Theorem 2.6.2 The capacity of a wagging composite

Let $X, Y$ be buffers of capacity $k_X$, $k_Y$. Then the wagging composite $WAG(X,Y)$ is a buffer with capacity equal to the sum of the capacities of the buffers $X$ and $Y$ plus 2.

$$k_{WAG(X,Y)} = k_X + k_Y + 2$$

Proof

The number of basic components in the wagging composite $WAG(X,Y)$ equals the sum of basic components of buffers $X$ and $Y$, plus the basic components Split and Merge.

2.6.3 capacity and multi-wagging

Theorem 2.6.3 The capacity of a multi-wagging composite

Let $X_0, \ldots, X_{n-1}$ be buffers with capacity $k_{X_0}, \ldots, k_{X_{n-1}}$. Then the multi-wagging composite $MW(X_0, \ldots, X_{n-1})$ is a buffer with capacity equal to the sum of the capacities of the buffers $X_0, \ldots, X_{n-1}$ plus 2n.

$$k_{MW(X_0, \ldots, X_{n-1})} = k_{X_0} + \ldots + k_{X_{n-1}} + 2n$$

Proof

The number of basic components in the serial composite $SER(X,Y)$ equals the sum of basic components of the buffers $X_0, \ldots, X_{n-1}$, plus the basic components in the multi-splitter $MS_n$ and the multi-merger $MM_n$ Merge.

In the previous section we have defined some buffer families. The capacities of these buffer families are given in table 2.6.1.
Table 2.6.1. capacities of some standard buffer families

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Capacity κ</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBufₙ</td>
<td>n</td>
</tr>
<tr>
<td>BBufₙ</td>
<td>3\cdot2^{n-1} - 2</td>
</tr>
<tr>
<td>RBufₘₙ</td>
<td>m•n</td>
</tr>
</tbody>
</table>

As an example consider the LongShortRun buffer \( WAG_{0,2}(RBuf₃,₃, SER(LBuf₈, BBuf₂)) \). The LongShortRun buffer is the wagging composite of the rectangular buffer \( RBuf₃,₃ \) and a serial composite of a linear buffer \( LBuf₈ \) and a binary tree buffer \( BBuf₂ \). The LongShortRun buffer is characterised by the fact that it has distinct path lengths through its subcomponents. The path on the left side through the buffer, via the subcomponent with the rectangular buffer is short compared to the path on the right side. Figure 2.6.1 shows a diagram for this buffer.

![Diagram](image)

*Figure 2.6.1 The LongShortRun buffer \( WAG_{0,2}(RBuf₃,₃, SER(LBuf₈, BBuf₂)) \)*
De capacity of the LongShortRun buffer is given by:

\[
\begin{align*}
K_{WAG} &= K_{WAG(X,Y)} = K_X + K_Y + 2 \\
K_{SER} &= K_{SER(LBu,BBu)} + K_{RBu,BBu} + 2 \\
K_{LBu} + K_{BBu} + K_{RBu,BBu} + 2 \\
&= \{ \text{def capacity} \} \\
&= 6 + 4 + 2 + 2 \\
&= 12
\end{align*}
\]

2.7 The I/O-distance of a buffer

The I/O-distance of a buffer is the minimal number of variables visited by any value on its passage through the buffer. In the sequel the I/O-distance will be denoted with the Greek letter \( \delta \).

For a buffer consisting of basis components, the I/O-distance is the number of basic components on the shortest path from its input port to its output port.

2.7.1 I/O-distance and serial composition

Theorem 2.7.1 The I/O-distance of a serial composite

Let \( X,Y \) be buffers with I/O-distance \( \delta_X, \delta_Y \). Then the serial composite \( \text{SER}(X,Y) \) is a buffer with I/O-distance equal to the sum of the I/O-distances of the buffers \( X \) and \( Y \)

\[
\delta_{\text{SER}(X,Y)} = \delta_X + \delta_Y
\]

Proof

The number of basic components on the shortest path from its input port to its output port in the serial composite \( \text{SER}(X,Y) \) equals the sum of the number basic components on the shortest path from \( X \)'s input port to \( X \)'s output port and the number basic components on the shortest path from \( Y \)'s input port to \( Y \)'s output port.

2.7.2 I/O-distance and wagging

Theorem 2.7.2 The I/O-distance of a wagging composite

Let \( X,Y \) be buffers with I/O-distance \( \delta_X, \delta_Y \). Then the wagging composite \( \text{WAG}(X,Y) \) is a buffer with I/O-distance equal to the minimum of the I/O-distances of the buffers \( X \) and \( Y \) plus 2.
\[ \delta_{WAG(X,Y)} = \text{MIN}(\delta_X, \delta_Y) + 2 \]

**Proof**

The number of basic components on the shortest path from its input port to its output port in the wagging composite WAG \((X,Y)\) equals the minimum of the number basic components on the shortest path from \(X\)'s input port to \(X\)'s output port and the number basic components on the shortest path from \(Y\)'s input port to \(Y\)'s output port plus the basic components Split and Merge.

### 2.7.3 I/O-distance and multi-wagging

**Theorem 2.7.3** The I/O-distance of a multi-wagging composite

Let \(X_0, \ldots, X_n\) be buffers with i/o-distance \((\delta_{X_0}, \ldots, \delta_{X_n})\). Then the multi-wagging composite \(MW(X_0, \ldots, X_n)\) is a buffer with i/o-distance equal to the minimum of the i/o-distances of the buffers \(X_0, \ldots, X_n\) plus \(n+1\).

\[ \delta_{MW(X_0, \ldots, X_n)} = \text{MIN}(\delta_{X_0}, \ldots, \delta_{X_n}) + n + 1 \]

**Proof**

The number of basic components on the shortest path from its input port to its output port in the multi-wagging composite \(MW(X_0, \ldots, X_n)\) equals for all \(i, 0 \leq i < n\), the minimum of the number basic components on the shortest path from \(X_i\)'s input port to \(X_i\)'s output port plus the number of basic components on the shortest path in the multi-splitter MS\(_n\) and the multi-merger MM\(_n\) Merge.

The I/O-distance of some standard buffer families are given in table 2.7.1

<table>
<thead>
<tr>
<th>Buffer</th>
<th>I/O-distance (\delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBuf(_n)</td>
<td>(N)</td>
</tr>
<tr>
<td>BBuf(_n)</td>
<td>(2n - 1)</td>
</tr>
<tr>
<td>RBuf(_m,n)</td>
<td>(m + n - 1)</td>
</tr>
</tbody>
</table>

*Table 2.7.1, I/O-distances of some standard buffer families*

As an example again consider the LongShortRun buffer \(WAG_{0.2}(RBuf_{3.3}, \text{SER}(LBuf_6, BBuf_2))\) of figure 2.6.1 The i/o-distance of the LongShortRun buffer is given by:

\[
\delta_{WAG_{0.2}(RBuf_{3.3}, \text{SER}(LBuf_6, BBuf_2))} = \{ \text{MIN}(\delta_X, \delta_Y) + 2 \}
\]

\[\text{MIN}(\delta_{\text{SER}(LBuf_6, BBuf_2)}, \delta_{RBuf_{3.3}}) + 2\]

\[= \{ \text{MIN}(\delta_{X}, \delta_{Y}) = 2 \}
\]

\[= \{ \text{MIN}(\delta_{LBuf_6} + \delta_{BBuf_2}), \delta_{RBuf_{3.3}}) + 2 \]

\[= \{ \text{MIN}(6 + 3, 5) + 2 \]

\[= \{ \text{calc} \}
\]

7
Note that the length of the path that a value traverses through the buffer is not for every value the same. One half of the input stream travels through the rectangular buffer RBuf\(_3,5\) with path length 7, the other half travels through the serial buffer SER(LBuf\(_6,8\), BBuf\(_2\)). In the future we will define the average path length of a buffer.

2.8 Limitations of construction methods

In the previous sections we have described a limited set of construction methods and a limited set of basic components. The basic components have a maximum fan-in and fan-out of 2. Extending the set of basic components with components having a fan-in and fan-out of more than 2 extends the set of buffers that can be constructed. Similarly extending the construction methods with parallel composition extends the set of buffers that can be constructed. An example of a buffer that cannot be constructed using serial composition, wagging and/or multi-wagging is a component similar to the baroque buffer ([Mak]). This buffer is given in the figure of diagram 2.8.1.

![Diagram 2.8.1 the baroque buffer](image)

The baroque buffer cannot be designed using serial composition, wagging and multi-wagging. In the figure of diagram 2.8.1 the basic component Split\(_{1,3}\) divides the streams into two substreams. However these two substreams are not recombined together in a merge component. So this buffer cannot be constructed using wagging.

For the time being we limit our interest to the set of buffers that can be constructed using the set of construction methods serial composition, wagging and multi-wagging and the set of basic components Buf, Split and Merge.

Finally note that how the internal streams are divided and recombined is determined by the structural design of the buffer. Its communication behavior is independent of the values it sends and receives. Hence the processes we discuss are data independent.[Zwa]
3 Communication behavior

In the previous chapter we have given some methods to construct large systems out of basic components. There however, we have only looked at structural properties and functional correctness. In addition to that, the order in which a system performs, its internal and external communications, has to be specified. To provide this ordering we will define the basic components by means of a program text. Furthermore, we will use program texts to show the functional (partial) correctness of various buffer designs. The program texts of the basic components will show that those basic components have a total ordering. The events are sequential ordered. Constructed systems however have a partial ordering. Not all the components of a constructed system are related. The partial ordering of a constructed system can not easily be shown using program texts. Therefore, to describe the communication behavior of a constructed system we introduce the formalism of a sequence function. The sequence function maps the events onto discrete time slots. Furthermore, we use the formalism of a sequence function both as means to establish absence of deadlock ([Zwa]) and more importantly as a vehicle for performance analysis. We define common performance parameters such as cycle time, latency and occupancy in terms of these sequence functions, and show that they satisfy a simple relationship. This in turn enables us to establish lower and upper bounds for the latency and occupancy in terms of two static properties of a buffer design: the capacity and the i/o-distance.

3.1 Program text

A parallel program is a collection of communicating processes. The processes communicate via input ports and output ports. Processes can be defined by means of program texts. Consider the following program for the basic component Split0,2

\[ Split_{0,2} = \text{proc}( \text{in } a; \text{out } b, c) \]
\[ \left[ \text{var } x ; (a?x ; b!x ; a?x ; c!x)^* \right] \]

where the program header (\text{in } a; \text{out } b, c) denotes that component Split0,2 has one input port a and two output ports b and c. Here, the variable declaration var x declares an internal variable x of the component, and the command \((a?x ; b!x ; a?x ; c!x)^*\) defines the possible behaviors of the process. The asterisk expresses repetition. The part that is repeated (forever) consists of an input statement followed by an output statement followed by an input statement and again followed by an output statement. Input statement a?x denotes the receipt of a value via port a, and the assignment of that value to variable x. Output statements b!x and c!x denote the sending of the value of variable x via ports b and c respectively.

Recall that component Split0,2 satisfies specification B = A(0 : 2) and C= A(1 : 2). This can be made explicit by adding i as 'ghost variable' to the program text:

\[ \left[ x : \text{data} ; i: \text{int} ; i:=0 ; (a\#i)?x ; (b\#i)!x ; (a\#i)?x ; (c\#i)!x ; i:=i+1^* \right] \]

The program text shows that the events are sequential ordered and related. This can be shown by the input/output relation of the component Split0,2.

Let \(a\#i\) denote the i-th communication event along port a. Let \(a(i)\) denote the value communicated in that event. Let \(b(i)\) and \(c(i)\) be defined analogously. Then the input/output relation of the component Split0,2 is given by

\[ b(i) = a(2i) \quad , 0 \leq i \]
\[ c(i) = a(2i+1) \quad , 0 \leq i \]

This input/output relation defines the same relation as specification B = A(0 : 2) and C= A(1 : 2).
For the basic components the general communication behaviors are given by the following program texts:

\[
\text{Buf} = \text{proc } (\text{in } a; \text{out } b) [\text{var } x; (a?x;b!x)^*]
\]

\[
\text{Split}_{k,l} = \text{proc } (\text{in } a; \text{out } c,d) [\text{var } x; ((a?x;d!x)^k; (a?x;c!x); (a?x;d!x)^{l-k})^*]
\]

\[
\text{Merge}_{k,l} = \text{proc } (\text{in } e,f; \text{out } b) [\text{var } x; ((f?x;b!x)^k; (e?x; b!x); (f?x; b!x)^{l-k})^*]
\]

As the same as we have shown for \(\text{Split}_{0,2}\) that the program text for \(\text{Split}_{0,2}\) satisfies specification \(B = A(0 : 2)\) and \(C = A(1 : 2)\), it can be shown that the program text for the basic components \(\text{Buf}\) satisfies specification \(B = A(0 : 1) = A\). Furthermore \(\text{Split}_{k,l}\) satisfies specification \(C = A(k : l)\) and \(D = A(k : l)\), and component \(\text{Merge}_{k,l}\) satisfies specification \(E = B(k : l)\) and \(F = B(k : l)\).

### 3.2 Sequence functions

In the previous section we have defined the total ordering of the events in which basic components can participate by means of program texts. A program text contains a block of sequential statements. The next step is to add parallelism to this sequential formalism. In an effort to add parallelism to the traditional sequential program languages, Hoare [Hoa] developed Communicating Sequential Processes (CSP). The behavior of a parallel system can be described in CSP terms. Here, we introduce a CSP-like formalism to describe the communication behavior of a parallel system. We introduce the formalism of a sequence function.

A sequence function is defined by:

A sequence function is a function that maps all events, both internal and external, of a parallel system onto the natural numbers, which are interpreted as a discrete time domain.

In the sequel sequence functions will be denoted with the Greek letter \(\sigma\).

#### 3.2.1 Sequence functions for the basic components

Consider a parallel system with a port \(a\) and let \(\sigma\) be a sequence function for that system. Then \(\sigma(a#i)\) denotes the time slot in which event \(a#i\) is scheduled, where \(a#i\) is the \(i\)-th communication event along port \(a\). Note that time slots are numbered from 0 upwards.

Sequence functions must respect the ordering of events as specified by the sequencing operators in the program text prescribing the communication behavior.

As an example consider the basic component \(\text{Split}_{0,2}\). A sequence function \(\sigma\) for this component is:

\[
\begin{align*}
\sigma(a#i) &= 2i \\
\sigma(b#i) &= 2i+1 \\
\sigma(c#i) &= 2i+3
\end{align*}
\]
This function can also be shown in a schedule. The schedule for this sequence function $\sigma$ for $\text{Split}_{0,2}$ is given by:

<table>
<thead>
<tr>
<th>Component</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Split}_{0,2}$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>......</td>
</tr>
</tbody>
</table>

Table 3.2.1 Schedule for $\text{Split}_{0,2}$

A set of sequence functions for the basic components is given in table 3.2.2. We denote the sequence functions for the basic components $\text{Buf}$, $\text{Split}_{k,1}$ and $\text{Merge}_{k,1}$ respectively with $\sigma_{\text{Buf}}$, $\sigma_{\text{Split}_{k,1}}$, and $\sigma_{\text{Merge}_{k,1}}$.

| Basic component | $a|t_i$ | $e|t_i$ | $d|(i-1)|i+j$, $0 \leq j < 1$ | $e|t_i$ | $f|(i-1)|j$, $0 \leq j < 1$ | $b|t_i$ |
|-----------------|-------|-------|----------------|-------|----------------|-------|
| $\text{Buf}$    | 2i    |       |                |       |                | 2i+1  |
| $\text{Split}_{k,1}$ | 2i   | (2i)+2k+1 | (2i)+2j+1, $0 \leq j < k$ | (2i)+2(j+1)+1, $k \leq j < 1$ |       |       |
| $\text{Merge}_{k,1}$ |       |       | (2i)+2k     | (2i)+2j, $0 \leq j < k$ | (2i)+2(j+1), $k \leq j < 1$ | 2i+1  |

Table 3.2.2 Sequence functions for the basic components

Note that parallel systems composed of basic components lack internal choice. Therefore the existence of a sequence function for such a parallel system implies the absence of (total) deadlock, or in other words, ensures total correctness [Zwa].

3.2.2 Sequence functions for parallel systems

So far we have defined sequence functions for the basic components. We need to define sequence functions for systems composed of basic components. First we show how to get a sequence function for a buffer by using diagrams. In the next section we introduce a calculus for sequence functions and show how to use this calculus to construct a sequence function of a composite system out of the sequence functions of its basic components.

As an example consider the rectangular buffer $R\text{Buf}_{3,3}$ in figure 3.2.1 that consists of two split components, two merge components and five buffer components. The buffer receives its values on port $a$ of its $\text{Split}_{0,2}$ component. The basic sequence function of component $\text{Split}_{0,2}$ states that $\sigma_{\text{Split}_{0,2}}(a|t_i)=2i+j$. According to the sequence functions, the events on channel $e$ and $c$ are scheduled at $\sigma_{\text{Split}_{0,2}}(c|t_i)=6i+2|j+1$ and $\sigma_{\text{Split}_{0,2}}(e|t_i)=6i+5$. Next consider the right-hand side neighbor of component $\text{Split}_{0,2}$, i.e. component $\text{Split}_{1,2}$. It receives its input values at time slot $6i+1$ and at time slot $6i+3$. Therefore it must output the value received at time slot $6i+1$ at the next time slot $6i+2$. So the events for internal channel $d$ are scheduled at $\sigma_{\text{Split}_{1,2}}(d|t_i)=6i+2$. For the output stream on channel $f$ we have a choice. The component $\text{Split}_{1,2}$ receives the next value from channel $c$ at time slot $6i+1+1$. Therefore the component $\text{Split}_{1,2}$ must have sent the last value it has received from channel $c$ before time slot $6i+7$. So component $\text{Split}_{1,2}$ may output its value on channel $f$ at time slots $6i+4$, $6i+5$, and $6i+6$. We have chosen to output a value immediately after it has been received. Hence the events on internal channel $f$ are scheduled by $\sigma_{\text{Split}_{1,2}}(f|t_i)=6i+4$. The scheduling of the remaining events follow from similar reasoning.

Note that in the above construction of a sequence function for buffer $R\text{Buf}_{3,3}$ we consistently have chosen to schedule an event at the earliest time slot possible whenever there was a range of possible time slots to choose from. The resulting sequence functions are shown in figure 3.2.1.
As already mentioned subcomponent Split_{1,2} can postpone its output on channel f. This can easily be shown in the schedule for the rectangular buffer RBuf_{3,3} in Table 3.2.3. This table shows that component Split_{1,2} outputs a value at time slot 4 on channel f. However subcomponent Split_{1,2} receives a value from channel c at time slot 7. So the output of the value on channel f could be postponed until time slot 6. In the next section we will construct sequence functions where the output of values are postponed.

<table>
<thead>
<tr>
<th>time slot</th>
<th>Split_{1,2}</th>
<th>Split_{2,3}</th>
<th>Buf_{11}</th>
<th>Buf_{12}</th>
<th>Buf_{21}</th>
<th>Buf_{22}</th>
<th>Buf_{41}</th>
<th>Buf_{42}</th>
<th>Merge_{2,2}</th>
<th>Merge_{2,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>m</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k</td>
<td>b</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>e</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>a</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>m</td>
</tr>
<tr>
<td>11</td>
<td>e</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>a</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2.3 A schedule for RBuf_{3,3}

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>\sigma(#i) = 2i</td>
<td>\sigma(#i) = 6i+5</td>
<td>\sigma(#i) = 6i+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\sigma(#2i) = 6i+1</td>
<td>\sigma(#2i) = 6i+4</td>
<td>\sigma(#2i) = 6i+7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\sigma(#(2i+1)) = 6i+3</td>
<td>\sigma(#(2i+1)) = 6i+3</td>
<td>\sigma(#(2i+1)) = 6i+6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\sigma(#i) = 6i+2</td>
<td>\sigma(#i) = 6i+6</td>
<td>\sigma(#(2i+1)) = 6i+8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\sigma(#2i) = 2i+5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The sequence functions above are notated in the RBuf_{3,3} buffer in picture 3.2.1.
3.2.3 Early and late sequence functions

In the previous section we have in the construction of the sequence function consistently chosen to schedule an event into the earliest time slot possible whenever there was a range of possible time slots to choose from. A sequence function with the property that it outputs each data value immediately following the time slot at which that same data value was received is called an 'early sequence function'. Early sequence functions will be denoted by $\tilde{\sigma}$.

Similarly, a sequence function with the property that it outputs each data value directly before the time slot at which the next data value is received is called a 'late sequence function'. Late sequence functions will be denoted by $\tilde{\sigma}^\prime$.

Again consider the rectangular buffer $RBuf_{3,3}$. We construct the sequence functions as described in section 3.2.2. For the cases we have a range of possible sequence functions, we choose to output the value immediately before the next value will be received. These choices result in the late sequence function for the rectangular buffer $RBuf_{3,3}$. A late schedule for this buffer is shown in table 3.2.4.

<table>
<thead>
<tr>
<th>time slot</th>
<th>$\text{Split}_{1,1}$</th>
<th>$\text{Split}_{1,2}$</th>
<th>$\text{Buf}_{0,3}$</th>
<th>$\text{Buf}_{0,3}$</th>
<th>$\text{Buf}_{0,3}$</th>
<th>$\text{Buf}_{0,3}$</th>
<th>$\text{Merge}_{0,3}$</th>
<th>$\text{Merge}_{0,3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>a</td>
<td></td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>e</td>
<td>e</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>a</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>a</td>
<td></td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>e</td>
<td>e</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>a</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 3.2.4 A late schedule for $RBuf_{3,3}$*
The late sequence function is denoted in figure 3.2.2

Figure 3.2.2 RBuf\_3,3 with late sequence function

In the previous section we have given a set of sequence functions for the basic components. For these sequence functions the number of time slots between successive input events and between two successive output events equals 2. Next we give a set of sequence functions for any given number of time slots (at least 2).

Let \( \gamma \) be the number of time slots between successive input events or two successive output events. For \( 2 \leq \gamma \) the early sequence functions for Buf, Split and Merge are given in table 3.2.5.

<table>
<thead>
<tr>
<th>Early sequence function</th>
<th>a/bi</th>
<th>b/bi</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{\sigma}_{Buf} )</td>
<td>( \gamma )</td>
<td>( \gamma + 1 )</td>
<td>( d(k(l-1)+j) , 0 \leq j &lt; 1-1 )</td>
</tr>
<tr>
<td>( \bar{\sigma}_{Split} )</td>
<td>( \gamma )</td>
<td>( \gamma + k\gamma + 1 )</td>
<td>( \gamma + j\gamma + 1 ) , ( 0 \leq j &lt; k )</td>
</tr>
<tr>
<td>( \bar{\sigma}_{Merge} )</td>
<td>( \gamma + k\gamma )</td>
<td></td>
<td>( \gamma + (j+1)\gamma + 1 ) , ( k \leq j &lt; 1-1 )</td>
</tr>
</tbody>
</table>

Table 3.2.5 Early sequence functions for the basic components

Similarly, for \( 2 \leq \gamma \) the late sequence functions for Buf, Split and Merge are given in table 3.2.6.

<table>
<thead>
<tr>
<th>Late sequence function</th>
<th>a/bi</th>
<th>b/bi</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{\sigma}_{Buf} )</td>
<td>( \gamma )</td>
<td>( \gamma + \gamma - 1 )</td>
<td></td>
</tr>
<tr>
<td>( \bar{\sigma}_{Split} )</td>
<td>( \gamma )</td>
<td>( \gamma + k\gamma + \gamma - 1 )</td>
<td>( \gamma + j\gamma + \gamma - 1 ) , ( 0 \leq j &lt; k )</td>
</tr>
<tr>
<td>( \bar{\sigma}_{Merge} )</td>
<td>( \gamma + k\gamma )</td>
<td></td>
<td>( \gamma + (j+1)\gamma + \gamma - 1 ) , ( k \leq j &lt; 1-1 )</td>
</tr>
</tbody>
</table>

Table 3.2.6 Late sequence functions for the basic components

The tables with the early and late sequence functions show that there must be a range of sequence functions for the basic components. We will define general sequence functions for the basic components.
To be more precise, let g be an integer such that $0 < g < \gamma$. Then for $2 \leq \gamma$ the general sequence functions for Buf, Split and Merge are given in table 3.2.7.

<table>
<thead>
<tr>
<th>Sequence function</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_{Buf}$</td>
<td>a#i</td>
</tr>
<tr>
<td></td>
<td>$y_i$</td>
</tr>
<tr>
<td>$\sigma_{Split,ij}$</td>
<td>a#i</td>
</tr>
<tr>
<td></td>
<td>$y_i$</td>
</tr>
<tr>
<td></td>
<td>$y_i$</td>
</tr>
<tr>
<td>$\sigma_{Merg,ij}$</td>
<td>c#i</td>
</tr>
<tr>
<td></td>
<td>$y_i + ky$</td>
</tr>
</tbody>
</table>

**Table 3.2.7 General sequence functions for the basic components**

Note that the early and late sequence functions are obtained by instantiating the general sequence function with $g = 1$ and $g = \gamma - 1$ respectively.

So far we derived general sequence functions for the basic components. We continue with the derivation of the early and late sequence functions for the multi way splitter $MS_n$ and the multi way merger $MM_n$. We start with the derivation of the sequence function for the multi way splitter $MS_n$. Definition 2.2.2 shows that the multi way splitter is a generalisation of the basic split component. Definition 2.2.3 shows that the multi way merger is a generalisation of the basic merge component. Using the early and late sequence functions for the split components and the merge components, we derive the following sequence functions for the multi way splitter $MS_n$ and the multi way merger $MM_n$.

Again, let $\gamma$ be the number of time slots between two successive input events a#i or two successive output events b#i. For $2 \leq \gamma$ the early sequence functions for multi way splitter $MS_n$ and the multi way merger $MM_n$ are given in table 3.2.8.

<table>
<thead>
<tr>
<th>Early sequence function</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_{MS,n}$</td>
<td>a#i</td>
</tr>
<tr>
<td>$y_i$</td>
<td>$y_i + (\gamma - 1)y_i + n$</td>
</tr>
<tr>
<td>$\tilde{\sigma}_{MS,n}$</td>
<td>f#i</td>
</tr>
<tr>
<td>$\tilde{n}_i$</td>
<td>$y_i + (\gamma - 1)$</td>
</tr>
</tbody>
</table>

**Table 3.2.8 Early sequence functions for the multi way components**

For $2 \leq \gamma$ the late sequence functions for multi way splitter $MS_n$ and the multi way merger $MM_n$ are given in table 3.2.9.

<table>
<thead>
<tr>
<th>Late sequence function</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_{Split,ij}$</td>
<td>a#i</td>
</tr>
<tr>
<td>$y_i$</td>
<td>$y_i + (n-p)(\gamma - 1) + n - 1$</td>
</tr>
<tr>
<td>$\tilde{\sigma}_{Split,ij}$</td>
<td>f#i</td>
</tr>
<tr>
<td>$\tilde{n}_i$</td>
<td>$y_i + (\gamma - 1)$</td>
</tr>
</tbody>
</table>

**Table 3.2.9 Late sequence functions for the multi way components**
Note that the sequence functions of the basic components and the multi-way components schedule an initial event at time slot 0. Sequence functions that schedule an initial event at time slot 0 are called strict. Hence every strict sequence function $\sigma$ for an initially empty buffer has the property that $\sigma(\text{id}, 0) = 0$.

### 3.2.4 Sequence functions and transformers.

We have shown an ad hoc construction of a sequence function. We now show that we can construct a sequence function for a composite system in a systematic way using the sequence functions of its components. To this end, we define a class of transformers. The role of a transformer is to obtain a new sequence function from an existing one. For sequence function $\sigma$ and transformer $\tau : \mathbb{N} \rightarrow \mathbb{N}$ we define, for any event $e$, the function $\sigma ; \tau (e) = \tau (\sigma (e))$. Just like we construct buffers from a set of basic components, we now construct their sequence functions from the basic sequence functions using a few basic transformers.

