7. The Multi-Architecture Datapath Synthesis System

For the design space exploration process, the target applications have to be mapped onto the current architecture in order to get information about its suitability. A historic framework for the KressArray-I architecture was the Datapath Synthesis System (DPSS) [HaKr95] has been employed for this task. However, as the DPSS cannot be used to map applications on architectures other than the KressArray-I, a new Multi Architecture Datapath Synthesis System (MA-DPSS) [HHHN99a], [HHHN00a] has been developed.

MA-DPSS Features. The MA-DPSS is a redesign of the KressArray-I DPSS and aiming to preserve compatibility to this framework. So, the MA-DPSS can also be integrated into the CoDe-X environment for Xputer machines [Beck97]. In contrast to the DPSS, the MA-DPSS allows to map applications on all architectures of the KressArray design space (see section 6). While the DPSS is restricted to a fixed architecture with one unidirectional connection to the south and the east, the MA-DPSS can handle different numbers and types of nearest neighbor connections, which may each feature an individual torus structure (see section 6.3.2). Also, row and column backbuses are supported as a new type of communication resource for KressArrays, including according configuration data as described in sectio n6.3.3.

Another improvement of the MA-DPSS consists in the support of different types of memory I/O architectures. In addition to the historic I/O over the global bus, data input and output over edge ports as described in section 6.4.3 are supported by the system, including the consideration of restrictions regarding the location of these ports. Such restrictions may e.g. require an edge port to be located at a certain range of an array side, due to the general architecture embedding.

As a novel feature, the synthesis system supports also highly heterogeneous architectures as discussed in section 6.4.2. Such architectures allow the embedding of other resources in the reconfigurable array, like e.g. memory blocks. An existing architecture showing a heterogeneous structure is the CHESS array (see section 2.1.9) with alternating processing elements and switchboxes. However, the MA-DPSS allows a much more flexible definition of special cells, as the operator capabilities can be defined for each individual rDPU and operator.

In order to be suitable for the task of design space exploration, the MA-DPSS uses a special intermediate file format (see section A.1), with incremental information about the design process. This format allows e.g. the remapping of an application

by using the output directly as input again. Another part of the exploration process is realized by the architecture estimator, which resembles a simple analytical component in the process, which is otherwise mostly interactive.

Finally, the MA-DPSS supports virtual I/O ports inside the array and the fixing of rDPU positions. The fixing, which is described in detail below in section 7.3.4 allows to mark certain rDPU positions so they are not occupied by other operators in the synthesis process. This feature together with I/O ports in the array allows the implementation of a library concept.

MA-DPSS Overview. The MA-DPSS consists of four stages, which are the ALE-X compiler, the architecture estimator, the multi-architecture mapper, and the scheduler. The intermediate file format (section A.1) used by the framework features three flavors according to the stages in the synthesis process. These flavors are called α-, β-, and γ-intermediate format.

The application to be synthesized can be specified in the high-level language ALE-X (a C dialect) [HaKr95], [Kres96] or directly in α-intermediate format. An ALE-X specification is translated to α-format by the ALE-X compiler, which requires a hardware file containing the initial operator set for the technology mapping. While the ALE-X language provides a high level programming interface and compatibility to the Xputer programming environment CoDe-X [Beck97], the intermediate format allows fine-tuning of datapaths for the experienced user, providing access to all features of the synthesis framework. The α-format file contains the datapath of the application using a dataflow-graph representation. This datapath is then examined by the architecture estimator, which adds information for an initial KressArray architecture to the α-format, generating a file in β-format. This initial architecture is subject to the exploration process. In the placement and routing phase, which is performed by the MA-DPSS mapper, a mapping is generated and added to the intermediate format, which results in a file in γ-format. The mapping comprises the assignment of operators to physical rDPUs of the array as well as the assignment of data dependencies of the dataflow graph to physical communication resources of the KressArray. In a final step, the γ-format file is processed by the scheduler to generate data schedules for the global bus as well as statistic data for the exploration process, including performance estimations and critical path length. An overview on the synthesis process in the MA-DPSS is given in figure 7-1. The stages of the MA-DPSS will be described in more detail in the following sections.

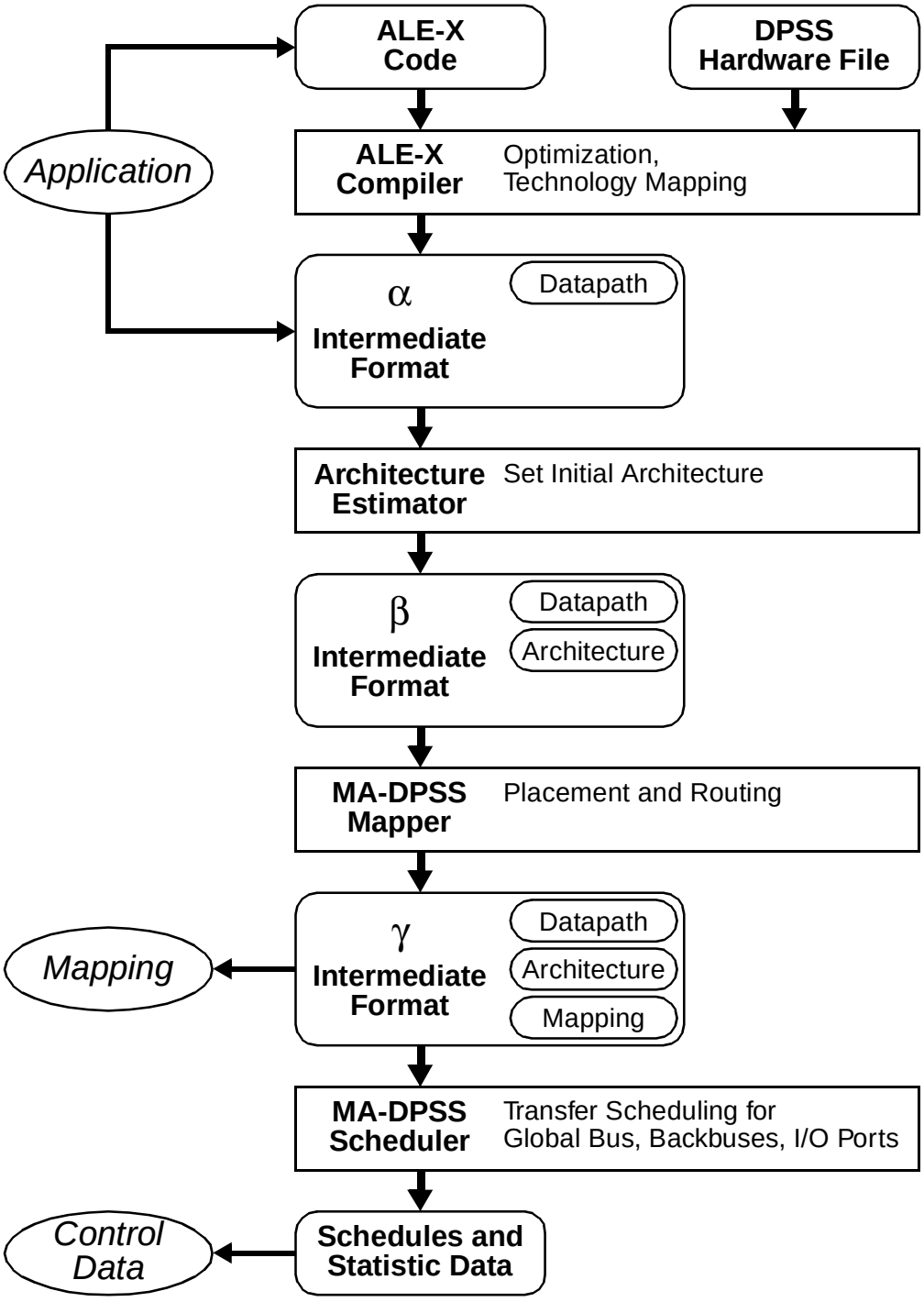

Figure 7-1: Overview on the MA-DPSS synthesis process

7.1 The ALE-X Compiler

The ALE-X (Arithmetic and Logic Expressions for Xputers) language [Kres96] allows the description of a complex datapath consisting of one or more data flows which are also referred to as subnets. The ALE-X compiler extracts a dataflow graph representation from such a description and generates an according representation in a file in α-intermediate format. The compiler uses also a hardware file describing the initial operator set. As the ALE-X compiler of the MA-DPSS resembles closely the one of the original DPSS described in [HaKr95], [Kres96], the files and the compiler itself are only briefly reviewed in this section.

7.1.1 The ALE-X Language

An ALE-X description contains the datapath specification itself in a syntax similar to that of the C programming language and additional information concerning the position of data in consecutive data sets. The latter part of the description is needed for compatibility to the Xputer design environment and contains the position of the data words of one data set in the memory and the change of these positions for the next data set. This specification follows the scanwindow model [HHRS91] of the Xputer-based MoM architectures [Rein99], assuming a two dimensional data memory as well as an offset vector describing the position change of data set positions.

This information is passed through to the scheduler (see below in section 7.4), which identifies corresponding memory positions of consecutive data words to optimize data memory accesses. In case this optimization is not needed, dummy values may be set for the position and change specification. The datapath description itself may contain constants and local variables. The prevailing statements are typically assignments, but also if-else statements, while loops, and do-while loops are allowed. An example for an ALE-X specification is shown in figure 7-2. The according mapping of this specification onto a five by five KressArray is pictured in figure 7-3. The mapping features nine input ports and one output port, the latter situated at the upper right corner.

The example shows the datapath description for an image processing filter. Nine pixels are processed at once, resulting in a new value for the center pixel. The nine pixel values represent the input data for the datapath. If a MoM-architecture is assumed, these values are arranged in the two-dimensional data memory like illustrated in the figure, resembling the scanwindow for this application. The variable declarations in the scanwindow description contain the corresponding data positions inside this scanwindow. In the following iteration, the next set of data is located one step right of the current one, which is described by the

```
rALUsubnet SubNet_0 (IMAGE)      // Subnet header

ScanWindow IMAGE                 // Scanwindow description
{     int i00 at [0][0] ;
      int i01 at [0][1] ;
      int i02 at [0][2] ;
      int i10 at [1][0] ;
      int i11 at [1][1] ;
      int i12 at [1][2] ;
      int i20 at [2][0] ;
      int i21 at [2][1] ;
      int i22 at [2][2] ;
      HandleOffset [0][1] ;
} ;

{     int hsum, vsum, csum ;     // Datapath description
      int lrsum, tbsum ;
      csum = i00 + i02 + i20 + i22 ;
      tbsum = i01 + i21 ;
      lrsum = i10 + i12 ;
      hsum = 2 * i11 + 2 * lrsum - ( csum + tbsum ) ;
      vsum = 2 * i11 + 2 * tbsum - ( csum + lrsum ) ;
      if ( hsum < 100 || vsum < 100 )
      {     i11 = 0 ;
      }
      else
      {     i11 = 255 ;
      }
}
```

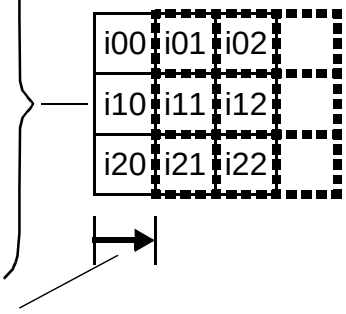

Figure 7-2: ALE-X language example showing an image processing filter. The according mapping is depicted in figure 7-3

HandleOffset statement. The datapath description contains five local variables for intermediate sums of pixel values. The final output value *i11* is then calculated in an if-clause.

7.1.2 The ALE-X Compiler Hardware File

The hardware file contains information about the current KressArray architecture. This file has been adopted from the historic DPSS, although most data is overridden in the later exploration and synthesis process or not used at all.

