Performance analysis of VLSI programs

Performance Analysis of VLSI Programs

by

E. van de Sluis A.F. van der Stappen

91/04

April, 1991

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Performance Analysis of VLSI Programs

E. van de Sluis A.F. van der Stappen

Eindhoven University of Technology
Dept. of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

The CP-0 programming language is described as an interface between the design of a system and its implementation as a VLSI layout. Before its translation into a VLSI layout, a CP-0 program is translated into a so-called handshake circuit. This circuit is optimised and its speed and size are estimated. The translation method and the optimisations are described. Furthermore, a formal method is introduced to compare CP-0 programs, by estimating their size and speed when implemented as handshake circuits. The method is applied to two CP-0 designs for dynamic programming.

1 Introduction

A VLSI program is the description of a VLSI circuit in an algorithmic language [BK91]. It is the task of a *silicon compiler* to translate VLSI programs into VLSI layouts. The language should be such, that VLSI programs can be written without any knowledge of the underlying communication protocol or implementation medium. This has as advantage that the programmer only needs to cope with the problem of writing a correct program that satisfies a given specification. Given this specification, the programmer makes a design of the VLSI program. This design step results in a network of Communicating Sequential Processes (CSP). To specify the processes, a CSP-like notation is adopted (cf. [Mar86, Pee90a, BK91]).

In general, there is not just one program that satisfies a given specification, but several. Therefore, we need criteria to compare programs. In traditional programming, programs are compared by estimating the amount of time and memory a program requires during execution. This finds its analogy in VLSI programming, where we can compare programs by estimating the *size* and *speed* of the programs, when implemented as VLSI layouts.

Just as performance analysis of traditional programs is done on an abstract, implementation independent level, we do not want to bother the VLSI programmer with the intricate details of VLSI circuits. Therefore, the performance analysis method should be

based on the VLSI programs, and not on their translation to VLSI layouts. However, such a method requires some knowledge of this translation to give useful results. So, what we need is an interface between the (high-level) VLSI programming language and its translation to (low-level) VLSI layouts.

In [BS88] and [BK91] Van Berkel *et al.* propose such an interface. They do not translate VLSI programs directly to VLSI layouts, but use an intermediate representation. Their approach is summarized in Figure 1.

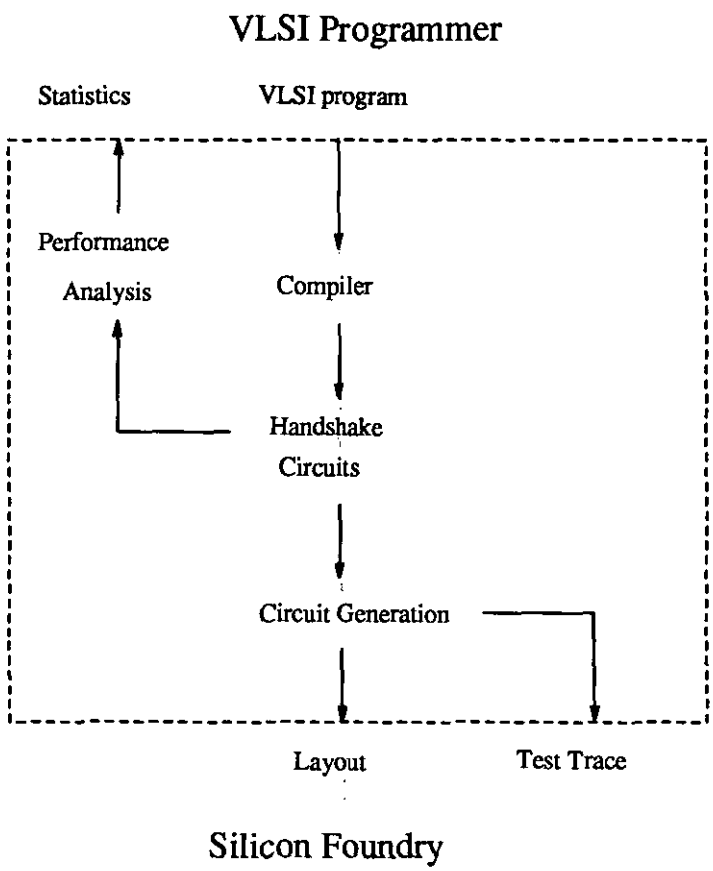

Figure 1: The development of VLSI circuits

A VLSI program consists of a number of concurrent processes that communicate via message passing over common channels. Each process of a VLSI program is first translated to an *abstract* or *handshake* circuit. Such a circuit consists of a list of basic or handshake components. Via a number of channels, each component communicates with other components on the list, or with the environment of the circuit. In the second step of the translation, each component is replaced by its corresponding implementation as a VLSI circuits, and the overall layout is generated. In this step, also a *test trace* can be generated to test a chip after fabrication.

In this paper, we introduce a method that estimates size and speed of single processes of VLSI programs. This method is based on the translation of these processes to handshake circuits. Therefore, our method is only useful to compare different VLSI processes, and should not be used to estimate the actual size and speed of VLSI layouts.

This paper is organised as follows. In Section 2 we describe the CP-0 programming language, which is a CSP-like language to specify VLSI processes. A complete CP-0 program consists of a finite number of these processes that communicate via common channels of the processes. We describe how CP-0 programs can be translated into handshake circuits. This translation can result in rather inefficient circuits, so two post-optimisations are applied to make them more efficient. In Section 3 we present our (formal) performance analysis method. This method results in formulae for both size and speed, which express the size and speed of a VLSI process in size and speed estimates of the handshake components. Given the formal framework of Section 3, it was straightforward to make an implementation of our method. This implementation was applied to the two different designs for dynamic programming of [MS89]. The results of this comparison are given in Section 4. We end this paper with some concluding remarks in Section 5. To illustrate the application of our method, an example of a size and speed derivation is included in the appendix.

2 Translation and optimisation of CP-0 processes

We first describe in Section 2.1 the CP-0 programming language by giving a BNF-grammar. Furthermore, we discuss how processes specified in this language can be translated into networks of "components" that interact by handshake signaling. Such networks are called *abstract* or *handshake* circuits (cf. [BS88], and [BK91]). The translation of CP-0 processes requires a relatively small set of different handshake components, basically one for each primitive concept of the language. This set of components is described in Section 2.2. Given this set, we describe a translation method in Section 2.3. This method consists of a sequence of syntax-directed decompositions until the level of the handshake components is reached. This method can result in rather inefficient circuits. Inefficiences can be eliminated by applying a number of post-optimisations. In Section 2.4 we describe two possible optimisations. These optimisations are incorporated in our performance analysis method of Section 3.

2.1 The CP-0 language

The CP-0 language was introduced by Van Berkel *et al.* in [BS88] as a notation for VLSI programs. In this paper, we consider a restricted class of this language. The processes that we consider are generated by the BNF-grammar of Table 1. In this grammar the semicolon expresses sequential composition. The comma, which takes priority over the semicolon, expresses concurrency. We have omitted any declaration part in this grammar, but it should be noticed that all variables and constants must be declared locally, while channels can be used to communicate with the environment of the process.

Program	::=	StatList
StatList	::=	Atom
		| StatList ; ... ; StatList
		| StatList , ... , StatList
		| (StatList)^n
Atom	::=	Ass
		| Ocomm
		| Icomm
Ass	::=	Var := Exp
Ocomm	::=	Chan ! Exp
Icomm	::=	Chan ? Var
Exp	::=	Con
		| Var
		| Exp □ ... □ Exp

Table 1: The CP-0 grammar

A sequence of statements S can be repeated n times by applying the so-called *repetition* operator. This is denoted by S^n.

At the heart of any CP-0 process lie the so-called *atomic statements*, or simply *atoms*. From the grammar we see that we distinguish three kinds of atoms: input actions, output actions, and assignment statements. With an input action, an incoming message on a channel a can be received in a variable x. This is denoted by $a?x$. With an output action we can send an evaluated expression E along a channel, as denoted by $a!E$. We also have the Pascal-like assignment statement to assign an expression to a variable. An expression consists of variables and constants, which can composed by binary operators (e.g. addition). The binary operators are represented by the '□' symbol.

2.2 The handshake components

The translation method of Section 2.3 consists of a sequence of decompositions until the level of the so-called *handshake* components is reached. In this section we give a *specification* of our set of components. For this specification we adopt the notation of [BS88]. The *implementation* of components is discussed in [Kam90].

A specification of a handshake component is based on its interface to the external world. This interface consists of a set of named *ports*. Ports are either *passive* or *active*, depending on their role during a handshake. When a channel a connects two components, then a must connect an active port with an passive port. A communication is requested by the active side of the channel and subsequently acknowledged by the other side. The communication interval $a^\bullet$ denotes the communication at the active side, and a° the communication at the passive side of channel a. The active communication interval $a^\bullet$ begins with sending a request and ends with the receipt of the corresponding acknowledgement. The passive interval a° starts with the receipt of the request and ends with the issue of an

acknowledgement. It is clear that the passive interval a° is enclosed in time by the active interval $a^\bullet$.

Now, a specification of a handshake component consists of a specification of the communication intervals, which are either passive or active. A specification can be made according to the following rules (cf. [Pee90b]):

- For communication channel a, $a^\bullet$ is an active, and a° a passive communication interval.
- If A and B are passive and active intervals respectively, then $A : B$ denotes the interval that starts with the receipt of requests on all channels in A, followed by interval B, and ends with the sending of acknowledgements on all channels in A. We say that B is "enclosed in time" by A. The communication interval $A : B$ is passive.
- If A and B are intervals of the same activity, then $A \bullet B$ defines an execution of A and B such that the two intervals overlap. The interval $A \bullet B$ has the same activity as A and B.
- If A and B are intervals of the same activity, then $A ; B$ denotes the sequential order of A and B. The interval is of the same activity as A and B.
- If A and B are intervals of the same activity, then A , B is the interval in which A and B may occur in either order, but may overlap as well. The interval has the same activity as A and B.
- For passive intervals A and B, $A \mid B$ defines the execution of either A or B. The environment makes the choice which interval is activated. No overlap of A and B is allowed. The interval $A|B$ is passive.
- For an interval A, A^n denotes the interval of the same activity in which A is activated exactly n times.
- For an interval A, $[A]$ denotes the interval of the same activity in which A is repeatedly activated; completion of this communication interval will never occur.

Similar to [Pee90b], we divide the handshake components into three classes: the *control* components, the *data-manipulation* components, and the *data-control* components.

In this paper, we distinguish four different control components. Their specification is given in Table 2. Each control component has an activation channel a, which is used to trigger the component. When triggered, control signals are sent according to their specification.