We define

- $u_k(i) = \begin{cases} \text{if } i < k \text{ then } i \text{ else } i + 1 \end{cases}$, $0 \leq k$
- $\chi_{x,m}(q_m + p) = q_{m+1} + p \text{ if } 0 \leq p < r \text{ then } 0 \text{ else } 1$
- $\mu_k(i) = i - k$
- $\alpha_{r,m}(i) = \begin{cases} \text{if } i \mod r = r \text{ then } i - 1 \text{ else } i \end{cases}$, $0 < r < m$
- $\pi_{r,m}(i) = \begin{cases} \text{if } i \mod r = m \text{ then } i + 1 \text{ else } i \end{cases}$, $0 \leq r < m - 1$

Transformer $\mu_k$ inserts a single empty time slot before time slot $k$. Transformer $\chi_{x,m}$ inserts a single empty time slot before every time slot $q_{m+r}$, for $0 \leq q$. Transformer $\mu_k$ inserts $k - 1$ time slots after each time slot. Transformer $\alpha_{r,m}$ advances all events that occur at time slot $q_{m+r}$, for $0 \leq q$, by a single time slot. Transformer $\pi_{r,m}$ postpones all events that occur at time slot $q_{m+r}$, for $0 \leq q$, by a single time slot.

From the definition of these transformers it follows that

- $\mu_k = \prod_{i=1}^{k} x_i$
- $\mu_k \cdot u_0^k = u_0 \cdot \mu_k$
- $\alpha_{r+1,m} \cdot u_0 = u_0 \cdot \alpha_{r,m}$
- $\pi_{r+1,m} \cdot u_0 = u_0 \cdot \pi_{r,m}$

Two sequence functions $\sigma_1$ and $\sigma_2$ are called matching when they are equal on their common domain, i.e.: 

$(\forall e. e \in \text{dom}(\sigma_1) \cap \text{dom}(\sigma_2): \sigma_1(e) = \sigma_2(e))$. Let $\sigma_X$ and $\sigma_Y$ be sequence functions for $X$ and $Y$. When $\sigma_X$ and $\sigma_Y$ match then $\sigma_X \cup \sigma_Y$ is a sequence function for $X \parallel Y$.

Next we derive a sequence function for the rectangular buffer $\text{RBuf}_{3,3}$. The sequence function $\sigma$ is given by $\sigma = \sigma_{\text{B}} \cup \sigma_{\text{B}_{1,2}} \cup \sigma_{\text{B}_{1,3}} \cup \sigma_{\text{B}_{2,2}} \cup \sigma_{\text{B}_{2,3}} \cup \sigma_{\text{B}_{4,3}} \cup \sigma_{\text{B}_{5,3}} \cup \sigma_{\text{M},0,2} \cup \sigma_{\text{M},0,3}$

where $\sigma_{\text{B}}, \sigma_{\text{B}_{k,i}}$ and $\sigma_{\text{M}_{k,i}}$ are the sequence functions of the subcomponents.

By construction the sequence functions of the subcomponents match. Therefore they define, when taken together, a sequence function for the buffer as a whole.

We define the sequence functions of the subcomponents as a generic sequence function of a basic component with successive application of a number of transformers.

The sequence function $\sigma_{\text{B}_{2,3}}$ for the split component Split$_{2,3}$ is an instance of the generic sequence function $\sigma_{\text{Split}_{2,3}}$. 
\[ \sigma_{5,2,3} = \sigma_{\text{Split}_{2,3}} \]

Channel \( c \) connects the split components \( \text{Split}_{2,3} \) and \( \text{Split}_{1,2} \). By construction the sequence functions \( \sigma_{5,1,2} \) and \( \sigma_{5,2,3} \) must be matching. Their common domain is channel \( c \).

\[
\sigma_{5,1,2}(c^#(2i+j)) = \sigma_{5,2,3}(c^#(2i+j))
\]
\[
= 6i+2j+1, \quad 0 \leq j < 2
\]
\[
\sigma_{5,1,2}(c^#(2i+j)) = (\sigma_{\text{Split}_{1,2}}; \chi_{4,4}; \chi_{5,5}; u_0)(c^#(2i+j))
\]
\[
= (\chi_{4,4}; \chi_{5,5}; u_0)(4i+2j), \quad 0 \leq j < 2
\]
\[
= (\chi_{5,5}; u_0)(5i+2j), \quad 0 \leq j < 2
\]
\[
= (u_0)(6i+2j), \quad 0 \leq j < 2
\]
\[
= 6i+2j+1, \quad 0 \leq j < 2
\]

A standard split has every two time slots an input event. The input events of \( \text{Split}_{1,2} \) are the output events of \( \text{Split}_{2,3} \). Every 6 time slots two events occur. So we have to insert two empty time slots into the standard sequence function \( \sigma_{5,2,3} \). Also a single empty time slot must be inserted before time slot 0.

\[ \sigma_{5,1,2} = \sigma_{\text{Split}_{1,2}}; \chi_{4,4}; \chi_{5,5}; u_0 \]

For the sequence function \( \sigma_{B2} \) of basic component buffer \( \text{Buf}(2) \) to be matching with \( \sigma_{5,2,3} \), the sequence functions must be equal for channel \( e \): \( \sigma_{B2}(e^#i) = \sigma_{5,2,3}(e^#i) = 6i+5 \). We successively apply the transformers \( u_0 \), \( \mu_3 \), \( \pi_{3,6} \) and \( \pi_{4,6} \) on the sequence function \( \sigma_{B2} \) for the generic basic component \( \text{Buf} \):

\[ \sigma_{B2}(e^#i) = (\sigma_{\text{Buf}}; u_0; \mu_3; \pi_{3,6}; \pi_{4,6})(e^#i) \]
\[ = (u_0; \mu_3; \pi_{3,6}; \pi_{4,6})(2i) \]
\[ = (\mu_3; \pi_{3,6}; \pi_{4,6})(2i+1) \]
\[ = (\pi_{3,6}; \pi_{4,6})(6i+3) \]
\[ = (\pi_{4,6})(6i+4) \]
\[ = 6i+5 \]

For the values on channel \( h \) the sequence function gives:

\[ \sigma_{B2}(h^#i) = (\sigma_{\text{Buf}}; u_0; \mu_3; \pi_{3,6}; \pi_{4,6})(h^#i) \]
\[ = (u_0; \mu_3; \pi_{3,6}; \pi_{4,6})(2i+1) \]
\[ = (\mu_3; \pi_{3,6}; \pi_{4,6})(2i+2) \]
\[ = (\pi_{3,6}; \pi_{4,6})(6i+6) \]
\[ = (\pi_{4,6})(6i+6) \]
\[ = 6i+6 \]

Note that the sequence function \( \sigma_{B2} \) also can be defined using the \( \alpha \) transformer

\[
\sigma_{B2} = \sigma_{\text{Buf}}; \mu_3; \alpha_{3,6}; \alpha_{3,6}; u_0^5
\]

\[
\sigma_{B2}(e^#i) = (\sigma_{\text{Buf}}; \mu_3; \alpha_{3,6}; \alpha_{3,6}; u_0^5)(e^#i)
\]
\[
= (\mu_3; \alpha_{3,6}; \alpha_{3,6}; u_0^5)(2i)
\]
\[
= (\alpha_{3,6}; \alpha_{3,6}; u_0^5)(6i)
\]
\[
= (u_0^5)(6i)
\]
\[
= 6i+5
\]

For the values on channel \( h \) the sequence function gives:

\[
\sigma_{B2}(h^#i) = (\sigma_{\text{Buf}}; \mu_3; \alpha_{3,6}; \alpha_{3,6}; u_0^5)(h^#i)
\]
\[
= (\mu_3; \alpha_{3,6}; \alpha_{3,6}; u_0^5)(2i+1)
\]
\[
= (\alpha_{3,6}; \alpha_{3,6}; u_0^5)(6i+3)
\]
\[
= (u_0^5)(6i+2)
\]
\[
= (u_0^5)(6i+1)
\]
\[
= 6i+6
\]
3.2 Sequence functions

The sequence functions for the remaining subcomponents can be derived analogously. The results are given in table 3.2.10.

<table>
<thead>
<tr>
<th>Subcomponent</th>
<th>Sequence function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split_{2,3}</td>
<td>(\sigma_{2,3}^{5} )</td>
</tr>
<tr>
<td>Split_{1,2}</td>
<td>(\sigma_{1,2}^{5} )</td>
</tr>
<tr>
<td>Buf_{(1)}</td>
<td>(\sigma_{B(1)}^{5} )</td>
</tr>
<tr>
<td>Buf_{(2)}</td>
<td>(\sigma_{B(2)}^{5} )</td>
</tr>
<tr>
<td>Buf_{(3)}</td>
<td>(\sigma_{B(3)}^{5} )</td>
</tr>
<tr>
<td>Buf_{(4)}</td>
<td>(\sigma_{B(4)}^{5} )</td>
</tr>
<tr>
<td>Merge_{0,2}</td>
<td>(\sigma_{M,0,2}^{5} )</td>
</tr>
<tr>
<td>Merge_{0,3}</td>
<td>(\sigma_{M,0,3}^{5} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subcomponent</th>
<th>Sequence function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split_{2,3}</td>
<td>(\sigma_{Split,2,3}^{5} )</td>
</tr>
<tr>
<td>BuF_{(1)}</td>
<td>(\sigma_{BuF_{(1)}}^{5} )</td>
</tr>
<tr>
<td>BuF_{(2)}</td>
<td>(\sigma_{BuF_{(2)}}^{5} )</td>
</tr>
<tr>
<td>BuF_{(3)}</td>
<td>(\sigma_{BuF_{(3)}}^{5} )</td>
</tr>
<tr>
<td>BuF_{(4)}</td>
<td>(\sigma_{BuF_{(4)}}^{5} )</td>
</tr>
<tr>
<td>Merge_{0,2}</td>
<td>(\sigma_{Merge_{0,2}}^{5} )</td>
</tr>
<tr>
<td>Merge_{0,3}</td>
<td>(\sigma_{Merge_{0,3}}^{5} )</td>
</tr>
</tbody>
</table>

Table 3.2.10 Sequence functions for the subcomponents of RBuf_{2,3}

So a sequence function for RBuf_{2,3} is given by

\[ \sigma = \sigma_{\text{Split}_{2,3}} \cup \sigma_{\text{Split}_{1,2}} \cup \sigma_{\text{BuF}_{(1)}} \cup \sigma_{\text{BuF}_{(2)}} \cup \sigma_{\text{BuF}_{(3)}} \cup \sigma_{\text{BuF}_{(4)}} \cup \sigma_{\text{Merge}_{0,2}} \cup \sigma_{\text{Merge}_{0,3}} \]

where the sequence functions for the subcomponents of RBuf_{2,3} are given in table 3.2.10.

3.3 Performance parameters

From the sequence functions of a buffer system we can derive various performance measures. In particular we will consider the average occupancy, which is a measure of the way the storage capacity of the buffer is used. Furthermore we consider the average latency, which is a measure of the time a value resides in a buffer. And finally we consider the average cycle time, which is a measure for the rate at which the buffer consumes and produces values. Ideally one would like to design buffers that have low cycle time, low latency and high occupancy. We start with some preliminary definitions. As we have seen components communicate with their environment through communication events on their ports. For an arbitrary system X of basic components we will distinguish

- \(pX\) the set of all internal and external ports of X
- \(iX\) the set of external input ports
- \(oX\) the set of external output ports
- \(cX\) the set of all internal ports also called channels

Let's take a look at the ports of the basic components

<table>
<thead>
<tr>
<th>Basic Component X</th>
<th>pX</th>
<th>iX</th>
<th>oX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buf</td>
<td>[a,b]</td>
<td>{a}</td>
<td>{b}</td>
</tr>
<tr>
<td>Split</td>
<td>[c,d,e]</td>
<td>{c}</td>
<td>{d,e}</td>
</tr>
<tr>
<td>Merge</td>
<td>[f,g,b]</td>
<td>{f,g}</td>
<td>{b}</td>
</tr>
</tbody>
</table>

Table 3.3.1 Ports of the Basic Components

Note that basic components don't have internal channels. So far the port sets of the basic components are defined. What remains to be defined are the port sets of a parallel composition.
Definition 3.3.1 Port of a parallel composition

For some systems X and Y the port sets of the parallel composition \( X \| Y \) are defined by

\[
\begin{align*}
p(X \| Y) &= p_X U p_Y \\
i(X \| Y) &= i_X o_Y U i_Y \circ o_X \\
o(X \| Y) &= o_X \circ i_Y U o_Y \circ i_X \\
e(X \| Y) &= c_X U c_Y U (o_X \cap i_Y) U (o_Y \cap i_X)
\end{align*}
\]

Furthermore note that with these definitions we can express a constraint on the parallel composition, that is the parallel composition \( X \| Y \) is only defined when \( i_X \cap i_Y = \emptyset \land o_X \cap o_Y = \emptyset \)

3.3.1 Cycle time and throughput

Definition 3.3.1.1 Individual cycle time at a port

For any given sequence function \( \sigma \) of a system \( X \) the \( i \)-th individual cycle time at port \( a, a \in p_X \), denoted as \( \gamma_i(\sigma,a) \), is defined as the number of time slots that pass between the successive input values \( i \) and \( i+1 \)

\[
\gamma_i(\sigma,a) = \sigma(a(\#(i+1))) - \sigma(a(\#(i))) \quad 0 \leq i
\]

Definition 3.3.1.2 Average cycle time at a port

For any sequence function \( \sigma \) of a system \( X \) the average cycle time at port \( a, a \in p_X \), denoted as \( \Gamma_a \), is defined as

\[
\Gamma_a(\sigma) = \lim_{i \to \infty} \frac{1}{i} \sum_{j=0}^{i-1} \gamma_j(\sigma,a)
\]

Note that the basic components are sequential. Their input and output events alternate. So the distance between two successive input events must be at least 2. Therefore we have the following property

Property 3.3.1.3 Average cycle time lower bound

For any sequence function \( \sigma \) of a system \( X \) the average cycle time at port \( a, a \in p_X \), \( \Gamma_a \) satisfies the inequality

\[
\Gamma_a(\sigma) \geq 2
\]

We have defined the average cycle time at a port as the average time between two successive input values. The inverse, that is the average number of input values per time slot, is also a meaningful quantity ans is defined as the throughput.

Definition 3.3.1.4 Average throughput at a port

For any sequence function \( \sigma \) of a system \( X \) the average throughput at port \( a, a \in p_X \), denoted as \( \Theta_a \), is defined as the inverse of the average cycle time \( \Gamma_a(\sigma) \)

\[
\Theta_a(\sigma) = \frac{1}{\Gamma_a(\sigma)}
\]
Definition 3.3.1.5 Average throughput

For any sequence function $\sigma$ of a system $X$ with input ports $iX$ and output ports $oX$ the *average throughput of system $X$*, denoted as $\Theta$, is defined as the sum of the average throughputs at the input ports of system $X$, which equals the sum of the average throughputs at the output ports of system $X$.

$$\Theta(\sigma) = \sum_{a \in iX} \Theta_a = \sum_{a \in oX} \Theta_a$$

Definition 3.3.1.6 Average cycle time

For any sequence function $\sigma$ of a system $X$ the *average cycle time*, denoted as $\Gamma$, is defined as the inverse of the average throughput of the system.

$$\Gamma(\sigma) = \frac{1}{\Theta(\sigma)}$$

Theorem 3.3.1.7

Let $\sigma$ a sequence function of a buffer $X$. Let $X$ have input ports $iX$ and output ports $oX$. Then the *average cycle time*, $\Gamma(\sigma)$ equals the average cycle time at both the input port and the output port of $X$.

$$\Gamma(\sigma) = \Gamma_a(\sigma) \quad a \in iX \cup oX$$

Proof

Let $X$ be a buffer with a given sequence function $\sigma$. Then by definition 3.3.1.5

$$\Gamma(\sigma) = \frac{1}{\Theta(\sigma)}$$

$$= \{ \text{calculus} \}$$

$$\frac{1}{\Gamma(\sigma)} = \Theta(\sigma)$$

$$\sum_{a \in iX} \Theta_a$$

$$\{ \text{definition 3.3.1.5} \}$$

$$\frac{1}{\Gamma(\sigma)} = \sum_{a \in iX} \Theta_a$$

$$\{ X \text{ is a buffer: } \# \{ a \in iX \} = 1 \}$$

$$\frac{1}{\Gamma(\sigma)} = \Theta_a$$

$$\{ \text{definition 3.3.1.4} \}$$

$$\frac{1}{\Gamma(\sigma)} = 1 / \Gamma_a(\sigma)$$

$$\{ \text{calculus} \}$$

$$\Gamma(\sigma) = \Gamma_a(\sigma)$$

$$a \in iX$$

The derivation for the output ports of buffer $X$ is similar. So equation $\Gamma(\sigma) = \Gamma_a(\sigma)$ holds.

Definition 3.3.1.8 Constant cycle time

For any sequence function $\sigma$, a buffer $X$ has *constant cycle time* when all its individual cycle times on the input and output ports are equal

$$\forall i, a: 0 \leq i \land a \in iX \cup oX: \gamma_{i+1}(\sigma, a) = \gamma_i(\sigma, a)$$
3.3.2 Occupancy and vacancy

**Definition 3.3.2.1 Instantaneous occupancy**

For any sequence function $\sigma$ of a system $X$ the *instantaneous occupancy at time slot* $t$, denoted as $\omega_t$, is defined as the number of values present in the system at time slot $t$

$$\omega_t(\sigma) = \#\{j|\sigma(a^tj) < t\} - \#\{j|\sigma(b^tj) < t\} \quad a \in iX, b \in oX, 0 \leq t$$

**Definition 3.3.2.2 Average occupancy**

For any sequence function $\sigma$ of a system the *average occupancy* of the system, denoted as $\Omega$, is defined as

$$\Omega(\sigma) = \lim_{i \to \infty} \frac{1}{i} \sum_{r=0}^{i-1} \omega_r(\sigma)$$

**Theorem 3.3.2.3 Instantaneous occupancy**

Let $\sigma$ be a sequence function of a system. Then the *instantaneous occupancy at time slot* $t$, $\omega_t$ is bounded from below by 0 and bounded from above by its capacity $\kappa$.

$$0 \leq \omega_t(\sigma) \leq \kappa$$

**Proof**

Naturally the number of values present in the system at time slot $t$ is at least 0. The number of values present in the system cannot exceed the maximum capacity of the buffer.

**Theorem 3.3.2.4 Average occupancy**

Let $\sigma$ be a given sequence function of a system. Then the *average occupancy* of the system, $\Omega$ is bounded from below by 0 and bounded from above by its capacity $\kappa$.

$$0 \leq \Omega(\sigma) \leq \kappa$$

**Proof**

Theorem 3.3.2.4 is a direct consequence of theorem 3.3.2.3 and definition 3.3.2.2.

Note that the occupancy is additive. Let $X||Y$ be a parallel composition of systems $X$ and $Y$, then

$$\Omega(\sigma_{X||Y}) = \Omega(\sigma_X) + \Omega(\sigma_Y)$$
Definition 3.3.2.5 Instantaneous vacancy

For any sequence function \( \sigma \) of a system the instantaneous vacancy at time slot \( t \), \( \varphi_i \), is defined as the capacity \( \kappa \) minus the instantaneous occupancy at time slot \( t \), \( \omega_i \),
\[
\varphi_i (\sigma) = \kappa - \omega_i (\sigma)
\]

Definition 3.3.2.6 Average vacancy

For any sequence function \( \sigma \) of a system the average vacancy of the system, denoted as \( \Phi \), is defined as the capacity minus the average occupancy of the system,
\[
\Phi (\sigma) = \kappa - \Omega (\sigma)
\]

Theorem 3.3.2.7 Instantaneous vacancy lower and upper bounds

Let \( \sigma \) be a sequence function of a system. Then the instantaneous vacancy at time slot \( t \), \( \varphi_i \), is bounded from below by 0 and bounded from above by its capacity.
\[
0 \leq \varphi_i (\sigma) \leq \kappa
\]

Proof

Theorem 3.3.2.7 is a direct consequence of theorem 3.3.2.3 and definition 3.3.2.5.

Theorem 3.3.2.8 Average vacancy lower and upper bounds

Let \( \sigma \) be a sequence function of a system. Then the average vacancy at time slot \( t \), \( \Phi \), is bounded from below by 0 and bounded from above by its capacity.
\[
0 \leq \Phi (\sigma) \leq \kappa
\]

Proof

Theorem 3.3.2.7 is a direct consequence of theorem 3.3.2.4 and definition 3.3.2.6.

3.3.3 Latency

Definition 3.3.3.1 Individual latency

For a given sequence function \( \sigma \) of a system \( X \) the individual latency, denoted as \( \lambda_i \), is defined as the number of time slots it takes value \( i \) to traverse the system
\[
\lambda_i (\sigma) = \sigma(b(i)) - \sigma(a(i))
\]
\[ a \in \mathcal{I}, b \in \mathcal{O}, 0 \leq i \]
3 Communication behavior

**Definition 3.3.3.2 Average latency**

For any sequence function \( \sigma \) of a system the *average latency*, denoted as \( \Delta \), is defined as

\[
\Delta (\sigma) = \lim_{i \to \infty} \frac{1}{i} \sum_{j=0}^{i-1} \lambda_j (\sigma)
\]

**Theorem 3.3.3.3 Individual latency lower bound**

Let \( \sigma \) a sequence function of a system. Then the *individual latency* \( \lambda_i \) of \( \sigma \) is bounded from below by its i/o-distance \( \delta \):

\[
\delta \leq \lambda_i (\sigma)
\]

**Proof**

The individual latency \( \lambda_i \) is defined as the number of time slots it takes value \( i \) to traverse the system, which is of course at least the shortest path through a system, i.e. de i/o-distance.

**Theorem 3.3.3.4 Average latency lower bound**

Let \( \sigma \) be a sequence function for a system. Then the average latency \( \Lambda \) of \( \sigma \) is bounded from below by its i/o-distance \( \delta \):

\[
\delta \leq \Lambda (\sigma)
\]

**Proof**

Theorem 3.3.3.4 is a direct consequence of theorem 3.3.3.3 and definition 3.3.3.2.

**Definition 3.3.3.5 Constant latency**

Let \( \sigma \) be a sequence function for a system. Then \( \sigma \) has *constant latency* when all its individual latencies are equal

\[
(\forall i: 0 \leq i: \lambda_{i+1} (\sigma) = \lambda_i (\sigma))
\]

We have defined some performance characteristics for buffers. When we apply these definitions to the late sequence function of the rectangular buffer \( \text{RBuf}_{3,3} \) of figure 3.2.2, we see that it has constant cycle time 2, constant latency 13, and average occupancy 6½. Note that the latency is the product of the cycle time and the occupancy. As we shall see in the next chapter this is not a coincidence but a property shared by all sequence functions.
3.4 Construction methods and sequence functions

In the previous sections we have defined some performance parameters. When applying these definitions to a buffer we find the matching performance characteristics. Moreover we are interested in the performance characteristics of buffers constructed using these methods. For each of the construction methods we like to derive the relationship of the latency (occupancy) of the constructed buffer and the latencies (occupancies) of the buffers out of which it is composed. We start to derive the relations for the serial composition.

3.4.1 Serial composition and sequence functions

Theorem 3.4.1 Serial composition theorem

For buffers $X$ and $Y$, let $\sigma_X$ and $\sigma_Y$ be sequence functions with equal and constant cycle time $\gamma$. Let $\sigma_X$ have constant latency $\lambda_x$. Then there exists a sequence function $\sigma$ for the serial composite SER $(X, Y)$ with constant cycle time $\gamma$ and latency $\lambda_{SER} = \lambda_x + \lambda_y (\sigma_Y)$.

Proof
Without loss of generality assume that $\sigma_X$ and $\sigma_Y$ are strict. Then sequence functions $\sigma_X$ and $\sigma_Y$ are defined by

\[
\begin{align*}
\sigma_X (a\#i) &= \gamma_i \\
\sigma_X (c\#i) &= \gamma_i + \lambda_x \\
\sigma_Y (c\#i) &= \gamma_i \\
\sigma_Y (b\#i) &= \gamma_i + \lambda_y (\sigma_Y)
\end{align*}
\]

Furthermore define $\sigma = \sigma_X \cup (\sigma_Y, u_0 \lambda_Y)$. Then $\sigma$ is a sequence function for SER(X,Y)

\[
\Gamma (\sigma) = \Gamma (\sigma_X) = \gamma
\]

- Sequence function $\sigma$ has constant cycle time $\gamma$

\[
\lambda_1 (\sigma) = \sigma(b\#(i)) - \sigma(a\#(i)) = \sigma(c\#(i)) - \sigma(a\#(i)) + \sigma(b\#(i)) - \sigma(c\#(i)) = \lambda_x + \lambda_y (\sigma_Y).
\]

- Sequence function $\sigma$ has latency $\lambda_{SER} = \lambda_x + \lambda_y (\sigma_Y)$.

As already be mentioned the occupancy is additive. The occupancy of the serial composite can also be found by the sum of the occupancies of $X$ and $Y$: $\omega_{SER} = \omega_X + \omega_Y$

3.4.2 Latency and occupancy of linear buffers

In the previous section we have shown the general relationship between the latency (occupancy) of a serial composite and the latency (occupancy) of its subcomponents. In this section we show how this relationship turns out for a specific case, the family of linear buffers.
Consider the linear buffer \( L_{\text{Buf}} \). For integers \( g \) and \( \gamma \), such that \( 2 \leq \gamma \) and \( 0 < g < \gamma \), there is a sequence function for Buf with constant cycle time \( \gamma \) and constant latency \( g \). Then by theorem 3.4.1 for any integer \( \gamma \), \( 2 \leq \gamma \), and any sequence of integers \( g_0, \ldots, g_n, \) such that \( (\forall j: 0 \leq j < n : 0 < g_j < \gamma) \) there exists a sequence function \( \sigma_{L_{\text{Buf}}} \) for the linear buffer \( L_{\text{Buf}} \) with constant cycle time \( \gamma \) and constant latency \( \lambda_{L_{\text{Buf}}}b \)
\[ \sigma_{L_{\text{Buf}}} = \sigma_{\text{Buf}} U (\sigma_{\text{Buf}}; \sigma; v_0 \sigma(1)) U (\sigma_{\text{Buf}}; v_0 \sigma(0)) U \ldots U (\sigma_{\text{Buf}}; v_0 \sigma(n-1)) \]
where \( S(p) = (\Sigma i: 0 \leq i < p; g_i) \)

Furthermore
\[ \Gamma (\sigma) = \Gamma (\sigma_{\text{Buf}}) = \gamma \]

- Sequence function \( \sigma_{L_{\text{Buf}}} \) has constant cycle time \( \gamma \)
- Sequence function \( \sigma_{L_{\text{Buf}}} \) has latency \( \lambda_{L_{\text{Buf}}} = \sum g_j \)

For the early sequence function the following condition holds: \( (\forall j: 0 \leq j < n ; g_j = 1) \). So early sequence function \( \sigma_{L_{\text{Buf}}} \) has latency \( \lambda_{L_{\text{Buf}}} = n \). Note that the latency for the early sequence function equals the i/o-distance \( \delta \). For the late sequence function the following condition holds: \( (\forall j: 0 \leq j < n ; g_j = \gamma - 1) \). So late sequence function \( \sigma_{L_{\text{Buf}}} \) has latency \( \lambda_{L_{\text{Buf}}} = n(\gamma - 1) \). Note that the latency for the late sequence function equals \( \lambda_{L_{\text{Buf}}} = n - \delta \).