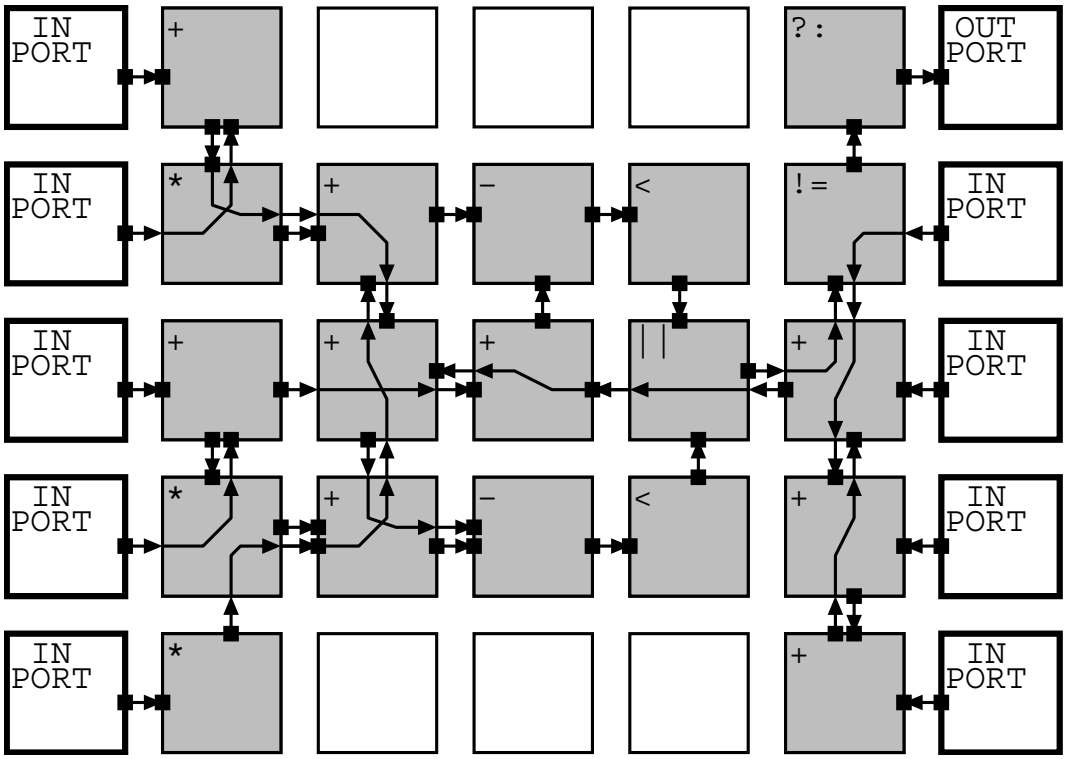

Figure 7-3: Mapping derived from the ALE-X specification shown in figure 7-2 onto 5 by 5 KressArray

However, the ALE-X compiler relies on this file to perform the technology mapping step. A detailed description of the hardware file is given in [Kres96]. In this section, a brief review and an identification of the used parts is provided. The file contains the following types of information:

- Array properties like size and torus structure.
  This data is not used, as it is subject to the exploration process. An initial array description is produced by the architecture estimator.
- Properties of the KressArray controller, like register file size and delay times of the controller commands.
  Although a KressArray-I-style architecture using a controller is no more mandatory, this information is forwarded to the scheduler in order to enable this type of implementation. A detailed description of these parameters can be found in [Molk95].
- Delay times for data transfers over communication resources.
  This information has been extended to match the flexibility of the MA-DPSS. Delay times are specified for each type of communication resource. The delay of a resource is modeled differently for the three resource types global bus, backbuses, and nearest neighbor connections.

- Available data types, operators, and user-defined functions.
  This is the most important part of the hardware file, which is used for the technology mapping step of the ALE-X compiler. The supported data types and the available operators are listed in this part of the file and resemble the initial operator set for the exploration process.

The specification of the data types, the available operators and functions, and the delay times are the only information used in the MA-DPSS, with the data type and operator description being the part with the most relevance for the ALE-X compiler. The delay times are forwarded to the scheduler tool described in section 7.4. The relevant parts of the hardware file will be discussed in more detail in the following.

7.1.2.1 Data Types

The operators of the KressArray may work on different data types. In order to allow the ALE-X compiler to manage these types, they have to be specified in the hardware file, using the *TYPE* statement. An example data type specification is shown in figure 7-4.

```
TYPE    char, short, unsigned int, int, float:17 ;
```

Figure 7-4: Example data type specification in the ALE-X compiler hardware file

The example specification declares the char, short, unsigned integer, integer and float data types to be available. The format of the float data type may be fix-point or IEEE standard single-precision. A constant, which is separated by a colon from the type name, gives the number of bits for the fractional part after the binary point for fix-point format.

7.1.2.2 Operator Specifications

For the technology mapping step, the ALE-X compiler requires a library of the available operators. Although the mapping step is mainly straight-forward due to the coarse grained structure of the KressArray, the choice of operators depends also on the types of the operands and the result data. Therefore, one function may require several operator specifications for different data types. An operator specification includes the name of the operator, the symbol used in the ALE-X code, the number and data types of the operands, the result type, and the delay time. The maximal number of operands is six, if the rDPUs do not support packing of two operators on one rDPU, and four otherwise. An example showing specifications for a multiplication is given in figure 7-5.

```
OPERATOR   multi    IS   *   :   int, int -> int, 66 ;
OPERATOR   mults    IS   *   :   short, short -> int, 36 ;
OPERATOR   multf    IS   *   :   float, float -> float, 82 ;
```

Figure 7-5: Example operator specification in the ALE-X compiler hardware file (integers indicate delay times)

The multiplication results in three operators for different data types. The first operator is named *multi*, takes two integer values as operands, and produces an integer value as result. This operator takes 66 time units to complete. The second operator, *mults*, multiplies two short values into one integer, and the third operator *multf* works on float data. All three operators use the same symbol "*" in the ALE-X source code. The compiler selects the correct operator by considering the data types of the operands.

7.1.2.3 User Defined Functions

In addition to common operators described above, there may be also cases for rDPUs capable of special application specific functions. For example, DSP applications may take advantage of multiply-accumulate operators or the calculation of the hamming-distance of two bit vectors. To allow the use of such special functions also at the high level of the ALE-X language, user defined functions may be specified by defining the function name, the number of operands, the data types of operands and result, and the delay time. The MA-DPSS supports up to six input operands for one function. An example specification of three user defined functions is shown in figur e7-6.

```
FUNCTION   mac      :   int, int -> int, 68 ;
FUNCTION   hamming  :   int, int -> int, 4 ;
FUNCTION   max      :   int, int -> int, 4 ;
```

Figure 7-6: Example for user defined functions in the ALE-X compiler hardware file (integers indicate delay times)

The three example functions are multiply-accumulate ("*mac*"), hamming distance calculation ("*hamming*"), and maximum ("*max*"). All functions produce one integer result from two integer operands.

7.1.2.4 Delay Times

The delay times are expressed as global parameters for all communication resources of one type. They are used by the scheduler tool (section 7.4) for performance estimation and data scheduling. The original set of delay

parameters has been extended to match the MA-DPSS requirements, keeping the syntax of the original approach. An example excerpt showing a specification of delay parameters is shown in figure 7-7.

In the following, the delay models for the communication resources and the according parameters are described briefly. A more detailed description can be found in [Haas99].

```
NC_Delay_OnChip      2 ;
NC_Delay_OffChip     18 ;
Delay_Cont_Dpu       20 ;
Delay_Dpu_Cont       20 ;
```

Figure 7-7: Example for global delay specifications in the ALE-X compiler hardware file

Global Bus. For the delay modeling of global bus transfers, the original KressArray-I setup is assumed, with a control unit managing all such transfers. For compatibility reasons, there are two separate parameters for the delay from a rDPU to the controller (*Delay_Dpu_Cont*) and vice versa (*Delay_Cont_Dpu*). Thus, the total delay for one global bus transfer inside the array is given by:

$$Delay_{GlobalBus} = Delay\_Dpu\_Cont + Delay\_Cont\_Dpu$$

Row and Column Backbuses. Depending on the time scale selected, the delay for a data transfer over a backbus may also be affected by the length of the bus segments. Thus, the total delay has a component independent of this length (*RCB_Delay_Base*) and one component, which depends on the segment length (*RCB_Delay_Step*). The total delay for a transfer between one sender and one receiver is then calculated by:

$$Delay_{BackBusSingle} = RCB\_Delay\_Base + SegLength \cdot RCB\_Delay\_Step$$

For a multicast transfer, with one rDPU sending a value to multiple recipients, the delay depends on the implementation of the multicast, which is described by a global flag parameter (*RCB_Multicast*). If the backbuses handle a multicast by sending the data to all receivers at one time (*RCB_Multicast*=1), then the equation above holds. If, on the other hand, a multicast is done using one single transfer for each receiver (*RCB_Multicast=0*), the delay times add up. Thus, if the total number of receivers is *n*, the delay is given by:

$$Delay_{BackBusMulti} = n \cdot (RCB\_Delay\_Base + SegLength \cdot RCB\_Delay\_Step)$$

Nearest Neighbor Connections. The delay model for routing paths using nearest neighbor connections consists of two components, the delay caused by using the connections themselves, and the delay caused by routing through rDPUs, as shown by the following equation:

$$Delay_{Neighbor} = Delay_{Connect} + Delay_{Routing}$$

The connection delay consists of one base delay (*NC_Delay_Base*) and the sum of delays for the connections used along the path. These depend on the type of connection being used. There are different times for on-chip connections (*NC_Delay_OnChip*), connections over chip-boundaries (*NC_Delay_OffChip*) and over array boundaries, using a torus structure (*NC_Delay_Torus*). Let n_{on}, n_{off} and n_{torus} denote the number of on-chip, off-chip and torus connections on the path. Then the connection delay is given by:

$$\begin{aligned} Delay_{Connect} = {} & NC\_Delay\_Base \\ & + n_{off} \cdot NC\_Delay\_OffChip \\ & + n_{off} \cdot NC\_Delay\_OffChip \\ & + n_{torus} \cdot NC\_Delay\_Torus \end{aligned}$$

The routing delay consists of two components. The first one is again a base value, which applies for each rDPU the path is routed through (*Rout_Delay_Base*). The second component resembles the additional delay caused by a split in the routing path for multi-terminal nets. This delay (*Rout_Delay_Step*) is added for each split in each rDPU. Let *n* be the number of rDPUs the path is routed through and let *split(i)* denote the number of splits of the path in rDPU number *i*. Then the routing delay is given by:

$$Delay_{Routing} = n \cdot Rout\_Delay\_Base + \sum_{i=1}^{n} split(i) \cdot Rout\_Delay\_Step$$

7.1.3 The Compilation Process

The MA-DPSS ALE-X compiler analyzes the input program and generates a description of the dataflow graph in an α-intermediate format file. The compilation process takes place in several steps. First, the conditional statements are transformed into single-assignment code using the functional if-expression (see section 6.1.1). An analogous conversion is done for loop statements, whereby loops and sequences of assignments are considered as basic blocks.

The basic blocks are then converted to directed acyclic graphs (DAGs). On these graphs, several netlist optimizations are performed (see section 3.2). These optimizations include the removal of local variables, dead code, identical assignments, and common subexpressions. Furthermore, constant folding and reduction in strength is applied, if possible. Also, unary operators are combined with the following operator, if the merged operator is available in the operator library.

In contrast to the original ALE-X compiler, which executed also the placement and routing step, the MA-DPSS ALE-X compiler stops after the technology mapping, generating an output file in α-intermediate format. This file contains the dataflow graph of the application in a representation describing the operators and the data dependencies among them. Additional information is passed on from the hardware file and the ALE-X source. This information includes the program structure, delay times, array flags, and scanwindow information for variables. These issues will be discussed in the following.

7.1.3.1 Program Structure

From the ALE-X file, the basic blocks of the input program are identified in the intermediate file by adequate subsections. The subsections are marked with the block types determined by the ALE-X compiler. According to the discussion above, there are three types of valid blocks: Assignment blocks, While blocks, and Do-While blocks.