The *sequencer* is a component that, after receiving a request via its a-channel, communicates once over all its b-channels, in sequential order, and finally sends an acknowledgement via its a-channel. It is used to implement the semicolon in CP-0 processes. The *concursor*, when initiated via its a-channel, independently triggers its b-channels in any order. After

Name	Specification	Notation
sequencer	$[a^\circ : (b_0^\bullet; \ldots; b_{k-1}^\bullet)]$	$seq(k)$
concursor	$[a^\circ : (b_0^\bullet, \ldots, b_{k-1}^\bullet)]$	$conc(k)$
repeater	$[a^\circ : (b^\bullet)^n]$	rep_n
mixer	$[(a_0^\circ \mid \ldots \mid a_{k-1}^\circ) : b^\bullet]$	$mix(k)$

Table 2: The control components

completion of communication on all b-channels, the concursor sends an acknowledgement via its a-channel. When triggered, the *repeater* component communicates exactly n times via its b-channel, after which an acknowledgement is sent via its a-channel. The *mixer* communicates via its b-channel if one of its a-channels is triggered; the choice is left to the environment. However, the control communication on the k input channels must be in mutual exclusion.

The data-manipulation components are used for the distribution, gathering, storage, and operation of data. We distinguish six different data-manipulation components. They are specified in Table 3.

Name	Specification	Notation
multiplexer	$[(a_0^\circ?v \mid \ldots \mid a_{k-1}^\circ?v) : b^\bullet!v]$	$mux(k)$
demultiplexer	$[(a_0^\circ!v \mid \ldots \mid a_{k-1}^\circ!v) : b^\bullet?v]$	$dmx(k)$
constant c	$[[c.w_0^\circ!c], \ldots, [c.w_{k-1}^\circ!c]]$	$con(k)$
variable x	$[x.r^\circ?x \mid ([x.w_0^\circ!x], \ldots, [x.w_{k-1}^\circ!x])]$	$var(k)$
passivator	$[a^\circ?v \bullet b^\circ!v]$	$pass$
k-ary oper. $\Box$	$[a^\circ?(v_0 \Box \ldots \Box v_{k-1}) : (b_0^\bullet?v_0 \bullet \ldots \bullet b_{k-1}^\bullet?v_{k-1})]$	$\Box(k)$

Table 3: The data-manipulation components

The *multiplexer* component is used to merge k data channels on one data channel. The data communication on the k input channels must be in mutual exclusion. The *demultiplexer* is the counterpart of the multiplexer. It is used for the splitting of data on k output channels. Communication is initiated by the demanding side. These demands must be in mutual exclusion.

To allow storage of data, we have two handshake components. The *constant* is a component that upon request on one of its $c.w$ channels, sends its value over the same channel. The *variable* can store data via its $x.r$ channel, and send this data via one of its $x.w$ channels.

The *passivator* is used as a connector for communication channels between two active communication partners. It is used to connect the communicating channels of different processes. In our translation method, the two sides of such channels are active, so they cannot be connected directly.

Operators are used to perform operations on data. They have k input channels and one output channel to communicate the result of the operation. The k-ary operator is triggered

by the input side. We do not have the *reverse* component, as in [BS88], i.e., where the activities are reversed.

There is only one data-control component, viz. the *transferrer*. Is is used to trigger data flow by connecting a control component with two data-manipulation components. It is specified as follows:

$$[a^\circ : (b^\bullet ?v \bullet c^\bullet !v)].$$

The transferrer is denoted by *trf*. When triggered via its a-channel, its reads data via its b-channel, and transfers this data via its c-channel.

2.3 The translation method

The first step in the translation of CP-0 processes consists of two program transformations. The first transformation concerns assignment statements $x := E$ where x appears in E. To avoid read/write conflicts, the following transformation must be applied to these assignment statements (cf. [BS88]):

$$x := E \Rightarrow (x0 := x\,;\; x := E^x_{x0}).$$

The second transformation concerns multiple occurrences of an expression E in a process, where $E = E_0 \Box \ldots \Box E_{n-1}$. Due to the second optimisation that we apply (single realisation of expressions, see Section 2.4), these occurrences cannot be evaluated *concurrently*, as for instance in the process

$$(a!x + y\,,\; b!x + y).$$

This kind of processes has to be transformed such, that concurrent evaluation of the same expression cannot take place. The example above could for instance be transformed to

$$(a!x + y\,,\; b!y + x) \text{ or } (xy := x + y\,;\; a!xy\,,\; b!xy),$$

since $x + y$ and $y + x$ are considered different expressions, and concurrent read from the same variable *is* possible. We note that, strictly speaking, a similar transformation has to be performed with respect to the first optimisation of Section 2.4, i.e., single realisation of atoms. However, the reader can check that atoms that have multiple concurrent occurrences can simply be replaced by a single occurrence.

We illustrate the translation of CP-0 processes with an example. This example is the following process S, which is not a very meaningful process, but shows most features of the CP-0 language:

$$S = (a?x\,,\; b?y\,;\; z := (x \max y)\,;\; d!(x + y + z))^n$$

Process S concurrently stores values from channels a and b in variables x and y respectively, then assigns the maximum of x and y to z, and finally outputs the sum of the three variables via channel d. This process is repeated n times ($n > 0$).

After application of the above program transformation, the following (informally described) translation of a CP-0 process results in a circuit of handshake components. Such a circuit is called a *handshake circuit.*

1. Make a parse tree of the process via a syntax directed decomposition. The result of this step for process S is shown in Figure 2.

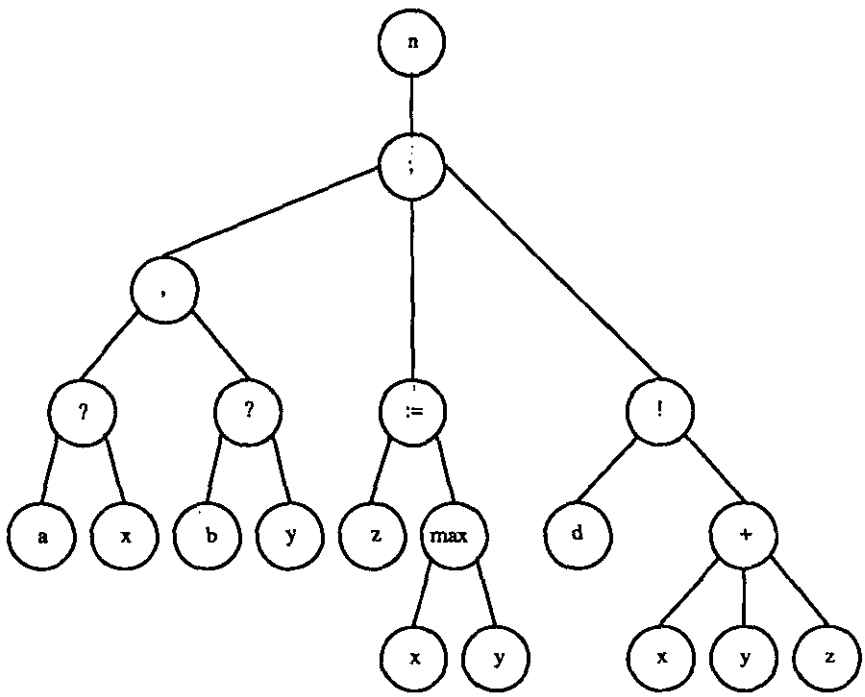

Figure 2: The parse tree of process S

2. Replace each *node* in the parse tree by its corresponding handshake component. For the nodes that correspond with control and arithmetic operators, this step is straightforward. The input, output, and assignment symbols ('?', '!', and ':=' respectively) are replaced by transferrers. For process S, this step is depicted in Figure 3. From this figure we see that the arcs of the parse tree are replaced by channels, which connect active and passive ports of components. The active ports are indicated by small filled circles, the passive ports by open ones.

3. Replace each *leave* in the tree by either channels, variables, or constants. If a variable (constant) is read k times ($k > 0$) in the process, then a $var(k)$ ($con(k)$) component is introduced. Furthermore, if we *write* to a variable k times, with $k > 1$, then we have to place a $mux(k)$ component in front of the variable. Similarly, if the process contains k "writes" to a channel, then a $mux(k)$ component is also required. Finally,