- Sequence function \( \sigma_{L_{\text{Buf}}} \) has occupancy \( \omega_{L_{\text{Buf}}} = (\sum g_j) / \gamma \).

Then \( \Omega_{(\gamma, \gamma)} = n / \gamma \) and \( \Omega_{(\gamma, \gamma)} = n(\gamma - 1) / \gamma \).

**Definition 3.4.2 \((\gamma, k, l)\) periodic**

For integers \( \gamma, k \), and \( l \), such that \( 0 \leq k < l \) and \( 2 \leq \gamma \), a sequence function \( \sigma_X \) for a buffer \( X \) with input port \( a \) and output port \( b \) is \((\gamma, k, l)\) periodic when

\[
\sigma_X (a^j b) = \begin{cases} 
\gamma i + j & , 0 \leq j < k \\
\gamma i + (j+1) & , k \leq j < 1\gamma - 1
\end{cases}
\]

Note that sequence function \( \sigma_X \) has average cycle time \( \gamma / (l - 1) \).

Furthermore note that when a sequence function \( \sigma_X \) has constant latency. Then there exists integers \( \gamma, k \) and \( l \), where \( 0 \leq k < 1 \) and \( 2 \leq \gamma \), such that \( \sigma_X \) is \((\gamma, k, l)\) periodic.

**3.4.3 Wagging composition and sequence functions**

Next we derive some relations for the performance parameters, latency and occupancy, between a buffer constructed using wagging composition and its subcomponents.

**Theorem 3.4.3 Wagging theorem**

Let \( W \) be the wagging composite \( WAG_{k,l}(X,Y) \) of buffers \( X \) and \( Y \). For integer \( \gamma \) let \( \sigma_X \) be a sequence function with constant cycle time \( \gamma \) and let \( \sigma_Y \) be a sequence function that is \((\gamma, k, l)\) periodic.

Furthermore let \( \sigma_X \) and \( \sigma_Y \) have constant and equal latency \( \lambda \). Then for any integers \( g_1 \) and \( g_2 \), such that \( 0 < g_1 < \gamma \) and \( 0 < g_2 < \gamma \), there exists a sequence function \( \sigma_W \) for the wagging composite \( WAG_{k,l}(X,Y) \) with constant cycle time \( \gamma \) and constant latency \( \lambda + g_1 + g_2 \).
Proof

Define sequence functions $\sigma_{\text{Split},k,i}$ and $\sigma_{\text{Merge},k,i}$ by

$$
\begin{align*}
\sigma_{\text{Split},k,i} (a\#i) &= \gamma_i \\
\sigma_{\text{Split},k,i} (c\#i) &= ly_i + ky + g_1 \\
\sigma_{\text{Split},k,i} (d\#i) &= ly_i + jy + g_1 \\
&= ly_i + (j+1)y + g_1, \quad 0 \leq j < k \\
\sigma_{\text{Merge},k,i} (e\#i) &= ly_i + ky \\
\sigma_{\text{Merge},k,i} (f\#i) &= ly_i + jy \\
&= ly_i + (j+1)y, \quad k \leq j < 1 - 1 \\
\sigma_{\text{Merge},k,i} (b\#i) &= \gamma_i + g_2
\end{align*}
$$

Furthermore define $\sigma_W = \sigma_{\text{Split},k,i} \cup (\sigma_X; u_0^{\beta i}) \cup (\sigma_Y; u_0^{\gamma i}) \cup (\sigma_{\text{Merge},k,i}; u_0^{\lambda g_1})$

Then

$$
\Gamma(\sigma_W) = \Gamma(\sigma_{\text{Split},k,i}) = \gamma
$$

- Sequence function $\sigma_W$ has constant cycle time $\gamma$

$$
\lambda_i(\sigma_W) = \sigma_W(b\#i) - \sigma_W(a\#i) = (\sigma_{\text{Merge},k,i} (b\#i) + \lambda + g_1) - \sigma_{\text{Split},k,i} (a\#i) = (\gamma_i + g_2 + \lambda + g_1) - \gamma_i = \lambda + g_1 + g_2
$$

- Sequence function $\sigma_W$ has constant latency $\lambda_W = \lambda + g_1 + g_2$

- Sequence function $\sigma_W$ has occupancy $\omega_W = \lambda_W / \gamma_W = (\lambda + g_1 + g_2) / \gamma$

The occupancy of the wagging composite can also be found by the sum of the occupancies of the subcomponents $X, Y, \text{Split}_{k,i}$ and $\text{Merge}_{k,i}$

$$
\omega_{\text{SER}} = \omega_x + \omega_y + \omega_{\text{Split}} + \omega_{\text{Merge}}
= \lambda / \gamma + \lambda / (\gamma - 1) / \gamma + g_1 / \gamma + g_2 / \gamma
= (\lambda + g_1 + g_2) / \gamma
$$

Corollary 3.4.4

Application of theorem 3.4.3 with $k=0$ and $l=2$ gives the Wagging theorem of [Mak]. Our wagging theorem is more general.

3.4.4 Latency and occupancy of binary tree buffers

In the previous section we have shown the general relationship between the latency (occupancy) of a wagging composite and the latency (occupancy) of its subcomponents. In this section we show how it turns out for a specific case, the family of binary tree buffers.

Consider the binary tree buffer $\text{BBuf}_n$. Then by theorem 3.4.3 for any integer $\gamma$, $2 \leq \gamma$, and any sequence of integers $g_0, \ldots, g_{2n-2}$ such that $(\forall j; 0 \leq j < n: 0 < g_j < 2^j \cdot \gamma)$ and $(\forall j; n \leq j < 2n - 1: 0 < g_j < 2^{2n-1-j} \cdot \gamma)$ there exists a sequence function $\sigma_{\text{BBuf}_n}$ for the binary buffer $\text{BBuf}_n$ with constant cycle time $\gamma$ and constant latency $\lambda_{\text{BBuf}_n} = g_0 + g_1 + \ldots + g_{2n-2}$. 
Define $\sigma_{BBUF,n}$ by

$$
\sigma_{BBUF,n} = \sigma_{Split,0,2} \cup \\
(\sigma_{Split,0,2,2,1,0}^1 \cup \sigma_0^1) \cup (\sigma_{Split,0,2,2,1,0}^1 \cup \sigma_0^1) \cup \\
(\sigma_{Split,0,2,2,1,0}^2 \cup \sigma_0^2) \cup (\sigma_{Split,0,2,2,1,0}^2 \cup \sigma_0^2) \cup (\sigma_{Split,0,2,2,1,0}^2 \cup \sigma_0^2) \cup \\
(\sigma_{Split,0,2,2,1,0}^3 \cup \sigma_0^3) \cup (\sigma_{Split,0,2,2,1,0}^3 \cup \sigma_0^3) \cup (\sigma_{Split,0,2,2,1,0}^3 \cup \sigma_0^3) \cup \\
(\sigma_{Split,0,2,2,1,0}^4 \cup \sigma_0^4) \cup (\sigma_{Split,0,2,2,1,0}^4 \cup \sigma_0^4) \cup (\sigma_{Split,0,2,2,1,0}^4 \cup \sigma_0^4) \cup \\
(\sigma_{Split,0,2,2,1,0}^{n-2} \cup \sigma_0^{n-2}) \cup (\sigma_{Split,0,2,2,1,0}^{n-2} \cup \sigma_0^{n-2}) \cup (\sigma_{Split,0,2,2,1,0}^{n-2} \cup \sigma_0^{n-2}) \cup \\
(\sigma_{Split,0,2,2,1,0}^n \cup \sigma_0^n) \cup (\sigma_{Split,0,2,2,1,0}^n \cup \sigma_0^n) \cup (\sigma_{Split,0,2,2,1,0}^n \cup \sigma_0^n) \\
$$

where $S(p) = (\Sigma; 0 \leq i < p; g_i)$

Furthermore

$$
\Gamma(\sigma_{BBUF,n}) = \Gamma(\sigma_{Split,0,2}) = \gamma
$$

Sequence function $\sigma_{BBUF,n}$ has constant cycle time $\gamma$

$$
\lambda(\sigma_{BBUF,n}) = \sum_{k=0}^{2n-2} (g_k), \quad 1 \leq n
$$

We are interested in the late and early sequence functions.

- For the early sequence function the following condition holds: $(\forall j; 0 \leq j < 2n-1; g_j=1)$. So early sequence function $\sigma_{BBUF,n}$ has latency $\lambda_{BBUF,n} = 2n-1$. Note that the latency for the early sequence function equals the i/o-distance $\delta$.

- For the late sequence function the following condition holds:

  $(\forall j; 0 \leq j < n; g_j = 2^{2^j} - 1) \land (\forall j; n \leq j < 2n-1; g_j = 2^{2n-2-j} \cdot \gamma - 1)$. So late sequence function $\sigma_{BBUF,n}$ has latency

$$
\lambda(\sigma_{BBUF,n}) = \sum_{k=0}^{2n-2} (y_k^2 - 1) + \gamma 2^{n-1} - 1 + \sum_{k=0}^{2n-2} (y_k^{2^2-2k} - 1)
$$

$$
\lambda(\sigma_{BBUF,n}) = \sum_{k=0}^{2n-2} (y_k^2 - 1) + \gamma 2^{n-1} - 1 + \sum_{k=0}^{2n-2} (y_k^{2^2-2k} - 1)
$$

$$
= \sum_{k=0}^{2n-2} (y_k^2 - 1) + 2 \gamma 2^{n-1} - 1
$$

$$
= \sum_{k=0}^{2n-2} (y_k^2) + 2 \sum_{k=0}^{n-2} (y_k^2) + 2 \gamma 2^{n-1} - 1
$$
\[ 2\gamma \sum_{k=0}^{n-2} (2^k) - 2(n-1) + \gamma 2^{n-1} - 1 \]
\[ = \{ \text{calculus} \} \]
\[ 2\gamma (2^{n-1} - 2(n-1) + \gamma 2^{n-1} - 1) \]
\[ = \{ \} \]
\[ 3\gamma 2^{n-1} - 2\gamma - 2n+1 \]

The above expression for \( \lambda \) can also be derived as a recurrent definition

\[ \lambda(\bar{\sigma}_{BBuf,n+1}) \]
\[ = \{ \text{definition } \lambda \} \]
\[ 2 \sum_{k=0}^{n-1} (\gamma 2^k - 1) + \gamma 2^n - 1 \]
\[ = \{ \text{splitting on domain } k=n-1 \} \]
\[ 2 \sum_{k=0}^{n-2} (\gamma 2^k - 1) + 2(\gamma 2^{n-1} - 1) + \gamma 2^n - 1 \]
\[ = \{ \} \]
\[ 2 \sum_{k=0}^{n-2} (\gamma 2^k - 1) + 2\gamma 2^{n-1} - 2 + 2\gamma 2^{n-1} - 1 \]
\[ = \{ \} \]
\[ 2 \sum_{k=0}^{n-2} (\gamma 2^k - 1) + \gamma 2^{n-1} - 2 \]
\[ = \{ \text{definition } \lambda \} \]
\[ \lambda(\bar{\sigma}_{BBuf,n}) = 3\gamma 2^{n-1} - 2 \]

The recurrent definition for \( \lambda(\sigma_{BBuf,1}) \)

\[ \lambda(\bar{\sigma}_{BBuf,1}) = \gamma - 1 \]
\[ \lambda(\bar{\sigma}_{BBuf,n+1}) = \lambda(\bar{\sigma}_{BBuf,n}) + 3\gamma 2^{n-1} - 2 \quad , \quad 1 \leq n \]

Late sequence function \( \bar{\sigma}_{BBuf,n} \) has latency \( \lambda_{BBuf,n} = 3\gamma 2^{n-1} - 2\gamma - 2n+1 \)
Note that the latency equals \( \gamma_k - \delta = \gamma (3\cdot 2^{n-1} - 2) - (2n-1) \)

- Sequence function \( \sigma_{BBuf,n} \) has occupancy

\[ \omega_{BBuf,n} = \frac{\sum_{k=0}^{n-2} (2\gamma g_k) + g_{n-1} \gamma}{\gamma} \quad , \quad 1 \leq n \]

Then \( \Omega_{r=\gamma} = (2n-1)/\gamma \) and \( \Omega_{r=2^{n-1}} = 3\cdot 2^{n-1} - (2n-1)/\gamma \).
3.4.5 Multi-Wagging and sequence functions

So far we derived some relations for the performance parameters, latency and occupancy, between a constructed buffer using serial composition or wagging and its subcomponents. The remaining relations we derive are the performance parameters, latency and occupancy, between a constructed buffer using multi-wagging and its subcomponents.

Theorem 3.4.5 Multi-wagging theorem

Let $MW_n (X_0, \ldots, X_{n-1})$ be the multi-wagging composite of buffers $X_0, \ldots, X_{n-1}$. For integer $\gamma$ let $\sigma_{X_0}, \ldots, \sigma_{X_{n-1}}$ be sequence functions for $X_0, \ldots, X_{n-1}$ with constant cycle time $\gamma$. Furthermore let $\sigma_{X_0}, \ldots, \sigma_{X_{n-1}}$ have constant and equal latency $\lambda$.

Then for any integers $g_0, \ldots, g_n$ such that $(\forall j: 0 \leq j \leq n : 0 < g_j < \gamma)$, and integers $v_1, \ldots, v_{n-1}$ that such that $(\forall j: 0 < j < n : 0 \leq v_j < \gamma)$ there exists a sequence function $\sigma_{MW}$ for the wagging composite $MW_n (X_0, \ldots, X_{n-1})$ with constant cycle time $\gamma$ and constant latency $\lambda + g_0 + g_1 + \ldots + g_n + v_1 + \ldots + v_{n-1}$.

Proof

For any integers $g_0, \ldots, g_n$ such that $(\forall j: 0 \leq j \leq n : 0 < g_j < \gamma)$, and integers $v_1, \ldots, v_{n-1}$ $(0 < k < n)$ such that such that $(\forall j: 0 < j < n : 0 \leq v_j < \gamma)$ define sequence functions $\sigma_{MS,n}$ and $\sigma_{MM,n}$ by

$$
\sigma_{MS,n} (a_{i\#}) = \gamma_i
$$

$$
\sigma_{MS,n} (c_{p\#}) = n\gamma_i + p\gamma + G(p) + V(p)
$$

$$
\sigma_{MM,n} (b_{i\#}) = \gamma_i + \lambda + G(0) + g_n + V(0)
$$

$$
\sigma_{MM,n} (e_{p\#}) = n\gamma_i + p\gamma + \lambda + G(p) + V(p)
$$

where $G(p) = (\Sigma: p_i \leq i < n: g_{i-1,i})$

and $V(p) = (\Sigma: p_i < i < n: v_{i-1,i})$

Define $\sigma_{MW} = \sigma_{MS,n} U (\sigma_{X_0,1}^{(a\#)}; v_0 (0\gamma) + G(0) + V(0)) U \ldots \sigma_{X_0,1}^{(a\#)}; v_0 (p\gamma + G(p) + V(p)) U \ldots \sigma_{X_0,1}^{(a\#)}; v_0 (0\gamma + V(0)) U \sigma_{MM,n}$

$$
\Gamma(\sigma_{MW}) = \Gamma(\sigma_{MS,n}) = \gamma
$$

Sequence function $\sigma_{MW}$ has constant cycle time $\gamma$

$$
\lambda' (\sigma_{MW}) = \sigma_{MW} (b_{i\#}) - \sigma_{MW} (a_{i\#})
$$

$$
= (\sigma_{MM,n} (b_{i\#}) - \sigma_{MS,n} (a_{i\#}))
$$

$$
= (\gamma_i + \lambda + G(0) + g_n + V(0) - \gamma_i)
$$

$$
= \lambda + G(0) + g_n + V(0)
$$

$$
= \lambda + g_0 + g_1 + \ldots + g_n + v_1 + \ldots + v_{n-1}
$$

Sequence function $\sigma_{MW}$ has constant latency $\lambda + g_0 + g_1 + \ldots + g_n + v_1 + \ldots + v_{n-1}$.

The sequence functions of the subcomponents are notated in the diagram of figure 3.4.1.
3.4.6 Latency and occupancy of rectangular buffers

In the previous section we have shown the general relationship between the latency (occupancy) of a multi-wagging with the latency (occupancy) of its subcomponents. In this section we show how it turns out for a specific case, the family of rectangular buffers.

Consider the rectangular buffer RBufm,n. Applying theorem 3.4.5 gives a sequence function σRBufm,n for the rectangular buffer RBufm,n with latency constant latency λRBufm,n = λLBuf,m,n + g0 + g1 + ... + gn + v1 + ... + vn for any integers g0, ..., gn, such that (∀j: 0 ≤ j ≤ n: 0 ≤ gj < γ), and v1, ..., vn, that such that (∀j: 0 < j < n: 0 ≤ vj ≤ γ).

Sequence function σRBufm,n has constant latency λ = λLBuf,m,n + g0 + g1 + ... + gn + v1 + ... + vn where λLBuf,m,n = w0 + w1 + ... + wm for such that (∀j: 0 ≤ j < m: 0 < wj < γ).

So sequence function σRBufm,n has constant latency λ = w0 + w1 + ... + wm + g0 + g1 + ... + gn + v1 + ... + vn.

We are interested in the late and early sequence functions.

For the early sequence function the following conditions hold
- (∀j: 0 ≤ j ≤ n: gj = 1)
- (∀j: 0 < j ≤ n: vj = 0)
- λ(σRBufm,n) = m - 2

λ(σRBufm,n) = λ(σLBufm,n) + g0 + g1 + ... + gn + v1 + ... + vn
= m - 2 + n + 1
= m + n - 1

Figure 3.4.1 Multi-wagging with general sequence
Early sequence function \( \bar{\sigma}_{RBuf,m,n} \) has constant latency \( \lambda = m + n - 1 \)
Note that the latency equals the I/O-distance \( \delta_{RBuf,m,n} \).

For the late sequence function the following conditions hold
- \( (\forall j: 0 < j < n : g_j = \gamma - 1) \)
- \( (\forall j: 0 < j < n : v_j = \gamma) \)
- \( \lambda (\bar{\sigma}_{LBuf,m-2}) = (n \gamma - 1)(m-2) \)

\[
\lambda(\bar{\sigma}_{RBuf,m,n}) = \lambda(\bar{\sigma}_{LBuf,m-2}) + g_0 + \cdots + g_n + v_1 + \cdots + v_{n-1} \\
= (n \gamma - 1)(m-2) + (n+1)(\gamma - 1) + (n-1) \gamma \\
= \gamma (mn) - m + 2 - 2n \gamma + n \gamma - n + \gamma - 1 - n \gamma - \gamma \\
= \gamma (mn) - (m+n-1)
\]

Late sequence function \( \bar{\sigma}_{RBuf,m,n} \) has constant latency \( \lambda_{RBuf,m,n} = \gamma (mn) - (m+n-1) \)
Note that the latency equals \( \gamma \lambda - \delta = \gamma (mn) - (m+n-1) \)

Sequence function \( \sigma_{RBuf,m,n} \) has occupancy \( \omega_{RBuf,m,n} = (w_0 + \cdots + w_{m-2} + g_0 + \cdots + g_n + v_1 + \cdots + v_{n-1}) / \gamma \).
Then \( \Omega_{\gamma} = (m+n-1)/\gamma \) and \( \Omega_{\gamma} = (mn) - (m+n-1)/\gamma \).
4 Optimal buffers

In the previous chapters we have defined basic components, component construction methods, and have given the definition of a buffer. In this chapter we will consider optimal buffers with respect to two parameters $\kappa$ and $\delta$, which are the capacity and i/o-distance of a buffer. We introduce the definition of a $(\kappa, \delta)$ - optimal buffer. Next we show how to design $(\kappa, \delta)$ - optimal buffers, i.e. we solve

Problem 4.1

Given a pair of values $\kappa$ and $\delta$, design a $(\kappa, \delta)$ - optimal buffer with capacity $\kappa$ and i/o-distance $\delta$.

4.1 Defining $(\kappa, \delta)$ - optimality

Theorem 4.1.1 Queueing formula for buffers

Let $X$ be a buffer with capacity $K_x$ and i/o-distance $\delta_x$. Then for every sequence function $\sigma_x$ of $X$ with average cycle time $\gamma_x$ and average occupancy $\omega_x$ the average latency $\lambda_x$ satisfies the equality

$$\lambda_x = \gamma_x \cdot \omega_x$$

Proof

See [Mak].

This formula can be seen as a special case of the queuing formula of [Lit].

Theorem 4.1.2 Latency lower bound

Let $X$ be a buffer with capacity $K_x$ and i/o-distance $\delta_x$. Then for every sequence function $\sigma_x$ of $X$ with average cycle time $\gamma_x$, the average latency $\lambda_x$ is bounded from below by

$$\delta_x \leq \lambda_x$$

Proof

Note that it is sufficient to show that the individual latency for any value is at least $\delta_x$. Moreover note that that in each time slot a value can move at most one variable towards the output port. Since $\delta_x$ is the minimum number of variables on any path from the buffer’s input port to the buffer’s output port, it takes a value at least $\delta_x$ time slots to traverse the buffer. Hence the individual latency, which is the number of time slots a value resides in the buffer, is at least $\delta_x$.

Corollary 4.1.3 Occupancy lower bound

Let $X$ be a buffer with capacity $K_x$ and i/o-distance $\delta_x$. Then for every sequence function $\sigma_x$ of $X$ with average cycle time $\gamma_x$, the average occupancy $\omega_x$ is bounded from below by

$$\delta_x / \gamma_x \leq \omega_x$$

Proof
Let $X$ be a buffer with capacity $\kappa_x$ and i/o-distance $\delta_x$. Let $\sigma_x$ be a sequence function for $X$ with average cycle time $\gamma_x$. Then by theorem 4.1.2 the average latency $\lambda_x$ is bounded from below by the i/o-distance $\delta_x$.

\[
\begin{align*}
\delta_x &\leq \lambda_x \\
&= \{ \text{theorem 4.1.1} : \lambda_x = \gamma_x \cdot \omega_x \} \\
\delta_x &\leq \gamma_x \cdot \omega_x \\
&= \{ \text{calculus, } \gamma_x > 0 \} \\
\delta_x / \gamma_x &\leq \omega_x
\end{align*}
\]

Theorem 4.1.4 Vacancy lower bound

Let $X$ be a buffer with capacity $\kappa_x$ and i/o-distance $\delta_x$. Then for every sequence function $\sigma_x$ of $X$ with average cycle time $\gamma_x$, the average vacancy $\Phi_x$ is bounded from below by

\[
\delta_x / \gamma_x \leq \Phi_x
\]

Proof

Let $X$ be a buffer with capacity $\kappa_x$ and i/o-distance $\delta_x$. Let $\sigma_x$ be a sequence function for $X$ with average cycle time $\gamma_x$. Note that when the buffer outputs a value this can be also viewed as if the buffer inputs a bubble (empty space). Similarly when a value propagates from one variable to the next this can also be viewed as a bubble propagating the reverse direction. Then the smallest number of moves for any bubble to traverse the buffer is also the i/o-distance $\delta_x$. Hence it follows that the average latency for the bubbles which is denoted as $\overline{\lambda}_x$ is also bounded from below by $\delta_x$, $\delta_x \leq \overline{\lambda}_x$.

Then by theorem 4.1.1 in which the average occupancy $\omega_x$ is replaced by the average vacancy $\Phi_x$ and which the "value latency" $\lambda_x$ is replaced by the "bubble latency" $\overline{\lambda}_x$ it follows that $\overline{\lambda}_x = \gamma_x \cdot \Phi_x$.

Hence we find $\delta_x \leq \overline{\lambda}_x = \gamma_x \cdot \Phi_x$, which after rearranging gives the required upper bound $\delta_x / \gamma_x \leq \Phi_x$.

Corollary 4.1.5 Occupancy upper bound

Let $X$ be a buffer with capacity $\kappa_x$ and i/o-distance $\delta_x$. Then for every sequence function $\sigma_x$ of $X$ with average cycle time $\gamma_x$, the average occupancy $\omega_x$ is bounded from above by

\[
\omega_x \leq \kappa_x - \delta_x / \gamma_x
\]

Proof

Let $X$ be a buffer with capacity $\kappa_x$ and i/o-distance $\delta_x$. Let $\sigma_x$ be a sequence function for $X$ with average cycle time $\gamma_x$. Then by theorem 4.1.2 the average vacancy $\Phi_x$ is bounded from below by

\[
\begin{align*}
\delta_x / \gamma_x &\leq \Phi_x \\
&= \{ \text{definition vacancy } \Phi_x = \kappa_x - \omega_x \} \\
\delta_x / \gamma_x &\leq \kappa_x - \omega_x \\
&= \{ \text{calc} \} \\
\omega_x &\leq \kappa_x - \delta_x / \gamma_x
\end{align*}
\]

Theorem 4.1.6 Latency upper bound

Let $X$ be a buffer with capacity $\kappa_x$ and i/o-distance $\delta_x$. Then for every sequence function $\sigma_x$ of $X$ with average cycle time $\gamma_x$ the average latency $\lambda_x$ is bounded from above by
4.1 Defining \((\kappa, \delta)\) - optimality

\[
\lambda_x \leq \gamma_x \cdot \kappa_x - \delta_x
\]

\textbf{Proof}

Let \(X\) be a buffer with capacity \(\kappa_x\) and \(i/o\)-distance \(\delta_x\). Let \(\sigma_x\) be a sequence function for \(X\) with average cycle time \(\gamma_x\). Then by corollary 4.1.5

\[
\omega_x \leq \kappa_x - \delta_x / \gamma_x
\]

\(= \{\ \text{calculus}, \gamma_x > 0\} \)

\[
\gamma_x \cdot \omega_x \leq \gamma_x \cdot \kappa_x - \delta_x
\]

\(= \{\ \text{theorem 4.1.1} : \lambda_x = \gamma_x \cdot \omega_x\} \)

\[
\gamma_x \cdot \lambda_x / \gamma_x \leq \gamma_x \cdot \kappa_x - \delta_x
\]

\(= \{\ \text{calculus}\} \)

\[
\lambda_x \leq \gamma_x \cdot \kappa_x - \delta_x
\]

\textbf{Definition 4.1.7} \((\kappa, \delta)\) - minimal

Let \(X\) be a buffer with capacity \(\kappa_x\) and \(i/o\)-distance \(\delta_x\). Then \(X\) is \((\kappa, \delta)\) - minimal if for each cycle time \(\gamma_x\), such that \(2 \leq \gamma_x\), there exists a sequence function \(\sigma_x\) with average cycle time \(\Gamma(\sigma_x) = \gamma\) where for each value that traverses buffer \(X\) the individual latency equals the \(i/o\)-distance. Note that this implies that \(\sigma_x\) has constant latency \(\lambda_x = \delta_x\) and average occupancy \(\omega_x = \delta_x / \gamma_x\).

\textbf{Definition 4.1.8} \((\kappa, \delta)\) - maximal

Let \(X\) be a buffer with capacity \(\kappa_x\) and \(i/o\)-distance \(\delta_x\). Then \(X\) is \((\kappa, \delta)\) - maximal if for all cycle time \(\gamma_x\), such that \(2 \leq \gamma_x\), there exists a sequence function \(\sigma_x\) with average cycle time \(\Gamma(\sigma_x) = \gamma\) where for each value that traverses buffer \(X\) the individual latency equals the product of the cycle time and capacity minus the \(i/o\)-distance. Note that this implies that \(\sigma_x\) has constant latency \(\lambda_x = \gamma_x \cdot \kappa_x - \delta_x\) and average occupancy \(\omega_x = \kappa_x - \delta_x / \gamma_x\).