7.1.3.2 Delay Times

From the hardware file, the delay times for KressArray operations are stored in the intermediate file. These times are used by the scheduler (see section 7.4) in order to produce a data schedule and a performance estimation. The delay times can be divided into four groups: The delays for operators, for data transfers over I/O ports, for data transfers over communication resources, and for operations of the KressArray controller, if present. While the delays for operators and I/O ports are specified separately for each operator and port respectively, the delays for the communication resources and for the controller operations are globally described in according parameters. These parameters correspond directly to the ones discussed in section 7.1.2.

7.1.3.3 Array Flags

The array flags are only needed for compatibility with the KressArray-I. They describe the interaction of the KressArray with a data sequencer [Rein99] via status bits. Those flags consist of four groups of four bits each. A detailed description of them is given in [Kres96].

7.1.3.4 Scanwindow Information

Each I/O port of a KressArray configuration corresponds to an extern variable in the ALE-X file. Being associated with the MoM architecture [Rein99] and the Xputer paradigm [HHRS91], these variables had to belong to a scanwindow in the KressArray-I DPSS. To retain compatibility, for each I/O port the MA-DPSS ALE-X compiler outputs the scanwindow identification and the position inside the scanwindow of the corresponding variable. This information is used by the scheduler to generate an I/O schedule and to perform memory cycle optimizations.

7.1.4 ALE-X Restrictions

Although ALE-X is suitable for general datapath description in the Xplorer environment, the current version of the compiler has some restrictions, which require manual workarounds in the α-intermediate format. Those restrictions are based on the original design of the ALE-X language for the KressArray-I in an Xputer environment.

The first restriction applies to the select operator "?:" (see section 6.1.1), which is implemented by two separate operators, and thus also occupying two rDPUs. Furthermore, there is no possibility to specify cycles in the datapath (see section 6.1.3), which would be necessary e.g. for recursive functions. Also, double-output operators (see section 6.2.4) are not supported in the language. Another relict from the historic KressArray-I architecture consists in the implementation of while- and do-while loops, which is done using the control unit and does not follow the concepts described in section 6.1.4. Finally, a specification of I/O ports is not possible, as all ports are assumed to be global bus ports. Thus, edge ports and inner ports have to be added manually using the architecture editor described in sectio n8.5.3.

7.2 The Architecture Estimator

The architecture estimator is a novel tool inside the MA-DPSS synthesis subsystem, which has been developed to implement a part of the design space exploration process. Its task consists in determining an initial architecture for a set of applications. In this regard, it can be seen as a kind of analytical component in the exploration process, according to the concept of e.g. the design space exploration for Raw microprocessors described in section 5.2.4. However, in spite of complicated mathematical models, a simple approach has been taken, as this component is meant only to support the mostly interactive exploration.

The estimator does not consider the complexity of the datapath. Instead, it uses a simple strategy to define an architecture description satisfying the minimal requirements in regard of array size and number of nearest neighbor connects for all datapaths in the application domain. This basic description is then used as a starting point in the exploration process to be enhanced for an optimized architecture.

In the typical exploration, the estimator is applied once to the ALE-X compiler outputs of all applications to provide a starting architecture for the following process (see figure 7-1). However, in a later state of the exploration, it may be desirable to redo the estimation for additional applications without destroying the data collected so far. Thus, the estimation step preserves existing architecture information and considers it for the calculation of the new architecture parameters.

The initial architecture consists of parameters for the array size and for the communication resources. If no existing architecture data is found in the intermediate file, the basic pattern envisioned for a KressArray architecture consists of an array with a quadratic aspect ratio, featuring edge ports at the east and west sides. Unless specified otherwise, the array will consist of one single chip featuring all the rDPUs. The communication resources consist only of a number of bidirectional nearest neighbor connections. The determination of the size parameters and number of neighbor connects is described in the following sections.

7.2.1 Determination of Array Size

The array size required depends mainly on the maximum number of operators of all applications. Further constraints are added by existing architecture information, port considerations, and the desired aspect ratio. The calculation of the array size is done by a simple algorithm in the following steps:

- The horizontal and vertical array dimensions are set to consider the requirements of edge ports, ports inside the array, and array capability declarations (see section 6.4.2). For ports, the according position ranges have to lie inside the array limits, while for capabilities, the limit parameters of the definition have to fit inside the array. The result of this step is a set of minimal horizontal and vertical dimensions used in the further calculations.
- The total number of I/O ports is determined. A coarse approach for a reasonable architecture is assumed, with all ports not yet assigned located on the vertical edges of the array, with the input and output ports on opposing sides. The vertical array dimension is adjusted to enable this structure.

- For the calculation of the final size, the number of rDPUs required is determined. The further progress depends on the availability of previous size information.
- If a complete specification of the array size is already present, it is checked if it satisfies the minimal dimensions determined in the previous steps and the minimal number of rDPUs. If this is the case, the array size is kept. Otherwise, the dimensions are aligned to the chip size. Then, the vertical array size is increased until the array contains enough rDPUs.
- A specification without chipsize information will result in an architecture comprising one chip with the whole array. First, the estimator checks again, if the minimum array dimensions meet the number of rDPUs. If not, the square root of the number of rDPUs is calculated. The estimator tries, if the root multiplied with one of the minimal dimensions results in a sufficient array size, preferring the lesser value. If this is not the case, the array dimensions are set to this square root, resulting in a quadratic array.

This algorithm retains existing size information as much as possible, and leads to an architecture enabling edge ports, which is a reasonable approach for a KressArray. However, the architecture can be changed arbitrarily during the exploration process.

7.2.2 Determination of Communication Resources

Like the size calculation, the estimation of the communication resources tries to retain existing information and adds resources on demand. The estimation of the communication requirements does not consider the complexity of the datapath. Instead, it is made sure that every operator in the intermediate file can be potentially mapped without using the global bus. The interconnect requirements are estimated by calculating the maximal fan-in of all operators and the number of outputs. The latter value is either one or two, depending on the existence of double-output operators or double-ALU rDPUs (see section 6.2.3 and section 6.2.4). While the datapath complexity is ignored, the number of rDPUs is considered by an additional logarithmic component.

When the minimal requirements are determined this way, the existing communication resources are checked, if they already satisfy the estimation. If not, then a pair of one horizontal and one vertical bidirectional nearest neighbor connections is added, until the minimal requirements are met.

7.3 The MA-DPSS Mapper

The mapper of the MA-DPSS framework performs the task of placement and routing of the datapath (see section 3.4 and section 3.5). The objective of the placement consists in assigning each operator to a rDPU in a way, that the necessary routing between all operators is optimized. The quality of the routing is expressed by a cost function, which is associated with each mapping configuration and which is to be minimized in the mapping process. The cost function depends on the exploration aim and is determined by parameters in the intermediate file, which allow to define preferences for communication resources, limit the search depth for the routing algorithm, or set the duration of the annealing process.

The input for the mapper is a file in β-intermediate format. The information is enhanced by the mapping of the datapath, and a file in γ-format is generated. In contrast to most synthesis approaches for reconfigurable architectures, but similar to the classic DPSS mapping technique, the placement and routing step are performed concurrently rather than in two consecutive steps. Similar techniques have been proposed in other works [NaRu95], [Ebel00]. The disadvantage of simultaneous placement and routing is a higher computation time, caused by the repeated application of the relatively complex routing algorithm for each change of the placement. However, as the objective function for the placement resembles the actual optimization gain instead of an estimation, the simultaneous approach has the potential for better solutions. For the KressArray mapping, the number of cells to be placed is relatively low compared to FPGAs, so that the computation times are still moderate.

For the exploration process, it is possible to do a remapping of an already mapped datapath by feeding the mapper output again into the mapper without further modifications. As an additional feature, it is possible to mark operator positions and routing connections in an existing mapping to be fixed. The mapper will not alter the positions of fixed mappings, nor break any fixed routing connections. This way, a remapping can be done without destroying parts which have been found to have a good mapping.

In the following sections, the general placement approach is described. Then, the routing algorithm is discussed. After the algorithms have been presented, the cost parameters accounting for the cost function of the annealing process are described. Finally, the fixing of operator positions is addressed.

7.3.1 The Placement Algorithm

The MA-DPSS placement approach is an extension of the classic DPSS simulated annealing algorithm, which follows by itself the general approach described in section 3.4.3.2. New features include an optional adaptive cooling schedule as

well as the choice of different stopping conditions. Furthermore, the generation of new configurations (see below) is much more complex than in the historic DPSS due to the possibilities of double-ALU rDPUs (section 6.2.3) and the special handling of rDPU fixings and edge port placement. These issues are addressed in the subsections below. Starting from an initial placement, the algorithm generates new configurations and decides to keep or discard them, depending on the change of an associated cost function. The annealing procedure is performed following a specific cooling schedule, until a stopping condition is reached.

7.3.1.1 Initial Placement

The initial placement is produced by simply arranging the operators in row-major order in the sequence they appear in the datapath. However, due to fixings, array capabilities, and the possibility of double ALU rDPUs, some precautions have to be taken to avoid an illegal placement The general procedure consists in first placing all the fixed rDPUs. Then, while lining up the other rDPUs, the system tries to place suitable operators in one rDPU, if double ALU rDPUs are assumed. A special technique is used for the placement of edge ports, as those may have ranges, which may also overlap. An example showing overlapping edge port ranges and the resulting placement is given in figure 7-8.

To place a number *n* edge ports, a simple backtracking algorithm is employed, as sketched in the following:

- The ports are placed one by one in the order they appear in the datapath. For each port number *i*, the possible positions of its range are traversed until a free position is found. Then, the port *i* is placed on this position.The process is continued with ports *i*+1, *i*+2, ..., *n*. When all ports are placed, the algorithm stops.
- If all possible range positions for port *i* have been found to be occupied, than the algorithm backtracks, i.e. the previous port *i*-1 is moved from its current position to the next free one in its range. Then, the algorithm tries again to place port *i,* starting from the beginning of its range.
- If in the case described above, it may happen that there is also no other free position for port *i*-1, then the algorithm backtracks to port *i*-2, and so on. If this way, the range of the first port has been traversed to the end without finding a placement for all ports, then a valid port placement is not possible.

7.3.1.2 Generation Of New Configurations

The generation of new configurations is done by randomly exchanging two rDPU positions. This is performed in the following steps:

a)

Port0 West [0][2]
Port1 West [1][3]
Port2 West [0][1]
Port3 West [0][1]

b)

Array Port Position	Port Ranges
0	Port0, Port2, Port3
1	Port0, Port1
2	Port0, Port1
3	Port1

c)

Array Port Position	Actual Port Placement
0	Port3
1	Port2
2	Port0
3	Port1

Figure 7-8: Example edge port placement:
a) Definition of four ports on western edge
b) Illustration showing overlapping ranges of the ports
c) Resulting port placement on four available positions

- An operator of the datapath is selected randomly. The rDPU position of this operator is determined.
- A second rDPU position is selected randomly. This selection may be constrained by several conditions. If the first rDPU position is marked to be fixed horizontally or vertically, the second rDPU has to be in the same column or row respectively. If the second rDPU does not allow to host the operator due to array capabilities (see section 6.4.2) or the pairing attribute of operators for double ALU arrays (see section 6.2.3), then a different position has to be found.
- All routing connections emerging from or going to the operator in the first rDPU are broken up. The same is done for the operator in the second rDPU, if present.
- The operators of both selected rDPU positions are exchanged.
- All routing connections are rebuilt.

The removal and rebuilding of the routing connections will normally result in a change of the cost function, as the use of a communication resource is affected by the cost, and the routing before the exchange is typically different from the one afterwards. This change of the cost function is used to determine the acceptance or discarding of the current configuration.

7.3.1.3 Cooling Schedule

The cooling schedule for the annealing process is determined by the start temperature, the decrease of the temperature after some iterations have been executed in the inner loop, the number of iterations in this inner loop, and the stopping condition. For the exploration process, the mapper offers both a

parameter based cooling schedule, and an adaptive schedule. The adaptive schedule is based on the technique used in [BRM99], with parameters being found by experiments.