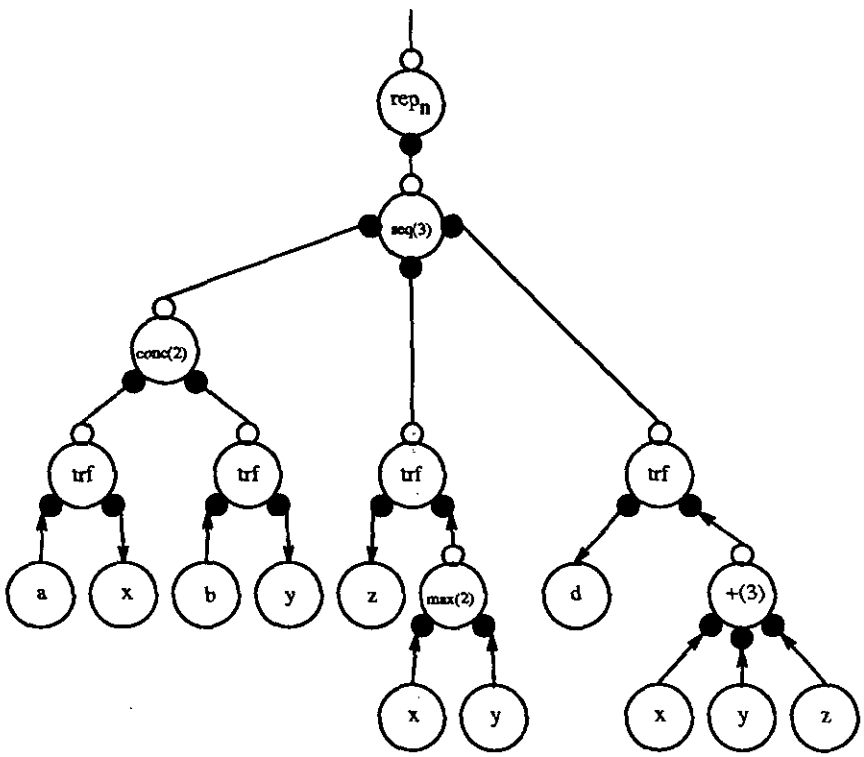

Figure 3: The parse tree of S after step 2

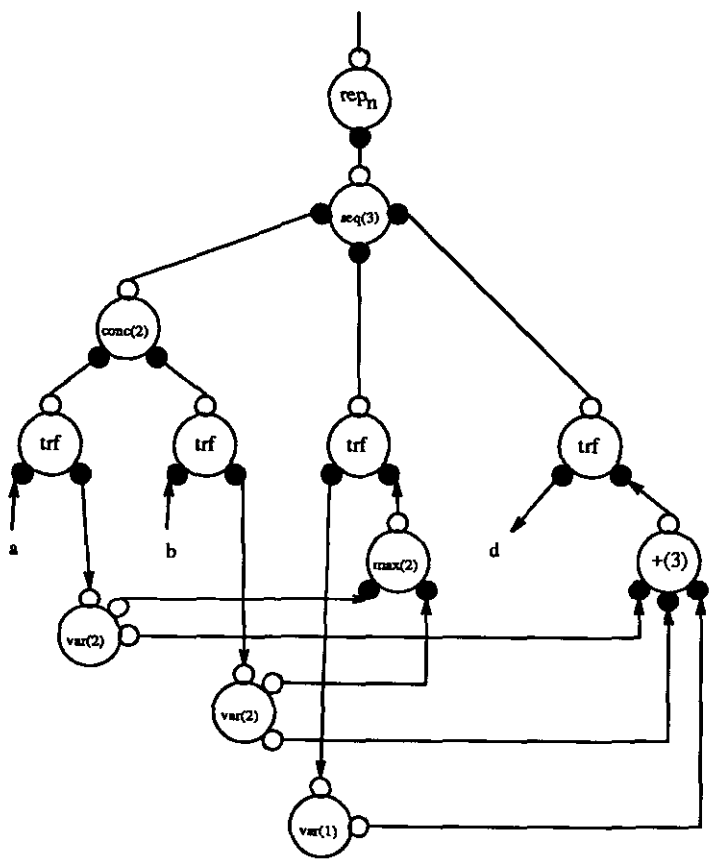

Figure 4: The handshake circuit of S

for k "reads" from a channel a $dmx(k)$ component is introduced. For process S, this final step results in the handshake circuit of Figure 4.

For process S, we had to introduce three components: two $var(2)$ components for variables x and y, and a $var(1)$ component for variable z. Note that for the translation of S no (de)multiplexers are required. In Section 2.4 we give examples where (de)multiplexers *are* required.

Steps one and two in our method are similar to the command and expression decomposition steps of [BS88]. Step three also consists of two steps in [BS88], and are called variable and channel decomposition there.

The translation strategy above applies only to the translation of single CP-0 processes. Remember that CP-0 programs consist of a number of these processes that communicate over common channels. So, these programs can be translated by translating their processes, and connecting their common channels. Since all channels are active in our translation, this connection cannot be done directly. This problem is solved by making one adding *passivator* components to these channels.

2.4 Optimisation of handshake circuits

In this section we discuss two possible optimisations that can be applied to a handshake circuit that is the result of a translation as described in Section 2.3. These optimisations concern multiple occurrences of *atoms* and *expressions* in a CP-0 process. Since these introduce "expensive" data components in the handshake circuit, we want to realise them only once, and realise multiple invocations by introducing "cheap" control components. The optimisations are described in subsequent sections, and are illustrated with small examples.

2.4.1 Single realisation of atoms

Suppose we have a CP-0 process that contains the atom $a!x$ three times, and the atom $a!y$ just once. Here we assume that these atoms occur in mutual exclusion. When we follow our translation method, the data-flow of these statements would result in a circuit as given in Figure 5. Notice the demultiplexer ($dmx(4)$) for the multiple writes to channel a.

Figure 6 shows a different translation of the same program. It is not difficult to see that this circuit performs the same function. Since we replaced a considerable part of the area-consuming data components by less expensive control components, this translation yields a considerable reduction of the size (area) of the circuit, especially when a large wordlength is used (e.g. 16 bits).

Above, we have only discussed the optimisation for an output action. It is not difficult to see that for the other atoms (input actions and assignments) similar optimisations can be applied, which means that *demultiplexers* can be smaller, or sometimes disappear completely.

where

$$\alpha_d(E) = \begin{cases} 0 & E \in Var \cup Con \\ a_{dmx(\#_e(E))} + \alpha_e(E) & E = E_0 \,\square\, \ldots \,\square\, E_{n-1} \end{cases}$$

and

$$\alpha_e(E) = \begin{cases} 0 & E \in Var \cup Con \\ a_{\square(n)} + (\Sigma i : 0 \leq i < n : \alpha_e(E_i)) & E = E_0 \,\square\, \ldots \,\square\, E_{n-1} \end{cases}$$

□

We assume that $a_{mux(1)} = a_{dmx(1)} = a_{mix(1)} = 0$.

3.3 Speed estimates

The timing analysis for the optimised realisation is based on the syntax of the process. An estimate for the 'speed of a CP-0 process' is given by a function $\tau : (StatList \cup Exp) \rightarrow \mathcal{R}$. Here we mean by the speed of a CP-0 process the time that is spent within the handshake components. So, no delays are included for wires or communication with the environment.

The speed estimates for sequential composition, concurrent composition, and repetition is rather straightforward. They equal the internal switching time of the (control) component, plus a speed estimate for the statements that are activated. Since we assume that the statements in a concurrent composition are executed in parallel, this estimate equals the *maximum* of all the speed estimates of these statements.

The speed estimates for input actions, output actions, and assignment statements depend on the multiplicity functions. If, for example, an atom appears n times, $n > 1$, in a CP-0 process S then the speed estimate for this atom is increased by a delay $t_{mix(n)}$ for an n-ary mixer. A delay t_{trf} is always included for a transferrer. In case of an n-ary write to a single variable or a single channel, a delay $t_{mux(n)}$ is added. Similarly, we add a delay $t_{dmx(n)}$, in case of a n-ary read from a single channel. A delay $t_{read(n)}$ is added in case of an n-ary read from a variable or constant. Similarly, the delay for writing to a n-ary variable is denoted by $t_{write(n)}$. As will be explained in Section 4, variables can be implemented such, that the write delay depends on the number of *read* ports of a variable.

For expressions, we assume that the evaluation of all operands of an n-ary $\square$ expression (e.g. an n-ary sum) start at the same time. A delay $t_{dmx(n)}$ is added for a multiplicity n of each expression. When we denote the delay in component $\square(n)$ by $t_{\square(n)}$, we can now give the definition of speed estimate function τ.

Definition 3.12 *(Speed estimate function τ)*

$$\begin{aligned} \tau(S_0; \ldots; S_{n-1}) &= t_{seq(n)} + (\Sigma i : 0 \leq i < n : \tau(S_i)) \\ \tau(S_0, \ldots, S_{n-1}) &= t_{conc(n)} + (\mathbf{MAX}\, i : 0 \leq i < n : \tau(S_i)) \\ \tau((S)^n) &= t_{rep_n} + n \cdot \tau(S) \end{aligned}$$

optimisation again with an example. Consider the following CP-0 process:

$$(a!x + y\,;\; b!x + y)$$

With our translation method, the data-flow part of this process is translated to the circuit of Figure 7. We see that the area-consuming addition operator is duplicated.

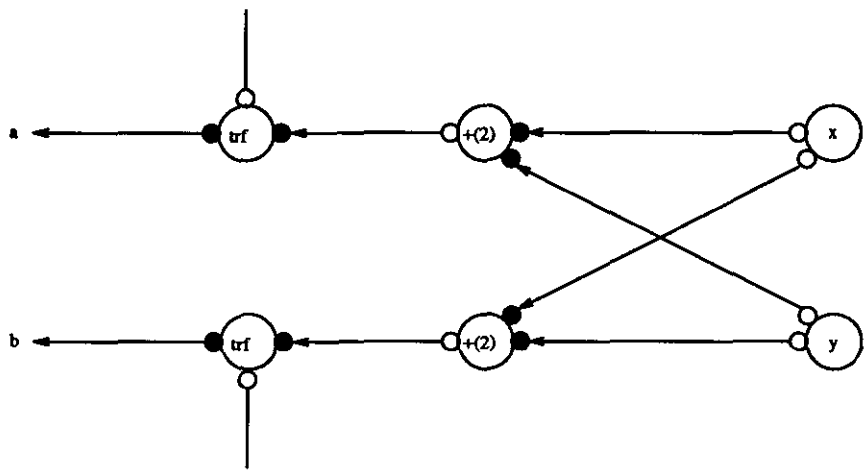

Figure 7: Two additions with two addition operators

In Figure 8 another translation for the same program is depicted. With respect to area, this circuit is an optimisation, since we need only one addition operator. However, the introduction of a demultiplexer means that the circuit is slower. We decided to apply this (area) optimisation, since we think that in this case the loss of speed is neglectable when compared with the gain in area.

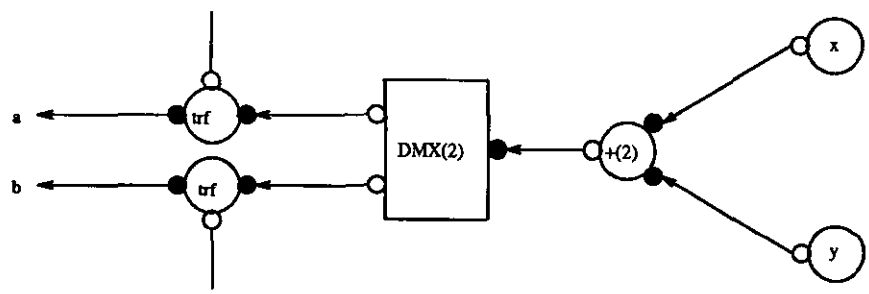

Figure 8: Two additions with just one addition operator

Note that a demultiplexer requires the demanding sides to be in mutual exclusion. For multiple invocations of expressions, this is guaranteed by our second program transformation of Section 2.3.

Note also that this optimisation is applied only if *complete* expressions on the right hand sides of output or assignment statements are duplicated. So, we apply no common *sub*expression elimination, which is suggested in [Mak90] as an optimisation.

3 Performance analysis of CP-0 processes

Now that we have a unique translation for an arbitrary CP-0 program into its optimised realisation, we can use this translation for the performance analysis of CP-0 programs. The method that we describe in this section is only applicable to *single* CP-0 processes. To derive size and speed estimates for complete CP-0 programs, we have to add size and speed estimates for the passivators, which are necessary to connect the communicating channels of the processes.