\textbf{Definition 4.1.9} \((\kappa, \delta)\) - optimal

A buffer \(X\) with capacity \(\kappa_x\) and \(i/o\)-distance \(\delta_x\) is \((\kappa, \delta)\) - optimal when a buffer \(X\) is \((\kappa, \delta)\) - minimal and \((\kappa, \delta)\) - maximal.

4.2 Construction of \((\kappa, \delta)\) - optimal buffers

In the previous section we have introduced the definition of a \((\kappa, \delta)\) - optimal buffer. In this section we show how to design \((\kappa, \delta)\) - optimal buffers. We start by proving that the basic component BUF is a \((\kappa, \delta)\) - optimal buffer. In chapter two we have given three construction methods, serial composition, wagging and multi-wagging. Applying these construction methods using a set of \((\kappa, \delta)\) - optimal buffers not necessarily yields another \((\kappa, \delta)\) - optimal buffer. Therefore we derive some restrictions for the application of the three construction methods such that when obeyed these methods indeed yield a \((\kappa, \delta)\) - optimal buffer.

\textbf{Theorem 4.2.1} BUF is \((\kappa, \delta)\) - optimal

The basic component BUF is a \((\kappa, \delta)\) - optimal buffer.
Proof

Let $X$ be buffer $BUF$. Then $\kappa_x = 1$ and $\delta_x = 1$. Then by table 3.4.2 the early sequence for $X$ is given by $\sigma_{Buf}$. Hence the latency is given by $\lambda_x = (y_x \cdot i + 1) - \gamma_x \cdot i = 1$ which equals defi of i/o-distance $\delta_x$. By table 3.4.3 the late sequence function is given by $\lambda_x = (y_x \cdot i + y_x - 1) - (\gamma_x \cdot i) = y_x - 1$ which equals $y_x \cdot \kappa_x - \delta_x$.

4.2.1 Serial composition and constructing $(k, \delta)$ - optimal buffers

Theorem 4.2.2 Serial composition optimality

Let $X$, $Y$ be $(k, \delta)$ - optimal buffers with capacities $k_x$, $k_y$ and i/o-distances $\delta_x$, $\delta_y$. Then the serial composite $SER(X, Y)$ is a $(k, \delta)$ - optimal buffer with i/o-distance $\delta_{SER} = \delta_x + \delta_y$ and capacity $k_{SER} = k_x + k_y$.

Proof

Let $X$, $Y$ be $(k, \delta)$ - optimal buffers with capacities $k_x$, $k_y$ and i/o-distances $\delta_x$, $\delta_y$. Buffer $X$ is $(k, \delta)$ - optimal. Hence there exists a sequence function $\sigma_x$ with constant latency $\lambda_x$, and constant cycle time $y_x$. Buffer $Y$ is $(k, \delta)$ - optimal. Hence there exists a sequence function $\sigma_y$ with constant latency $\lambda_y$, and constant cycle time such that $y_x \geq y_x$. Then by theorem 3.4.1 there exists a sequence function $\sigma_{SER}$ for the serial composite $SER(X, Y)$ with constant cycle time $y_{SER} \geq y_x$ and latency $\lambda_{SER} = \lambda_x + \lambda_y$. According to theorems 2.6.1 and 2.7.1 the serial composite $SER(X, Y)$ is a buffer with i/o-distance $\delta_{SER} = \delta_x + \delta_y$ and capacity $k_{SER} = k_x + k_y$. Hence it remains to be shown that when we take the early sequence functions $\sigma_x$ and $\sigma_y$ the latency $\lambda_{SER}$ equals the i/o-distance $\delta_{SER}$ and for the late sequence functions $\sigma_x$ and $\sigma_y$ that the latency $\lambda_{SER}$ equals $y_{SER} \cdot k_{SER}$ - $\delta_{SER}$. We first start with the early sequence function

\[ \lambda_{SER} = \{ \text{theorem 3.4.1} \} \]
\[ \lambda_x + \lambda_y \]
\[ = \{ \sigma_{x(y)} \text{ early so } \lambda_{x(y)} = \delta_{x(y)} \} \]
\[ \delta_x + \delta_y \]
\[ = \{ \text{theorem 2.7.1: I/O distance of a serial composite } SER(X, Y) \ \delta_{SER} = \delta_x + \delta_y \} \]
\[ \delta_x \]

The latency $\lambda_{SER}$ of the late sequence function equals

\[ \lambda(\sigma_{SER}) = \{ \text{theorem 3.4.1: latency of serial composition} \} \]
\[ \lambda_x + \lambda_y \]
\[ = \{ \sigma_{x(y)} \text{ late so } \lambda_{x(y)} = y_{x(y)} \cdot k_{x(y)} \cdot \delta_{x(y)} \} \]
\[ \gamma_x \cdot \kappa_x + \delta_x + \gamma_y \cdot \kappa_y - \delta_y \]
\[ = \{ \text{theorem 2.7.1: I/O distance of a serial composite } SER(X, Y) \ \delta_{SER} = \delta_x + \delta_y \} \]
\[ \gamma_y \cdot \kappa_x + \delta_y \]
\[ = \{ \text{theorem 2.6.1: capacity of a serial composite } SER(X, Y) \ \kappa_y = k_x + k_y \} \]
\[ \gamma_{SER} \cdot k_{SER} - \delta_{SER} \]

Theorem 4.2.3 Linear buffers are $(k, \delta)$ - optimal

The family of linear buffers $\{L_{Buf_i}\}_{i=1}^{\infty}$ is a family of $(k, \delta)$ - optimal buffers.
4.2 Construction of \((\kappa, \delta)\) - optimal buffers

Proof

The theorem is proved by mathematical induction on \(n, 1 \leq n\).

Basis \(n=1\)

The linear buffer \(Lbuf_1\) is the basic component \(BUF\). According theorem 4.2.1 \(BUF\) is a \((\kappa, \delta)\) - optimal buffer.

Step \(n=n+1\)

Let \(Lbuf_n\) be a \((\kappa, \delta)\) - optimal buffer with capacity \(K_{Lbuf_n}\) and i/o-distance \(\delta_{Lbuf_n}\). Then by theorem 4.2.2 the serial composite \(SER(\text{Buf}, Lbuf_n)\) is an optimal buffer with i/o-distance \(\delta_{SER} = \delta_{Lbuf_n} + \delta_{BUF}\) and capacity \(K_{SER} = K_{Lbuf_n} + K_{BUF}\).

4.2.2 Wagging and constructing \((\kappa, \delta)\) - optimal buffers

Theorem 4.2.4 Balanced Wagging on a pair of identical \((\kappa, \delta)\) - optimal buffers

Let \(X\) be a \((\kappa, \delta)\) - optimal buffer with capacity \(\kappa_x\) and i/o-distance \(\delta_x\). Then the wagging composite \(WAG_{0,2}(X, X)\) is a \((\kappa, \delta)\) - optimal buffer with capacity \(K_{WAG} = 2\kappa_x + 2\) and i/o-distance \(\delta_{WAG} = \delta_x + 2\).

Proof

Let \(X\) be a \((\kappa, \delta)\) - optimal buffer with capacity \(\kappa_x\) and i/o-distance \(\delta_x\). Buffer \(X\) is \((\kappa, \delta)\) - optimal. So there exists a sequence function \(\sigma_x\) with constant latency \(\lambda_x\) and constant cycle time \(\gamma_x\). Then by theorem 3.4.3 there exists a sequence function \(\sigma_{WAG}\) for the wagging composite \(WAG_{0,2}(X, X)\) with constant cycle time \(\gamma_{WAG}\), such that \(2\gamma_{WAG} = \gamma_x\) and constant latency \(\lambda_{WAG} = \lambda_x + 2\). According theorems 2.6.2 and 2.7.2 the wagging composite \(WAG_{0,2}(X, X)\) is a buffer with capacity \(K_{WAG} = 2\kappa_x + 2\) and i/o-distance \(\delta_{WAG} = \delta_x + 2\).

Hence it remains to be shown that when we take the early sequence function \(\delta_x\) the latency \(\lambda_{WAG}\) equals the i/o-distance \(\delta_{WAG}\) and for the late sequence function \(\delta_x\) that the latency \(\lambda_{WAG}\) equals \(\gamma_{WAG}\). We start with the early sequence function

\[
\lambda(\delta_{WAG}) = \{\text{theorem 3.4.3: } g=1 \text{ early sequence}\}
\]

\[
\lambda_x + 2
\]

\[
= \{\sigma_x \text{ early so } \lambda_x = \delta_x\}
\]

\[
\delta_x + 2
\]

\[
= \{\delta_{WAG} = \delta_x + 2\}
\]

\[
\delta_{WAG}.
\]

The latency \(\lambda_{WAG}\) of the late sequence function equals

\[
\lambda(\delta_{WAG}) = \{\text{theorem 3.4.3: late sequence}\}
\]

\[
\lambda_x + 2(\gamma_{WAG} - 1)
\]

\[
= \{\sigma_x \text{ late so } \lambda_x = \gamma_x \kappa_x - \delta_x\}
\]

\[
\gamma_x \kappa_x - \delta_x + 2\gamma_{WAG} - 2
\]

\[
= \{\gamma_x = 2\gamma_{WAG}\}
\]

\[
\gamma_x \kappa_x - \delta_x + \gamma_x - 2
\]

\[
= \{\text{ calculus}\}
\]

\[
\gamma_x (\kappa_x + 1) - (\delta_x + 2)
\]
\[
\begin{align*}
\gamma_x &= 2\gamma_{\text{WAG}} \\
\gamma_{\text{WAG}} &= (2k_x + 2) - (\delta_x + 2) \\
\delta_{\text{WAG}} &= \delta_x + 2 \\
\gamma_{\text{WAG}} &= (2k_x + 2) - \delta_{\text{WAG}} \\
\delta_{\text{WAG}} &= k_{\text{WAG}} - \delta_{\text{WAG}}
\end{align*}
\]

**Theorem 4.2.5** A binary - tree buffer is \((\kappa, \delta)\) - optimal

The family of binary - tree buffers \(\{\text{BBuf}_n\}_{n=1}^\infty\) is a family of \((\kappa, \delta)\) - optimal buffers.

**Proof**

The theorem is proved by mathematical induction on \(n, 1 \leq n\).

**Basis** \(n = 1\)

The binary tree buffer \(\text{BBuf}_1\) is the basic component \(\text{BUF}\). According theorem 4.2.1 \(\text{BUF}\) is a \((\kappa, \delta)\) - optimal buffer.

**Step** \(n = n + 1\)

Let \(\text{BBuf}_n\) be a \((\kappa, \delta)\) - optimal buffer with capacity \(k_{\text{BBuf}_n}\) and i/o-distance \(\delta_{\text{BBuf}_n}\). Then by theorem 4.2.4 the wagging buffer \(\text{WAG}_{0,2}(\text{BBuf}_n, \text{BBuf}_n)\) is also \((\kappa, \delta)\) - optimal with i/o-distance \(\delta_{\text{WAG}} = \delta_{\text{BBuf}_n} + 2\) and capacity \(k_{\text{WAG}} = 2k_{\text{BBuf}_n} + 2\).

Theorem 4.2.4 shows that the wagging composite of two identical \((\kappa, \delta)\) - optimal buffers is a \((\kappa, \delta)\) - optimal buffer. In the sequel we will strengthen this result. First we drop the requirement that the subcomponents must be identical, and subsequently we allow unbalanced wagging, i.e. more values sent to one subcomponent than the other.

**Theorem 4.2.6 Balanced Wagging on any pair of \((\kappa, \delta)\) - optimal buffers**

Let \(X, Y\) be \((\kappa, \delta)\) - optimal buffers with capacity \(k_X = k_Y\) and equal i/o-distance \(\delta_X = \delta_Y\). Then the wagging composite \(\text{WAG}_{0,2}(X, Y)\) is a \((\kappa, \delta)\) - optimal buffer with capacity \(k_{\text{WAG}} = k_X + k_Y + 2\) and i/o-distance \(\delta_{\text{WAG}} = \delta_X + 2 = \delta_Y + 2\).

**Proof**

Let \(X, Y\) be \((\kappa, \delta)\) - optimal buffers with capacity \(k_X = k_Y\) and equal i/o-distance \(\delta_X = \delta_Y\). Buffer \(X\) is \((\kappa, \delta)\) - optimal. So there exists a sequence function \(\sigma_X\) with constant latency \(\lambda_X\) and constant cycle time \(y_X\). Buffer \(Y\) is \((\kappa, \delta)\) - optimal. So there exists a sequence function \(\sigma_Y\) with constant latency \(\lambda_Y\) and constant cycle time \(y_Y\), such that \(y_X = y_Y\) and \(\lambda_X = \lambda_Y\). Then by theorem 3.4.3 there exists a sequence function \(\sigma_{\text{WAG}}\) for the wagging composite \(\text{WAG}(X, Y)\) with constant cycle time \(y_{\text{WAG}}\), such that \(2\gamma_{\text{WAG}} = y_X = y_Y\) and constant latency \(\lambda_{\text{WAG}} = \lambda_X + 2 = \lambda_Y + 2\). According theorems 2.6.2 and 2.7.2 the wagging composite \(\text{WAG}(X, Y)\) is a buffer with capacity \(k_{\text{WAG}} = k_X + k_Y + 2\) and i/o-distance \(\delta_{\text{WAG}} = \delta_X + 2 = \delta_Y + 2\). Hence it remains to be shown that when we take the early sequence function \(\sigma_X\) that the latency \(\lambda_{\text{WAG}}\) equals the i/o-distance \(\delta_{\text{WAG}}\) and for the late sequence function \(\sigma_X\) that the latency \(\lambda_{\text{WAG}}\) equals \(\gamma_{\text{WAG}} \cdot k_{\text{WAG}} - \delta_{\text{WAG}}\). We first start with the early sequence function.

\[
\lambda(\sigma_{\text{WAG}}) = \{\text{theorem 3.4.3 : } g = 1 \text{ early sequence} \}
\]

\[
\lambda_{\text{WAG}} + 2
\]
4.2 Construction of \((\kappa, \delta)\) - optimal buffers

\[
\begin{align*}
\lambda_{\text{WAG}} &= \{ \text{theorem 3.4.3: late sequence} \\
\lambda_{\text{WAG}} &= 2(\gamma_{\text{WAG}}) \} \\
\lambda_{\text{WAG}} &= \{ \text{calculus} \} \\
\gamma_{\text{WAG}} &= \gamma_{\text{WAG}} \cdot (2\kappa_{\text{WAG}} + 2) \cdot (\delta_{\text{WAG}} + 2) \\
\lambda_{\text{WAG}} &= \lambda_{\text{WAG}} \cdot \kappa_{\text{WAG}} - \delta_{\text{WAG}}
\end{align*}
\]

The latency \(\lambda_{\text{WAG}}\) of the late sequence function equals

**Theorem 4.2.7 Wagging optimality**

For some \(k, l, 0 \leq k, l\), let \(X, Y\) be \((\kappa, \delta)\) - optimal buffers with capacity \((l-1)\kappa_x=\kappa_y\) and equal i/o-distance \(\delta_x=\delta_y\). Then the wagging buffer \(\text{WAG}_{k,l}(X, Y)\) is also \((\kappa, \delta)\) - optimal with i/o-distance \(\delta_{\text{WAG}} = \delta_x+2 = \delta_y+2\) and capacity \(\kappa_{\text{WAG}} = \kappa_x+\kappa_y+2\).

**Proof**

For some \(k, l, 0 \leq k, l\), let \(X, Y\) be \((\kappa, \delta)\) - optimal buffers with capacity \((l-1)\kappa_x=\kappa_y\) and equal i/o-distance \(\delta_x=\delta_y\). Buffer \(X\) is \((\kappa, \delta)\) - optimal. So there exists a sequence function \(\sigma_x\) with constant latency \(\lambda_x\), and constant cycle time \(\gamma_x=\gamma_y\). Buffer \(Y\) is \((\kappa, \delta)\) - optimal. So there exists a sequence function \(\sigma_y\) with constant latency \(\lambda_y\), and \((\gamma, k, l)\) periodic cycle time \(\gamma_y\) such that \(\lambda_x=\lambda_y\). Then by theorem 3.4.3 there exists a sequence function \(\sigma_{\text{WAG}}\) for the wagging composite \(\text{WAG}(X, Y)\) with constant cycle time \(\gamma_{\text{WAG}}\), such that \(\gamma_{\text{WAG}}=\gamma_y\) and constant latency \(\lambda_{\text{WAG}} = \lambda_x + 2 = \lambda_y + 2\). According theorems 2.6.2 and 2.7.2 the wagging composite \(\text{WAG}(X, Y)\) is a buffer with capacity \(\kappa_{\text{WAG}} = \kappa_x+\kappa_y+2\) and i/o-distance \(\delta_{\text{WAG}} = \delta_x+2 = \delta_y+2\). Hence it remains to be shown that when we take the early sequence function \(\sigma_x\), the latency \(\lambda_{\text{WAG}}\) equals the i/o-distance \(\delta_{\text{WAG}}\) and for the late sequence function \(\sigma_y\) that the latency \(\lambda_{\text{WAG}}\) equals \(\gamma_{\text{WAG}}-\kappa_{\text{WAG}} = \delta_{\text{WAG}}\). We first start with the early sequence function.

\[
\begin{align*}
\lambda(\sigma_{\text{WAG}}) &= \{ \text{theorem 3.4.3: } g=1 \text{ early sequence} \} \\
\lambda(\sigma_{\text{WAG}}) &= \{ \text{calculus} \} \\
\lambda(\sigma_{\text{WAG}}) &= \{ \sigma_{\text{WAG}} \text{ early so } \lambda_{\text{WAG}} = \delta_{\text{WAG}} \} \\
\lambda(\sigma_{\text{WAG}}) &= \{ \delta_{\text{WAG}} = \delta_{\text{WAG}} + 2 \} \\
\delta_{\text{WAG}}
\end{align*}
\]
The latency $\lambda_{WAG}$ of the late sequence function equals

$$\lambda(\tilde{F}_{WAG})$$

$$=\{\text{theorem 3.4.3: late sequence}\}$$

$$(\alpha_0^2 + 2(\gamma_{WAG} - 1)) \uparrow (\alpha_0^2 + 2(\gamma_{WAG} - 1))$$

$$\{\tilde{F}_{s(v)} \text{ late so } \lambda(s(v)) = \gamma_{s(v)} \cdot \kappa_{s(v)} - \delta_{s(v)}\}$$

$$\{\gamma_0 \cdot \kappa_0 + 2(\gamma_{WAG} - 2) \uparrow (\gamma_0 \cdot \kappa_0 + 2(\gamma_{WAG} - 2))$$

$$=\{\delta_{WAG} = \delta_{s(v)} + 2\}$$

$$\{\gamma_0 \cdot \kappa_0 + 2(\gamma_{WAG} - 2) \uparrow (\gamma_0 \cdot \kappa_0 + 2(\gamma_{WAG} - 2))$$

$$=\{\gamma_{WAG} = 1/l(\gamma_{WAG} \cdot \gamma_{WAG} - 1/l(1-l) \cdot \gamma_{WAG})\}$$

$$=\{\text{calculus}\}$$

$$\gamma_{WAG}((1/l(1-l) \cdot \kappa_0 + 2) - \delta_{WAG}$$

$$=\{\kappa_{WAG} = 1/l(1-l) \cdot \kappa_0 + 2 \land \kappa_{WAG} = 1/l(1-l) \cdot \kappa_0 + 2\}$$

$$\gamma_{WAG} = \delta_{WAG}$$

As an example consider the wagging composite $WAG_{0,3}$ of the binary buffer $BBuf_3$ and the linear buffer $LBuf_5$. For graphical reasons we call this system the Diamond buffer. A diagram of the Diamond buffer with the early sequence function is shown in figure 4.2.1. The late sequence function is shown in figure 4.2.2. The Diamond buffer has capacity $\kappa = 17$ and i/o-distance $\delta = 7$. According to definition 4.1.7 for the diamond buffer to be $(\kappa, \delta)$-minimal there must exist a sequence function such that the latency equals the i/o-distance. A sequence function with this latency, $\lambda = 7$, is presented in the diagram of figure 4.2.1. So the Diamond buffer is $(\kappa, \delta)$-minimal. Note that the same conclusion can be made using theorem 4.2.5, because the Diamond buffer is the wagging composite of two optimal buffers: the linear buffer $LBuf_5$ and the binary tree buffer $BBuf_3$, composed with a generic instance of a split component $Split_{0,3}$ and a generic instance of a merge component $Merge_{0,3}$. Hence we then apply theorem 4.2.5 with the value of $l=3$, which requires that one subcomponent has twice the capacity of the other one, which is indeed the case. So the Diamond buffer is $(\kappa, \delta)$-optimal, hence $(\kappa, \delta)$-minimal.

![Figure 4.2.1 the Diamond buffer with early sequence](image-url)
4.2 Construction of \((\kappa, \delta)\) - optimal buffers

![Diagram](image)

**Figure 4.2.2 the Diamond buffer with late sequence**

The Diamond buffer has capacity \(\kappa = 17\) and i/o-distance \(\delta = 7\). According to definition 4.1.6 for the diamond buffer to be \((\kappa, \delta)\)-maximal there must exist a sequence function such that the latency equals the product of the cycle time and capacity minus the i/o-distance. A sequence function with this latency, \(\lambda = 27\), is presented in the diagram of figure 4.2.2. So the Diamond buffer is \((\kappa, \delta)\)-maximal. Note that the same conclusion is already made. We already concluded that the Diamond buffer is \((\kappa, \delta)\) - optimal, hence \((\kappa, \delta)\) - maximal.

4.2.3 Multi-wagging and constructing \((\kappa, \delta)\) - optimal buffers

**Theorem 4.2.8 Multi-wagging optimality**

Let \(X_0, \ldots, X_{n-1}\) be \((\kappa, \delta)\) - optimal buffers with equal capacities \(\kappa_x\) and i/o-distances \(\delta_x\). Then the multi-wagging composite \(MW(X_0, \ldots, X_{n-1})\) is an \((\kappa, \delta)\) - optimal buffer with capacity \(\kappa_{mw} = n(\kappa_x + 2)\) and i/o-distance \(\delta_{mw} = \delta_x + n + 1\).

**Proof**

Let \(X_0, \ldots, X_{n-1}\) be \((\kappa, \delta)\) - optimal buffers with equal capacities \(\kappa_x\) and i/o-distances \(\delta_x\). Buffers \(X_0, \ldots, X_{n-1}\) are \((\kappa, \delta)\) – optimal. So there exist sequence functions \(\sigma_{x0}, \ldots, \sigma_{xn-1}\) with equal and constant latency \(\lambda_x\), and equal and constant cycle time \(y_x\). Then by theorem 3.4.5 for some integers \(g_0, \ldots, g_n\) such that 
\((\forall j): 0 \leq j < n : 0 < g_j < \gamma\), and integers \(v_1, \ldots, v_{n-1}\) such that 
\((\forall j): 0 < j < n : 0 \leq v_j \leq \gamma\) there exists a sequence function \(\sigma_{mw}\) for the multi-wagging composite \(MW\) \((X_0, \ldots, X_{n-1})\) with constant cycle time \(y_{mw}\), such that 
\(y_{mw} = n y_{mn}\) and constant latency \(\lambda_{mw} = \lambda + g_0 + g_1 + \cdots + g_n + v_1 + \cdots + v_{n-1}\).
According theorems 2.6.3 and 2.7.3 the wagging composite MW is a buffer with capacity $\kappa_{MW} = n'(\kappa_x+2)$ and i/o-distance $\delta_{MW} = \delta_x + n + 1$. Hence it remains to be shown that when we take the early sequence function $\tilde{\sigma}_x$ the latency $\lambda_{MW}$ equals the i/o-distance $\delta_{MW}$ and for the late sequence function $\hat{\sigma}_x$ that the latency $\lambda_{MW}$ equals $y_{MW} - \kappa_{MW} - \delta_{MW}$. We first start with the early sequence function.

$$
\lambda(\tilde{\sigma}_{MW}) = \lambda_x + n + 1
$$

- theorem 3.4.5 : early sequence 

- $\{\tilde{\sigma}_x$ early so $\lambda_x = \delta_x\}$
- $\delta_x + n + 1$

$$
\delta_{MW} = \delta_x + n + 1
$$

The latency $\lambda_{MW}$ of the late sequence function equals

$$
\lambda(\hat{\sigma}_{MW}) = \lambda_x + (n + 1)(y_{MW} - \delta_x) + (n - 1)y_{MW}
$$

- $\{\hat{\sigma}_x$ late so $\lambda_x = y_{MW} - \delta_x\}$
- $y_{MW} - \delta_x + (n + 1)(y_{MW} - \delta_x) + (n - 1)y_{MW}$

$$
\delta_{MW} = n\cdot y_{MW}
$$

- $n\cdot y_{MW} - \delta_x + 2n\cdot y_{MW} - n - 1$

$$
\delta_{MW} = \delta_x + n + 1
$$

$$
\delta_{MW} = \delta_x + n + 1
$$

### 4.3 Classes of $(\kappa, \delta)$ - optimal buffers

In the previous section we have concluded that, when using the construction methods with $(\kappa, \delta)$ - optimal buffers as its subcomponents, new $(\kappa, \delta)$ - optimal buffers are created. For every combination $(\kappa, \delta)$, $\delta \leq \kappa$ we like to find a buffer that is $(\kappa, \delta)$ - optimal. We will divide the buffers to be obtained into three buffer classes:

- **class a**
  - Let $X$ be a buffer then
  
  - $X \in \text{class } a = X \in B_1 \land X \text{ is } (\kappa, \delta) - \text{optimal}$

- **class b**
  - Let $X$ be a buffer then
  
  - $X \in \text{class } b = X \in B_2 \land X \text{ is } (\kappa, \delta) - \text{optimal}$

- **class c**
  - Let $X$ be a buffer then
  
  - $X \in \text{class } c = X \in B = \bigcup_{n=3}^{\infty} B_1 \land X \text{ is } (\kappa, \delta) - \text{optimal}$

where $B$ is the family of buffers as defined in section 2.4

As a consequence, class a $\subseteq$ class b $\subseteq$ class c.
Of course the pairs of combination \((k, \delta)\) is bounded by \(\delta \leq k\). The capacity of a buffer is always at least its i/o-distance. The design of the basic components raises another boundary. The basic components split and merge have a fan-in and fan-out of 2. One expects the relation between the capacity and i/o-distance for binary tree buffers to be another boundary: \(\delta \leq \log((k+2)/3)+1\).

Another boundary is given by more practical design. The length of the wire in the design of a buffer is the bottleneck. The i/o-distance of a buffer is at least the square root of its capacity: \(\delta \geq \text{Upper}(\sqrt{k})\). In table 4.1 the pairs of combination with \(\delta < \text{Upper}(\sqrt{k})\) are filled with a X.

4.3.1 Class a of \((k, \delta)\) - optimal buffers

The buffers of class a are buffers that can be composed, are buffers consisting of basic component Buf. The only construction method that uses only basic component Buf is the serial composition (SER). The buffers that can be designed are the linear buffers LBuf\(_n\), \(1 \leq n\). The linear buffers are shown in table 4.1 with letter a.