Parameter Based Schedule. The intermediate file contains parameters for the start temperature (*MaxTemp*), as well as the temperature decrease (*TempFactor*). During each temperature step, a number of iterations of the inner loop is performed. This number depends on the amount of operators to be mapped and on the according parameter *Iterations*. Let *MappedExpr* denote the number of operators in the datapath, and *StepsPerTemp* denote the number of iterations in one temperature step. Then *StepsPerTemp* calculates to:

$StepsPerTemp = Iterations * MappedExpr ;$

After *StepsPerTemp* iterations have been performed, a new temperature is calculated using *TempFactor*. Let *NewTemp* denote the temperature for the next set of iterations, and *CurTemp* denote the current temperature. Then the new temperature is given by:

$NewTemp = CurTemp * TempFactor ;$

The annealing stops, when the final temperature (*MinTemp*) is reached, or when no change in the configuration has occured for a number of iterations. This number can be set by an according parameter (*NoChangeDie*).

Adaptive Cooling Schedule. The adaptive cooling schedule allows the automatic calculation of the starting temperature, the temperature decrease, and the end temperature, thus overriding the according parameters in the intermediate file. The adaptive schedule can be activated individually for all three parameters, retaining the values in the intermediate file for those parameters not overridden.

The adaptive cooling schedule is based on the approach described in [BRM99] (see also section 3.4.3.2). This approach uses by itself experience gained from earlier works, which will be described in the following. The parameters *Iterations*, *MappedExpr*, *MaxTemp*, *MinTemp* and *TempFactor* are supposed to have the same meaning like in the section about the parameter based schedule.

The start temperature for the adaptive cooling schedule is calculated from statistic data gained by some trial moves (i.e. random exchanges of rDPU positions). This statistic resembles the standard deviation of the cost function during these moves, which is multiplied with a constant *C* to gain the start temperature. This technique is described in [BRM99] and is based on the earlier work in [HRS86]. While Rose et al. perform $n = MappedExpr$ trials, the MA-DPSS mapper uses $n = 3 * MappedExpr$ moves to get a better base for the standard

deviation due to the lower cell count. The factor C for the calculation of the final temperature, has been chosen to be 10. Let $Cost_i$ denote the value of the cost function for the ith trial move. Then the initial temperature $MaxTemp$ is given by:

$$MaxTemp = C \cdot \sqrt{\frac{\sum_{i=1}^{n} Cost_i^2 - \frac{\left(\sum_{i=1}^{n} Cost_i\right)^2}{n}}{n}}$$

For the decrease of the temperature, the constant *TempFactor* is replaced with an adaptive value depending on the acceptance rate of the previous temperature step. It has been found, that the acceptance rate, which denotes the ratio of the number of accepted configurations to the number of all configurations generated, should be kept at about 0.44, as this rate allows the fastest cooling without losing quality of the annealing result [LaDe88], [SwSe90]. Instead of a sophisticated scheme, the more simple approach of [BRM99] has been used, which employs a table lookup with different values of *TempFactor* for different ranges of the acceptance rate. The *TempFactor* is kept high, if the acceptance rate lies within the range around 0.44 and is lower for the other ranges. However, in each temperature step, the number of iterations to be performed is given by:

StepsPerTemp = Iterations ∗ MappedExpr ;

as in the parameter based cooling schedule. The values for *TempFactor*, which have been found by experiments, are shown in table 7-1.

Acceptance Rate *r*	***TempFactor***
$0 \le r \le 0.01$	0.7
$0.01 \le r \le 0.15$	0.96
$0.15 < r \le 0.5$	0.98
$0.5 < r \le 0.95$	0.9
$0.95 < r \le 1$	0.5

Table 7-1: Adaptive temperature decrease for simulated annealing

For the stopping criterion of the annealing process, the constant value *MinTemp* is again replaced by an adaptive term, which depends on the current value of the cost function and the total number of operators [MBR99]. The end temperature $MinTemp_k$ is calculated at each temperature step k, and the annealing process

stops, if ever the temperature drops below $MinTemp_k$. If $Cost_k$ denotes the value of the cost function for the current configuration at the end of temperature step k, then $MinTemp_k$ is given by:

$$MinTemp_k = 0.05 \cdot \frac{Cost_k}{MappedExpr}$$

The effect of the adaptive cooling schedule is illustrated by an example mapping of a datapath with 70 operators in figure 7-9, done with an adaptive schedule and a parameterized schedule. The chosen or calculated parameters and result values for both mappings are shown in table 7-2.

	Parameterized Cooling	**Adaptive Cooling**
MappedExpr	70	
Iterations	15	
MaxTemp	1000.0	6026.0
TempFactor	0.95	changing
MinTemp	1.0	changing
Final Temperature	0.98	0.50
Final Cost	1675	1211
Total Iterations	188325	172980

Table 7-2: Experiment with adaptive and parameterized cooling schedule

The decrease of the temperature during the annealing in figure 7-9a shows a constant value for *TempFactor* for the parameterized cooling, while the adaptive decrease depends on the acceptance rate. While the adaptive schedule starts at a much higher temperature, it quickly decreases, avoiding an exaggerated heating phase without enhancement of the configuration at the beginning. When large enhancements occur, the adaptive schedule keeps the temperature decrease higher. At the end, when there is hardly any decrease of the cost function, the temperature is reduced faster. As a result, the cost decrease is more evenly distributed over the annealing time, as shown in figur e7-9b.

Although the computation time in the example is shorter with a better result, the adaptive schedule does not guarantee a better solution compared to the parameterized schedule. Generally, the adaptive schedule is meant to be used during exploration especially for bigger datapaths, as no manual specification of

parameters is needed. However, for generation of final mappings, or for fine-tuning of the annealing process by an experienced user, a parameterized schedule may eventually produce better results.

7.3.2 The Routing Algorithm

The routing algorithm of the MA-DPSS mapper is used to connect rDPUs each time a new configuration is being generated. Only the two operators involved in the exchange operation are re-routed to keep the computation time moderate. The actual number of connections to be routed depends on the fan-in and the fan-out of the involved operators. In the following, first the general approach is sketched, and then the algorithm for the nearest neighbor routing is described.

7.3.2.1 General Routing Approach

For each connection to be routed, the mapper tests three possible types of connections, if they are applicable:

- Row or column backbuses.
  A connection over a row or column backbus is possible, if the source and destination rDPU lie in the same row or column respectively, and in the same segment of a backbus, which is not yet occupied to its limit. The latter condition is met, if the maximal number of writers on the bus is not yet reached or if the source rDPU is already one of the writers.
- Nearest neighbor connections.
  Establishing a routing between two rDPUs comprising nearest neighbor connections is equivalent to the routing problem for reconfigurable architectures discussed in section 3.5. The feasibility of such a routing depends on the number and type of existing neighbor connections per cell, and the complexity of the application datapath. The routing technique employed in the MA-DPSS mapper is based on Dijkstra's algorithm and described below.
- Global bus connections.
  A connection using the global bus is always possible. In order to avoid excessive use of this resource due to its universality, it is typically discouraged by assigning it a higher cost in contrast to the other resources.

The mapper tries all three possibilities, and determines the potential change of the cost function for every communication resource. This change depends on the selection of the according cost parameters. Then, the cheapest connection is actually implemented.

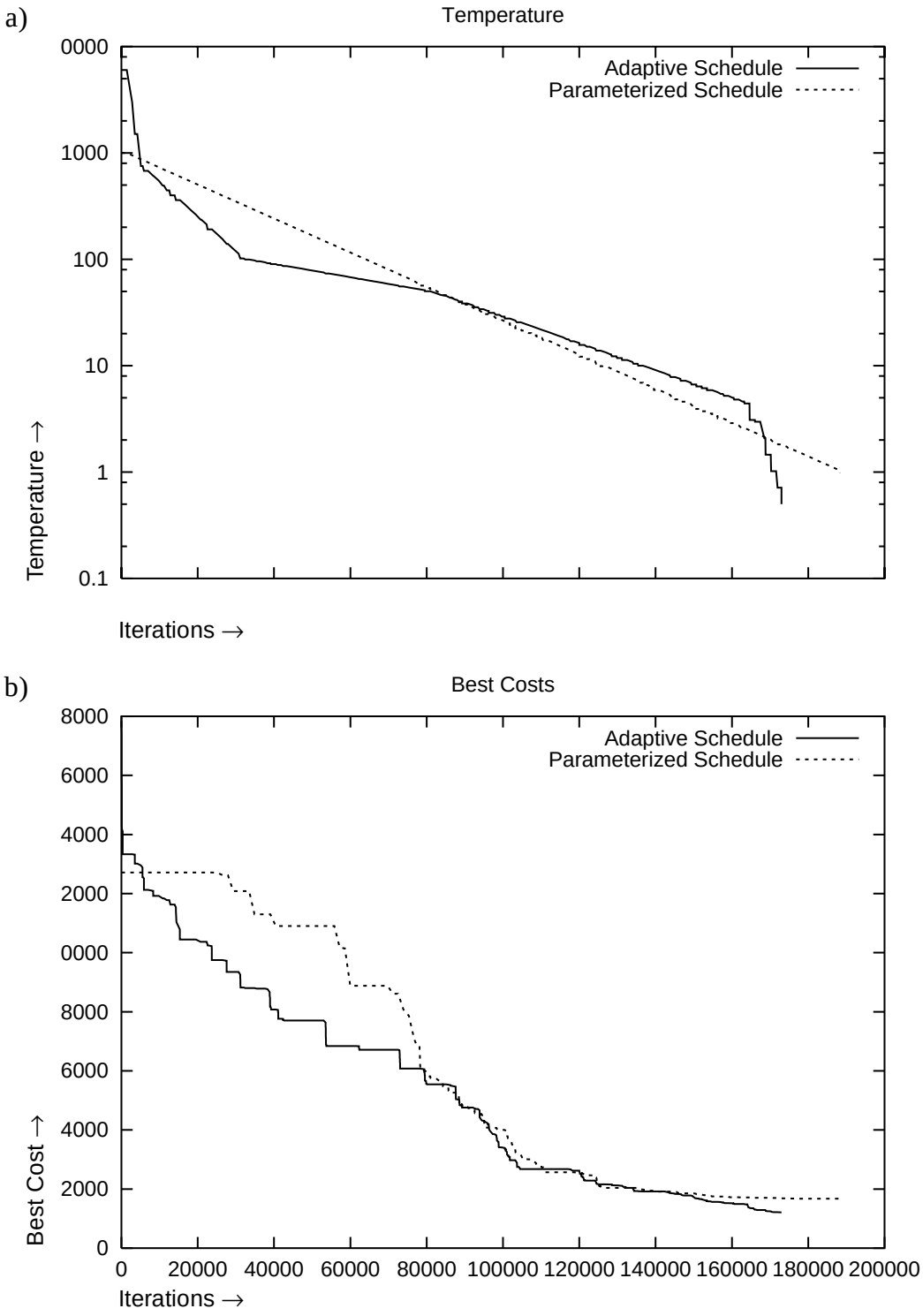

Figure 7-9: Adaptive versus parameterized cooling schedule:
a) Temperature decrease (logarithmic scale)
b) Best cost decrease

As existing routing connections are not destroyed when a new path is constructed, the sequential application of this approach would involve the same problem encountered with the simple sequential application of the Lee or Dijkstra algorithms. As described in section 3.5, this problem consists in a significant impact of the global routing order on the quality of the routing. Common approaches meet this problem by explicit rip-up and re-route cycles. The MA-DPSS mapper does this automatically by embedding the routing in the simulated annealing process, which results in a rip-up of routing connections each time two rDPU positions are exchanged. Thus, the routing and placement of the datapath are optimized concurrently.

7.3.2.2 Nearest Neighbor Routing Algorithm

The technique used for nearest neighbor routing is based on the Dijkstra algorithm [Dijk59], as described in section 3.5.3. Generally, a connection is routed from the data sink (routing source) to the data source (routing target) by recursively building a path consisting of nearest neighbor segments between two rDPUs. When there are more than one nearest neighbor segments possible, the one with the lowest cost is selected, and the accumulated cost of the path is stored in the rDPU position corresponding to Dijkstra's approach. The algorithm is sketched in figur e7-10.