In order to give the formulae for speed and size estimation, we need to define a number of sets and functions that formalise the notions of different atoms and multiple read/write that are used in the description of the translation strategy. This will be done in Section 3.1. Then in Section 3.2, we present our method to estimate size of CP-0 processes. In Section 3.3, we give our method for speed estimates. The method expresses the size and speed estimate of a process in estimates for the handshake components that are used in the translation of the process. Therefore we give in Section 4 the size and speed of each handshake component. We conjecture that the values we give are not very realistic, but they are necessary to show an application of our method in Section 4.

In this section we use the following notation:

- $S, S_0, \ldots, S_{n-1}$: statements, denoted by $S, S_0, \ldots, S_{n-1} \in StatList$,
- $E, E_0, \ldots, E_{n-1}$: expressions, denoted by $E, E_0, \ldots, E_{n-1} \in Exp$,
- x, y: variables, denoted by $x, y \in Var$, and
- a: a channel, denoted by $a \in Chan$.

The number of elements of a set X is denoted by $|X|$. The formal framework that we introduce in Section 3.1 consists of the definition of a number of recursive functions. One of the arguments of these functions is a statement (an element of $StatList$). As a notational convenience, we allow ourselves to omit this argument when we give a complete process as an argument.

3.1 The formal framework

The first step of the translation strategy is the realisation of all variables, constants and channels in the CP-0 process. We introduce a function $ChanSet : StatList \rightarrow \mathcal{P}(Chan)$. Set $ChanSet(S)$ contains all channels that are used in CP-0 process S.

Definition 3.1 *(ChanSet)*

$$
\begin{array}{ll}
ChanSet(a?x) & = \{a\} \\
ChanSet(a!E) & = \{a\} \\
ChanSet(x := E) & = \emptyset \\
ChanSet(S_0; \ldots; S_{n-1}) & = (\cup i : 0 \leq i < n : ChanSet(S_i)) \\
ChanSet(S_0, \ldots, S_{n-1}) & = (\cup i : 0 \leq i < n : ChanSet(S_i)) \\
ChanSet(S^n) & = ChanSet(S)
\end{array}
$$

□

A second function $VarSet : StatList \cup Exp \rightarrow \mathcal{P}(Var)$ gives the set of all variables and constants that occur in a process.

Definition 3.2 *(VarSet)*

$$
\begin{array}{ll}
VarSet(a?x) & = \{x\} \\
VarSet(a!E) & = VarSet(E) \\
VarSet(x := E) & = \{x\} \cup VarSet(E) \\
VarSet(S_0; \ldots; S_{n-1}) & = (\cup i : 0 \leq i < n : VarSet(S_i)) \\
VarSet(S_0, \ldots, S_{n-1}) & = (\cup i : 0 \leq i < n : VarSet(S_i)) \\
VarSet(S^n) & = VarSet(S) \\
VarSet(E) & = \begin{cases} \{E\} & E \in Var \cup Con \\ (\cup i : 0 \leq i < n : VarSet(E_i)) & E = E_0 \Box \ldots \Box E_{n-1} \end{cases}
\end{array}
$$

□

The second step of the translation strategy realises all different input actions, output actions, and assignment statements in a CP-0 process S. Let $AtomSet : StatList \rightarrow \mathcal{P}(Atom)$; $AtomSet(S)$ is the set of all input actions, output actions, and assignment statements in S.

Definition 3.3 *(AtomSet)*

$$
\begin{array}{ll}
AtomSet(a?x) & = \{a?x\} \\
AtomSet(a!E) & = \{a!E\} \\
AtomSet(x := E) & = \{x := E\} \\
AtomSet(S_0; \ldots; S_{n-1}) & = (\cup i : 0 \leq i < n : AtomSet(S_i)) \\
AtomSet(S_0, \ldots, S_{n-1}) & = (\cup i : 0 \leq i < n : AtomSet(S_i)) \\
AtomSet(S^n) & = AtomSet(S)
\end{array}
$$

□

Function $ExpSet : StatList \rightarrow \mathcal{P}(Exp)$ computes the collection of right-hand sides of assignment statements and output actions. However, only right-hand sides that contain operators are included in the collection. The function is defined as follows:

Definition 3.4 *(ExpSet)*

$$
\begin{aligned}
ExpSet(a?x) &= \emptyset \\
ExpSet(a!E) &= \begin{cases} \emptyset & E \in Var \cup Con \\ \{E\} & E = E_0 \,\square\, \ldots \,\square\, E_{n-1} \end{cases} \\
ExpSet(x := E) &= \begin{cases} \emptyset & E \in Var \cup Con \\ \{E\} & E = E_0 \,\square\, \ldots \,\square\, E_{n-1} \end{cases} \\
ExpSet(S_0; \ldots; S_{n-1}) &= (\cup i : 0 \leq i < n : ExpSet(S_i)) \\
ExpSet(S_0, \ldots, S_{n-1}) &= (\cup i : 0 \leq i < n : ExpSet(S_i)) \\
ExpSet(S^n) &= ExpSet(S)
\end{aligned}
$$

□

These four sets facilitate the definition of functions that express the multiplicity of read and write operations from/to variables and channels, and the multiplicity of atoms (input actions, output actions, assignment statements) in CP-0 processes. These functions offer us the possibility to count the number and determine the arity of multiplexers, demultiplexers, variables, constants, and mixers. The multiplicity functions are denoted by '#' symbols. First we give two functions for the multiplicity of channels; $\#_?, \#_! : Chan \times StatList \to \mathcal{N}$. They are defined as follows:

Definition 3.5 *(Channel Multiplicities)*

Let $a \in Chan$. Then:

$$
\begin{aligned}
\#_?(a, S) &= |\{x : x \in VarSet(S) \wedge a?x \in AtomSet(S) : x\}| \\
\#_!(a, S) &= |\{x : x \in VarSet(S) \wedge a!x \in AtomSet(S) : x\}| \\
&\quad + \\
&\quad |\{E : E \in ExpSet(S) \wedge a!E \in AtomSet(S) : E\}|
\end{aligned}
$$

□

To compute read and write multiplicities of variables (and constants), we next define $\#_w : Var \times StatList \to \mathcal{N}$, and $\#_r : (Var \cup Con) \times (StatList \cup Exp) \to \mathcal{N}$. Note that we do not distinguish between variables and constants and consider a constant as a variable with a zero write multiplicity, so without a write port.

Definition 3.6 *(Read and Write Multiplicities)*

Let $x \in Var \cup Con$. Then:

$$
\begin{aligned}
\#_w(x,S) &= |\{a : a \in ChanSet(S) \wedge a?x \in AtomSet(S) : a\}| \\
&\quad + \\
&\quad |\{y : y \in VarSet(S) \wedge x := y \in AtomSet(S) : y\}| \\
&\quad + \\
&\quad |\{E : E \in ExpSet(S) \wedge x := E \in AtomSet(S) : E\}| \\
\#_r(x,S) &= |\{a : a \in ChanSet(S) \wedge a!x \in AtomSet(S) : a\}| \\
&\quad + \\
&\quad |\{y : y \in VarSet(S) \wedge y := x \in AtomSet(S) : y\}| \\
&\quad + \\
&\quad (\Sigma E : E \in ExpSet(S) : \#_r(x,E))
\end{aligned}
$$

Where:

$$
\#_r(x,E) = \begin{cases} 1 & E \in Var \cup Con \wedge x = E \\ 0 & E \in Var \cup Con \wedge x \neq E \\ (\Sigma i : 0 \leq i < n : \#_r(x,E_i)) & E = E_0 \Box \ldots \Box E_{n-1} \end{cases}
$$

□

Function $\# : Atom \times StatList \rightarrow \mathcal{N}$ gives the multiplicity of each atom in a CP-0 process.

Definition 3.7 *(Atom Multiplicities)*

Let $g \in Atom$. Then:

$$
\begin{aligned}
\#(g, a?x) &= \begin{cases} 1 & g = a?x \\ 0 & g \neq a?x \end{cases} \\
\#(g, a!E) &= \begin{cases} 1 & g = a!E \\ 0 & g \neq a!E \end{cases} \\
\#(g, x := E) &= \begin{cases} 1 & g = (x := E) \\ 0 & g \neq (x := E) \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\#(g, S_0; \ldots; S_{n-1}) &= (\Sigma i : 0 \leq i < n : \#(g, S_i)) \\
\#(g, S_0, \ldots, S_{n-1}) &= (\Sigma i : 0 \leq i < n : \#(g, S_i)) \\
\#(g, S^n) &= \#(g, S)
\end{aligned}
$$

□

Function $\#_e : Exp \times StatList \rightarrow \mathcal{N}$ gives the multiplicity of each expression in a CP-0 process. With the definition we have to be careful not to count expressions more than once when they appear in atoms that have several occurrences. For this purpose we use function *AtomSet* in the definition.

Definition 3.8 *(Expression Multiplicities)*

Let $E \in ExpSet(S)$. Then:

$$\#_e(E,S) \quad = \quad \begin{cases} 1 & S = (a!E) \\ 1 & S = (x := E) \\ 0 & S = (a?x) \\ (\Sigma g : g \in AtomSet(S) : \#_e(E,g)) & otherwise \end{cases}$$

□

3.2 Size estimates

The multiplicity functions make it quite easy to give the number and arity of multiplexers, demultiplexers, variables (constants), and mixers in a translated CP-0 process. The functions are defined in a way that takes into account the optimisation of single realisation of atoms and expressions. Our size estimates 'only' include the handshake components that are used in a CP-0 process, not the wires that are used in an actual VLSI layout. This is because we do not have any knowledge at this level (the level of the VLSI programmer) about this layout.

The number of occurrences of control components in the realisation of a CP-0 process S, and the arities of these occurrences depend on the structure of S. They are not influenced by our optimisations. The estimate for the total size of all elements of these components in S will be given by $\alpha_c(S)$.

In contrast to the control components, the number of occurrences and the arities of data manipulation components and transferrers are influenced *are* influenced by our optimisations. The estimate for the total size of all these components is given by $\alpha_d(S)$. The size estimate $\alpha(S)$ for a process S is the sum of both estimates $\alpha_c(S)$ and $\alpha_d(S)$.

Definition 3.9 *(Size estimate function α)*

$$\alpha(S) = \alpha_c(S) + \alpha_d(S)$$

□

Function $\alpha_c : StatList \rightarrow \mathcal{N}$ depends on the syntactic structure of the process. It gives the total area of all repetitors, sequencers, and concursors. The definition of α_c is straightforward. We denote the area of a handshake component p by a_p.