An algorithm for generating the buffers of class a is given by

```plaintext
//------------------------------------------------------------------------
Const \(\delta_{\text{MAX}}\): ...
Const \(k_{\text{MAX}}\): ...
Array IsOpt[\(\delta_{\text{MAX}}, k_{\text{MAX}}\)] of Boolean

Proc CalcClassA(k : int, d : Int)
    if \(k < k_{\text{MAX}} \land d < \delta_{\text{MAX}} \land d \leq k\)
        if IsOpt(k, d) \rightarrow skip
        if not IsOpt(k, d) \rightarrow
            IsOpt(k, d) := True
            {serial composition}
            Call CalcClassA(k + 1, d + 1)
    fi
End Proc
//------------------------------------------------------------------------

Algorithm 4.1 CalcClassA

The procedure call is CalcClassA(1,1)

4.3.2 Class b of \((k, \delta)\) - optimal buffers

The buffers of class b are buffers that can be composed, are buffers consisting of the basic components Buf, SPLIT\(_k\), MERGE\(_k\), \(0 \leq k \leq 2\). According to the wagging theorem we can construct new buffers \(Y\) that are \((k, \delta)\) - optimal with i/o-distance \(\delta = \delta + 2\) and capacity \(k = 2 \cdot k + 2\) by \(Y = \text{WAG}_{2}(X, X)\) where \(X\) is also \((k, \delta)\) - optimal. According to the serial composition theorem we can construct new buffers \(Y\) that are \((k, \delta)\) - optimal with i/o-distance \(\delta = \delta + n\) and capacity \(k = k + n\) by \(Y = \text{WAG}_{2}(X, \text{LBuf}_n)\) where \(X\) is also \((k, \delta)\) - optimal.

Define the family of \(b_{\delta}\) buffers \{\(b_{\delta, k}\)\}_{\delta = 1 \text{ to } \infty, \ k = 1 \text{ to } \infty} \ by

\[\begin{align*}
b_{1,1} &= \{\text{LBuf}_1\} \\
b_{2,2} &= \{\text{LBuf}_2\} \\
b_{3,3} &= \{\text{LBuf}_3\}
\end{align*}\]
\( b_{3,k} = \{ BB_2 \} \)
\( b_{2,k} = \{ \text{Ser}(b_{2,1:k-1}, \text{Buf}) \}, 2 \leq \delta \leq k \land k \mod 2 = 1 \)
\( b_{2,k} = \{ WAG_{i,2}(b_{2,2:k-2}, b_{2,2:k-2}) \} \cup \{ \text{Ser}(b_{2,1:k-1}, \text{Buf}) \}, 2 \leq \delta \leq k \land k \mod 2 = 0 \)

An algorithm for generating the buffers of class b is given by

\[
\begin{align*}
\text{Const } & \delta_{\text{MAX}} = \ldots \\
\text{Const } & \kappa_{\text{MAX}} = \ldots \\
\text{Array } & \text{IsOpt}[\delta_{\text{MAX}}, \kappa_{\text{MAX}}] \text{ of Boolean} \\
\text{Proc } & \text{CalcClassB}(k : \text{int}, d : \text{Int}) \\
\text{var } & i : \text{int}; \\
& \text{if } k < \kappa_{\text{MAX}} \land d < \delta_{\text{MAX}} \land d \leq k \rightarrow \\
& \quad \text{if } \text{IsOpt}(k, d) \rightarrow \text{skip} \\
& \quad [] \not\text{IsOpt}(k, d) \rightarrow \\
& \quad \text{IsOpt}(k, d) := \text{True} \\
& \quad \{ \text{serial composition} \} \\
& \quad \text{Call } \text{CalcClassB}(k + 1, d + 1) \\
& \quad \{ \text{wagging } WAG_{i,2} \} \\
& \quad \text{Call } \text{CalcClassB}(2 \cdot k + 2, d + 2) \\
& \text{fi} \\
& \text{fi} \\
\text{End Proc}
\end{align*}
\]

Algorithm 4.2 CalcClassB

The procedure call is CalcClassB(1,1)

4.3.3 Class c of \((\kappa, \delta)\) - optimal buffers

The buffers of class c are buffers that can be composed, are buffers consisting of the basic components Buf, Split\(k\), Merge\(k\), \(0 \leq k < \ell\).

An algorithm for generating the buffers of class c is given by

\[
\begin{align*}
\text{Const } & \delta_{\text{MAX}} = \ldots \\
\text{Const } & \kappa_{\text{MAX}} = \ldots \\
\text{Const } & M_{\text{MAX}} = \ldots \\
\text{Array } & \text{IsOpt}[\delta_{\text{MAX}}, \kappa_{\text{MAX}}] \text{ of Boolean} \\
\text{Proc } & \text{CalcClassC}(k : \text{int}, d : \text{Int}) \\
\text{var } & n : \text{int} ; i : \text{int} \\
& \text{if } k < \kappa_{\text{MAX}} \land d < \delta_{\text{MAX}} \land d \leq k \rightarrow \\
& \quad \text{if } \text{IsOpt}(k, d) \rightarrow \text{skip} \\
& \quad [] \not\text{IsOpt}(k, d) \rightarrow \\
& \quad \text{IsOpt}(k, d) := \text{True} \\
& \quad \{ \text{serial composition} \} \\
& \quad \text{Call } \text{CalcClassC}(k + 1, d + 1) \\
& \quad \{ \text{wagging } WAG_{i,2} \}
\end{align*}
\]
4.3 Classes of \((k, \delta)\) - optimal buffers

Call CalcClassC\((2 \cdot k + 2, d + 2)\)
\{wagging WAG_{k1}, t \geq 2\}
i := 2 \{ i := i + 1 \}
do \ k \geq d \ast i \{ \kappa_y < \delta_y * i \land \delta_x = \delta_y \land \delta_x \leq \kappa_y \Rightarrow \kappa_x < \kappa_x * i \} 
If \ k \mod i = 0 \rightarrow \{ \forall i : i \ast \kappa_x = \kappa_y \} 
Call CalcClassC \((i + 1) \ast k / i + 2, d + 2)\)
If
i := i + 1
od
\{multi-wagging\}
for \ n = 2 \ to \ \text{MAX}\nCall CalcClassC\((n \cdot (k + 2), n + d + 1)\)
\text{ref} \ n
\text{fi}
\text{fi}
End Proc

Algorithm 4.3 CalcClassC

The procedure call is CalcClassC\((1, 1)\)

The next step is to provide an algorithm that for given capacity \(k\) and i/o-distance \(\delta\) a Boolean value returns indicating a \((k, \delta)\) - optimal buffer of class a, b or c exists.

Function BufferIsKappaDeltaOptimalOfClassABC\((k: \text{int}, d: \text{int}) : \text{Boolean}\)

\text{var} \ b : \text{Boolean}
\text{m : int}
if \ k = 1 \land d = 1 \rightarrow
\ b := \text{True}
\fi
b := \text{False}
\{ serial composition \}
if \ k > 1 \land d > 1 \rightarrow
\ b := b \lor \text{BufferIsKappaDeltaOptimalOfClassABC}(k - 1, d - 1)
\fi
\{wagging\}
if \ k > 2 \land d > 2 \rightarrow
\ if \ (k - 2) \mod 2 = 0 \rightarrow
\ b := b \lor \text{BufferIsKappaDeltaOptimalOfClassABC}((k - 2) / 2, d - 2)
\fi
\fi
\{multi-wagging\}
for \ m = 1 \ to \ k - 1
\ if \ k \mod m = 0 \rightarrow
\ if \ k / m > 2 \rightarrow
\ if \ d + m + 1 > 0 \rightarrow
\ b := b \lor \text{BufferIsKappaDeltaOptimalOfClassABC}(k / m - 2, d - m - 1)
\fi
Algorithm 4.4 BufferIsKappaDeltaOptimalOfClassABC

The procedure call is BufferIsKappaDeltaOptimalOfClassABC (k, d)
The results of the algorithms are shown in table 4.1.

| $\kappa$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 |
| $\delta$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Table 4.1 ($\kappa, \delta$) - optimal buffers of classes $a$, $b$ and $c$

Table 4.1 is interpreted as follows. For example take position $(\kappa, \delta) = (16, 8)$. We are able to construct a $(\kappa, \delta)$ - optimal buffer of class $b$. The construction of this buffer can be derived by following the diagonal up, where $\kappa - \delta$ is constant i.e. $\kappa - \delta = 8$, until the last $b$ can be found. The diagonal step of size $n$ is a
4.3 Classes of \((\kappa, \delta)\) - optimal buffers

serial composition of a linear buffer \(LBuf_a\) and the \((\kappa, \delta)\) - optimal buffer \((\kappa - n, \delta - n)\). For the example we can follow the diagonal up 2 steps where the last \(b\) on this diagonal can be found, on position \((\kappa, \delta) = (14, 6)\). So buffer \((\kappa, \delta) = (16, 8)\) can be constructed by a serial composition of a linear buffer \(LBuf_2\) and the \((\kappa, \delta)\) - optimal buffer \((14, 6)\). Optimal buffer \((\kappa, \delta) = (14, 6)\) can be constructed by a wagging composition. We construct a wagging composition of two equal buffers. The buffer in the wagging construction has capacity \((\kappa - 2) / 2\) and i/o-distance, \(\delta - 2\). So optimal buffer \((\kappa, \delta) = (14, 6)\) is constructed using a wagging composition of two optimal buffers \((\kappa, \delta) = (6, 4)\). Again we construct optimal buffer \((\kappa, \delta) = (6, 4)\) using a wagging composition of two optimal buffers \((\kappa, \delta) = (2, 2)\) we can follow the diagonal up 1 step where the last \(b\) on this diagonal can be found, on position \((\kappa, \delta) = (1, 1)\) which equals the basic component Buf. As a result the buffer \((\kappa, \delta) = (16, 8)\) can be defined by

\[
\text{SER}(\text{WAG}_{1,2}(\text{WAG}_{0,2}(\text{SER}(\text{Buf}, LBuf_1), \text{SER}(	ext{Buf}, LBuf_1)), \text{WAG}_{0,2}(\text{SER}(\text{Buf}, LBuf_1), \text{SER}(	ext{Buf}, LBuf_1))), LBuf_2)
\]

Of course this notation equals

\[
\text{SER}(\text{WAG}_{1,2}(\text{WAG}_{0,2}(LBuf_2, LBuf_2), \text{WAG}_{0,2}(LBuf_2, LBuf_2)), LBuf_2)
\]

Note that we can derive a different construction for the \((\kappa, \delta)\) - optimal buffer with capacity \(\kappa = 16\) and \(\delta = 8\) when we start the above method by first constructing the optimal buffer \((\kappa, \delta) = (16, 8)\) using a wagging composition of two optimal buffers \((\kappa, \delta) = (7, 6)\). Then the optimal buffer \((\kappa, \delta) = (7, 6)\) is constructed using a serial composition of \(LBuf_2\) and an optimal buffer \((\kappa, \delta) = (4, 3)\). The optimal buffer \((\kappa, \delta) = (4, 3)\) is a wagging composition of the basic component Buf. This method also derives a \((\kappa, \delta)\) - optimal buffer with capacity \(\kappa = 16\) and \(\delta = 8\) with definition

\[
\text{WAG}_{0,2}(\text{SER}(\text{WAG}_{3,2}(\text{Buf}, Buf), LBuf_1), \text{SER}(\text{WAG}_{0,2}(\text{Buf}, Buf), LBuf_1))
\]

A range of buffers with varying i/O-distance can be constructed for some given capacity. The last step is to provide an algorithm that for given capacity \(\kappa\) the smallest value of the capacity of a \((\kappa, \delta)\) - optimal buffer of class \(a\), \(b\) or \(c\) provides.

//------------------------------------------------------

Function BufferGivenKappaCalcMinDelta (k :int) : int

var b : boolean
    d : int
    d:=1
    do d < k \& not b
        if BufferIsKappaDeltaOptimalOfClassABC(k,d) ->
            b:=true
            []
        d:=d+1
    fi
    od
BufferGivenKappaCalcMinDelta:=d

End Function

//------------------------------------------------------

Algorithm 4.5 BufferGivenKappaCalcMinDelta

The procedure call is BufferGivenKappaCalcMinDelta(k)
In table 4.1 the possible values of the I/O-distance for a fixed capacity are given by a horizontal row. The smallest value for a buffer of classes a, b or c is given by the most left filled position with the corresponding letter of that buffer class. This relation, the minimum I/O-distance as function of the capacity of a buffer of class a, b or c is shown graphically in the diagram of figure 4.3.1.

![Figure 4.3.1 Minimum I/O-distance as function of the capacity (κ ≤36)](image)

Our goal in designing buffers is to construct a buffer with maximal capacity and minimal i/o-distance. Figure 4.3.1 shows that using the Wagging composition with the buffers from buffer class a, i.e. buffer class b, the minimum i/o-distance significantly decreases. If we add the construction method Multi-Wagging, hence construct buffer class c, for some values of the capacity we find a buffer with a smaller i/o-distance. The results are shown in figure 4.3.2.

![Figure 4.3.2 Minimum I/O-distance as function of the capacity (κ ≤1000)](image)
4.4 \((\kappa, \delta)\) - minimal and maximal buffers

In the previous sections we defined and constructed \((\kappa, \delta)\) - optimal buffers. A \((\kappa, \delta)\) - optimal buffer has for any value of the cycle time an early and a late sequence function that attains its optimal bounds. In this section we are interested in buffers that for certain values of the cycle time have sequence functions that attain their optimal bounds. We start our search for those buffer designs that are constructed using wagging composition.

Consider the wagging composition \(WAG_{0,2}\) of two \((\kappa, \delta)\) - optimal subcomponents \(X\) and \(Y\).

![Wagging Composition Diagram](image)

*Figure 4.4.1 Wagging \(WAG_{0,2}(X,Y)\)*

The wagging composite has cycle time \(\gamma\), so the cycle times of buffers \(X\) and \(Y\) equal half the cycle time of the wagging composite, \(\gamma_x = \gamma_y = 2\gamma\). The buffers \(X\) and \(Y\) are optimal. Hence there exist sequence functions for \(X\) and \(Y\) that attain their optimal bounds. The wagging composition \(WAG_{0,2}\) requires that the latencies of \(X\) and \(Y\) are equal, \(\lambda_x = \lambda_y\). We split our search for the values of the cycle time of sequence functions that attain their optimal bounds in the upper bound, i.e. maximal optimal, and lower bound, i.e. minimal optimal.

4.4.1 \((\kappa, \delta)\)-maximal buffers

The wagging composition \(WAG_{0,2}\) requires that the latencies \(\lambda_x\) and \(\lambda_y\) of the sequence functions are equal. Consider the upper bound for the latency of a sequence function for optimal buffers.

\[
\begin{align*}
\lambda_x &= \lambda_y \\
&= \{ \text{X and Y are} \ (\kappa, \delta) \ - \text{optimal} \} \\
2\gamma \kappa_x - \delta_x &= 2\gamma \kappa_y - \delta_y \\
&= \{ \text{calc} \} \\
2\gamma \ (\kappa_x - \kappa_y) &= \delta_x - \delta_y
\end{align*}
\]

Let us find two buffers that satisfy the equation such that \(\gamma = 2\) and \(\kappa_x \neq \kappa_y\). The first buffers \(X\) and \(Y\) that are \((\kappa, \delta)\) - optimal and satisfy equation \(4(\kappa_x - \kappa_y) = \delta_x - \delta_y\) are buffer \(X\) with capacity \(\kappa=10\) and i/o-distance \(\delta=9\) and buffer \(Y\) with \(Y(\kappa, \delta) = (9,5)\). Buffer \(X\) is serial composition of \(LBuf_9\) and \(BBuf_2\). Buffer \(Y\) is a rectangular buffer \(RBuf_{3,3}\). The wagging composition of buffers \(X\) and \(Y\), \(WAG_{0,2}(X,Y)\), gives the buffer \(WAG_{0,2}(SER(LBuf_9, BBuf_2), RBuf_{3,3})\). In figure 4.4.2 and 4.4.3 the early and late sequence functions are shown for this buffer \(WAG_{0,2}(SER(LBuf_9, BBuf_2), RBuf_{3,3})\).
Figure 4.4.2 $WAG_{0,2}(SER(LBuf_{6}, BBuf_{3}), RBuf_{3,3})$ with early sequence
Figure 4.4.3 $WAG_{0,2}(SER(LBuf_5, BBuf_3), RBuf_{3,3})$ with late sequence
Figure 4.4.3 shows that for this buffer the latency of the late sequence function equals $\lambda_y=33$. According to definition 4.1.8 the buffer is $(\kappa,\delta)$- maximal if $\lambda = \gamma \cdot \kappa - \delta = 2 \cdot 21 - 7 = 35$. So according to definition 4.1.8 the buffer is not $(\kappa,\delta)$- maximal. Nevertheless the buffer is optimal occupied. Hence we introduce a new definition for buffers that are $(\kappa,\delta)$-average maximal.

**Definition 4.4.1 $(\kappa,\delta)$-average maximal**

Let $X$ be a buffer with capacity $\kappa_x$ and i/o-distance $\delta_x$. Then $X$ is $(\kappa,\delta)$-average maximal if for all cycle time $\gamma_x$, such that $2 \leq \gamma_x$, there exists a sequence function $\sigma_x$ with average cycle time $\Gamma(\sigma_x) = \gamma$ where the average latency equals the product of the average cycle time and capacity minus the average i/o-distance $\Delta_x$.

$$\Delta(\sigma) = \lim_{i \to \infty} \frac{1}{i} \sum_{j=0}^{i-1} \sigma_j(\sigma)$$

The wagging buffer however does not suffice this definition. The buffer is only $(\kappa,\delta)$-average maximal for $\gamma=2$. Buffers, which are only for certain values of the cycle time optimal, are called *partial* $(\kappa,\delta)$-optimal.

Hence the wagging buffer WAG$_{0,2}$(SER(LBuf$_8$, BBuf$_6$), RBuf$_{3,3}$) is partial $(\kappa,\delta)$-average maximal for $\gamma=2$.

We conclude that, for the wagging composition WAG$_{0,2}$ of two $(\kappa,\delta)$-optimal buffers $X$ and $Y$ with capacities $\kappa_x$ and $\kappa_y$ and i/o-distances $\delta_x$ and $\delta_y$, such that $\kappa_x \neq \kappa_y$, exactly one value for the cycle time exists that satisfies equation $2\gamma (\kappa_x - \kappa_y) = \delta_x - \delta_y$. Then the wagging composite is partial $(\kappa,\delta)$-average maximal.

**4.4.2 $(\kappa,\delta)$-minimal buffers**

The wagging composition WAG$_{0,2}$ requires that the latencies $\lambda_x$ and $\lambda_y$ of the sequence functions are equal. Consider the lower bound for the latency of a sequence function for optimal buffers.

$$\lambda_x = \lambda_y$$

$$= \{ X \text{ and } Y \text{ are } (\kappa,\delta) - \text{optimal} \}$$

$$\delta_x = \delta_y$$

Hence we can conclude that when the latencies $\lambda_x$ and $\lambda_y$ are equal for any cycle time $\gamma$, that the i/o-distance of the subcomponents must be equal, $\delta_x = \delta_y$.

Let us take a look at equal latencies for a particular range of values for the cycle time. Consider the wagging of figure 4.4.4.

![Figure 4.4.4 Wagging WAG$_{0,2}$ (Empty Buf,Buf)](image-url)
Figure 4.4.4 shows that for this buffer the latency of the early sequence function equals $\lambda = 2\gamma$. According to definition 4.1.9 the buffer is $(\kappa, \delta)$-minimal if $\lambda = \delta = 2$. So according to definition 4.1.9 the buffer is not $(\kappa, \delta)$-minimal. Nevertheless the buffer is minimal occupied. The latency equals the average i/o-distance. Hence we introduce a new definition for buffers that are $(\kappa, \delta)$-average minimal.

**Definition 4.4.2** $(\kappa, \delta)$ – average minimal

Let $X$ be a buffer with capacity $\kappa_x$ and i/o-distance $\delta_x$. Then $X$ is $(\kappa, \delta)$-average minimal if for all cycle time $Y_x$, such that $2 \leq Y_x$, there exists a sequence function $\sigma_x$ with average cycle time $\Gamma(\sigma_x) = \gamma$ where the average latency equals the average i/o-distance $\Delta_x$.

The wagging buffer of figure 4.4.4 however does not suffice this definition. The buffer is only $(\kappa, \delta)$-average minimal for $\gamma \geq 2\gamma$. Hence, analogously, the wagging buffer $WAG_{0.2}$(SER(LBuf6, BBuf2), RBuf3,3) is partial $(\kappa, \delta)$-average minimal for $\gamma \geq 2\gamma$.

Our next step is to derive a lower bound for the cycle time of a wagging composite such that the wagging composite is partial $(\kappa, \delta)$-average minimal.

![Diagram of wagging buffer](image)

**Figure 4.4.5 Wagging $WAG_{0.2}(X, Y)$**

Consider the wagging buffer in figure 4.4.5. We define de i/o-distance of a value that passes through the buffer along buffer X and Y respectively as $\delta_x$ and $\delta_y$. We assume $\delta_x > \delta_y$.

The first input in the wagging buffer is at time slot 0: a(0)=0. To let the buffer be minimal occupied, this value passes along buffer X. Hence the first output is at time slot b(0) = $\delta_x$. The second output is a value that passes along buffer Y. The second output must be at least at time slot b(1) = $\delta_y + 2$. This value was the second input value of the buffer at time slot a(1) = $\delta_x + 2 - \delta_y$. The next input that travels along buffer X is at time slot a(2)= $\delta_x + 2 - \delta_y$. From time slot a(0) and before time slot a(2) we have made two values input in the buffer. Hence $2\gamma \geq \delta_x + 2 - \delta_y + 2$. We conclude that a wagging composite is partial $(\kappa, \delta)$-minimal for $\gamma \geq (\delta_x - \delta_y)/2 + 1$.

Consider the buffer in figure 4.4.4. The i/o-distances are given by $\delta_x = 2$ and $\delta_y = 3$. Hence the wagging buffer is partial minimal for cycle time $\gamma \geq 2\gamma$, which was already shown before.
5 Advanced buffer design

In the previous chapters the number of variables of each basic component used to construct systems is precisely one. In this chapter we explore the consequences of increasing the number of variables (by one), while keeping the i/o-distance the same. As a result we will expect to find optimal buffers that for a fixed i/o-distance have a larger capacity than we found before. The same effect, larger capacity with a fixed i/o-distance, can be reached by increasing the fan-in and fan-out. Extending the number of variables for a basic component increases the occupancy, but still the i/o-distance remains one. We shall present some examples of constructed buffers using basic components with two variables. Furthermore we introduce split and merge components with respectively a fan-out and a fan-in of three. As an example the cubic buffer is introduced.

5.1 Two-place basic components

5.1.1 the two-place buf

The counter part of the one-place buffer with one variable is the two-place buffer, called double buf, and denoted by DBuf. Naturally the buffer specification B=A(0:1)=A holds.

Figure 5.1.1 shows a diagram for this component.

![Diagram of DBuf]

Figure 5.1.1 DBuf

This component is already described in [vBer] where it is known as the wagging buffer, because it alternately stores its input values in respectively variable x and variable y. In this thesis however, we have reserved the name wagging for a more general construction method. Hence we simply refer to this component by the name DBuf.

The communication behavior for the Double Buf component is given by the following program text:

```
DBuf  =  proc (in a; out b)
  [ [ var x,y
     ; a?x;
     (a?y ,b!x ; a?x,b!y)*
     ] ]
```

Here the part that is repeated (forever) is again a sequence of two statements The first statement a?y, b!x prescribes the execution of two events: Input a?y and output b!x. The comma separator indicates that these events may be executed in any order or even simultaneously. Hence a two-place buffer has internal parallelism. Similarly the second statement a?x,b!y prescribes the execution in arbitrary order of an input and an output event.

Note that there are no internal assignments ("x:=y" or y:=x") in the buffer. Therefore no input value passes through two variables. Hence the i/o-distance of DBuf still remains δ=1.
Note that for this component a cycle time of $y=1$ is possible. As we shall see this will also hold for the two remaining two place components. In general we have

**Property 5.1.1.1 Average cycle time lower bound for two-place components**

For any sequence function $\sigma$ of a system $X$ containing two-place components the average cycle time $\Gamma_\sigma$ satisfies the inequality

$$\Gamma_\sigma (\sigma) \geq 1$$

5.1.2 the two-place split

The counter part of the one-place split component $\text{Split}_{k,l}$ is the two-place split, called $\text{DSplit}_{k,l}$. Double split component $\text{DSplit}_{k,l}$ (see figure 5.1.2) has the same functionality as a split component but uses two variables. Hence its functionality is specified by $C = A(k : l)$ and $D = A(k : l)$, $0 \leq k < l$.

![Figure 5.1.2 DSplit_{k,l}](image)

Since the communication behavior is dependent on the values for $k$ and $l$, we need to describe the communication behavior for the odd and even values of $k$ and $l$ separately. Hence the communication behavior of the double split component is given by the following program texts:

```plaintext
DSplit_{k,2l} = proc ( in a; out c,d )
  [ var x,y
    ; a?x
    ; ( (d!x,a?y ; d!y,a?x)^k
      ; c!x,a?y ; d!y,a?x
      ; (d!x,a?y ; d!y,a?x)^k
    )^*
  ], 0 \leq k < l