The algorithm uses a recursive approach for the search. The recursion depth can be controlled by the parameter *RoutingDepth*, which limits the search. Another implemented means for limiting the recursion depth consists in storing the global minimum for the path cost. The recursion stops immediately, if the current accumulated cost exceeds this global minimum. This criterion has shown to efficiently limit the search time very fast once a valid path has been found.

The recursion stops also, when a valid routing target is found. The set of these valid routing targets depends on the operation mode of the router. There are two operation modes, which can be selected by the user. The first mode does not allow nonlinear nets, which results in the use of a new resource for each fan out connection of an operator. Thus, the set of routing targets contains only the source rDPU itself. In the second mode, nonlinear nets are routed by allowing a path to connect to an existing routing segment carrying the data word generated by the routing target (which is the data source, as mentioned above). Thus, the set of routing targets contains the target rDPU and additionally all rDPUs which carry the data word. The concept of linear and nonlinear nets as well as the target rDPUs are illustrated in an example in figure 7-11.

The figure shows a scenario with a source operator *src*, which has to be routed to three sinks *t1*, *t2* and *t3*. In figure 7-11a, the routing is done using linear nets only. As there are three data sinks, also three connections are needed starting from the source rDPU. With the given placement of the operators, at least two vertical

```
ALGORITHM MA-DPSS Router
{   PROC'D RecursiveSearch( rDPU, RoutingDepth, CurrentCost )
    {   /* The search depth is limited by RoutingDepth */
        IF ( RoutingDepth == 0 )
        {   RETURN ;   }
        /* Stop, if there already exists a cheaper path to rDPU */
        IF ( CurrentCost Š MinCost OR CurrentCost Š Marker[ rDPU ] )
        {   RETURN ;   }
        /* Stop, if a target is found */
        IF ( rDPU ∈ TargetSet )
        {   MinCost = CurrentCost ;
            RETURN ;
        }
        /* Enter recursion by stepping to neighbors */
        FOREACH d IN NeighborsOf( rDPU )
        {   Link = FindCheapestConnect( d, rDPU ) ;
            Cost = CostParam( Link ) ;
            RecursiveSearch( d, RoutingDepth-1, CurrentCost+Cost ) ;
        }
        RETURN ;
    }   /* End of procedure RecursiveSearch */

    /* TargetSet contains all possible routing target rDPUs */
    TargetSet = FindAllRoutingTargets();
    /* The global minimal cost is stored for effectivity */
    MinCost = ∞ ;
    /* Initialize the markers for the rDPUs */
    FOREACH p IN AllrDPUs
    {   Marker[ p ] = ∞ ;
    }
    /* Invoke recursive search procedure */
    RecursiveSearch( StartrDPU, RoutingDepth, 0 ) ;
}   /* End of Algorithm MA-DPSS Router */
```

Figure 7-10: MA-DPSS nearest neighbor routing algorithm

nearest neighbor ports are needed to map the datapath without employing a global bus connection. In figure 7-11b, the routing uses just one nonlinear net, which splits up twice to connect to *t1*, *t2*, and *t3*. However, this time, only one connection leaves the rDPU of operator *src*, and the whole mapping can be accomplished with only one horizontal and one vertical nearest neighbor port. To

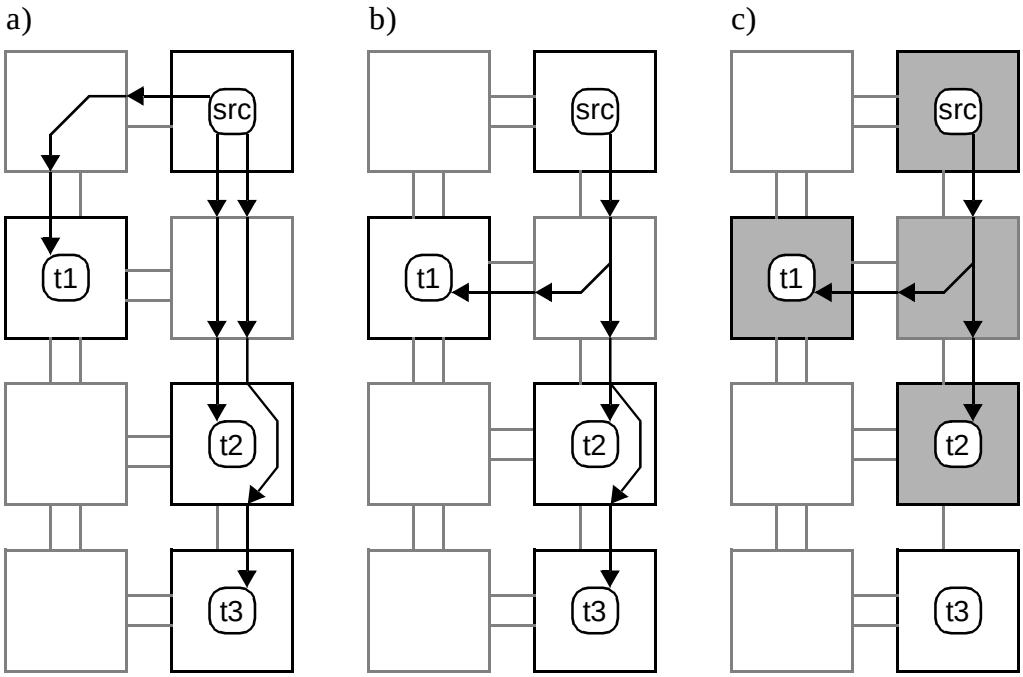

Figure 7-11: Routing modes of MA-DPSS:
a) Routing using linear nets only
b) Routing using a nonlinear net
c) Set of possible data sources for operator t3 (shaded)

accomplish this routing, the set of routing targets for each rDPU is extended to all rDPUs being already connected to *src*. This situation is illustrated in figur e7-11c, which shows a snapshot after the routing of *t1* and *t2* to *src*, and before *t3* is routed. The possible routing targets for operator *t3* are shaded in the figure. As described in figure 7-11b, the routing for *t3* is established by connecting to the rDPU holding *t2*, which results in a connection path of length one.

Nonlinear nets require the rDPUs to be capable of splitting up a connection and distributing a data word onto several outputs. Depending on the hardware implementation, this capability may imply additional delays or hardware efforts. For example, a KressArray emulator implemented as a research project [Zimm99] described in section 6.5.2 needs several CPU cycles to handle a splitting of a communication path, while nonlinear nets are routed in one cycle. However, the additional requirements for connection resources implies severe drawbacks in regard of chip area, making array structures only capable of linear nets inefficient for most application domains.

7.3.3 Cost Parameters

The simulated annealing process of the placement algorithm is based on a cost function, which describes the quality of the mapping. The cost of a complete mapping is composed of the costs of all the routing connections in it. For a specific routing connection, the cost depends on the type of resource used, the distance of the rDPUs to be linked, the number of rDPUs without an operator on the path, and if the target rDPU is part of a broadcast.

The MA-DPSS allows the cost function to be controlled from outside by according parameters in the intermediate file. These parameters allow the user to make the mapper prefer certain communication resources, while others are discouraged. A typical scenario uses a high cost for the global bus, as a high usage of this resource implies a performance decrease due to its serial character. On the other hand, nearest neighbor paths or row and column backbuses are normally encouraged by keeping the cost parameters for these resources low.

The actual cost for a connection is calculated differently for the three possible routing resources global bus, row and column backbuses, and nearest neighbor connections. In the following sections, the definitions of the according cost functions and the parameters involved are described in more detail.

7.3.3.1 Global Bus Cost Function

There are three cost parameters, which can be set for the global bus to steer the synthesis process. These are called *CostBusBase*, *CostBusStep* and *CostBusWrite*. Basically, the higher the cost parameters are set, the more discouraged the use of the global bus for a connection will be, thus favoring other communication resources. The meaning of the parameters are described in the following.

The *CostBusBase* parameter specifies a basic cost, which is accounted every time a connection is routed over the global bus. To this basic cost, another value is added, which depends on the manhattan distance of the rDPUs to be linked. The manhattan distance is the sum of the horizontal and vertical grid distances between the rDPUs. By including a distance component into the cost function, the proximity of cells is encouraged. Thus, an additional cost is added, which is given by *CostBusStep * Distance*.

A final component of the total cost leads to preference of operator broadcasts. The idea here is, that though the global bus can handle only one data word at a time, this data word may as well be broadcast to several rDPUs when it is transferred over the bus, thus using only one communication resource. The favoring of broadcasts is implemented by the third cost parameter *CostBusWrite*, which is

only accounted for new connections, but left out for further links transferring the same data from the source rDPU to another target. Following this discussion, the total cost for a global bus connection is given by:

$$TotalCost_{GlobalBus} = CostBusBase + (CostBusStep \cdot Distance) + \begin{cases} CostBusWrite & \text{if connection source is new} \\ 0 & \text{else} \end{cases}$$

7.3.3.2 Row and Column Backbuses

Backbuses have three parameters describing the components which add up to the cost for one connection over the specified bus. These parameters are called *CostBase*, *CostStep* and *CostWrite*. Their meaning is analog to the parameters *CostBusBase*, *CostBusStep* and *CostBusWrite* for the serial global bus (see section 6.3.1 above). The final total cost for a backbus connection is given by a formula similar to the one for the global bus:

$$TotalCost_{BackBus} = CostBase + (CostStep \cdot Distance) + \begin{cases} CostWrite & \text{if connection source is new} \\ 0 & \text{else} \end{cases}$$

7.3.3.3 Nearest Neighbor Connections

As a part of the definition in the intermediate file (see section A.1), each nearest neighbor connection features two cost parameters, which contribute to the total cost that is accounted for one use of the according connection. The two cost parameters are the *OnChipCost* and the *OffChipCost*. Here, *OnChipCost* is the base cost accounted for every connection. The second parameter *OffChipCost* has been adapted from the classic KressArray-I DPSS [HaKr95] to support a physical structure of a KressArray which is built from several chips containing smaller rectangular arrays. A connection over chip boundaries is expected to be slower or more power consuming than one which stays inside the chip. This fact is considered by the additional cost *OffChipCost*, which is added to the cost of a connection crossing chip borders.

Routing Elements. A further issue for the construction of routing paths are routing elements. A routing element is a rDPU, which is only used for routing, i.e. which passes data from an input to an output without implementing a computational operator. As such routing elements use up rDPUs without contributing to the datapath, they are generally discouraged. This is realized by accounting an additional cost for a routing element. This cost is added to the nearest neighbor cost whenever the starting rDPU for the connection is a routing element. The global parameter describing this cost is called *CostRoutElem*. A high cost for routing elements will generally lead to a compact mapping. However, it may also cause additional global bus connections due to the lack of other routing resources. Such a scenario is illustrated by two example mappings in figure 7-13 and figure 7-12. The mappings realize a division of two complex numbers *G* and *H* into the result *F* by implementing the following formula:

$$F = \frac{G}{H}; \qquad F = (f_r, f_i); \qquad G = (g_r, g_i); \qquad H = (h_r, h_i)$$

$$denom = h_r^{\,2} + h_i^{\,2}$$

$$f_r = \begin{cases} \dfrac{g_r h_r + g_i h_i}{denom} & denom \neq 0 \\ 0 & else \end{cases} \qquad f_i = \begin{cases} \dfrac{g_i h_r - g_r h_i}{denom} & denom \neq 0 \\ 0 & else \end{cases}$$

The mapping in figure 7-13, featuring a value for *CostRoutElem* of 0, uses an area of six by nine rDPUs for 14 operators. There are nine routing elements. However, the upper row of rDPUs could potentially be used by another design for computational resources, retaining the routing connections. This mapping needs two global bus connections in addition to the nearest neighbor paths, as denoted by the dotted arrows. The mapping in figure 7-12 features a high cost for routing elements, which leads to only one of them being used. The mapping takes only a four by four area, which is the optimum considering the given array size. However, due to the tight packing of the operators, another global bus connection is needed.