Definition 3.10 *(α_c)*

Let $g \in Atom$. Then:

$$\begin{array}{ll} \alpha_c(g) & = 0 \\ \alpha_c(S_0; \ldots; S_{n-1}) & = a_{seq(n)} + (\Sigma\, i : 0 \leq i < n : \alpha_c(S_i)) \\ \alpha_c(S_0, \ldots, S_{n-1}) & = a_{conc(n)} + (\Sigma\, i : 0 \leq i < n : \alpha_c(S_i)) \\ \alpha_c(S^n) & = a_{rep_n} + \alpha_c(S) \end{array}$$

□

Function $\alpha_d : (StatList \cup Exp) \rightarrow \mathcal{N}$ is somewhat more complex. It does not depend on the syntactic structure of the process, but uses the multiplicity functions. In order to give a formal definition of α_d we need to examine the number of occurrences and arities of each of the remaining handshake components.

- **Variables**
  The realisation of a CP-0 process S contains a $var(\#_r(x))$ component for each $x \in VarSet(S) \cap Var$, and a $con(\#_r(x))$ for each $x \in VarSet(S) \cap Con$.

- **Multiplexers**
  The realisation of S contains a $mux(\#_!(a))$ for each $a \in ChanSet(S)$, with $\#_!(a) > 1$, and a $mux(\#_w(x))$ for each $x \in VarSet(S)$, with $\#_w(x) > 1$.

- **Demultiplexers**
  The realisation of S contains a $dmx(\#_?(a))$ for each $a \in ChanSet(S), \#_?(a) > 1$ and a $dmx(\#_e(E))$ for each $E \in ExpSet(S)$ of the form $E = E_0 \Box \ldots \Box E_{n-1}$ and $\#_e(E) > 1$.

- **Transferrers**
  The realisation of S contains a trf for each $g \in AtomSet(S)$.

- **Mixers**
  The realisation of S contains a $mix(\#(g))$ for each $g \in AtomSet(S), with \#(g) > 1$.

- **Arithmetic operators**
  For each $E \in ExpSet(S)$ of the form $E = E_0 \ \Box \ \ldots \ \Box \ E_{n-1}$ with, $n \geq 1$, the realisation of CP-0 process S contains a $\Box(n)$ component.

Combining these results we can easily deduce the following definition for α_d:

Definition 3.11 *(α_d)*

$$
\begin{aligned}
\alpha_d(S) \quad = \ & (\Sigma\, c : c \in ChanSet(S) : a_{mux(\#_!(c))} + a_{dmx(\#_?(c))}) \\
& + \\
& (\Sigma\, x : x \in VarSet(S) \cap Var : a_{mux(\#_w(x))} + a_{var(\#_r(E))}) \\
& + \\
& (\Sigma\, x : x \in VarSet(S) \cap Con : a_{con(\#_r(E))}) \\
& + \\
& (\Sigma\, g : g \in AtomSet(S) : a_{mix(\#(g))} + a_{trf}) \\
& + \\
& (\Sigma\, E : E \in ExpSet(S) : \alpha_d(E))
\end{aligned}
$$

where

$$\alpha_d(E) = \begin{cases} 0 & E \in Var \cup Con \\ a_{dmx(\#_e(E))} + \alpha_e(E) & E = E_0 \,\square\, \ldots \,\square\, E_{n-1} \end{cases}$$

and

$$\alpha_e(E) = \begin{cases} 0 & E \in Var \cup Con \\ a_{\square(n)} + (\Sigma i : 0 \leq i < n : \alpha_e(E_i)) & E = E_0 \,\square\, \ldots \,\square\, E_{n-1} \end{cases}$$

□

We assume that $a_{mux(1)} = a_{dmx(1)} = a_{mix(1)} = 0$.

3.3 Speed estimates

The timing analysis for the optimised realisation is based on the syntax of the process. An estimate for the 'speed of a CP-0 process' is given by a function $\tau : (StatList \cup Exp) \rightarrow \mathcal{R}$. Here we mean by the speed of a CP-0 process the time that is spent within the handshake components. So, no delays are included for wires or communication with the environment.

The speed estimates for sequential composition, concurrent composition, and repetition is rather straightforward. They equal the internal switching time of the (control) component, plus a speed estimate for the statements that are activated. Since we assume that the statements in a concurrent composition are executed in parallel, this estimate equals the *maximum* of all the speed estimates of these statements.

The speed estimates for input actions, output actions, and assignment statements depend on the multiplicity functions. If, for example, an atom appears n times, $n > 1$, in a CP-0 process S then the speed estimate for this atom is increased by a delay $t_{mix(n)}$ for an n-ary mixer. A delay t_{trf} is always included for a transferrer. In case of an n-ary write to a single variable or a single channel, a delay $t_{mux(n)}$ is added. Similarly, we add a delay $t_{dmx(n)}$, in case of a n-ary read from a single channel. A delay $t_{read(n)}$ is added in case of an n-ary read from a variable or constant. Similarly, the delay for writing to a n-ary variable is denoted by $t_{write(n)}$. As will be explained in Section 4, variables can be implemented such, that the write delay depends on the number of *read* ports of a variable.

For expressions, we assume that the evaluation of all operands of an n-ary $\square$ expression (e.g. an n-ary sum) start at the same time. A delay $t_{dmx(n)}$ is added for a multiplicity n of each expression. When we denote the delay in component $\square(n)$ by $t_{\square(n)}$, we can now give the definition of speed estimate function τ.

Definition 3.12 *(Speed estimate function τ)*

$$\begin{aligned} \tau(S_0; \ldots; S_{n-1}) &= t_{seq(n)} + (\Sigma i : 0 \leq i < n : \tau(S_i)) \\ \tau(S_0, \ldots, S_{n-1}) &= t_{conc(n)} + (\mathbf{MAX}\, i : 0 \leq i < n : \tau(S_i)) \\ \tau((S)^n) &= t_{rep_n} + n \cdot \tau(S) \end{aligned}$$

$$\tau(x := E) = t_{mux(\#_w(x))} + t_{write(\#_r(x))} + t_{trf} + t_{mix(\#(x:=E))} + \begin{cases} t_{read(\#_r(E))} & E \in Var \cup Con \\ \tau(E) + t_{dmx(\#_e(E))} & E = E_0 \Box \ldots \Box E_{n-1} \end{cases}$$

$$\tau(a!E) = t_{mux(\#_!(a))} + t_{trf} + t_{mix(\#(a!E))} + \begin{cases} t_{read(\#_r(E))} & E \in Var \cup Con \\ \tau(E) + t_{dmx(\#_e(E))} & E = E_0 \Box \ldots \Box E_{n-1} \end{cases}$$

$$\tau(a?x) = t_{dmx(\#_?(a))} + t_{trf} + t_{mux(\#_w(x))} + t_{write((\#_r(x))} + t_{mix(\#(a?x))}$$

Where:

$$\tau(E) = \begin{cases} t_{read(\#_r(E))} & E \in Var \cup Con \\ t_{\Box(n)} + (\mathbf{MAX}\, i : 0 \leq i < n : \tau(E_i)) & E = E_0 \,\Box\, \ldots \,\Box\, E_{n-1} \end{cases}$$

□

3.4 Size and speed of the handshake components

When we apply our performance analysis method to a CP-0 process, this results in two expression: one for the size, and one for the speed of the corresponding handshake circuit. To get actual speed and size estimates, we have to substitute in these expressions the values for the size and speed of the handshake components. Clearly, these values depend on the design and implementation of the components. The components can be designed as single standard cells that consist of transistors (cf. [BK91]). Important factors that influence the size and speed are the design of these cells and the IC-technology that is used for their implementation. We only touch upon the design decisions that we made with respect to our set of handshake components.

In Table 4 the size and speed estimates for our set of handshake components is given. Since the goal of our performance analysis method is to compare different CP-0 processes, we express the size and speed in the abstract measures "F" and "T" respectively.

We decided to implement the sequencer as a binary tree of $seq(2)$ components. The size and speed of this components is estimated as F and T, respectively. The concursor, mixer, (de)multiplexer, and the operator components are implemented similarly, but the size and speed may differ somewhat. Note that the size of the data-manipulation components depend not only on k, but also on the wordlength that is used. Note also that the sequencer the *shape* of the binary tree does not influence the speed, whereas for the other components it does. For these components (i.e., the concursor, mixer, (de)multiplexer, and operator) the speed is proportional to the *depth* of the tree. Since the depth of a binary tree with k leaves ($k > 1$) lies between $\lceil \log k \rceil$ and $k - 1$, the lower and upper bounds on the speed of the components follow. We decided to take the lower bound for the mixer and the (de)multiplexer, while the upper bound was taken for the concursor and the operator.

The estimates for the repetitor, constant, passivator, and transferrer are due to [Mak90] and [Sch91]. Note that the speed and size of a rep_n component is assumed to be independent of n.

Furthermore, a variable component $var(k)$ is implemented such, that delay of a read action is *independent* of k. However, the delay of a write action *does* depend on k. According to [Sch91], a variable is implemented such, that on a write action the written values is

Component	Size	Speed
$seq(k)$	$(k-1)\cdot F$	$(k-1)\cdot T$
$conc(k)$	$(k-1)\cdot F$	$(k-1)\cdot T$
rep_n	F	T
$mix(k)$	$(k-1)\cdot F$	$\lceil \log k \rceil \cdot T$
$mux(k)$	$(k-1)\cdot F/bit$	$\lceil \log k \rceil \cdot T$
$dmx(k)$	$2(k-1)\cdot F/bit$	$\lceil \log k \rceil \cdot T$
$con(k)$	$(1.25+0.75k)\cdot F/bit$	T
$var(k)$	$(1.25+0.75k)\cdot F/bit$	$\begin{cases} t_{read(k)} = T \\ t_{write(k)} = \lceil \log(k+1) \rceil \cdot T \end{cases}$
$pass$	$2\cdot F/bit$	T
$\square(k)$	$3(k-1)\cdot F/bit$	$(k-1)\cdot 8T$
trf	$0F$	T

Table 4: Size and speed of handshake components

distributed to the k read ports. Again this distribution is done by using a tree, which is assumes to be balanced, resulting in a lower bound on the delay.

We do not know whether the values of Table 4 are realistic. Currently, it is not known what the best implementation of the handshake components is, so realistic values cannot be given [Sch91]. However, note that, in order to compare processes, the *ratios* between the values matter, not the values themselves.

4 The comparison of two CP-0 programs

In this section we compare the two different CP-0 programs for dynamic programming of [MS89] by giving size and speed estimates for their processes. Both programs consist of a network of processes that can be specified in the CP-0 language. We call these two programs S_{DP} and S_{DDP}. An example of an actual derivation of a size and speed estimate is given in the appendix. The estimates that we give in this section were not derived by hand, but are determined by a small program that is a straightforward implementation of our performance analysis method of Section 3.