DSplit_{k+1,2l} = proc ( in a; out c,d )
  [ var x,y
    ; a?x
    ; ( (d!x,a?y ; d!y,a?x)^k
      ; d!x,a?y ; c!y,a?x
      ; (d!x,a?y ; d!y,a?x)^{k-1}
    )^*
  ], 0 \leq k < l
```
\[ \text{DSplit}_{3k,2l+1} = \text{proc} \ (\text{in} \ a; \text{out} \ c,d) \ \\
\begin{align*}
\var x,y \\
a?x \\
( (dx,a?y; dly,a?x)^k \\
; clx,a?y \\
; (dlx,a?x; dly,a?y)^l \\
; cly,a?x \\
; (dlx,a?y; dly,a?x)^{2k-l} \\
)\end{align*} \\
\] , 0 \leq k \leq 1

\[ \text{DSplit}_{2k+1,2l+1} = \text{proc} \ (\text{in} \ a; \text{out} \ c,d) \ \\
\begin{align*}
\var x,y \\
a?x \\
( (dx,a?y; dly,a?x)^k \\
; dlx,a?y; clx,a;x; dlx,a?y \\
; (dlx,a?x; dly,a?y)^{2l} \\
; dly,a?x; clx,a?y; dly,a?x \\
; (dlx,a?y; dly,a?x)^{2k+2l-1} \\
)\end{align*} \\
\] , 0 \leq k \leq 1

5.1.3 the two-place merge

The counter part of the one-place merge component \( \text{Merge}_{x,i} \) is the two-place merge, called double merge. Double merge component \( \text{DMerge}_{x,i} \) has the same functionality as the merge component but uses two variables. The functionality of a \( \text{DMerge}_{2,k} \) component is specified by \( E = B(k:1) \) and \( F = B(k:1^?) \), 0 \leq k < 1.

![Diagram](image)

Figure 5.1.3 \( \text{DMerge}_{2,k} \)

Similar to the communication behavior of the double split, the communication behavior for the double merge is dependent on the values for \( k \) and \( l \) separately. Note that for the double merge component with a value \( k = 0 \), the first input must come along channel \( a \). Therefore we need to describe the communication behavior not only for the odd and even values of \( k \) and \( l \), but also for the case where \( k = 0 \). Hence the communication behavior for the double merge component is given by the following program texts:
5.1 Two-place basic components

\[ \text{DMerge} \] = \[ \text{proc} \ ( \text{in e, f; out b} ) \]
\[ \{ \text{var x,y} \]
\[ ; e?x \]
\[ ; ( (blx,f;y ; bly,f;x)^k \]
\[ ; blx,e;y ; bly,f;x \]
\[ )^* \]
\[ ||, 0 < l \]
\[ \} \]

\[ \text{DMerge} \] = \[ \text{proc} \ ( \text{in e, f; out b} ) \]
\[ \{ \text{var x,y} \]
\[ ; f?x \]
\[ ; ( (blx,f;y ; bly,f;x)^k \]
\[ ; blx,e;y ; bly,f;x \]
\[ ; (blx,f;y ; bly,f;x)^k \]
\[ )^* \]
\[ ||, 0 < k < l \]
\[ \} \]

\[ \text{DMerge} \] = \[ \text{proc} \ ( \text{in e, f; out b} ) \]
\[ \{ \text{var x,y} \]
\[ ; e?x \]
\[ ; ( (blx,f;y ; bly,f;x)^l \]
\[ ; blx,e;y ; \]
\[ ; (bly,f;x ; blx,f;y)^l \]
\[ ; bly,e;x \]
\[ )^* \]
\[ ||, 0 < l \]
\[ \} \]

\[ \text{DMerge} \] = \[ \text{proc} \ ( \text{in e, f; out b} ) \]
\[ \{ \text{var x,y} \]
\[ ; f?x \]
\[ ; ( (blx,f;y ; bly,f;x)^k \]
\[ ; blx,e;y ; bly,f;x \]
\[ ; (blx,f;y ; bly,f;x)^k \]
\[ ; bly,f;x \]
\[ )^* \]
\[ ||, 0 < k \leq l \]
\[ \} \]
\[ \text{DMerge}_{2k+1,2l+1} = \text{proc (in e, f; out b)} \]
\[ \text{|| var x,y } \]
\[ ; f?x \]
\[ ; (b!x,f?y ; b!y,f?x)^k \]
\[ ; b!x,e?y \]
\[ ; (b!y,f?x ; b!x,f?y)^l \]
\[ ; b!y,e?x \]
\[ ; (b!x,f?y ; b!y,f?x)^k \]
\[ \text{||}, 0 \leq k < l \]

5.2 Two-place construction methods

Analogous to the definitions of the construction methods that make use of basic components with one variable, we define construction methods that make use of basic components with two variables.

Of course the definition of the serial composition method is applicable for basic components with more than one variable. No additional subcomponents with two variables for the construction method itself are being used. However applying serial composition using two-place components creates new systems. For instance, consider the family of double linear buffers.

**Definition 5.2.1 Double Linear buffer**

The family of double linear buffers \( \{\text{DLBuf}_n\}_{n=1}^{\infty} \) is defined by

\[
\text{DLBuf}_1 = \text{DBuf} \\
\text{DLBuf}_{n+1} = \text{SER(DBuf}_n,\text{DLBuf}_n), 1 \leq n
\]

Note that the capacity of a double linear buffer, constructed out of \( n \) double bufs, equals the capacity of a linear buffer, constructed out of \( 2n \) bufs. The double buf, however, can operate at cycle time \( \gamma=1 \).

Next we consider the construction method two-place wagging.

**Definition 5.2.2 two-place wagging**

For any pair of two components \( X \) and \( Y \) we define the two-place wagging composite \( \text{DWAG}_{k,l}(X,Y) \) by

\[
\text{DWAG}_{k,l}(X,Y) = \text{proc (in a; out b)} \]
\[ \text{|| chan c,d,e,f} \]
\[ | \text{DSplit}_{k,l}(a, c, d) \]
\[ | X(c, e) \]
\[ | Y(d, f) \]
\[ | \text{DMerge}_{k,l} (e, f, b) \]
\[ \text{||}, 0 \leq k < l \]

Note that subcomponents \( X \) and \( Y \) themselves may contain a mixture of one-place and two-place basic components. Recursive application of double wagging introduces the family of double binary-tree buffers.
5.2 Two-place construction methods

**Definition 5.2.3 Double Binary - Tree buffer**

The family of double binary - tree buffers \( \{DBBuf_n\}_{n=1}^{\infty} \) is defined by

\[
\begin{align*}
DBBuf_1 &= DBuf \\
DBBuf_{n+1} &= DWAC_{0,2}(DBBuf_n, DBBuf_n), \ 1 \leq n \\
&= \text{proc } (\text{in } a; \text{ out } b) \\
&\quad || \text{ chan } c, d, e, f \\
&\quad \quad || DSplit_{0,2}(a, c, d) || DBBuf_n(c, e) || DBBuf_n(d, f) || DMerge_{0,2}(e, f, b) \\
&\quad , \ 1 \leq n
\end{align*}
\]

The last construction method using two-place components to be defined is the two-place multi-wagging. For this construction method we first need to introduce the family of two-place multi-way splitters and the family of two-place multi-way mergers.

**Definition 5.2.4 Two-place multi-way splitter**

The family \( \{DMS_{n}\}_{n=1}^{\infty} \) is defined by

\[
\begin{align*}
DMS_1 &= Buf \\
DMS_{n+1} &= \text{proc } (\text{in } a; \text{ out } c_0, \ldots, c_n) \\
&\quad || \text{ chan } d \\
&\quad \quad || DSplit_{0,1}(a, c_0, d) \\
&\quad \quad || DMS_n(d, c_{n+1}, \ldots, c_n) \\
&\quad , \ 1 \leq n
\end{align*}
\]

**Definition 5.2.5 Two-place multi-way merger**

Next the family \( \{DMM_n\}_{n=1}^{\infty} \) is defined by

\[
\begin{align*}
DMM_1 &= Buf \\
DMM_{n+1} &= \text{proc } (\text{in } f_0, \ldots, f_n; \text{ out } b) \\
&\quad || \text{ chan } e \\
&\quad \quad || DMM_n(f_0, \ldots, f_1, e) \\
&\quad \quad || DMerge_{0,1}(e, f_0, b) \\
&\quad , \ 1 \leq n
\end{align*}
\]

Note that the two-place multi-way splitter and two-place multi-way merger for \( n=1 \) are equal to the one-place buffer \( Buf \). The reason for choosing the one-place buffer in stead of the two-place buffer is given by the intention to create \((k, \delta)\)-optimal buffers. In the remaining part of this section we will introduce the double rectangular buffer, that is created using two-place multi-wagging. As we shall see in later sections this buffer can only attain its optimal bounds if the two ‘corners’ of this design contain two one-place buffers.
Definition 5.2.6 Two-place multi-wagging

For any set of components \(\{X_i\}_{i=0}^{n-1}\) we define the two-place multi-wagging composite \(DMW_n(X_0, \ldots, X_{n-1})\) by

\[
DMW_n(X_0, \ldots, X_{n-1}) = \text{proc} (\text{in} \ a; \text{out} \ b) \\
[\text{chan} \ c_0, \ldots, c_d \\
\text{DMS}_n(a, c_0, \ldots, c_0) \\
\text{DM}_n(X_{n-1}(c_{n-1}, f_{n-1})); \ldots; X_0(c_0, f_0) \\
\text{DM}_n(f_{n-1}, \ldots, f_0, b) \\
], \ 1 \leq n
\]

As an example for this construction method consider the family of double rectangular buffers.

Definition 5.2.7 Double Rectangular buffer

The family of double rectangular buffers \(\{DRBuf_{n,m}\}_{m=2}^{\infty}, \ n=1, 2 \leq m\) is defined by

\[
DRBuf_{n,m} = DMW_n(DLBuf_{n-2}, \ldots, DLBuf_{m-2}), \ 1 \leq n, 2 \leq m
\]

Similar to the family of square buffers we define the family of double square buffers.

Definition 5.2.8 Double Square buffer

The family of double square buffers \(\{DSBuf_n\}_{n=2}^{\infty}\) is defined by

\[
DSBuf_n = DRBuf_{n,n}, \ 2 \leq n
\]

Now we have defined buffer families containing two-place components, we present some characteristics of these buffer designs.

<table>
<thead>
<tr>
<th>Buffer</th>
<th>i/o-distance (\delta)</th>
<th>capacity (\kappa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLBuf_n</td>
<td>(n)</td>
<td>(2n)</td>
</tr>
<tr>
<td>DBBuf_n</td>
<td>(2n+1)</td>
<td>(3 \cdot 2^n - 4)</td>
</tr>
<tr>
<td>DRBuf_{m,n}</td>
<td>(m+n-1)</td>
<td>(2 \cdot m \cdot n\cdot 2)</td>
</tr>
</tbody>
</table>

Table 5.2.1 capacities and i/o-distances of some buffer families with two-place components

5.3 Performance of two-place buffers

Next we derive some relations for the performance parameters, latency and occupancy, between a constructed two-place buffer using the two-place composition methods and its subcomponents. First we start with the two-place serial composition.

Note that the serial composition theorem is still valid for the two-place components. This is because the theorem depends on sequence functions, not on the nature of components being used. The double wagging construction, however, explicitly uses a two-place split and a two-place merge component. Hence we need to define a new theorem.
Theorem 5.3.2 Double Wagging theorem

Let DW be the double wagging composite DWAG_{i,j} (X,Y) of buffers X and Y. For integer γ let \( \sigma_X \) be a sequence function with constant cycle time \( t_1 \) and let \( \sigma_Y \) be a sequence function that is \((\gamma, k, l)\) periodic. Furthermore let \( \sigma_X \) and \( \sigma_Y \) have constant and equal latency \( \lambda \). Then for any integers \( g_1 \) and \( g_2 \), such that \( 0 < g_1 < 2\gamma \) and \( 0 < g_2 < 2\gamma \), there exists a sequence function \( \sigma_{DW} \) for the double wagging composite DWAG_{i,j} (X,Y) with constant cycle time \( \gamma \) and constant latency \( \lambda + g_1 + g_2 \).

**Proof**

Analogous to the proof of theorem 3.4.3. Note that the only difference with respect to the wagging theorem, is the range of values for \( g_1 \) and \( g_2 \).

Note that the double wagging composite can operate at cycle time \( \gamma = 1 \). The subcomponents run at twice the cycle time. Hence it is possible to use one-place basic components in the design of the subcomponents.

As an example consider the double wagging composite DWAG_{0,3} (DBuf, Buf). A diagram for this buffer is shown in figure 5.3.1.

![Diagram 5.3.1 DWAG_{0,3} (DBuf, Buf) with early sequence.](image)

The latency of the sequence functions for the subcomponents DBuf and Buf equals \( \lambda = 1 \). According to the double wagging theorem there must exist for any integers \( g_1 \) and \( g_2 \), such that \( 0 < g_1 < 2\gamma \) and \( 0 < g_2 < 2\gamma \), a sequence function \( \sigma_{DW} \) for the double wagging composite with latency \( \lambda_{DW} = \lambda + g_1 + g_2 \). In figure 5.3.1 the DWAG_{0,3} (DBuf, Buf) with early sequence is shown. For \( g_1 = 1 \) and \( g_2 = 1 \) we have \( \lambda_{DW} = 3 \). Note that the latency equals the i/o-distance and attains the lower bound for a minimal buffer: \( \lambda_{DW} = \delta_{DW} = 3 \).

A sequence function for this double wagging composite with latency that attains the upper bound for a maximal buffer is shown in figure 5.3.2: \( \lambda_{DW} = \gamma k - \delta = 4 \). Hence the buffer is a \((k, \delta)\)-optimal.

![Diagram 5.3.2 DWAG_{0,3} (DBuf, Buf) with late sequence.](image)

Theorem 5.3.3 Double Multi-wagging theorem

Let DMW_{\gamma} (X_0, \ldots, X_{n-1}) be the double multi-wagging composite of buffers \( X_0, \ldots, X_{n-1} \). For integer \( \gamma \) let \( \sigma_{X_0}, \ldots, \sigma_{X_{n-1}} \) be sequence functions for \( X_0, \ldots, X_{n-1} \) with constant cycle time \( \gamma \).

Furthermore let \( \sigma_{X_0}, \ldots, \sigma_{X_{n-1}} \) have constant and equal latency \( \lambda \).

Then for any integers \( g_0, \ldots, g_{n-1} \), such that \( (\forall j: 0 < j < n: 0 < g_j < 2\gamma) \), and integers \( v_1, \ldots, v_{n-1} \) such that \( (\forall j: 0 < j < n: 0 \leq v_j \leq \gamma) \) there exists a sequence function \( \sigma_{DMW} \) for the double wagging composite.
DMW_n (X_0, ..., X_{n-1}) with constant cycle time γ and constant latency λ^t g_0 + g_1 + ... + g_n + v_1 + ... + v_{n-1}.

Proof

Analogous to the proof of theorem 3.4.5 Note that the only difference with respect to the wagging theorem, is the range of values for g_0, ..., g_n. Furthermore note that the range of values for v_1, ..., v_{n-1} is not changed.

As an example consider the double rectangular buffer DRBuf_{3,2}. A diagram of this construction is given in figure 5.3.3.

![Diagram of DRBuf_{3,2} with early sequence.]

The latency of the sequence functions for the subcomponents DBuf equals λ=1. According to the double multi-wagging theorem there must exist, for any integers g_0, ..., g_n, such that (∀j: 0 ≤ j ≤ n: 0 ≤ g_j ≤ 2γ), and integers v_1, ..., v_{n-1} that such that (∀j: 0 ≤ j ≤ n: 0 ≤ v_j ≤ γ), a sequence function σ_{DRBuf,3,3} for the multi-wagging composite with latency λ_{DRBuf,3,3} = λ^t g_0 + g_1 + ... + g_n + v_1 + ... + v_{n-1}. In figure 5.3.3 the DRBuf_{3,2} with early sequence is shown. For g_0 = 1 and v_2 = 0 we have λ_{DRBuf,3,3} = 5. Note that the latency equals the i/o-distance and attains the lower bound for a minimal buffer.

A sequence function for this multi-wagging composite that attains the upper bound for a maximal buffer is shown in figure 5.3.4. The latency of the sequence functions for the subcomponents DBuf equals λ=5. For g_0 = 2γ - 1 = 1 and v_2 = γ = 1 we have λ_{DRBuf,3,3} = 5+4+2=11. Note that the latency equals the i/o-distance and attains the lower bound for a minimal buffer. Hence the buffer is a (κ, δ)-optimal.

![Diagram of DRBuf_{3,2} with late sequence.]

Figure 5.3.4 DRBuf_{3,2} with late sequence.
5.4 Constructing two-place \((\kappa, \delta)\) - optimal buffers

*Theorem 5.4.1* **DBUF is \((\kappa, \delta)\) – optimal**

The two-place basic component DBUF is a \((\kappa, \delta)\) – optimal buffer.

**Proof**

Analogous to the proof of theorem 4.2.1.

*Theorem 5.4.2 Double Wagging optimality*

For some \(k\) and \(l\), \(0 \leq k < l\), let \(X, Y\) be \((\kappa, \delta)\) - optimal buffers with capacity \((l-1)k_x = k_y\) and equal i/o-distance \(\delta_x = \delta_y\). Then the double wagging buffer \(DWAG_{kl}(X,Y)\) is also \((\kappa, \delta)\) - optimal with i/o-distance \(\delta_{dw} = \delta_x + 2 = \delta_y + 2\) and capacity \(k_{dw} = k_x + k_y + 4\).

**Proof**

Analogous to the proof of theorem 4.2.7, but now using the double wagging theorem in stead of the wagging theorem.

*Theorem 5.4.3 Double Multi-wagging optimality*

Let \(X_0, \ldots, X_{n-1}\) be \((\kappa, \delta)\) - optimal buffers with equal capacities \(k_x\) and i/o-distances \(\delta_x\). Then the double multi-wagging composite \(DMW(X_0, \ldots, X_{n-1})\) is a two-place \((\kappa, \delta)\) - optimal buffer with capacity \(k_{dmw} = n(k_x + 4) - 2\) and i/o-distance \(\delta_{dmw} = \delta_x + n + 1\).

**Proof**

Analogous to the proof of theorem 4.2.8, but now using the double multi-wagging theorem in stead of the multi-wagging theorem.

As an example consider the two-place wagging composite \(DWAG_{0,2} (DBuf, DBuf)\)

![Diagram](image)

*Figure 5.4.1* **DWAG_{0,2} (DBuf, DBuf) with early sequence**
This two-place wagging buffer has an early sequence function with cycle time \( \gamma = 1 \). This buffer is \((\kappa, \delta)\)-minimal if the latency for the early sequence function equals the i/o-distance. We defined the i/o-distance as the minimum number of variables that is visited by a value on this path through the buffer. Hence the latency equals the i/o-distance: \( \lambda = \delta = 3 \), and the buffer is \((\kappa, \delta)\)-minimal. This buffer is \((\kappa, \delta)\)-maximal if the latency for the late sequence function equals the product of the cycle time and the capacity minus the i/o-distance. We defined the capacity as the number of variables in the buffer. Hence the latency equals \( \lambda = \gamma \kappa - \delta = 8 - 3 = 5 \), and the buffer is \((\kappa, \delta)\)-maximal.

![Diagram of buffer](image)

*Figure 5.4.2 DWAG\(_{0,2}\) (DBuf, DBuf) with late sequence*

The buffer DWAG\(_{0,2}\) (DBuf, DBuf) is \((\kappa, \delta)\)-optimal with capacity \( \kappa = 8 \) and i/o-distance \( \delta = 3 \).

### 5.5 Classes of two-place \((\kappa, \delta)\)-optimal buffers

In section 4.4 we have shown the classes a, b and c. These buffer classes are \((\kappa, \delta)\)-optimal. The classes were created using the set of basic components \( C \) and the set of construction methods \( M \). The classes a, b and c were represented respectively by buffer classes \( B_1, B_2 \) and \( B_3 \). In the previous sections we have shown the basic components with two variables. Next we define the set of two-place basic components \( DC \) by:

The family of two-place basic components \( \{ DC_i \}_{i=1}^{\infty} \) is defined by

\[
\begin{align*}
DC_1 &= \{ \text{DBuf} \} \cup C_1 \\
DC_{i+1} &= DC_i \cup \{ \text{DSplit}_{k,i} \mid 0 \leq k < i \} \cup \{ \text{DMerge}_{k,i} \mid 0 \leq k < i \} \cup C_{i+1}
\end{align*}
\]

Furthermore, we define the family of construction methods \( \{ DM_i \}_{i=1}^{\infty} \) by

\[
\begin{align*}
DM_1 &= \{ \text{SER} \} = M_1 \\
DM_{i+1} &= DM_i \cup \{ \text{DWAG}_{2,i} \mid 0 \leq k < i \} \cup \{ DMW_{2,i} \mid 2 \leq i \} \cup M_{i+1}
\end{align*}
\]

Let \( DB_i \) be the set of all buffers that can be constructed using the set of basic components \( DC_i \) and using the set of construction methods \( DM_i \).
In section 4.3 we divide the buffers to into three buffer classes: a, b and c. Next we define the buffer classes having buffers containing two-place components:

- **class da**
  Let $X$ be a buffer then
  $X \in \text{class da} = X \in \text{DB}_1 \land X \text{ is } (\kappa, \delta) - \text{optimal}$

- **class db**
  Let $X$ be a buffer then
  $X \in \text{class db} = X \in \text{DB}_2 \land X \text{ is } (\kappa, \delta) - \text{optimal}$

- **class dc**
  Let $X$ be a buffer then
  $X \in \text{class dc} = X \in \bigcup_{k=3}^{\infty} \text{DB}_1 \land X \text{ is } (\kappa, \delta) - \text{optimal}$

The algorithms for generating the classes da, db and dc are given in the next sections.

### 5.5.1 Class da of $(\kappa, \delta)$ - optimal buffers

The buffers of class da are buffers that can be composed, are buffers consisting of the one place buffer Buf and the two place buffer DBuf. The only construction method that uses only basic components without dividing the stream into substreams is the serial composition (SER). The buffers that can be designed are generated by algorithm CalcClassDA. These buffers are shown in table 5.2 with letter a.

An algorithm for generating the buffers of class da is given by

```plaintext
//------------------------------------------------------------------------------------------------
Const \( \delta_{\text{MAX}} \) = ...
Const \( \kappa_{\text{MAX}} \) = ...
Array IsOpt[\( \delta_{\text{MAX}}, \kappa_{\text{MAX}} \)] of Boolean

Proc CalcClassDA(k : int, d : Int)
  if \( k < \kappa_{\text{MAX}} \land d < \delta_{\text{MAX}} \land d \leq k \rightarrow 
    if IsOpt(k, d) \rightarrow \text{skip}
    [] not IsOpt(k, d) \rightarrow 
      IsOpt(k, d) := True
      {serial composition with Buf}
      Call CalcClassDA(k + 1, d + 1)
      {serial composition with DBuf}
      Call CalcClassDA(k + 2, d + 1)
  fi
fi
End Proc
//------------------------------------------------------------------------------------------------

Algorithm 5.1 CalcClassDA

The procedure call is CalcClassDA(1,1) U CalcClassDA(2,1)
```

### 5.5.2 Class db of $(\kappa, \delta)$ - optimal buffers

The buffers of class db are buffers that can be composed, are buffers consisting of the one place basic components Buf, Split\(_{k,2}\), Merge\(_{k,2}\), and the two-place basic components DBuf, DSplit\(_{k,2}\), D Merge\(_{k,2}\), for
0 ≤ k < 2. According to the wagging theorem we can construct new buffers Y that are (κ, δ) – optimal with i/o-distance δ = δ_x + 2 and capacity κ = 2 * κ_x + 2. Where X is also (κ, δ) – optimal. According to the double wagging theorem we can construct new buffers Y that are (κ, δ) – optimal with i/o-distance δ = δ_x + 2 and capacity κ = 2 * κ_x + 4. According to serial composition theorem we can construct new buffers Y that are (κ, δ) – optimal with i/o-distance δ = δ_x + n and capacity κ = κ_x + n. We also can apply the serial composition theorem using 2 place components which results in new buffers Y that are (κ, δ) – optimal with i/o-distance δ = δ_x + n and capacity κ = κ_x + 2n.

An algorithm for generating the buffers of class db is given by

```
Const δ_MAX = ...
Const κ_MAX = ...

Array IsOpt[δ_MAX, κ_MAX] of Boolean

Proc CalcClassDB(k : int, d : Int)
    var i : int;
    if k < κ_MAX ∧ d < δ_MAX ∧ d ≤ k →
        if IsOpt(k, d) → skip
    [ ] not IsOpt(k, d) →
        IsOpt(k, d) := True
        { serial composition Buf }
        Call CalcClassDB(k + 1, d + 1)
        { serial composition DBuf }
        Call CalcClassDB(k + 2, d + 1)
        { wagging WAG0,2 }
        Call CalcClassDB(2 * k + 2, d + 2)
        { wagging DWAG0,2 }
        Call CalcClassDB(2 * k + 4, d + 2)
    fi
fi
End Proc
```

Algorithm 5.2 CalcClassDB

The procedure call is CalcClassDB(1,1) U CalcClassDB(2,1)

5.5.3 Class dc of (κ, δ) - optimal buffers

The buffers of class c are buffers that can be composed, are buffers consisting of the basic components Buf, Split_k, Merge_k, 0 ≤ k ≤ l.

An algorithm for generating the buffers of class dc is given by

```
Const δ_MAX = ...
Const κ_MAX = ...
```
5.5 Classes of two-place \((\kappa, \delta)\)- optimal buffers

\[ M_{\text{MAX}} = \ldots \]

Array IsOpt[\(\delta_{\text{MAX}}, \kappa_{\text{MAX}}\)] of Boolean

**Proc** CalcClassDC(k : int, d : Int)

```
var n : int ; i : int

if \(k < \kappa_{\text{MAX}} \land d < \delta_{\text{MAX}} \land d \leq k \rightarrow\)
  if IsOpt(k, d) \rightarrow skip
[] not IsOpt(k, d) \rightarrow
  IsOpt(k, d) \(:=\) True
{serial composition Buf}
Call CalcClassC(k + 1, d + 1)
{serial composition DBuf}
Call CalcClassDC(k + 2, d + 1)
```

{wagging \(\text{WAG}_{0,2}\)}
Call CalcClassDC(2 \cdot k + 2, d + 2)
{wagging \(\text{DWAG}_{0,2}\)}
Call CalcClassDC(2 \cdot k + 4, d + 2)

{wagging \(\text{WAG}_{k,1}, \text{WAG}_{k,1} \mid> 2\)}

```
i := 2

for k / i := d \{ \(\kappa_{x} < \delta_{x} \land i \land \delta_{y} = \delta_{x} \land \delta_{y} \leq \kappa_{y} \implies \kappa_{x} < \kappa_{y} \land i\}\}
  if k Mod i = 0 \rightarrow \{(\forall i : \kappa_{x} = \kappa_{y})\}
    Call CalcClassDC((i + 1) \cdot k / i + 2, d + 2)
    Call CalcClassDC((i + 1) \cdot k / i + 4, d + 2)
  if
    i := i + 1
od
```

{\((\text{double})\)-multi-wagging}