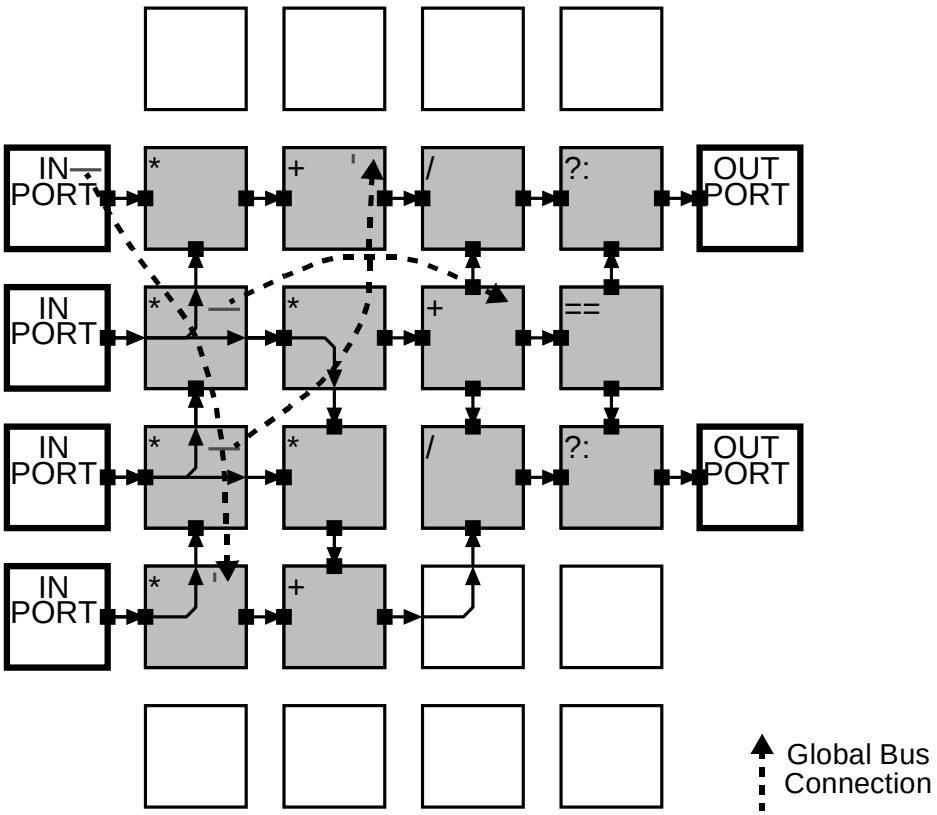

Figure 7-12: Example mapping of the complex division datapath with high cost for routing elements, showing explicitly global bus connections

In summary, the total cost of a path built from nearest neighbor connections is calculated by the sum of the costs for each nearest neighbor links involved, with the cost for a specific link established by a nearest neighbor connection being given by the following formula:

$$LinkCost_{Neighbor} = OnChipCost + \begin{cases} OffChipCost & \text{if chip border is crossed} \\ 0 & \text{else} \end{cases} + \begin{cases} CostRoutElem & \text{if source rDPU is a routing element} \\ 0 & \text{else} \end{cases}$$

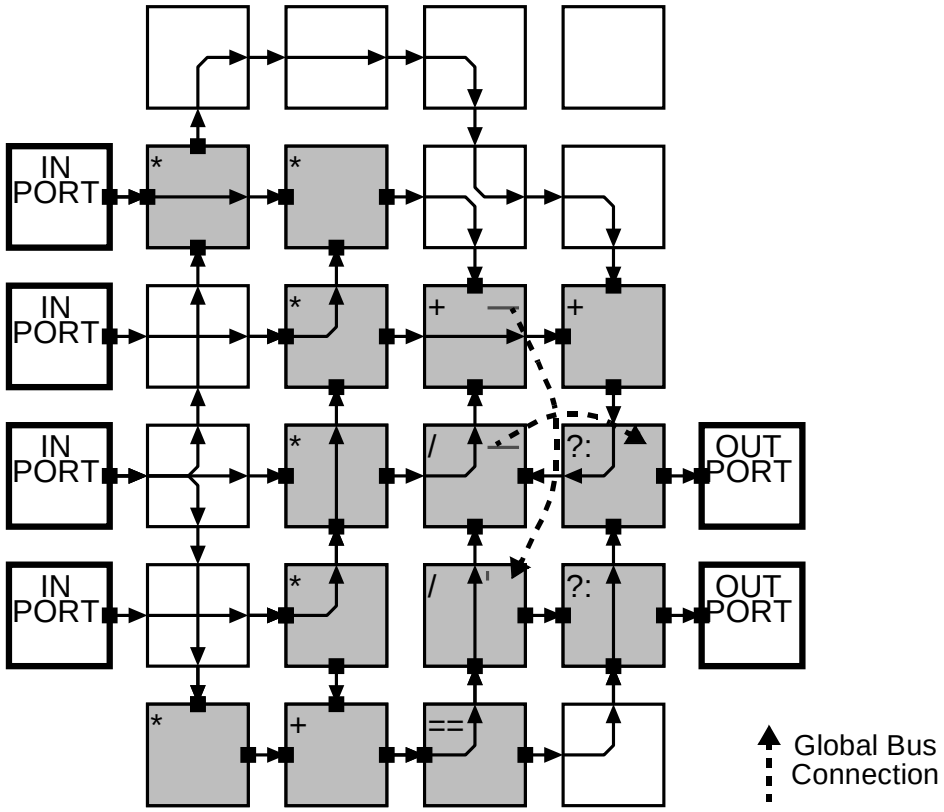

Figure 7-13: Example mapping of complex division datapath with low cost for routing elements, showing explicitly global bus connections

7.3.4 Fixing of Operator Positions

Besides the normal way of feeding a β-intermediate to the mapper, it is also possible to use a γ-intermediate file with an existing mapping as an input file. This allows for example a remapping of a datapath with other cost parameters. However, certain parts of the existing mapping may be considered to be worth keeping, so they should be protected from alterations by the simulated annealing algorithm.

The fixing is specified in the intermediate file (sectio nA.1) for each operator and each routing connection. A routing connection can be either "Variable" or "Fixed", with "Variable" being the default state, allowing the routing algorithm to change this connection during the mapping process. If, on the other hand, the connection is marked as "Fixed", the mapper will retain the connection in its current state.

For the operators, the fixing information can be specified on a finer granularity. The fixing attribute of an operator position in a mapping has one of four values:

- "Variable": This is the default value, indicating that the mapper is allowed to change the position of this operator as far as other constraints, like array capabilities (see section 6.4.2) or pairing attributes (see section 6.2.3) allow.
- "X-Fixed": This means, that the horizontal coordinate of the current operator position will not be altered. However, the operator can still be moved inside the same row.
- "Y-Fixed": This is the analog case to "X-Fixed", enabling a change of the horizontal coordinate of the operator position, while the operator is fixed to its current row.
- "Fixed": This value indicates, that the position of the operator will not be altered at all by the mapper. It is implemented by marking the operator both "X-Fixed" and "Y-Fixed".

Edge ports are a special case for fixing, as they are by default bound already to one edge of the array. Therefore, ports at the north and south edge are for themselves "Y-Fixed", while east and west ports are "X-Fixed". Furthermore, an edge port can be locked into one position by making it "Fixed".

The effect of operator fixing will be illustrated by a simple experiment shown in figure 7-14 and figure 7-15. The mapping in figure 7-14a is suboptimal as the six operators in the right half are arranged so that the whole mapping has height five, although four rows would be sufficient from the number of operators. However, the left eight operators as well as the input ports show a good arrangement. In order to remap the datapath without destroying the left part, the left eight operator positions are locked by marking them "Fixed". To reduce the height of the mapping, one would try to get the six operators in the right half of the array upwards. A reasonable approach consists in forcing each of the output ports one position to the north and fixing it there. These preparations are sketched in figure 7-14b.

During the annealing process, only the six operators in the right half of the array are allowed to change positions, while the ports and the eight operators in the left half are locked in their places. A snapshot taken after 100 iterations of the annealing shows this effect in figure 7-15a. After the annealing has finished, the final mapping shown in figure 7-15b features the left half of the array unchanged compared to figure 7-14a, with the six mobile operators having now a more optimized arrangement, reducing the required array size by one row.

7.3.5 Restrictions of the Mapper

Although the mapper is capable of handling all architectures in the design space defined in section 6, the current version has restrictions concerning the maximal numbers of some architectural features. First, for each horizontal and vertical

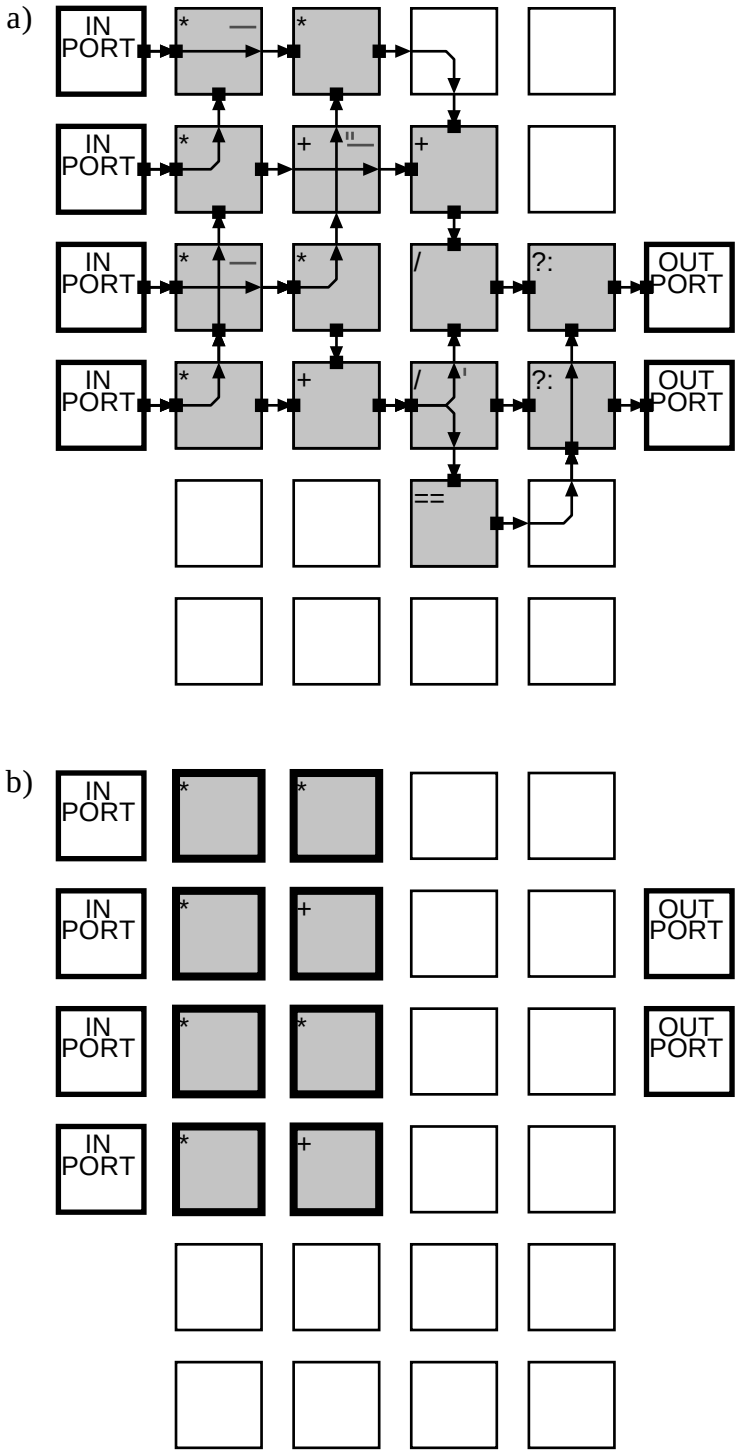

Figure 7-14: Experiment for the fixing of operator positions (1), using the example from figure 7-12:
a) Initial mapping with extra area usage
b) Fixing the position of eight operators