Program S_{DP} consists of processes S_{ij} ($0 \leq i \leq j \leq N$, N the problem size). Three cases are distinguished in [MS89]:

case 1 $0 \leq i = j \leq N$,

case 2 $0 \leq i < j \leq N$, where $(j-i)$ **mod** $2 = 0$, and

case 3 $0 \leq i < j \leq N$, where $(j-i)$ **mod** $2 = 1$.

In this section we restrict ourselves to cases 2 and 3. We denote the corresponding CP-0 processes by $S_{DP'}$ and $S_{DP''}$ respectively.

Program S_{DDP} is meant as an optimisation, and is constructed by combining four neighbouring processes of S_{DP} (two $S_{DP'}$ and two $S_{DP''}$ cells). In order to get a fair comparison, we should take one process of S_{DDP} and compare its performance with the performance of a cluster of four neighbouring S_{DP} processes.

4.1 Size and speed estimates for S_{DP}

Using the multiplicity tables and the size estimate formulae of the previous section, we deduce a size estimate for a cluster of two processes $S_{DP'}$ and two processes $S_{DP''}$. The size of the cluster is the sum of the sizes of the four processes, increased by the size of a number of passivators: one for each output (or input) channel in each of the four processes. Hence, we should add the size of 16 passivators.
For $S_{DP'}$, assuming a 16 bit wordlength, this gives the following size:

$$\alpha(S_{DP'}) = 1203F.$$

For $S_{DP''}$ we obtain the following size:

$$\alpha(S_{DP''}) = 1141F.$$

The size of a cluster of four processes (twice case 2, twice case 3) equals $2 \cdot \alpha(S_{DP'}) + 2 \cdot \alpha(S_{DP''})$ plus the size of 16 passivators: one for each output (or input) channel in each of the four cells. We assume that the size of a passivator is $2F/bit$, so for a 16 bit wordlength we get $32F$ per passivator. Hence, the size of a cluster of four cells is: $2 \cdot 1203F + 2 \cdot 1141F + 16 \cdot 32F = 5200F$.

With respect to the speed of a cluster, we assume that the four processes work in parallel, so the speed is determined by the 'slowest' process. Since $S_{DP'}$ is of the form $S_0; S_1; S_2; S_3; S_4$ and $S_{DP''}$ is of the form $S_0; S_2; S_3; S_4$ (see [MS89]), we conclude that the speed of $S_{DP''}$ is less than the speed of $S_{DP'}$. Therefore, we assume that the speed of the cluster of four processes equals $\tau(S_{DP'})$. For $S_{DP'}$, we obtain the following speed estimate:

$$\tau(S_{DP'}) = (28 \cdot (j - i) + 10)T.$$

In order to compare this speed with the speed of S_{DDP}, we should compute the maximum value for $\tau(S_{DP'})$. It is easy to see that this results in taking $j = N$ and $i = 0$, which gives the following speed estimate for program S_{DP}:

$$\tau(S_{DP}) = (28 \cdot N + 10)T.$$

4.2 Size and speed estimates for S_{DDP}

For an S_{DDP} process $S_{DDP(i,j)}$ $(0 \leq i \leq j \leq \frac{N}{2})$, again assuming a 16 bit wordlength, we get the following size:

$$\alpha(S_{DDP(i,j)}) = 4680F.$$

This size estimate should be increased by the size of four passivators (one for each output channel). Taking these into account, the total size of $S_{DDP(i,j)}$ becomes $4680F + 4 \cdot 32F = 4808F$. With respect to speed, we get the following result for process $S_{DDP(i,j)}$:

$$\tau(S_{DDP(i,j)}) \;=\; (169 \cdot (j - i) + 13)T.$$

If we want to compare this speed with $\tau(S_{DP})$, we should take $j = \frac{N}{2}$ and $i = 0$. This results in:

$$\tau(S_{DDP}) = (84.5 \cdot N + 13)T.$$

Comparing the speed and size estimates for both systolic designs, we conclude that the size of the clustered design is about 8% smaller than the size of the 'fine-grained' design, whereas it is three times slower.

5 Conclusion

We presented a method for performance analysis of processes of CP-0 programs. With this method these processes can be compared by estimating their size and speed. The method was developed such that it could be implemented quite easily. As a test case, the resulting program was applied to the systolic designs for dynamic programming of [MS89].

Our method features the single realisation of atoms and expressions as optimisations. For the second optimisation we have to perform a program transformation that establishes mutual exclusion of expression evaluations. Since this might not be desirable in all situations, the optimisations should be made *optional* when the performance analysis method is incorporated in a VLSI programming environment.

We have tried to make our method independent from the implementation of the handshake components. If the implementation of a component changes, this should only affect the size and speed estimates of the component, and not the entire method. However, this independence has its limitations. If, for instance, variables are implemented with only one read channel (as is suggested in [Pee90b]), this would require a change in the translation method, and would therefore also require a change in the performance analysis method.

We suppose that our method can be extended with a number of language constructs, like selection ("if-then") and iteration ("while-do"). This extension is a topic for future research. Another extension could be the addition of more (post-)optimisations, e.g. common subexpression elimination. However, we conjecture that the introduction of more optimisations will lead to more program transformations, something we consider undesirable. A way to avoid these transformations is to introduce new handshake components (e.g., the *fork* component of [Pee90a]). A disadvantage of this approach is that it might make our performance analysis rather complex.

Acknowledgements

We would like to thank Martin Rem and Huub Schols for introducing us to the subject. Ad Peeters is acknowledged for his suggestions and stimulating discussions on earlier versions of this paper. Finally, we are specially grateful to Alex Jansen, who implemented our method.

References

[NB88] C. Niessen, C.H. (Kees) van Berkel, M. Rem, and R.W.J.J. Saeijs. *VLSI Programming and Silicon Compilation; A Novel Approach from Philips Research*, Proceedings ICCD '88, 1988.

[BS88] C.H. (Kees) van Berkel, and R.W.J.J. Saeijs. *Compilation of Communicating Processes into Delay-Insensitive Circuits*, Proceedings ICCD '88, 1988.

[BK91] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. *The VLSI-programming language Tangram and its translation into handshake circuits*, submitted to EDAC '91.

[Kam90] M. Kamps. *Translation of Basic Components into Gate Circuits*, In: Designing Delay-Insensitive Circuits, ISBN 90-5282-076-7.

[Mar86] Alain J. Martin. *Compiling communicating processes into delay-insensitive circuits*, Distributed Computing, 1, pp. 247-260, 1986.

[MS89] R.H. Mak and P. Struik. *A Systolic Design for Dynamic Programming*, In: Proceedings CSN 89, eds. P.M.G. Apers, D. Bosman, J. van Leeuwen, pp. 355-375, 1989.

[Mak90] R.H. Mak. *Private communication.*

[Pee90a] A. Peeters. *A Notation for Systolic Computations* In: Designing Delay-Insensitive Circuits, ISBN 90-5282-076-7.

[Pee90b] A. Peeters. *Syntax-Directed Translation of the Regular Language Acceptors*, Ibidem.

[Sch91] F. Schalij. *Private communication.*

A An example of a derivation

In this appendix we consider CP-0 program S_{DP} of [Mak89]. For the example derivation, we restrict ourselves to case 2, so process $S_{DP'}$. The statements of this process are given

below:

$$S_{DP'} = S0; S1; S2; S3; S4$$

where:

$$\begin{array}{lcl}
S0 & = & \mathrm{a?ya, b?xb, d?yd, e?xe} \\
S1 & = & \mathrm{a!xb, b!xb, d!xe, e!xe} \\
S2 & = & \mathrm{m := (w + ya + xe)\ \mathbf{min}\ (w + xb + yd)} \\
S3 & = & \mathrm{(a?xa, b?xb, d?xd, e?xe} \\
 & & \mathrm{; a!ya, b!xb, d!yd, e!xe} \\
 & & \mathrm{; (m0 := m; m := m0\ \mathbf{min}\ (w + xa + xe)\ \mathbf{min}\ (w + xb + xd)), ya := xa, yd := xd} \\
 & &)^{\frac{1}{2}(j-i)-1} \\
S4 & = & \mathrm{a!ya, b!m, d!yd, e!m}
\end{array}$$

In order to apply our method, we need to determine sets $ChanSet$, $VarSet$, $AtomSet$ and $ExpSet$, and the multiplicity functions for this process. We do this by constructing three tables:

- A table in which for each variable $x \in VarSet(S_{DP'})$, the values $\#_r(x)$ and $\#_w(x)$ are given. Set $VarSet(S_{DP'})$ is the set of all elements in the left column of the table. The values $\#_r(x)$ for $x \in ExpSet(S_{DP'}) \cap Con$ are also given.
- A table in which for each channel $a \in ChanSet(S_{DP'})$, the values $\#_?(a)$ and $\#_!(a)$ are given. Set $ChanSet(S_{DP'})$ is the set of all elements in the left column of the table.
- A table in which for each atom of $g \in AtomSet(S_{DP'})$, the value $\#(g)$ is given. Set $ExpSet(S_{DP'})$ is the set of right-hand sides of assignment statements in the left column of the table; $AtomSet(S_{DP'})$ is the set of all elements in the left column.