```
for n = 2 to \(M_{\text{MAX}}\)
  Call CalcClassDC(n \cdot (k + 2), n + d + 1)
  Call CalcClassDC(n \cdot (k + 4) - 2, n + d + 1)
rof n
fi
fi
```

**End Proc**

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Algorithm 5.3 CalcClassDC

The procedure call is CalcClassDC(1,1) U CalcClassDC(2,1)

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
All the composed two-place buffers of classes da, db and dc for $\kappa \leq 36$ and $\delta \leq 36$ are given in table 5.2. The classes da, db and dc are represented respectively with a, b and c.

| $\kappa$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 |
|----------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| $\delta$| a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a |
|         | b | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a |
|         | b | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a |

*Table 5.2 ($\kappa$, $\delta$) - optimal buffers of classes da, db and dc*
5.6 Cubic buffers

In chapter two we defined split and merge components with respectively a fan-out and a fan-in of two. Increasing the fan-out and a fan-in for these components may lead to constructions of \((\kappa, \delta)\)-optimal buffers with values for \( \kappa \) and \( \delta \) that weren’t found until this section. In this section we define split and merge components with respectively a fan-out and a fan-in of three. As an example we introduce the family of cubic buffers.

5.6.1 Split3

Component \( \text{Split3}_{\kappa} \) (see figure 5.6.1) divides a stream into three substreams. Its functionality is specified by \( C = A(k-1:k)(l-1 : l) \) and \( D = A(k-1:k)(l-1 \Rightarrow l) \) and \( E = A(k-1 \Rightarrow k) \), \( 0 < k \), \( 0 < l \).

![Figure 5.6.1 Split3_{\kappa}(a,c,d,e)]](image)

5.6.2 Merge3

We combine three substreams by means of a \( \text{Merge3}_{\kappa} \) component (see figure 5.6.2). The functionality of a \( \text{Merge3}_{\kappa} \) component is specified by \( F = B(0:k)(0 : l) \) and \( G = B(0 \Rightarrow k)(0 \Rightarrow l) \) and \( H = (0 \Rightarrow l) \), \( 0 < k \), \( 0 < l \).

![Figure 5.6.2 Merge3_{\kappa}(f,g,h,b)](image)
Let $g$ be an integer such that $0 < g < \gamma$. Then for $2 \leq \gamma$ the general sequence functions for Split3 and Merge3 are given in table 5.6.1.

<table>
<thead>
<tr>
<th>Sequence function</th>
<th>$a_{i1}$</th>
<th>$a_{i2}$</th>
<th>$d_{iQ1}$, $0 \leq q &lt; 1 - 1$</th>
<th>$e_{i}(p,q;k), 0 \leq p &lt; k, 0 \leq q &lt; 1 - 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_{\text{split}3,k}$</td>
<td>$\gamma_i$</td>
<td>$1 \cdot k \cdot \gamma_i + (k-1) \cdot \gamma_i + (l-1) \cdot k \cdot \gamma + g$</td>
<td>$1 \cdot k \cdot \gamma_i + (k-1) \cdot \gamma_i + q \cdot k \cdot \gamma + g$</td>
<td>$0 \leq q &lt; 1 - 1$</td>
</tr>
<tr>
<td>$\sigma_{\text{merge}3,k}$</td>
<td>$\delta_i$</td>
<td>$g_{j1}, 0 &lt; p &lt; k$</td>
<td>$b_{i}(p,q;k), 0 &lt; p &lt; k, 0 &lt; q &lt; 1 - 1$</td>
<td>$b_{i}$</td>
</tr>
<tr>
<td>$\sigma_{\text{merge}3,k}$</td>
<td>$1 \cdot k \cdot \gamma_i$</td>
<td>$1 \cdot k \cdot \gamma_i + p \cdot \gamma_i, 0 &lt; p &lt; k$</td>
<td>$1 \cdot k \cdot \gamma_i + q \cdot \gamma_i + p \cdot \gamma_i + g$</td>
<td>$0 &lt; p &lt; k, 0 &lt; q &lt; 1 - 1$</td>
</tr>
</tbody>
</table>

Table 5.6.1 General sequence functions for the components Split3 and Merge3

5.6.3 Cubic buffer design

For designing cubic buffers we present two new components: the rectangular splitter and the rectangular merger.

The rectangular splitter, denoted as $R_{n,m}$ for $2 \leq n$ and $2 \leq m$, divides a stream into $n \cdot m$ equal substreams. Next we show how to construct the rectangular splitter using several multi-way splitters.

Definition 5.6.1 Rectangular splitter

The family $\{R_{n,m}\}_{n=2}^{\infty}, \{m=2}^{\infty}$ of rectangular splitters is defined by

$$RS_{n=1, m=1} = \text{proc (in a; out c_{0,0}, c_{0,1}, \ldots, c_{0,m-1}, c_{0,m})}$$

$$\text{chan d_{0,0}, \ldots, d_{0,m}, e_{1}, \ldots, e_{n-1}}$$

$$\text{Split3}_{n=1, m=1}(a, c_{0,0}, c_{0,1}, c_{0,2}) \parallel MS_{m}(d_{0,0}, d_{0,1}, \ldots, d_{0,m-1}, d_{0,m})$$

$$\text{Split3}_{n=1, m=1}(c_{0,1}, c_{0,2}, d_{1,0}, d_{0,1}, c_{0,2}) \parallel MS_{m}(d_{0,1}, d_{0,2}, \ldots, d_{0,2, m-1}, d_{0,2,m})$$

$$\text{Split3}_{n=1, m=1}(c_{0,2}, c_{0,2, m-1}, d_{2,0}, d_{0,2}, e_{1}) \parallel MS_{m}(d_{2,0}, c_{2,0}, \ldots, c_{2,0})$$

$$\text{Split3}_{2,m}(c_{1,1}, c_{1, m}, d_{2,0}, c_{0,0}) \parallel MS_{m}(d_{1,0}, c_{1,0}, \ldots, c_{1,0})$$

$$\text{Split3}_{m,m}(c_{0,0}, c_{0, m}, d_{0,0}) \parallel MS_{m}(d_{0,0}, c_{0,0}, \ldots, c_{0,0})$$

$$\ldots$$

$$, 1 \leq n, 1 \leq m$$
Figure 5.6.1 shows a diagram for this component.

Figure 5.6.1 rectangular splitter $RS_{n,m}$

The output streams of a rectangular splitter can be recombined into the original stream by means of a rectangular merger. Next we show how to construct the rectangular merger using several multi-way mergers.

Definition 5.6.2 Rectangular merger

The family $\{RM_{n,m}\}_{n=2}^{\infty}, m=2 \rightarrow \infty$, of rectangular mergers is defined by

$$RM_{n+1,m+1} = \text{proc (in } f_{0,0}, f_{1,0}, \ldots, f_{m,1}, f_{m,m}; \text{ out } b)$$

$$\begin{align*}
&\text{[ chan } g_0, g_1, \ldots, g_m, h_{0,1}, \ldots, h_{m,1} \\
&\text{ MM}_m(f_{m,0}, \ldots, f_{m,1}, g_0) || \text{ Merge}_{0,m+1}(f_{0,0}, g_0, h_{0,1}) \\
&\text{ MM}_m(f_{m-1,m}, \ldots, f_{m-1,1}, g_0) || \text{ Merge}_{1,m+1}(h_{0,1}, f_{1,0}, g_0, h_{0,2}) \\
&\text{ MM}_m(f_{m-2,m}, \ldots, f_{m-2,1}, g_0) || \text{ Merge}_{2,m+1}(h_{0,2}, f_{2,0}, g_0, h_{0,3}) \\
&\text{ MM}_m(f_{m-3,m}, \ldots, f_{m-3,1}, g_0) || \text{ Merge}_{3,m+1}(h_{0,3}, f_{3,0}, g_0, h_{0,4}) \\
&\ldots \\
&\text{ MM}_m(f_{m-1,1}, g_0) || \text{ Merge}_{m+1,m+1}(h_{0,m}, f_{m,0}, g_0, h_{0,1}) \\
&\text{ MM}_m(f_{m-1,1}, g_0) || \text{ Merge}_{m+1,m+1}(h_{1,0}, f_{1,0}, g_0, h_{1,1}) \\
&\text{ MM}_m(f_{m-1,1}, g_0) || \text{ Merge}_{m+1,m+1}(h_{2,0}, f_{2,0}, g_0, h_{2,1}) \\
\text{ ]} \\
&\quad, 1 \leq n, 1 \leq m
\end{align*}$$
Figure 5.6.2 shows a diagram for this component.

Figure 5.6.2 rectangular merger $RM_n$

Definition 5.6.3 Cubic composition

For any components $X_{0,0}, X_{0,1}, \ldots, X_{n-1,m-2}, X_{n-1,m-1}$, we define the cubic composite $Cubic_{n,m}(X_{0,0}, X_{0,1}, \ldots, X_{n-1,m-2}, X_{n-1,m-1})$ by

$$
Cubic_{n,m}(X_{0,0}, X_{0,1}, \ldots, X_{n-1,m-2}, X_{n-1,m-1}) = \begin{cases} \text{proc (in a; out b)} \\ \\
\quad [[ \text{chan } c_{0,0}, c_{0,1}, \ldots, c_{n-1,m-2}, c_{n-1,m-1}, f_{0,0}, f_{0,1}, \ldots, f_{n-2,m-2}, f_{n-2,m-1}, f_{n-1,m-1}] \\
\quad | \text{RS}_{n,m}(a, c_{0,0}, c_{0,1}, \ldots, c_{n-1,m-2}, c_{n-1,m-1}) ] \\
\quad | (\forall i,j: 0 \leq i < n \land 0 \leq j < m : X_{ij}(c_{ij}, f_{ij})) \\
\quad | RM_{n,m}(f_{0,0}, f_{0,1}, \ldots, f_{n-1,m-2}, f_{n-1,m-1}, b) \\
\quad ]], 2 \leq n, 2 \leq m
\end{cases}
$$

Now that we have defined the cubic composition, we present two families of buffers that are constructed using that composition method.
**Definition 5.6.4 Block buffer**

The family of block buffers \( \{ \text{BlockBuf}_{n,m,p} \} \) \( n \geq 2 \to \infty, m \geq 2 \to \infty, p \geq 2 \to \infty \) is defined by

\[
\text{BlockBuf}_{n,m,p} = \text{proc} \ (\text{in} \ a; \text{out} \ b)
\begin{align*}
&[\text{ch} \ c_0, c_1, \ldots, c_{n-1,m-2}, c_{n-1,m-1}, f_0, f_0, \ldots, f_{n-2,m-1}, f_{n-1,m-1} \\
&\quad | \text{RS}_{n,m}(a, c_0, c_1, \ldots, c_{n-1,m-2}, c_{n-1,m-1}) \\
&\quad | (\forall i; 0 \leq i < n, \forall j; 0 \leq j < m : \text{LBuf}_{p,2}(c_0, f_0) ) \\
&\quad | \text{RM}_{n,m}(f_0, f_0, \ldots, f_{n-1,m-2}, f_{n-1,m-1}, b) \\
&]\end{align*}
\]

Note that the block buffer \( \text{BlockBuf}_{n,m,p} \) is the cubic composition of \( n \cdot m \) linear buffers \( \text{LBuf}_{p,2} \).

\( \text{BlockBuf}_{n,m,p} = \text{Cubic}_{n,m}(\text{LBuf}_{p,2}, \ldots, \text{LBuf}_{p,2}), 2 \leq n, 2 \leq m, 2 \leq p \)

**Definition 5.6.5 Cubic buffer**

The family of cubic buffers \( \{ \text{CBuf}_n \} \) \( n \geq 2 \to \infty \) is defined by

\[
\text{CBuf}_n = \text{proc} \ (\text{in} \ a; \text{out} \ b)
\begin{align*}
&[\text{ch} \ c_0, c_0, \ldots, c_{n-1,n-2}, c_{n-1,n-1}, f_0, f_0, \ldots, f_{n-2,n-2}, f_{n-1,n-1} \\
&\quad | \text{RS}_{n,n}(a, c_0, c_1, \ldots, c_{n-1,n-2}, c_{n-1,n-1}) \\
&\quad | \text{LBuf}_{p,2}(c_0, f_0) \parallel \text{LBuf}_{p,2}(c_1, f_1) \\
&\quad | \ldots \ldots \ldots \\
&\quad | \text{LBuf}_{p,2}(c_{n-1,n-2}, f_{n-1,n-2}) \parallel \text{LBuf}_{p,2}(c_{n-1,n-1}, f_{n-1,n-1}) \\
&\quad | \text{RM}_{n,n}(f_0, f_0, \ldots, f_{n-2,n-2}, f_{n-1,n-1}, b) \\
&\quad |, 2 \leq n \\
&]\end{align*}
\]

Note that \( \text{CBuf}_n = \text{BlockBuf}_{n,n,n}, 2 \leq n \)

The core of this thesis is to design optimal buffers. Therefore we are interested in the optimality of cubic buffer designs. For instance consider the cubic buffer \( \text{CBuf}_n \). This buffer has i/o-distance \( \delta = 7 \) and capacity \( \kappa = 27 \). A buffer with these static properties was not been found using construction methods and components containing only one-place components. A diagram for this buffer with early sequence is shown in figure 5.6.4. Note that we take cycle time \( \gamma = 5 \). Sequence functions can also be shown for other cycle times.
Figure 5.6.4 CBuf3 with early sequence

The latency of the early sequence function for the cubic buffer CBuf3 equals $\lambda=7$. Note that the latency equals the I/O-distance and attains the lower bound for a minimal buffer. Hence the buffer is minimal optimal. Now we take a look at the buffer design with a late sequence function.

Figure 5.6.5 CBuf3 with late sequence
The latency of the late sequence function for the cubic buffer CBuF\(_2\) equals \(\lambda = 128\). Note that the latency attains the upper bound for a maximal buffer, \(\lambda = \gamma - \delta = 128\). Hence the buffer is maximal optimal. We can conclude that the cubic buffer CBuF\(_2\) is \((k, \delta)\) optimal. The cubic buffer is a special case of the block buffers. Next we prove that application of the cubic composition using a set of \((k, \delta)\) optimal buffers, constructs new buffers that are \((k, \delta)\) optimal. Therefore we first present the cubic theorem.

**Theorem 5.6.6 Cubic theorem**

Let Cubic\(_{n,m}(X_{0,0}, X_{0,1}, \ldots, X_{n,1, m}, \ldots, X_{n,1, m, 1})\) be the cubic composite of buffers \(X_{0,0}, X_{0,1}, \ldots, X_{n,1, m}, X_{n,1, m, 1}\). For integer \(\gamma\) let \(\sigma_{X_{0,0}}, \ldots, \sigma_{X_{n,1, m}, 1}\) be sequence functions for \(X_{0,0}, X_{0,1}, \ldots, X_{n,1, m}, X_{n,1, m, 1}\) with constant cycle time \(n \cdot \gamma\). Furthermore let \(\sigma_{X_{0,0}}, \ldots, \sigma_{X_{n,1, m}, 1}\) have constant and equal latency \(\lambda\). Then for any integers \(g_0, \ldots, g_n\) and \(v_1, \ldots, v_{n,1}\) and \(r_1, \ldots, r_{m-1}\) and \(d_1, \ldots, d_{m-1}\) such that \((\forall i: 0 \leq i < n: 0 < g_i < \gamma)\) and \((\forall i: 0 < i < n: 0 < v_i < n)\) and \((\forall i: 0 < i < m: 0 < r_i < n)\) and \((\forall i: 0 < i < m: 0 < d_i < n)\) there exists a sequence function \(\sigma_{CBuF}\) for the cubic composite Cubic\(_{n,m}(X_{0,0}, X_{0,1}, \ldots, X_{n,1, m}, X_{n,1, m, 1})\) with constant cycle time \(\gamma\) and constant latency \(\lambda_{CBuF} = \lambda + g_1 + \ldots + g_n + v_1 + \ldots + v_{n,1} + r_1 + \ldots + r_{m-1} + d_1 + \ldots + d_{m-1}\).

**Proof**

Then for any integers \(g_0, \ldots, g_n\) and \(v_1, \ldots, v_{n,1}\) and \(r_1, \ldots, r_{m-1}\) and \(d_1, \ldots, d_{m-1}\) such that \((\forall i: 0 \leq i < n: 0 < g_i < \gamma)\) and \((\forall i: 0 < i < n: 0 < v_i < n)\) and \((\forall i: 0 < i < m: 0 < r_i < n)\) and \((\forall i: 0 < i < m: 0 < d_i < n)\) define sequence functions \(\sigma_{MSC,n,m,p}\) and \(\sigma_{MMC,n,m,p}\) by

\[
\begin{align*}
\sigma_{MSC,n,m,p}(d_{q, i}(m + i)) & = n \cdot m \cdot \gamma_i + p \gamma + m \gamma + G(p) + V(p), \ 0 \leq i < m-1 \\
\sigma_{MSC,n,m,p}(c_{p,q}(i)) & = n \cdot m \cdot \gamma_i + p \gamma + q \gamma + G(p) + V(p) + R(q), \ 0 \leq i < m-1
\end{align*}
\]

for \(0 \leq p < n\) and \(0 \leq q < m-1\)

\[
\begin{align*}
\sigma_{MMC,n,m,p}(g_{n, i}(m + i)) & = n \cdot m \cdot \gamma_i + p \gamma + G(p) + V(p) + R(0) + D(0), \ 0 \leq i < m-1 \\
\sigma_{MMC,n,m,p}(f_{i, p,q})(i) & = n \cdot m \cdot \gamma_i + p \gamma + q \gamma + G(p) + V(p) + R(q) + D(q)
\end{align*}
\]

for \(0 \leq p < n\) and \(0 < q < m-1\)

where \(G(p) = (\Sigma i: p < i < n: g_i), V(p) = (\Sigma i: p < i < n: v_i), R(q) = (\Sigma i: p < i < m: r_i), D(q) = (\Sigma i: p < i < m: d_i)\)
Then
\[ \Gamma(\sigma_{\text{Cubic},n,m}) = \Gamma(\sigma_{\text{Split3},n,m}) = \gamma \]

- Sequence function \( \sigma_{\text{Cubic},n,m} \) has constant cycle time \( \gamma \)

\[
\lambda(\sigma_{\text{Cubic},n,m}) = \sigma_{\text{Cubic},n} (b\#i) - \sigma_{\text{Cubic},n} (a\#i) \\
= (\sigma_{\text{Merge3},n,m} (b\#i) + \lambda + G(0) + V(0) + R(0) + D(0)) - \sigma_{\text{Split3},n,m} (a\#i) \\
= (\gamma i + \lambda + G(0) + V(0) + R(0) + D(0) + g_0) - \gamma i \\
= \lambda + g_0 + g_1 + g_2 + v_1 + v_{n-1} + r_1 + r_{m-1} + d_1 + d_{m-1}
\]

Sequence function \( \sigma_{\text{Cubic},n,m} \) has constant latency \( \lambda + g_0 + g_1 + g_2 + v_1 + v_{n-1} + r_1 + r_{m-1} + d_1 + d_{m-1} \)

Early sequence function \( \sigma'_{\text{Cubic},n} \) has latency \( \lambda_{\text{Cubic},n,m} = \lambda + (n+1)+(m-1) \)

= \lambda + n+m

Late sequence function \( \sigma''_{\text{Cubic},n} \) has latency \( \lambda_{\text{Cubic},n} = \lambda + (n+1)(\gamma-1) + (n-1)\gamma + (m-1)(\eta-1) + (m-1)\eta \)

= \lambda + 2n \cdot m \cdot \gamma - n - m

- Sequence function \( \sigma_{\text{Cubic},n} \) has occupancy \( \omega_{\text{Cubic},n} = \lambda_{\text{Cubic},n} / \gamma_{\text{Cubic},n} \)

= (\lambda + g_0 + g_1 + g_2 + v_1 + v_{n-1} + r_1 + r_{m-1} + d_1 + d_{m-1}) / \gamma

(end of proof)
The diagram shown in figure 5.6.6 shows general sequence functions for the cubic composition. Because of layout reasons we restrict the figure to a part of the buffer design.

Figure 5.6.6 Cubic composition with general sequence
The next step after defining the cubic theorem is to show that application of the cubic composition under specific restrictions constructs new \((k, \delta)\) - optimal buffers.

**Theorem 5.6.4 Cubic composition optimality**

Let \(X_{0,0}, X_{0,1}, \ldots, X_{n-1,m-2}, X_{n-1,m-1}\) be \((k, \delta)\) - optimal buffers with equal capacities \(k_x\) and i/o-distances \(\delta_x\). Then the cubic composite \(Cubic_{n,m}(X_{0,0}, X_{0,1}, \ldots, X_{n-1,m-2}, X_{n-1,m-1})\) is an \((k, \delta)\) - optimal buffer with capacity \(K_{Cubic} = n \cdot m (k_x + 2)\) and i/o-distance \(\delta_{Cubic} = \delta_x + n + m\).

**Proof**

Let \(X_{0,0}, X_{0,1}, \ldots, X_{n-1,m-2}, X_{n-1,m-1}\) be \((k, \delta)\) - optimal buffers with equal capacities \(k_x\) and i/o-distances \(\delta_x\). Buffers \(X_{0,0}, X_{0,1}, \ldots, X_{n-1,m-2}, X_{n-1,m-1}\) are \((k, \delta)\) - optimal. So there exist sequence functions 
\[\sigma_{X_{0,0}}, \sigma_{X_{0,1}}, \ldots, \sigma_{X_{n-1,m-1}}\]
with equal and constant latency \(\lambda\), and equal and constant cycle time \(n \cdot m \cdot \gamma\). Then by the cubic theorem for some integers \(g_0, \ldots, g_n\) and \(v_1, \ldots, v_{n+1}\) and \(r_1, \ldots, r_m\) and \(d_0, \ldots, d_m\) such that
\[(\forall i: 0 \leq i \leq n: g_{i+1} \leq g_i) \quad \text{and} \quad (\forall i: 0 \leq i < n: 0 < v_{i+1} \leq v_i) \quad \text{and} \quad (\forall i: 0 \leq i < n: 0 < r_i < r_{i+1}) \quad \text{and} \quad (\forall i: 0 < i < m: 0 < d_i \leq d_{i+1})\]
there exists a sequence function \(\sigma_{CUBF}\) for the cubic composite \(Cubic_{n,m}(X_{0,0}, X_{0,1}, \ldots, X_{n-1,m-2}, X_{n-1,m-1})\) with constant cycle time \(\gamma\) and constant latency \(\lambda_{CUBF} = \lambda + g_0 + g_1 + \ldots + g_n + v_1 + \ldots + v_{n+1} + r_1 + \ldots + r_m + d_0 + \ldots + d_m\)

As can be seen in figure 5.6.3 the cubic composite \(Cubic_{n,m}(X_{0,0}, X_{0,1}, \ldots, X_{n-1,m-2}, X_{n-1,m-1})\) is a buffer with capacity \(K_{Cubic,n,m} = n \cdot m (k_x + 2)\) and i/o-distance \(\delta_{Cubic,n,m} = \delta_x + n + m\). Hence it remains to be shown that when we take the early sequence function \(\sigma_{e}\) the latency \(\lambda_{Cubic,n,m}\) equals the i/o-distance \(\delta_{Cubic,n,m}\) and for the late sequence function \(\sigma_{l}\) the latency \(\lambda_{Cubic,n,m}\) equals \(\gamma_{Cubic,n,m} \cdot K_{Cubic,n,m} - \delta_{Cubic,n,m}\). We first start with the early sequence function.

\[
\begin{align*}
\lambda(\sigma_{Cubic,n,m}) &= \{\text{cubic theorem : early sequence}\} \\
&= \lambda + n + m \\
&= \{\text{\(\sigma_{e}\) early so \(\lambda = \delta_x\)}\} \\
&= \{\text{\(\delta_{Cubic,n,m} = \delta_x + n + m\)}\} \\
&= \delta_{Cubic,n,m}
\end{align*}
\]

The latency \(\lambda_{Cubic,n,m}\) of the late sequence function equals

\[
\begin{align*}
\lambda(\sigma_{Cubic,n,m}) &= \{\text{cubic theorem : late sequence}\} \\
&= \lambda + 2n \cdot m \cdot \gamma - n - m \\
&= \{\text{\(\sigma_{l}\) late so \(\lambda_x = n \cdot m \cdot \gamma \cdot k_x - \delta_x\)}\} \\
&= \{\text{\(\gamma (2n \cdot m + n \cdot m \cdot k_x) - (\delta_x - n - m)\)}\} \\
&= \{\text{\(\delta_{Cubic,n,m} = \delta_x + n + m\)}\} \\
&= \{\text{\(\gamma (n \cdot m \cdot (k_x + 2)) - \delta_{Cubic,n,m}\)}\} \\
&= \{\text{\(K_{Cubic,n,m} = n \cdot m \cdot (k_x + 2)\)}\} \\
&= \gamma_{Cubic,n,m} \cdot K_{Cubic,n,m} - \delta_{Cubic,n,m}
\end{align*}
\]

Hence \(Cubic_{n,m}(X_{0,0}, X_{0,1}, \ldots, X_{n-1,m-2}, X_{n-1,m-1})\) is an \((k, \delta)\) - optimal buffer.

The Cubic optimality theorem shows us that we can generate \((k, \delta)\) - optimal Cubic buffers using a \((k, \delta)\) - optimal buffer \(X\) with the following relations: \(K_{Cubic,n,m} = n \cdot m \cdot (k_x + 2)\) and i/o-distance \(\delta_{Cubic,n,m} = \delta_x + n + m\).

For instance we can construct buffers \(Cubic_{n,m}(B_1, \ldots, B_n)\) using \(n^2\) Buffs with capacity \(k = 3n^2\) and i/o-distance \(\delta = 2n + 1\).
Adding cubic composition to the three standard construction methods, gives for the one-place buffers new values for $\kappa$ and $\delta$. Adding above relations to the algorithm for class $c$ presents a new algorithm that creates buffer designs with cubic composition.

Next we derive some characteristics of the block buffers and the cubic buffers.

### 5.6.4 Latency of cubic buffers

In the previous section we have shown the general relationship between the latency (occupancy) of a cubic composite and the latency (occupancy) of its subcomponents. In this section we show how it turns out for two specific cases, the family of block buffers and the family of cubic buffers. We start with the block buffers.

Consider the block buffer $\text{BlockBuf}_{n,m,p}$. Applying the cubic theorem 5.6.6 gives a sequence function $\overline{\sigma}_{\text{BlockBuf}_{n,m,p}}$ for the rectangular buffer $\text{BlockBuf}_{n,m,p}$ with latency constant latency $\lambda_{\text{BlockBuf}_{n,m,p}} = \lambda_{l_{\text{Buf}(p-2)}}, g_0 + g_1 + \ldots + g_n + v_1 + \ldots + v_{n-1} + r_1 + \ldots + r_{m-1} + d_1 + \ldots + d_{m-1}$ for any integers $g_0, \ldots, g_n$ and $v_1, \ldots, v_{n-1}$ and $r_1, \ldots, r_{m-1}$ and $d_1, \ldots, d_{m-1}$ such that $(\forall i: 0 \leq i < n: 0 < g_i < \gamma)$ and $(\forall i: 0 < i < n: 0 < v_i < \gamma)$ and $(\forall i: 0 < i < m: 0 < r_i < n\gamma - 1)$ and $(\forall i: 0 < i < m: 0 < d_i < n\gamma - 1)$.

Sequence function $\overline{\sigma}_{\text{BlockBuf}_{n,m,p}}$ has constant latency $\lambda_{l_{\text{Buf}(p-2)}} + g_0 + g_1 + \ldots + g_n + v_1 + \ldots + v_{n-1} + r_1 + \ldots + r_{m-1} + d_1 + \ldots + d_{m-1}$ where $\lambda_{l_{\text{Buf}(p-2)}} = \omega_0 + w_1 + \ldots + w_{p-2}$ such that $(\forall j: 0 \leq j < p-2: 0 < w_j < n\gamma - \gamma)$.

So sequence function $\overline{\sigma}_{\text{BlockBuf}_{n,m,p}}$ has constant latency $\lambda_{\text{BlockBuf}_{n,m,p}} = \omega_0 + w_1 + \ldots + w_{p-2} + g_0 + g_1 + \ldots + g_n + v_1 + \ldots + v_{n-1} + r_1 + \ldots + r_{m-1} + d_1 + \ldots + d_{m-1}$.

We are interested in the late and early sequence functions.

For the early sequence function the following conditions hold

- $(\forall j: 0 \leq j < n: g_j = 1)$
- $(\forall j: 0 < j < n: v_j = 0)$
- $(\forall j: 0 < j < m: r_j = 1)$
- $(\forall j: 0 < j < m: d_j = 0)$
- $\lambda(\overline{\sigma}_{l_{\text{Buf}(p-2)}}, n, m, p) = p - 2$

$$
\lambda(\overline{\sigma}_{\text{BlockBuf}_{n,m,p}}) = \lambda(\overline{\sigma}_{l_{\text{Buf}(p-2)}}) = g_0 + g_1 + \ldots + g_n + v_1 + \ldots + v_{n-1} + r_1 + \ldots + r_{m-1} + d_1 + \ldots + d_{m-1} = p - 2 + n + 1 + m - 1 = p + n + m - 2
$$

Early sequence function $\overline{\sigma}_{\text{BlockBuf}_{n,m,p}}$ has constant latency $\lambda = p + n + m - 2$

Note that the latency equals the i/o-distance $\delta_{\text{BlockBuf}_{n,m,p}}$.

For the late sequence function the following conditions hold

- $(\forall j: 0 \leq j < n: g_j = \gamma - 1)$
- $(\forall j: 0 < j < n: v_j = \gamma)$
- $(\forall j: 0 < j < m: r_j = n\gamma - 1)$
- $(\forall j: 0 < j < m: d_j = n\gamma)$
- $\lambda(\overline{\sigma}_{l_{\text{Buf}(p-2)}}, n, m, p) = (n+m)\gamma - 1$

$$
\lambda(\overline{\sigma}_{\text{BlockBuf}_{n,m,p}}) = \lambda(\overline{\sigma}_{l_{\text{Buf}(p-2)}}) = g_0 + g_1 + \ldots + g_n + v_1 + \ldots + v_{n-1} + r_1 + \ldots + r_{m-1} + d_1 + \ldots + d_{m-1} = (n+m)\gamma - 1 + (n+1)(\gamma - 1) + (n-1)(\gamma + (n-1)(n\gamma + (m-1)n\gamma = p + n + m + 2 + 2n + m + n = n + m + p - 2
$$
Late sequence function $\sigma_{\text{BlockBuf},n,m,p}$ has constant latency $\lambda_{\text{BlockBuf},n,m,p} = \gamma(p \cdot n \cdot m) - (n + m + p - 2)$

Note that the latency equals $\gamma \cdot (p \cdot n \cdot m) - (n + m + p - 2)$

Sequence function $\sigma_{\text{BlockBuf},m,n}$ has occupancy $\omega_{\text{BlockBuf},m,n} = (w_0 + \ldots + w_{m-2} + g_0 + \ldots + g_m + v_1 + \ldots + v_{n-1}) / \gamma$.

Then $\Omega_{\gamma=\gamma} = (n + m + p - 2) / \gamma$, and $\Omega_{\gamma=\gamma} = (p \cdot n \cdot m) - (n + m + p - 2) / \gamma$.

The characteristics of the cubic buffers $\text{CBuf}_n$ can be derived by using the characteristics of the block buffer. Early sequence function $\sigma_{\text{CBuf},n}$ has constant latency $\lambda = 3n - 2$. Late sequence function $\sigma_{\text{BlockBuf},m,n}$ has constant latency $\lambda_{\text{BlockBuf},m,n} = \gamma n^3 - 3n + 2$. Then $\Omega_{\gamma=\gamma} = (3n - 2) / \gamma$, and $\Omega_{\gamma=\gamma} = n^3 - (3n - 2) / \gamma$. 
6 Miscellaneous systems

In the previous chapters we constructed buffers. In our definition a buffer has exactly one input port and one output port. Furthermore the output stream must be a copy of the input stream. This buffer type is known as FIFO buffers. In this chapter we introduce two families of systems, not being buffers to our definition, and we show that the queueing formula still holds for those systems. The two families of systems we present are the family of permutations and the family of block reversers. We start with the family of permutations.

6.1 Permutations

The block permutator is a system design that rearranges its input stream. Its output stream is not a copy of the input stream. Hence this design is not a buffer.

Definition 6.1.1 Block Permutator

The family of block permutator \( \{BPer_n\}_{n=1}^{\infty} \) is defined by

\[