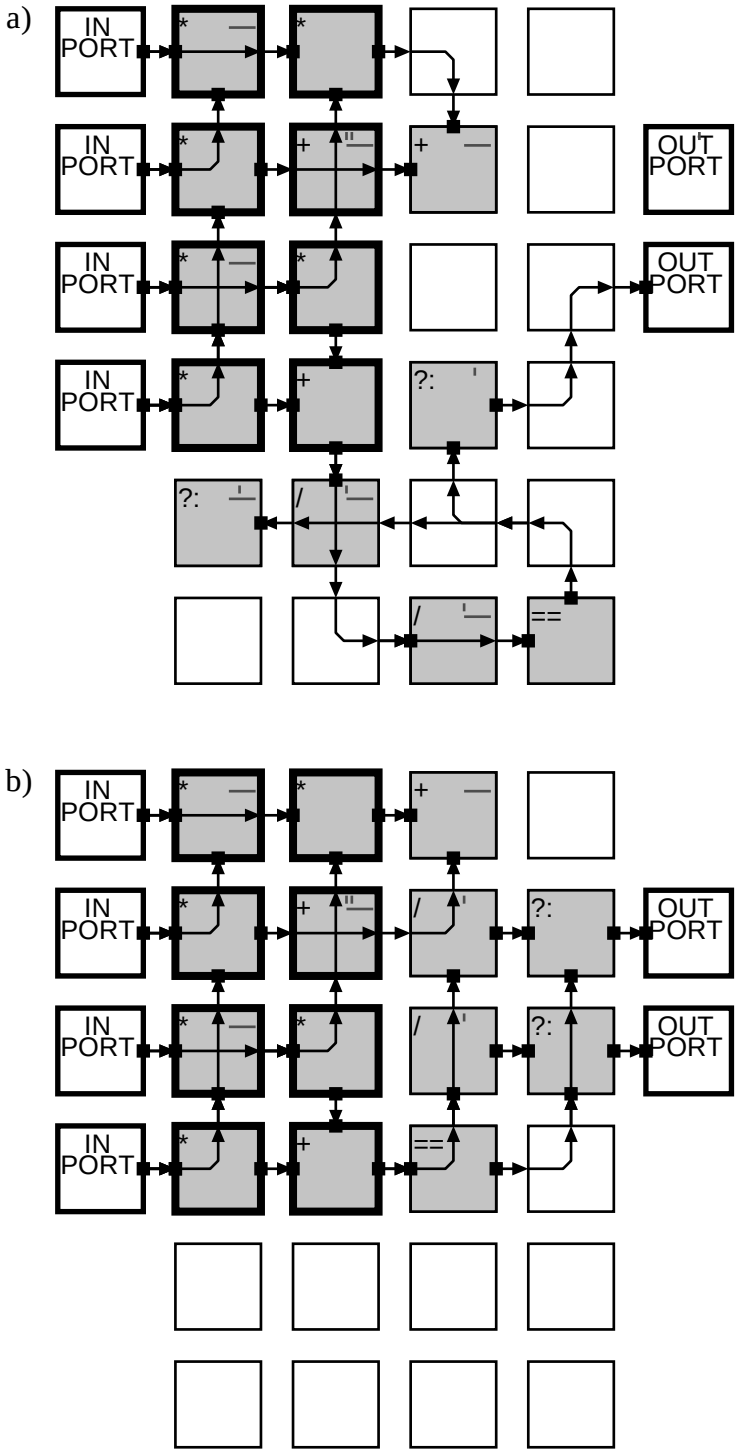

Figure 7-15: Experiment for the fixing of operator positions (2), using the example from figure 7-12:
a) Snapshot of annealing after 100 iterations
b) Final mapping

dimension, there are at most four nearest neighbor links and four backbuses allowed. Second, there must not be more than six inputs to any operator, if the array features only single ALU rDPUs. For double ALU rDPUs, as described in section 6.2.3, this number reduces to four.

7.4 The MA-DPSS Scheduler

The scheduler in the MA-DPSS framework [Haas99] is an extension of the according DPSS tool described in [Kres96]. The scheduling process determines the optimal sequence of the operations in the datapath considering data dependencies and resource constraints. For the calculation of the schedule, the delay times of the operations are used, and the resulting schedule is optimal in the sense of having the shortest total delay.

In the Xplorer framework, the scheduler has two main tasks:

- Like in the classic DPSS, the scheduler produces control code for data transfers over restricted communication resources and for data I/O. To retain compatibility to the KressArray-I, the MA-DPSS scheduler is also capable of performing several optimizations, which partially assume a surrounding Xputer architecture. These optimizations include loop unrolling, loop folding, vectorization and memory cycle optimization.
- For the exploration process, the scheduler derives statistical data from the schedule, which can be used to evaluate a given mapping. This data includes the estimated delay of the datapath, the length and location of the longest path, and the average waiting time caused by unavailable operators.

Generally, the necessity for a schedule arises from the restriction of some resources to handle only one operation at a time. Unlike scheduling in VLSI design, the computational resources are no such constraint, as the mapping process assigns one physical rDPU to each operator. Instead, the constrained resources are the communication facilities for data I/O and internal routing. They resemble the global serial bus, a memory bus connected to the global bus, row and column backbuses, and groups of edge ports. An example architecture featuring all these resources is shown in figure 7-16. In the following, the issues concerning the scheduling for these resources are discussed briefly.

Global Serial Bus and Memory Bus. The global bus is by itself a constrained resource, as it allows only serial data transfer with one word at a time. In this context, the bus is assumed to be controlled by a dedicated bus controller, which handles all transfers using an according control program.

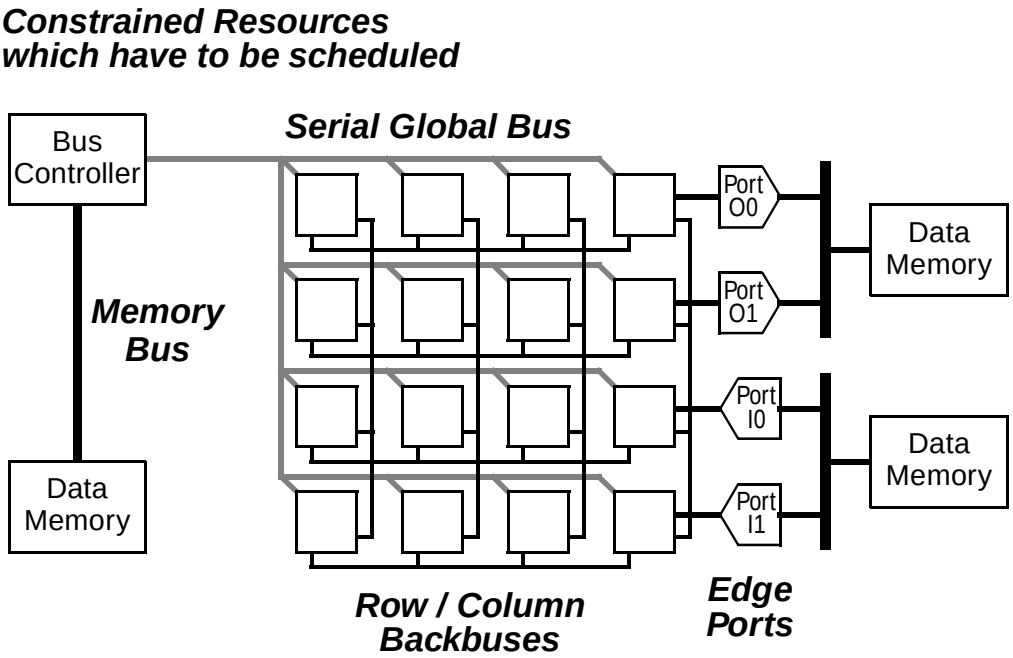

Figure 7-16: Example KressArray architecture giving an overview on all constrained resources of the KressArray design space which are subject to scheduling

The connection of the memory bus to the global bus controller has been introduced by the MoM architecture. The memory bus resembles also a resource constraint in the same sense as the global bus. The transfers over the memory bus are assumed to be handled as well by the control unit. Thus, the control instructions for memory bus transfers as well as the control instructions for the global bus transfers themselves are integrated in one control program generated by the scheduler. The scheduler assumes no other units accessing the memory bus. If this is not the case, additional delays will be observed during runtime, which cannot be estimated by the scheduler.

Row and Column Backbuses. A row or column backbus is only a potential resource constraint for the scheduler, if it allows multiple writers. In this case, each bus segment is scheduled, and an according control file is generated assuming the bus transfer handling mechanism described in section 6.3.3.2.

Edge Ports. Although I/O ports located at the edge can by themselves operate in parallel, each port is typically connected to a data memory, over a bus, which may be shared with other ports, thus being restricted to serial data transfers. As edge ports may be attached to memories in different ways, the groups of ports, which have to be scheduled together have to be declared explicitly using the *PortGroup* attributes of the ports as described in sectio n6.4.4.

The MA-DPSS scheduler relies on known scheduling techniques also employed by the historic DPSS tool. Though, the general scheduling algorithm differs significantly due to the additional resources to be considered. Furthermore, the scheduler is capable of performing several optimizations of the original DPSS scheduler, some of which assume architectural features as being described in [Kres96] for the KressArray-I and the MoM architecture. In the following subsections, first the scheduling approach of the MA-DPSS will be described. Then, the optimizations will be sketched briefly.

7.4.1 MA-DPSS Scheduling

The scheduling applies to the data transfers of restricted communication resources and is done on a non-hierarchical weighted data dependency graph (DDG), which is defined in a similar way as in section 3.5.3 as a directed graph $G(V, E)$ with a set of Vertices V, a set of edges E, an incidence function $Inc(e)$, $e \in E$, and a weight $delay(e)$, $e \in E$. Analog to $Inc(e)$, a function $Edge(u, v)$ is defined by $Edge(u, v) = e$, with $e \in E$ and $Inc(e) = (u, v)$.

The vertices $v \in V$ correspond to the operators of the datapath. The edges $e \in E$ correspond to the data dependencies between two operators. As data dependencies are directed, the incidence function $Inc(e) = (a, b)$ results in an ordered tuple of the vertices $a, b \in V$ for each edge $e \in E$.

Based on these definitions, the set of predecessors $pred(v)$ and the set of successors $succ(v)$ of a vertex $v \in V$ are given by:

$pred(v) = \{p \mid p \in V \wedge \exists\, e \in E : Inc(e) = (p, v)\}$, and

$succ(v) = \{p \mid p \in V \wedge \exists\, e \in E : Inc(e) = (v, p)\}$.

Let $s, t \in V, d \in E, Inc(d) = (s, t)$. Then $delay(d)$ corresponds to the delay of the data transfer from operator s to t. This delay includes the execution time of the operator as well as the delay afflicted by the communication resources. The delays are assumed to be known for all operators and communication resources and are considered to be data independent. Further, they are expected to be integer multiples of a basic time step. The scheduling assigns a start time $start(v)$ to each vertex $v \in V$. According to the assumptions for the delay times, $start(v)$ is also an integer multiple of the basic time step. With the start times provided, the total latency of a schedule is given by:

$latency = max(start(v) \mid v \in V) - min(start(v) \mid v \in V)$.

In the following, basic scheduling techniques are reviewed briefly, before the MA-DPSS approach is described. The ASAP and ALAP scheduling algorithms are used to calculate the important measure of mobility, which is again used in the list-based scheduling algorithm. The MA-DPSS scheduling resembles an

adaptation of this list-based scheduling. After the discussion of scheduling techniques, the statistic information generated by the scheduler is described briefly.