Below we give these three tables for the CP-0 process $S_{DP'}$. All multiplicities are given, except the expression multiplicities. It is not difficult to see that no expression occurs more than once, so these multiplicities are omitted.

x	$\#_?$(x)	$\#_!$(x)
a	2	2
b	1	2
d	2	2
e	1	2

Table 5: Channels and their multiplicities

Using these tables and the size estimate formulae, we derive a size estimate for $S_{DP'}$. This derivation is given below, with some hints.

x	$\#_w$(x)	$\#_r$(x)
m	2	3
m0	1	1
xa	1	2
xb	1	4
xd	1	2
xe	1	4
ya	2	2
yd	2	2
w	-	4

Table 6: Variables and their write and read multiplicities

x		#(x)
a?	ya	1
	xa	1
b?	xb	2
d?	yd	1
	xd	1
e?	xe	2
a!	xb	1
	ya	2
b!	xb	2
	m	1
d!	xe	1
	yd	2
e!	xe	2
	m	1
m:=	(w + ya + xe) **min** (w + xb + yd)	1
	m0 **min** (w + xa + xe) **min** (w + xb + xd)	1
m0:=	m	1
ya:=	xa	1
yd:=	xd	1

Table 7: Atoms, expressions, and atom multiplicities

$$
\begin{aligned}
\alpha(S_{DP'}) &= \{ \text{ Definition 3.9 } \} \\
&\quad \alpha_c(S_{DP'}) + \alpha_d(S_{DP'}) \\
\alpha_c(S_{DP'}) &= \alpha_c(S_0; S_1; S_2; S_3; S_4) \\
&= \{ \text{ Definition 3.10 } \} \\
&\quad a_{seq(5)} + (\Sigma\, i : 0 \leq i < 5 : \alpha_c(S_i)) \\
\alpha_c(S_0) &= \alpha_c(\text{a?ya, b?xb, d?yd, e?xe}) \\
&= \{ \text{ Definition 3.10 } \} \\
&\quad a_{conc(4)} + \alpha_c(\text{a?ya}) + \alpha_c(\text{b?xb}) + \alpha_c(\text{d?yd}) + \alpha_c(\text{e?xe}) \\
&= \{ \text{ Definition 3.10 } \} \\
&\quad a_{conc(4)} + 0 + 0 + 0 + 0 \\
&= a_{conc(4)} \\
\alpha_c(S_1) &= \alpha_c(\text{a!xb, b!xb, d!xe, e!xe}) \\
&= \{ \text{ Definition 3.10 } \} \\
&\quad a_{conc(4)} \\
\alpha_c(S_2) &= \alpha_c(\text{m} := (\text{w} + \text{ya} + \text{xe}) \textbf{ min } (\text{w} + \text{xb} + \text{yd})) \\
&= \{ \text{ Definition 3.10 } \} \\
&\quad 0 \\
\alpha_c(S_3) &= \alpha_c((S_{30}; S_{31}; S_{32})^{\frac{1}{2}(j-i)-1}) \\
&= \{ \text{ Definition 3.10 } \} \\
&\quad a_{rep_{\frac{1}{2}(j-i)-1}} + \alpha_c(S_{30}; S_{31}; S_{32}) \\
&= \{ \text{ Definition 3.10 } \} \\
&\quad a_{rep_{\frac{1}{2}(j-i)-1}} + a_{seq(3)} + \alpha_c(S_{30}) + \alpha_c(S_{31}) + \alpha_c(S_{32}) \\
\alpha_c(S_4) &= \alpha_c(\text{a!ya, b!m, d!yd, e!m}) \\
&= a_{conc(4)} \\
\alpha_c(S_{30}) &= \alpha_c(\text{a?xa, b?xb, d?xd, e?xe}) \\
&= a_{conc(4)} \\
\alpha_c(S_{31}) &= \alpha_c(\text{a!ya, b!xb, d!yd, e!xe}) \\
&= a_{conc(4)} \\
\alpha_c(S_{32}) &= \alpha_c(S_{320}, S_{321}, S_{322}) \\
\alpha_c(S_{320}) &= \alpha_c(\text{m0} := \text{m; m} := \text{m0 } \textbf{min } (\text{w} + \text{xa} + \text{xe}) \textbf{ min } (\text{w} + \text{xb} + \text{xd})) \\
&= a_{seq(2)} + \alpha_c(\text{m0} := \text{m}) + \alpha_c(\text{m} := \text{m0 } \textbf{min } (\text{w} + \text{xa} + \text{xe}) \textbf{ min } (\text{w} + \text{xb} + \text{xd})) \\
&= a_{seq(2)} + 0 + 0 \\
&= a_{seq(2)} \\
\alpha_c(S_{321}) &= \alpha_c(\text{ya} := \text{xa}) \\
&= 0 \\
\alpha_c(S_{322}) &= \alpha_c(\text{yd} := \text{xd}) \\
&= 0 \\
\alpha_d(S_{DP'}) &= \{ \text{ Definition 3.11 } \}
\end{aligned}
$$

$$
\begin{aligned}
& (\Sigma\, c : c \in ChanSet(S_{DP'}) : a_{mux(\#_!(c))} + a_{dmx(\#_?(c))}) + \\
& (\Sigma\, v : v \in VarSet(S_{DP'}) : a_{mux(\#_w(v))} + a_{var(\#_r(v))}) + \\
& (\Sigma\, b : b \in AtomSet(S_{DP'}) : a_{mix(\#(b))} + a_{trf}) + \\
& (\Sigma\, e : e \in ExpSet(S_{DP'}) : \alpha(e)) \\
= \; & a_{min(3)} + a_{min(2)} + 4a_{plus(3)} + \\
& 6a_{mix(2)} + 19a_{trf} + \\
& 7a_{mux(2)} + 2a_{dmx(2)} + \\
& 2a_{var(4)} + a_{con(4)} + a_{var(3)} + 4a_{var(2)} + a_{var(1)}
\end{aligned}
$$

For process $S_{DP'}$, assuming a 16 bit wordlength, we get the following size:

$$\alpha(S_{DP'}) \;=\; 1203F$$

Using the same tables and the speed estimate formulae, we deduce a speed estimate for $S_{DP'}$.

$$
\begin{aligned}
\tau(S_{DP'}) &= \tau(S_0; S_1; S_2; S_3; S_4) \\
&= \{ \text{ Definition 3.12 } \} \\
&\quad t_{seq(5)} + (\Sigma\, i : 0 \le i < 5 : \tau(S_i)) \\
\tau(S_0) &= \tau(\text{a?ya, b?xb, d?yd, e?xe}) \\
&= \{ \text{ Definition 3.12 } \} \\
&\quad t_{conc(4)} + max\{\tau(\text{a?ya}), \tau(\text{b?xb}), \tau(\text{d?yd}), \tau(\text{e?xe})\} \\
&= t_{conc(4)} + max\ \{t_{trf} + t_{write(2)} + t_{dmx(2)} + t_{mux(2)} + t_{mix(1)} \\
&\qquad\qquad , t_{trf} + t_{write(4)} + t_{dmx(1)} + t_{mux(1)} + t_{mix(2)} \\
&\qquad\qquad , t_{trf} + t_{write(2)} + t_{dmx(2)} + t_{mux(2)} + t_{mix(1)} \\
&\qquad\qquad , t_{trf} + t_{write(4)} + t_{dmx(1)} + t_{mux(1)} + t_{mix(2)}\} \\
&= t_{conc(4)} + t_{trf} + max\ \{t_{write(2)} + t_{dmx(2)} + t_{mux(2)} \\
&\qquad\qquad , t_{write(4)} + t_{dmx(1)} + t_{mux(1)} + t_{mix(2)}\} \\
\tau(S_1) &= \tau(\text{a!xb, b!xb, d!xe, e!xe}) \\
&= \{ \text{ Definition 3.12 } \} \\
&\quad t_{conc(4)} + max\{\tau(\text{a!xb}), \tau(\text{b!xb}), \tau(\text{d!xe}), \tau(\text{e!xe})\} \\
&= t_{conc(4)} + max\ \{t_{trf} + t_{read(4)} + t_{mux(2)} + t_{mix(1)} \\
&\qquad\qquad , t_{trf} + t_{read(4)} + t_{mux(2)} + t_{mix(2)} \\
&\qquad\qquad , t_{trf} + t_{read(4)} + t_{mux(2)} + t_{mix(1)} \\
&\qquad\qquad , t_{trf} + t_{read(4)} + t_{mux(2)} + t_{mix(2)}\} \\
&= t_{conc(4)} + t_{trf} + t_{mux(2)} + max\ \{t_{read(4)} + t_{mix(1)} \\
&\qquad\qquad\qquad\qquad , t_{read(4)} + t_{mix(2)} \\
&\qquad\qquad\qquad\qquad , t_{read(4)} + t_{mix(1)} \\
&\qquad\qquad\qquad\qquad , t_{read(4)} + t_{mix(2)}\} \\
&= t_{conc(4)} + t_{trf} + t_{read(4)} + t_{mux(2)} + t_{mix(2)}
\end{aligned}
$$

$$
\begin{aligned}
\tau(S_2) &= \tau(\text{m} := (\text{w} + \text{ya} + \text{xe}) \textbf{ min } (\text{w} + \text{xb} + \text{yd})) \\
&= \{ \text{ Definition 3.12 } \} \\
&\quad \tau((\text{w} + \text{ya} + \text{xe}) \textbf{ min } (\text{w} + \text{xb} + \text{yd})) + t_{trf} + t_{write(3)} + t_{mux(2)} + t_{mix(1)} \\
&= \{ \text{ Definition 3.12 } \} \\
&\quad t_{trf} + t_{write(3)} + t_{mux(2)} + t_{min(2)} + max \ \{\tau(\text{w} + \text{ya} + \text{xe}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , \tau(\text{w} + \text{xb} + \text{yd})\} \\
&= \{ \text{ Definition 3.12 } \} \\
&\quad t_{trf} + t_{write(3)} + t_{mux(2)} + t_{min(2)} + max \ \{t_{plus(3)} + max\{\tau(\text{w}), \tau(\text{ya}), \tau(\text{xe})\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , t_{plus(3)} + max\{\tau(\text{w}), \tau(\text{xb}), \tau(\text{yd})\}\} \\
&= t_{trf} + t_{write(3)} + t_{mux(2)} + t_{min(2)} + max \ \{t_{plus(3)} + max\{t_{read(4)}, t_{read(2)}, t_{read(4)}\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , t_{plus(3)} + max\{t_{read(4)}, t_{read(4)}, t_{read(2)}\}\} \\
&= t_{trf} + t_{write(3)} + t_{mux(2)} + t_{min(2)} + t_{plus(3)} + t_{read(4)} \\
\tau(S_3) &= \tau((S_{30}; S_{31}; S_{32})^{\frac{1}{2}(j-i)-1} \\
&= \{ \text{ Definition 3.12 } \} \\
&\quad t_{rep_{\frac{1}{2}(j-i)-1}} + (\tfrac{1}{2}(j-i) - 1) \cdot (t_{seq(3)} + \tau(S_{30}) + \tau(S_{31}) + \tau(S_{32})) \\
\tau(S_4) &= \tau(\text{a!ya, b!m, d!yd, e!m}) \\
&= \{ \text{ Definition 3.12 } \} \\
&\quad t_{conc(4)} + max\{\tau(\text{a!ya}), \tau(\text{b!m}), \tau(\text{d!yd}), \tau(\text{e!m})\} \\
&= t_{conc(4)} + max \ \{t_{trf} + t_{read(2)} + t_{mux(2)} + t_{mix(2)} \\
&\qquad\qquad , t_{trf} + t_{read(3)} + t_{mux(2)} + t_{mix(1)} \\
&\qquad\qquad , t_{trf} + t_{read(2)} + t_{mux(2)} + t_{mix(2)} \\
&\qquad\qquad , t_{trf} + t_{read(3)} + t_{mux(2)} + t_{mix(1)}\} \\
&= t_{conc(4)} + t_{trf} + t_{mux(2)} + max \ \{t_{read(2)} + t_{mix(2)} \\
&\qquad\qquad\qquad\qquad\qquad\qquad , t_{read(3)}\} \\
\tau(S_{30}) &= \tau(\text{a?xa, b?xb, d?xd, e?xe}) \\
&= \{ \text{ Definition 3.12 } \} \\
&\quad t_{conc(4)} + max\{\tau(\text{a?xa}), \tau(\text{b?xb}), \tau(\text{d?xd}), \tau(\text{e?xe})\} \\
&= t_{conc(4)} + max \ \{t_{trf} + t_{dmx(2)} + t_{mux(1)} + t_{mix(1)} + t_{write(2)} \\
&\qquad\qquad , t_{trf} + t_{dmx(1)} + t_{mux(1)} + t_{mix(2)} + t_{write(4)} \\
&\qquad\qquad , t_{trf} + t_{dmx(2)} + t_{mux(1)} + t_{mix(1)} + t_{write(2)} \\
&\qquad\qquad , t_{trf} + t_{dmx(1)} + t_{mux(1)} + t_{mix(2)}\} + t_{write(4)} \\
&= t_{conc(4)} + t_{trf} + t_{mux(1)} + max \ \{t_{dmx(2)} + t_{write(2)} \\
&\qquad\qquad\qquad\qquad\qquad\qquad , \ t_{dmx(1)} + t_{mix(2)} + t_{write(4)}\} \\
\tau(S_{31}) &= \tau(\text{a!ya, b!xb, d!yd, e!xe}) \\
&= \{ \text{ Definition 3.12 } \} \\
&\quad t_{conc(4)} + max\{\tau(\text{a!ya}), \tau(\text{b!xb}), \tau(\text{d!yd}), \tau(\text{e!xe})\} \\
&= t_{conc(4)} + max \ \{t_{trf} + t_{read(2)} + t_{mux(2)} + t_{mix(2)} \\
&\qquad\qquad , t_{trf} + t_{read(4)} + t_{mux(2)} + t_{mix(2)} \\
&\qquad\qquad , t_{trf} + t_{read(2)} + t_{mux(2)} + t_{mix(2)} \\
&\qquad\qquad , t_{trf} + t_{read(4)} + t_{mux(2)} + t_{mix(2)}\} \\
&= t_{conc(4)} + t_{trf} + t_{read(4)} + t_{mux(2)} + t_{mix(2)}
\end{aligned}
$$