\begin{align*}
BPer_1 & = \text{LBuf}_1 \\
BPer_{n+1} & = \text{proc} \ (\text{in a; out b}) \\
& \quad \| \text{chan c,d,f} \\
& \quad | \text{Split}_{a,n-1}(a,c,d) \| \text{Buf}(d,e) \| BPer_n(c,f) \| \text{Merge}_{n+1}(c,f,b) \\
& \quad \|, 1 \leq n
\end{align*}
\]

Figure 6.1.1 shows a diagram for the BPer

![Diagram of BPer](image_url)

Figure 6.1.1 Block Permutator BPer

\[ a \rightarrow \text{Split}_{a,b} \rightarrow \text{Buf} \rightarrow \text{Merge}_{n+1} \rightarrow b \]

\[ a \rightarrow \text{BPer}_n \rightarrow c \rightarrow d \rightarrow \text{Buf} \rightarrow \text{Merge}_{n+1} \rightarrow f \]

\[ a \rightarrow \text{BPer}_n \rightarrow c \rightarrow d \rightarrow \text{Buf} \rightarrow \text{Merge}_{n+1} \rightarrow f \]

\[ a \rightarrow \text{BPer}_n \rightarrow c \rightarrow d \rightarrow \text{Buf} \rightarrow \text{Merge}_{n+1} \rightarrow f \]
As an example consider the block reverser \( BPer_2 \)

![Diagram of Block Permutator BPer_2](image)

*Figure 6.1.2 Block Permutator BPer_2*

The \( BPer_2 \) has a sequence function with cycle time \( \gamma = 6 \). The latency for this sequence function equals \( \lambda = 11 \). The sum of the occupancies of the subcomponents equals \( \omega = 22/12 \). Hence the queuing formula is still valid: \( \lambda = \gamma \cdot \omega = 6 \cdot \frac{22}{12} = 11 \).

### 6.2 Block reverser

The block reverser is a system design that rearranges its input stream. It output stream is not a copy of the input stream. Hence this design is not a buffer.

**Definition 6.2.1 Block Reversers**

The family of block reversers \( \{BRev_n\}_{n=1}^{\infty} \) is defined by

\[
\begin{align*}
BRev_1 &= \text{LBuf}_1 \\
BRev_{n+1} &= \text{proc} (\text{in } a; \text{out } b) \\
&\quad \mid [\text{chan } c,d,f] \\
&\quad \mid Split_{n+1}(a,c,d) \parallel BRev_n(c,f) \parallel Merge_{n+1}(d,f,b) \\
&\quad ] , \text{Sn}
\end{align*}
\]

*Figure 6.2.1 shows a diagram for the BRev_n*

![Diagram of Block Reverser BRev_n](image)

*Figure 6.2.1 Block Reverser BRev_n*
A block reverser reverses its input stream in blocks of size $n$. As an example consider the block reverser $B_{Rev_4}$

![Diagram of the block reverser $B_{Rev_4}$]

The $B_{Rev_4}$ has a sequence function with cycle time $\gamma = 2$. The latency for this sequence function equals $\lambda = 8$. We see that the occupancy for each basic component equals $\frac{1}{2}$. The sum of the occupancies of the subcomponents equals $\omega = 8 \cdot \frac{1}{2} = 4$. Hence the queuing formula is still valid: $\lambda = \gamma \cdot \omega = 2 \cdot 4 = 8$. 
7 Summary

7.1 Summary

In this thesis we have shown how buffer systems, especially \((k,\delta)\) - optimal buffers that are characterised by our optimality criterium, can be constructed from basic components. Furthermore we have shown how the communication behavior of these systems can be described using sequence functions. Sequence functions were being introduced as a suitable formalism for showing the total correctness (absence of deadlock), and as a vehicle for performance analysis of system designs.

In addition to the above aim we have introduced buffers and their functional specification. We have presented a set of basic components, Buf, Split and Merge, that we have used as a base for constructing more complicated buffer designs. We have defined three construction methods to design those buffers: Serial composition, Wagging and Multi-Wagging. We have shown a calculus, taken from [Mak], consisting of only two operators that suffices to establish the functional correctness of buffer designs. This calculus precisely defines how the internal streams and substreams of a buffer are divided and recombined.

Furthermore we have introduced four buffer families: The linear buffers, the binary tree buffers, the rectangular buffers and the square buffers.

Next we have defined two static properties of a buffer design: the capacity and the i/o-distance. We have used these two properties as a base for performance analysis.

To describe the communication behavior of a constructed system, we have introduced the formalism of a sequence function. We have defined common performance parameters such as cycle time, latency and occupancy in terms of these sequence functions and have shown the relationship: the queuing formula. We have derived the relations for these performance parameters between a constructed buffer using one of the construction methods and its subcomponents. The relations have been shown in the serial composition theorem, the wagging theorem and the multi-wagging theorem. The multi-wagging theorem has been added as a new theorem. The wagging theorem was already given in [Mak]. In this thesis, however, we have defined a more general wagging theorem.

We have established lower and upper bounds for the latency and occupancy in terms of the capacity and the i/o-distance. This in turn has raised our definition of a \((k,\delta)\) - optimal buffer. Furthermore we have derived some restrictions for the construction methods such that, when obeyed, the application of the construction methods indeed yields a \((k,\delta)\) - optimal buffer. These restrictions have been shown in the serial composition optimality theorem, the wagging optimality theorem and the multi-wagging optimality theorem. Furthermore we have shown that the four buffer families, the linear buffers, the binary tree buffers, the rectangular buffers and the square buffers, are \((k,\delta)\) - optimal. We have defined a more precise definition of \((k,\delta)\) - optimal buffers where we have replaced the i/o-distance with the average i/o-distance.

In [Mak] the definition of \((k,\delta)\) - optimal buffer was given with a few examples of optimal buffers. Here we have searched systematically for \((k,\delta)\) - optimal buffers. We have introduced three classes of \((k,\delta)\)-optimal buffers, class a, class b and class c, and have presented for each of these classes an algorithm that generates the \((k,\delta)\) - pairs for which a \((k,\delta)\) - optimal buffer in that class exists. Furthermore we have presented an algorithm that provides for a fixed capacity the smallest i/o-distance such that a \((k,\delta)\) - optimal buffer exists. Finally, we have shown buffer designs that are partial \((k,\delta)\) - optimal, i.e. \((k,\delta)\) - optimal only for certain values of the cycle time.
We have extended our search for \((k, \delta)\) – optimal buffers. Our search was not confined to buffers with buffers composed of basic components with one variable. We have introduced two-place basic components. As we have shown we can create buffers in the same way, but now using two-place components. Therefore we have extended the set of construction methods with the double wagging composition and the double multi-wagging composition. Similar we have derived the relations for the performance parameters between a constructed buffer using the two-place construction methods and its subcomponents. The relations have been shown in the double wagging theorem and the double multi-wagging theorem. Furthermore we have derived some restrictions for the two-place construction methods such that, when obeyed, the application of the construction methods indeed yields a \((k, \delta)\) - optimal buffer. These restrictions have been shown in the double wagging optimality theorem and the double multi-wagging optimality theorem.

With those extensions we have presented three new algorithms that generate the classes of buffers class da, class db and class dc.

We have introduced basic components with a fan-in and fan-out of three. We have presented a new construction method: the cubic composition. We have presented the cubic theorem to show the relation between the performance parameters of the constructed buffer and its subcomponents. Using that composition method we have constructed two buffer families: the block buffers and the cubic buffers. Furthermore we have presented the cubic optimality, where we have shown some restrictions for the cubic composition such that, when obeyed, the application of the cubic composition indeed yields a \((k, \delta)\) - optimal buffer. Hence we have shown that the block buffers and the cubic buffers are \((k, \delta)\) – optimal.

Furthermore we have introduced two miscellaneous system designs: the permutators and blockreversers and we have shown that the queuing formula also holds for these systems.

Summarising we conclude that the material we have presented in the previous chapters provides adequate means to construct buffers, in particular the subclass of \((k, \delta)\)-optimal buffers we were interested in. We have systematically searched for \((k, \delta)\)-optimal buffers and have shown some algorithms to create them.

### 7.2 Further study

The core of this thesis is about designing \((k, \delta)\)-optimal buffers. To verify the theoretical results, one can implement the \((k, \delta)\)-optimal buffer designs from this thesis. To implement these designs a VLSI program language TANGRAM is given by Schalij [Sch]. For a more comprehensive understanding of VLSI programming we refer to C.H. van Berkel [vBer]. Van Berkel not only presents an introduction to VLSI programming, but also discusses some practical experiences.
References


## Appendices

### A Sequence functions for the basic components

#### A.1 Sequence functions for the basic component $\text{Split}_{k,l}$

<table>
<thead>
<tr>
<th>k</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a#i</td>
<td>2i</td>
<td>2i</td>
<td>2i</td>
<td>2i</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>4i+1</td>
<td>6i+1</td>
<td>8i+1</td>
<td>10i+1</td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>4i+3</td>
<td>6i+{3,5}</td>
<td>8i+{3,5,7}</td>
<td>10i+{3,5,7,9}</td>
</tr>
<tr>
<td>1</td>
<td>a#i</td>
<td>2i</td>
<td>2i</td>
<td>2i</td>
<td>2i</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>4i+3</td>
<td>6i+3</td>
<td>8i+3</td>
<td>10i+3</td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>4i+1</td>
<td>6i+{1,5}</td>
<td>8i+{1,5,7}</td>
<td>10i+{1,5,7,9}</td>
</tr>
<tr>
<td>2</td>
<td>a#i</td>
<td>2i</td>
<td>2i</td>
<td>2i</td>
<td>2i</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>6i+5</td>
<td>8i+5</td>
<td>10i+5</td>
<td>12i+5</td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>6i+{1,3}</td>
<td>8i+{1,3,7}</td>
<td>10i+{1,3,7,9}</td>
<td>12i+{1,3,7,9,11}</td>
</tr>
<tr>
<td>3</td>
<td>a#i</td>
<td>2i</td>
<td>2i</td>
<td>2i</td>
<td>2i</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>8i+7</td>
<td>10i+7</td>
<td>12i+7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>8i+{1,3,5}</td>
<td>10i+{1,3,5,9}</td>
<td>12i+{1,3,5,9,11}</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a#i</td>
<td>2i</td>
<td>2i</td>
<td>2i</td>
<td>2i</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>10i+9</td>
<td>12i+9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>10i+{1,3,5,7}</td>
<td>12i+{1,3,5,7,11}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a#i</td>
<td>2i</td>
<td>2i</td>
<td>2i</td>
<td>2i</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>12i+11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>12i+{1,3,5,7,11}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### A.2 Sequence functions for the basic component $\text{Merge}_{k,l}$

<table>
<thead>
<tr>
<th>k</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>e#i</td>
<td>4i</td>
<td>6i</td>
<td>8i</td>
<td>10i</td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>4i+2</td>
<td>6i+{2,4}</td>
<td>8i+{2,4,6}</td>
<td>10i+{2,4,6,8}</td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>2i+1</td>
<td>2i+1</td>
<td>2i+1</td>
<td>2i+1</td>
</tr>
<tr>
<td>1</td>
<td>e#i</td>
<td>4i+2</td>
<td>6i+2</td>
<td>8i+2</td>
<td>10i+2</td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>4i</td>
<td>6i+{0,4}</td>
<td>8i+{0,4,6}</td>
<td>10i+{0,4,6,8}</td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>2i+1</td>
<td>2i+1</td>
<td>2i+1</td>
<td>2i+1</td>
</tr>
<tr>
<td>2</td>
<td>e#i</td>
<td>6i+4</td>
<td>8i+4</td>
<td>10i+4</td>
<td>12i+4</td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>6i+{0,2}</td>
<td>8i+{0,2,6}</td>
<td>10i+{0,2,6,8}</td>
<td>12i+{0,2,6,8,10}</td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>2i+1</td>
<td>2i+1</td>
<td>2i+1</td>
<td>2i+1</td>
</tr>
<tr>
<td>3</td>
<td>e#i</td>
<td>8i+6</td>
<td>10i+6</td>
<td>12i+6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>8i+{0,2,4}</td>
<td>10i+{0,2,4,8}</td>
<td>12i+{0,2,4,8,10}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>2i+1</td>
<td>2i+1</td>
<td>2i+1</td>
<td>2i+1</td>
</tr>
<tr>
<td>4</td>
<td>e#i</td>
<td>10i+8</td>
<td>12i+8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>10i+{0,2,4,6}</td>
<td>12i+{0,2,4,6,10}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>2i+1</td>
<td>2i+1</td>
<td>2i+1</td>
<td>2i+1</td>
</tr>
<tr>
<td>5</td>
<td>e#i</td>
<td>12i+10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>12i+{0,2,4,6,8}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>2i+1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### A.3 Early sequence functions for the basic component $\text{Split}_{k,l}$

<table>
<thead>
<tr>
<th>$k$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a#i</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>$2y_i+1$</td>
<td>$3y_i+1$</td>
<td>$4y_i+1$</td>
<td>$5y_i+1$</td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>$2y_i+y+1$</td>
<td>$3y_i+(1,2)y+1$</td>
<td>$4y_i+(1,2,3)y+1$</td>
<td>$5y_i+(1,2,3,4)y+1$</td>
</tr>
<tr>
<td>1</td>
<td>a#i</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>$2y_i+y+1$</td>
<td>$3y_i+y+1$</td>
<td>$4y_i+y+1$</td>
<td>$5y_i+y+1$</td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>$2y_i+1$</td>
<td>$3y_i+(0,2)y+1$</td>
<td>$4y_i+(0,2,3)y+1$</td>
<td>$5y_i+(0,2,3,4)y+1$</td>
</tr>
<tr>
<td>2</td>
<td>a#i</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>$3y_i+2y+1$</td>
<td>$4y_i+2y+1$</td>
<td>$5y_i+2y+1$</td>
<td>$6y_i+2y+1$</td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>$3y_i+(0,1)y+1$</td>
<td>$4y_i+(0,1,3)y+1$</td>
<td>$5y_i+(0,1,3,4)y+1$</td>
<td>$6y_i+(0,1,3,4,5)y+1$</td>
</tr>
<tr>
<td>3</td>
<td>a#i</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>$4y_i+3y+1$</td>
<td>$5y_i+3y+1$</td>
<td>$6y_i+3y+1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>$4y_i+(0,1,2)y+1$</td>
<td>$5y_i+(0,1,2,4)y+1$</td>
<td>$6y_i+(0,1,2,4,5)y+1$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a#i</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>$5y_i+4y+1$</td>
<td>$6y_i+4y+1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>$5y_i+(0,1,2,3)y+1$</td>
<td>$6y_i+(0,1,2,3,5)y+1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a#i</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>$6y_i+5y+1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>$6y_i+(0,1,2,3,4)y+1$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.4 Late sequence functions for the basic component $\text{Split}_{k,l}$

<table>
<thead>
<tr>
<th>$k$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a#i</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>$2y_i+y-1$</td>
<td>$3y_i+y-1$</td>
<td>$4y_i+y-1$</td>
<td>$5y_i+y-1$</td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>$2y_i+2y-1$</td>
<td>$3y_i+(2,3)y-1$</td>
<td>$4y_i+(2,3,4)y-1$</td>
<td>$5y_i+(2,3,4,5)y-1$</td>
</tr>
<tr>
<td>1</td>
<td>a#i</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>$2y_i+2y-1$</td>
<td>$3y_i+2y-1$</td>
<td>$4y_i+2y-1$</td>
<td>$5y_i+2y-1$</td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>$2y_i+y-1$</td>
<td>$3y_i+(1,3)y-1$</td>
<td>$4y_i +(1,3,4)y-1$</td>
<td>$5y_i+(1,3,4,5)y-1$</td>
</tr>
<tr>
<td>2</td>
<td>a#i</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>$3y_i+3y-1$</td>
<td>$4y_i+3y-1$</td>
<td>$5y_i+3y-1$</td>
<td>$6y_i+3y-1$</td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>$3y_i+(1,2)y-1$</td>
<td>$4y_i+(1,2,4)y-1$</td>
<td>$5y_i+(1,2,4,5)y-1$</td>
<td>$6y_i+(1,2,4,5,6)y-1$</td>
</tr>
<tr>
<td>3</td>
<td>a#i</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>$4y_i+4y-1$</td>
<td>$5y_i+4y-1$</td>
<td>$6y_i+4y-1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>$4y_i+(1,2,3)y-1$</td>
<td>$5y_i+(1,2,3,5)y-1$</td>
<td>$6y_i+(1,2,3,5,6)y-1$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a#i</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>$5y_i+5y-1$</td>
<td>$6y_i+5y-1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>$5y_i+(1,2,3,4)y-1$</td>
<td>$6y_i+(1,2,3,4,5)y-1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a#i</td>
<td>$y_i$</td>
<td>$y_i$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c#i</td>
<td>$6y_i+6y-1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d#i</td>
<td>$6y_i+(1,2,3,4,5)y-1$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### A.5 Early sequence functions for the basic component Merge_{k,l}

<table>
<thead>
<tr>
<th>k</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>e#i</td>
<td>2yi</td>
<td>3yi</td>
<td>4yi</td>
<td>5yi</td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>2yi+y</td>
<td>3yi+{1,2}yi</td>
<td>4yi+{1,2,3}yi</td>
<td>5yi+{1,2,3,4}yi</td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>yi+1</td>
<td>yi+1</td>
<td>yi+1</td>
<td>yi+1</td>
</tr>
<tr>
<td>1</td>
<td>e#i</td>
<td>2yi+y</td>
<td>3yi+y</td>
<td>4yi+y</td>
<td>5yi+y</td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>2yi</td>
<td>3yi+{0,2}yi</td>
<td>4yi+{0,2,3}yi</td>
<td>5yi+{0,2,3,4}yi</td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>yi+1</td>
<td>yi+1</td>
<td>yi+1</td>
<td>yi+1</td>
</tr>
<tr>
<td>2</td>
<td>e#i</td>
<td>3yi+2yi</td>
<td>4yi+2yi</td>
<td>5yi+2yi</td>
<td>6yi+2yi</td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>3yi+{0,1}yi</td>
<td>4yi+{0,1,3}yi</td>
<td>5yi+{0,1,3,4}yi</td>
<td>6yi+{0,1,3,4,5}yi</td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>yi+1</td>
<td>yi+1</td>
<td>yi+1</td>
<td>yi+1</td>
</tr>
<tr>
<td>3</td>
<td>e#i</td>
<td>4yi+3yi</td>
<td>5yi+3yi</td>
<td>6yi+3yi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>4yi+{0,1,2}yi</td>
<td>5yi+{0,1,2,4}yi</td>
<td>6yi+{0,1,2,4,5}yi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>yi+1</td>
<td>yi+1</td>
<td>yi+1</td>
<td>yi+1</td>
</tr>
<tr>
<td>4</td>
<td>e#i</td>
<td>5yi+4yi</td>
<td>6yi+5yi</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>5yi+{0,1,2,3}yi</td>
<td>6yi+{0,1,2,3,5}yi</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>yi+1</td>
<td>yi+1</td>
<td>yi+1</td>
<td>yi+1</td>
</tr>
<tr>
<td>5</td>
<td>e#i</td>
<td>6yi+6yi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>6yi+{0,1,2,3,4}yi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>yi+1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.6 Late sequence functions for the basic component Merge_{k,l}

<table>
<thead>
<tr>
<th>k</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>e#i</td>
<td>2yi</td>
<td>3yi</td>
<td>4yi</td>
<td>5yi</td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>2yi+y</td>
<td>3yi+{1,2}yi</td>
<td>4yi+{1,2,3}yi</td>
<td>5yi+{1,2,3,4}yi</td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
</tr>
<tr>
<td>1</td>
<td>e#i</td>
<td>2yi+y</td>
<td>3yi+y</td>
<td>4yi+y</td>
<td>5yi+y</td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>2yi</td>
<td>3yi+{0,2}yi</td>
<td>4yi+{0,2,3}yi</td>
<td>5yi+{0,2,3,4}yi</td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
</tr>
<tr>
<td>2</td>
<td>e#i</td>
<td>3yi+2yi</td>
<td>4yi+2yi</td>
<td>5yi+2yi</td>
<td>6yi+2yi</td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>3yi+{0,1}yi</td>
<td>4yi+{0,1,3}yi</td>
<td>5yi+{0,1,3,4}yi</td>
<td>6yi+{0,1,3,4,5}yi</td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
</tr>
<tr>
<td>3</td>
<td>e#i</td>
<td>4yi+3yi</td>
<td>5yi+3yi</td>
<td>6yi+3yi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>4yi+{0,1,2}yi</td>
<td>5yi+{0,1,2,4}yi</td>
<td>6yi+{0,1,2,4,5}yi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
</tr>
<tr>
<td>4</td>
<td>e#i</td>
<td>5yi+4yi</td>
<td>6yi+5yi</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>5yi+{0,1,2,3}yi</td>
<td>6yi+{0,1,2,3,5}yi</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
<td>yi+y-1</td>
</tr>
<tr>
<td>5</td>
<td>e#i</td>
<td>6yi+6yi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>f#i</td>
<td>6yi+{0,1,2,3,4}yi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b#i</td>
<td>yi+y-1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B Sequence functions for the multi-wagging composite

B.1 Early sequence functions for the multi-wagging composite

\[ \text{Split}_{1,0} \xrightarrow{nyi + 1 + 0, y, \ldots, (n-2)y} \text{Split}_{2,0} \xrightarrow{\text{Split}_{1,0}} \text{Buf} \]

\[ \text{Split}_{2,0} \xrightarrow{\text{Split}_{1,0}} \text{Buf} \]

\[ f_{0,1} \xrightarrow{nyi + \lambda + (n-1)y + 1} \text{Buf} \]

\[ f_{0,2} \xrightarrow{nyi + \lambda + (n-2)y + 2} \text{Buf} \]

\[ f_{1} \xrightarrow{nyi + y + n - 1} \text{Buf} \]

\[ f_{0} \xrightarrow{nyi + \lambda + n} \text{Buf} \]

\[ \text{Merge}_{1} \xrightarrow{nyi + \lambda + (n-1)y + 1} \text{Buf} \]

\[ \text{Merge}_{2} \xrightarrow{nyi + \lambda + (n-2)y + 3 + (0, y)} \text{Buf} \]

\[ \text{Merge}_{0} \xrightarrow{nyi + \lambda + y + n + (0, y, \ldots, (n-1)y)} \text{Buf} \]

\[ y + \lambda + n + 1 \]
B.2 Late sequence functions for the multi-wagging composite

\[
\begin{align*}
\text{Split}_{1,1} & \quad n_{y_1} + (y_1-1) + (n-1)y + (n-2)y \\
\text{Split}_{1,2} & \quad \ldots \ldots \\
\text{Split}_{2,1} & \quad n_{y_1} + (n-1)(y_1-1) + (0) \\
\text{Buf} & \quad n_{y_1} + (n-1)y + (n-2)y + (n-1)y \\
\text{X}_{n_1} & \quad f_{n_1} + n_{y_1} + \lambda + (n-1)y + (n-1)y + (0) \\
\text{X}_{n_2} & \quad f_{n_2} + n_{y_1} + \lambda + (n-1)y + (n-1)y + (n-2)y \\
\text{X}_{n_3} & \quad f_{n_3} + n_{y_1} + \lambda + (n-1)y + (n-1)y + (0) \\
\text{X}_{n_4} & \quad f_{n_4} + n_{y_1} + \lambda + (n-1)y + (n-1)y + (n-1)y \\
\text{Buf} & \quad \ldots \\
\text{Merge}_{0,1} & \quad n_{y_1} + \lambda + n(n-1)y + (n-1)y + (n-1)y \\
\text{Merge}_{1,0} & \quad n_{y_1} + \lambda + n(n-1)y + (n-1)y + (0, y, \ldots (n-2)y) \\
\text{Merge}_{2,0} & \quad n_{y_1} + \lambda + (n+1)(y-1) + (n-1) y \\
\end{align*}
\]
B.3 General sequence functions for the multi-wagging composite

\[ a \gamma \]

\[ \text{Split}_{a,1} \rightarrow \text{Split}_{a,2a+1} \rightarrow \text{Split}_{a,2} \rightarrow \text{Buf} \]

\[ C_{a,1} \text{ nyi } + \frac{(n-1)y+g0}{y+g0} \]
\[ C_{a,2} \text{ nyi } + \frac{(n-2)y+g0+g1}{y+g0+g1} \]
\[ C_{1} \text{ nyi } + \frac{g0+g1+\ldots+gn-2+v1+v2+\ldots+vn-2}{y+g0+g1+\ldots+gn-2+v1+v2+\ldots+vn-2} \]
\[ C_{0} \text{ nyi } + \frac{g0+g1+\ldots+gn-1+v1+v2+\ldots+vn-1}{g0+g1+\ldots+gn-1+v1+v2+\ldots+vn-1} \]

\[ X_{a,1} \]
\[ f_{a,1} \text{ nyi } + \frac{\lambda+(n-1)y+g0}{\lambda+(n-1)y+g0} \]

\[ X_{a,2} \]
\[ f_{a,2} \text{ nyi } + \frac{\lambda+(n-2)y+g0+g1}{\lambda+(n-2)y+g0+g1} \]

\[ X_{1} \]
\[ f_{1} \text{ nyi } + \frac{\lambda+\lambda+\lambda+\lambda+\lambda+\lambda}{\lambda+\lambda+\lambda+\lambda+\lambda+\lambda} \]

\[ X_{0} \]
\[ f_{0} \text{ nyi } + \frac{\lambda+\lambda+\lambda+\lambda+\lambda+\lambda}{\lambda+\lambda+\lambda+\lambda+\lambda+\lambda} \]

\[ \text{Buf} \rightarrow \text{Merge}_{a,2} \rightarrow \text{Merge}_{a,3} \rightarrow \text{Merge}_{a,n} \]

\[ \text{nyi}(\lambda+(n-1)y+g0+g1+\ldots+vn-1+0) \]
\[ \text{nyi}(\lambda+y+g0+g1+\ldots+vn-1+1) \]
\[ \text{nyi}(\lambda+g0+g1+\ldots+vn-1+0) \]
\[ \text{nyi}(\lambda+g0+g1+\ldots+vn-1+0) \]