$$
\begin{aligned}
\tau(S_{32}) &= \tau(S_{320}, S_{321}, S_{322}) \\
&= \{ \text{ Definition 3.12 } \} \\
&\quad t_{conc(3)} + max\{\tau(S_{320}), \tau(S_{321}), \tau(S_{322})\} \\
\tau(S_{320}) &= \tau(\text{m0} := \text{m; m} := \text{m0 } \textbf{min } (\text{w} + \text{xa} + \text{xe}) \textbf{ min } (\text{w} + \text{xb} + \text{xd})) \\
&= t_{seq(2)} + \tau(\text{m0} := \text{m}) + \tau(\text{m} := \text{m0 } \textbf{min } (\text{w} + \text{xa} + \text{xe}) \textbf{ min } (\text{w} + \text{xb} + \text{xd})) \\
&= t_{seq(2)} + \tau(\text{m}) + t_{trf} + t_{write(1)} + t_{mux(1)} + t_{mix(1)} + \\
&\quad \tau(\text{m} := \text{m0 } \textbf{min } (\text{w} + \text{xa} + \text{xe}) \textbf{ min } (\text{w} + \text{xb} + \text{xd})) \\
&= t_{seq(2)} + t_{read(3)} + t_{trf} + t_{write(1)} + t_{mux(1)} + \\
&\quad \tau(\text{m} := \text{m0 } \textbf{min } (\text{w} + \text{xa} + \text{xe}) \textbf{ min } (\text{w} + \text{xb} + \text{xd})) \\
&= t_{seq(2)} + t_{read(3)} + t_{trf} + t_{write(1)} + t_{mux(1)} + \\
&\quad \tau(\text{m0 } \textbf{min } (\text{w} + \text{xa} + \text{xe}) \textbf{ min } (\text{w} + \text{xb} + \text{xd})) + t_{trf} + t_{write(3)} + t_{mux(2)} + t_{mix(1)} \\
&= t_{seq(2)} + t_{read(3)} + 2t_{trf} + t_{write(1)} + t_{mux(1)} + \\
&\quad t_{mux(2)} + t_{min(3)} + t_{write(3)} + max\ \{\tau(\text{m0}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad , \tau(\text{w} + \text{xa} + \text{xe}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad , \tau(\text{w} + \text{xb} + \text{xd})\} \\
&= t_{seq(2)} + t_{read(3)} + 2t_{trf} + t_{write(1)} + t_{mux(1)} + \\
&\quad t_{mux(2)} + t_{min(3)} + t_{write(3)} + max\ \{\tau(\text{m0}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad , t_{plus(3)} + max\{\tau(\text{w}), \tau(\text{ya}), \tau(\text{xe})\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad , t_{plus(3)} + max\{\tau(\text{w}), \tau(\text{xb}), \tau(\text{yd})\}\} \\
&= t_{seq(2)} + t_{read(3)} + 2t_{trf} + t_{write(1)} + t_{mux(1)} + \\
&\quad t_{mux(2)} + t_{min(3)} + t_{write(3)} + max\ \{t_{read(1)} \\
&\qquad\qquad\qquad\qquad\qquad\qquad , t_{plus(3)} + max\{t_{read(4)}, t_{read(2)}, t_{read(4)}\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad , t_{plus(3)} + max\{t_{read(4)}, t_{read(4)}, t_{read(2)}\}\} \\
&= t_{seq(2)} + 2t_{trf} + t_{min(3)} + t_{plus(3)} + t_{read(4)} + t_{read(3)} + t_{mux(2)} + t_{mux(1)} \\
\tau(S_{321}) &= \tau(\text{ya} := \text{xa}) \\
&= \tau(\text{xa}) + t_{trf} + t_{mux(2)} + t_{mix(1)} \\
&= t_{write(2)} + t_{read(2)} + t_{trf} + t_{mux(2)} \\
\tau(S_{322}) &= \tau(\text{yd} := \text{xd}) \\
&= \tau(\text{xd}) + t_{trf} + t_{mux(2)} + t_{mix(1)} \\
&= t_{write(2)} + t_{read(2)} + t_{trf} + t_{mux(2)}
\end{aligned}
$$

For $S_{DP'}$ process (i, j), this gives the following speed estimate:

$$\tau(S_{DP'}) \;=\; (28 \cdot (j - i) + 10)T$$

In this series appeared :

No.	Author(s)	Title
85/01	R.H. Mak	The formal specification and derivation of CMOS-circuits.
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes.
85/03	W.J.M. Lemmens	Use of a computer for evaluation of flow films.
85/04	T. Verhoeff H.M.L.J.Schols	Delay insensitive directed trace structures satisfy the foam the foam rubber wrapper postulate.
86/01	R. Koymans	Specifying message passing and real-time systems.
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A language for formal specification of information systems.
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures.
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several systems.
86/05	J.L.G. Dietz K.M. van Hee	A framework for the conceptual modeling of discrete dynamic systems.
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP.
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers.
86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arun Kumar	Compositional semantics for real-time distributed computing (Inf.&Control 1987).
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language.
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing.
86/11	W.P. de Roever	Questions to Robin Milner - A responder's commentary (IFIP86).
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes.
86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4).

86/14	R. Koymans	Specifying passing systems requires extending temporal logic.
87/01	R. Gerth	On the existence of sound and complete axiomati zations of the monitor concept.
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases.
87/03	G.J. Houben J.Paredaens	A formal approach to distributed information systems.
87/04	T.Verhoeff	Delay-insensitive codes - An overview.
87/05	R.Kuiper	Enforcing non-determinism via linear time temporal logic specification.
87/06	R.Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch).
87/07	R.Koymans	Specifying message passing and real-time systems with real-time temporal logic.
87/08	H.M.J.L. Schols	The maximum number of states after projection.
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P. van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics.
87/10	T.Verhoeff	Three families of maximally nondeterministic automata.
87/11	P.Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.
87/12	K.M. van Hee and A.Lapinski	OR and AI approaches to decision support systems.
87/13	J.C.S.P. van der Woude	Playing with patterns - searching for strings.
87/14	J. Hooman	A compositional proof system for an occam-like real-time language.
87/15	C. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts.
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas.
87/17	K.M. van Hee G.-J.Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples.

87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces.
87/19	A.J.Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen.
87/20	G.J. Houben J. Paredaens	The R^2 -Algebra: An extension of an algebra for nested relations.
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG.
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence.
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems.
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples.
88/04	G.J. Houben J.Paredaens D.Tahon	The Nested Relational Algebra: A Tool to Handle Structured Information.
88/05	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specifications for Information Systems.
88/06	H.M.J.L. Schols	Notes on Delay-Insensitive Communication.
88/07	C. Huizing R. Gerth W.P. de Roever	Modelling Statecharts behaviour in a fully abstract way.
88/08	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	A Formal model for System Specification.
88/09	A.T.M. Aerts K.M. van Hee	A Tutorial for Data Modelling.
88/10	J.C. Ebergen	A Formal Approach to Designing Delay Insensitive Circuits.
88/11	G.J. Houben J.Paredaens	A graphical interface formalism: specifying nested relational databases.
88/12	A.E. Eiben	Abstract theory of planning.
88/13	A. Bijlsma	A unified approach to sequences, bags, and trees.
88/14	H.M.M. ten Eikelder R.H. Mak	Language theory of a lambda-calculus with recursive types.

88/15	R. Bos C. Hemerik	An introduction to the category theoretic solution of recursive domain equations.
88/16	C.Hemerik J.P.Katoen	Bottom-up tree acceptors.
88/17	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable specifications for discrete event systems.
88/18	K.M. van Hee P.M.P. Rambags	Discrete event systems: concepts and basic results.
88/19	D.K. Hammer K.M. van Hee	Fasering en documentatie in software engineering.
88/20	K.M. van Hee L. Somers M.Voorhoeve	EXSPECT, the functional part.
89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a paralell program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.

89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.
90/1	W.P.de Roever-H.Barringer C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, pp. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.

90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences, p. 29.
90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses of "if..., then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.