7.4.1.1 ASAP Scheduling

The ASAP (As Soon As Possible) scheduling algorithm is a foundation for more sophisticated techniques. It determines the earliest possible starting time for each operator. The algorithm assumes, that there are no resource constraints. Thus, the start times depend only on the delays given by the weights *delay*(*e*) of the graph edges. The ASAP algorithm is outlined in figur e7-17.

```
ALGORITHM ASAP Scheduling
{     /* Determine operators at beginning and */
      /* build set Rest of yet unscheduled operators */
      Rest = ∅ ;
      FOREACH v IN V
      {     IF ( pred( v ) == ∅ )
            {     start_ASAP( v ) = 1 ;
            }
            ELSE
            {     start_ASAP( v ) = 0 ;
                  Rest = Rest ∪ v ;
            }
      }
      /* Schedule the operators */
      WHILE ( Rest != ∅ )
      {     FOREACH r IN Rest
            {     IF ( AlreadyScheduled( pred( r ) ) == TRUE )
                  {     / * Schedule r */
                        start_ASAP( r ) = max( start( s ) + delay( Edge( s, r ) ) |
                                                      s ∈ pred( v ) ) ;
                        Rest = Rest - { r } ;
                  }
            }
      }
}     /* End of algorithm ASAP Scheduling*/
```

Figure 7-17: ASAP scheduling algorithm

7.4.1.2 ALAP Scheduling

The ALAP (As Late As Possible) scheduling algorithm is similar to the ASAP algorithm, but determines the latest possible starting time for each operator. The ALAP algorithm does also assume no resource constraints and is outlined in figure 7-18.

```
ALGORITHM ALAP Scheduling
{     /* Determine operators at beginning and */
      /* build set Rest of yet unscheduled operators */
      Rest = ∅ ;
      FOREACH v IN V
      {     IF ( pred( v ) == ∅ )
            {     start_ALAP( v ) = max(start_ASAP( p ) | p ∈ V ) ;
            }
            ELSE
            {     start_ALAP( v ) = 0 ;
                  Rest = Rest ∪ v ;
            }
      }
      /* Schedule the operators */
      WHILE ( Rest != ∅ )
      {     FOREACH r IN Rest
            {     IF ( AlreadyScheduled( succ( r ) ) == TRUE )
                  {     / * Schedule r */
                        start_ALAP( r ) = min( start( s ) - delay( Edge( s, r ) ) |
                                                s ∈ succ( v ) ) ;
                        Rest = Rest - { r } ;
                  }
            }
      }
}     /* End of algorithm ALAP Scheduling*/
```

Figure 7-18: ALAP scheduling algorithm

7.4.1.3 Mobility

If the earliest starting point and the latest starting point for an operator $v \in V$ is provided by the ASAP and ALAP algorithms, the mobility range $M(v)$ of v can be defined by:

$$M(v) = \{ n \mid start_{ASAP}(v) \leq n \leq start_{ALAP}(v) \}$$

The mobility range of an operator *v* is the set of all time steps, in which *v* can be executed without increasing the latency of the schedule. The number of these time steps, the mobility of v, is given by:

$$mobility(v) = | M(v) | = start_{ALAP}(v) - start_{ASAP}(v)$$

7.4.1.4 List Based Scheduling

While both the ASAP and the ALAP scheduling assume no constraints for the scheduling, the list based scheduling algorithm [PaPa88] handles resource constraints. This algorithm can therefore be used, if the number of operators to be scheduled onto a specific resource type in one time step may be greater than the number of available resources of this type.

List based scheduling can be seen as a generalization of ASAP scheduling, and gives the same result as ASAP scheduling, if the resources are available in sufficient numbers. The basic approach consists in assigning operators to resources in each time step, until all resources are occupied. Then, the algorithm proceeds to the next time step. To select operators to be assigned, the algorithm uses a list of operators, which are ready to be scheduled in the time step. An operator is ready, if all its predecessors are already scheduled, and whenever an operator is scheduled, new operators may become ready. The list is sorted by a priority function, so operators with higher priority get scheduled earlier than those with lower priority.

There are two subtypes of the list based scheduling algorithms, which differ in the way the priorities are handled. In static list based scheduling, the priorities are assigned to the operators once at the beginning and do not change during the process. In dynamic list based scheduling, the priorities are recalculated in each time step. This allows to consider priorities, which depend on the current time step in the scheduling, like for example a deadline for an operation. An outline of the list based scheduling algorithm is given in figure 7-19.

7.4.1.5 MA-DPSS Scheduling Algorithm

The scheduling for the MA-DPSS resembles basically a dynamic list based scheduling as described above in section 7.4.1.4. However, some adaptations have been made to handle edge ports and backbuses as new constrained resources. In these adaptation, the MA-DPSS approach differs also from the DPSS scheduling. The algorithm uses is described in detail in the following.

The scheduling takes place in four phases. First, the global bus is scheduled, ignoring the restrictions implied by the row and column backbuses and the edge ports. In the second and third steps, the backbuses and the edge ports are scheduled based on the results of the previous phase. Finally, the global bus is

```
ALGORITHM List Based Scheduling
{     TimeStep = 1 ;
      ReadyList = ∅ ;
      DO
      {     /* Only for dynamic scheduling: calculate priorities */
            CalculatePriorities( V ) ;
            /* Determine ready operators and sort the ReadyList */
            R = FindReadyOperators( V ) ;
            V = V - R ;
            ReadyList = SortByPrio( ReadyList ∪ R ) ;
            /* Distribute ready operators onto available resources */
            FOR r=1 TO NumResources
            {     /* Schedule the next operator in the list onto r */
                  IF ( ReadyList != ∅ )
                  {     v = ListFirst( ReadyList ) ;
                        start_LIST( v ) = TimeStep ;
                        Resource[ r, TimeStep ] = v ;
                        ListDelete( ReadyList, v ) ;
                  }
            }

            TimeStep = TimeStep + 1 ;
      }     WHILE ( V != ∅ || ReadyList != ∅ ) ;
}     /* End of algorithm List Based Scheduling */
```

Figure 7-19: List based scheduling algorithm

scheduled once more, this time considering the transfer times of the backbuses and the edge ports. This general approach for the scheduling is depicted in figure 7-20a.

The algorithm for the four phases is the same, except that for the global bus a program for the control unit is generated, while for the backbuses parameters according to the architecture described in section 6.3.3 are produced. For the edge ports, a file for each port group (see section 6.4.4) is generated describing the read and write transfers of each group member. A single scheduling phase as sketched in figure 7-20b will be described in the following, using the first global bus scheduling as an example.

At the beginning, a list of all data transfers to be scheduled is generated. In the example first phase, these are the global bus transfers and the I/O transfers over the memory bus attached to the global bus. Then, the ASAP and ALAP times for all operators are determined using the approach described in section 7.4.1.1 and

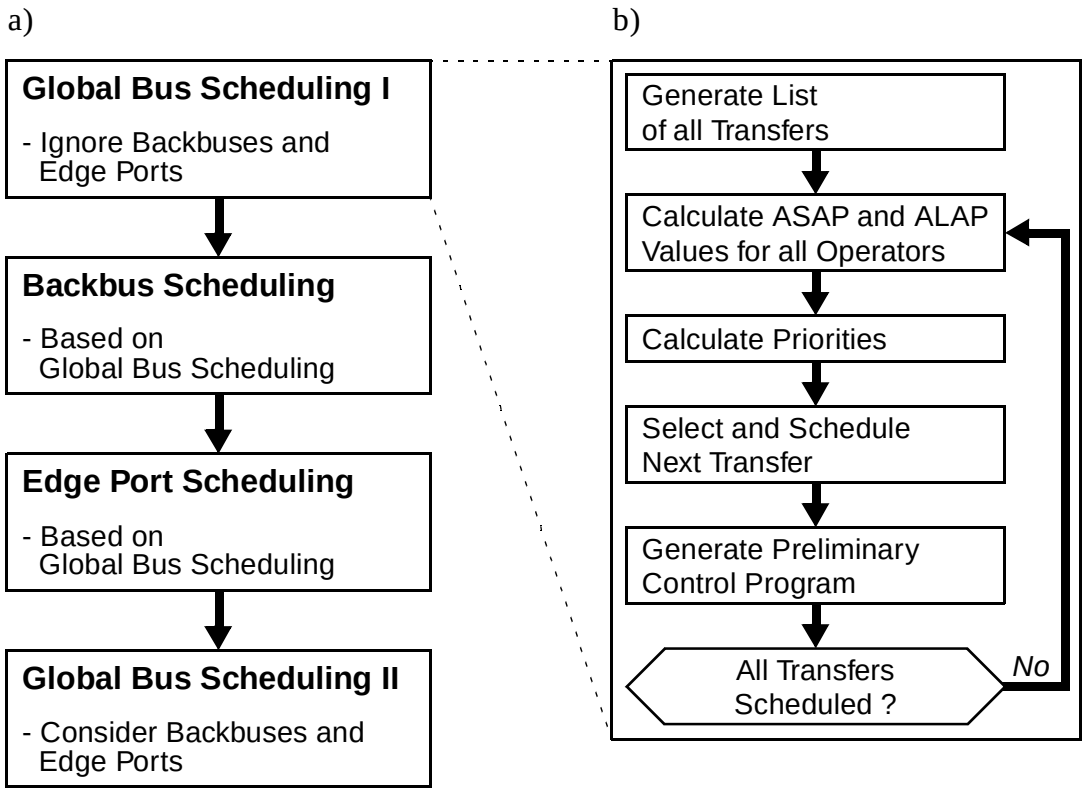

Figure 7-20: MA-DPSS Scheduling:
a) Total approach showing four phases
b) Dynamic list based scheduling (section 7.4.1.4) used in each single phase

section 7.4.1.2 respectively. Based on these values, a priority for each data transfer is calculated as described below. The transfer with the highest priority is then scheduled, allowing to generate a preliminary control sequence. The process iterates by calculating the ASAP and ALAP times of the operators again, under consideration of the preliminary control program, which is enhanced in each iteration. The scheduling is finished, when all data transfers are scheduled this way.

Transfer Priorities. The scheduling is controlled by the priority of the transfers, which expresses the urgency of a specific transfer to be scheduled. Several approaches for this priority function exist, for example the mobility (see section 7.4.1.3), which leads to a scheduling where the items with the least flexibility are scheduled first. This approach is also used in the historic DPSS scheduler. For the MA-DPSS, the priority is based on the ALAP times of the operators involved, or, if those happen to be equal, on the according ASAP times.

Additionally, the priorities have to satisfy conditions implied by the enhanced architectural possibilities. Those condition for the calculation of the priority for a specific transfer can be expressed by the following rules:

- Transfers for data input have to have a higher priority than those for data output so the general data processing sequence is retained.
- Transfers implementing a feedback line for datapath cycles, as described in section 6.1.3, have a lower priority than other transfers, as the linear part of the cycle has to be traversed once before the feedback can happen.
- If the optimizations loop folding or vectorization (see section 7.4.2) have been performed before, the priorities have to consider the timing dependencies between the operators in subsequent iterations.

7.4.1.6 Statistic Data Output

Due to its role as a performance estimator for the exploration process, the scheduler generates statistic data, which is then processed by the analyzer described in section 8.3. The following information is derived from the final schedule:

Datapath Latency. This value specifies the estimated performance of the datapath and is expressed in the number of execution cycles. This number is derived from the latency of the final schedule. If the datapath contains loops, the latency assumes all loops to be iterated only once, which results in an inexact estimation, which is still adequate for comparative purposes.

Average Waiting Time. Due to the transport triggered execution model (see section 6.1.2), an operation will wait until all operands are available at the inputs. The time from the arrival of the first operand until the start of the execution of the operator is denoted as the waiting time of this operator. The scheduler measures the average waiting time of all operators.

Longest Path. The scheduler identifies the path built from nearest neighbor links with the longest latency and outputs the delay of this path as well as the source and target rDPU. The longest path latency is an important measure for performance oriented exploration.

List of Unused rDPUs. This information is only generated, if a vectorization optimization (see section 7.4.2.1 below) has been performed before the scheduling. As this optimization leads to a use of less rDPUs than specified in the intermediate file, the scheduler produces as output the number of original rDPUs, the number of rDPUs in use after the vectorization, and a list of the rDPUs, which are no more needed.

7.4.2 Optimizations

As described above, the scheduler provides a performance estimation and other statistic data for the exploration process. This estimation is normally based on the mapping provided as input to the scheduler. However, the tool is able to produce additional performance estimations for other scenarios, which are achieved by applying certain optimizations to the datapath, thus typically overriding the mapping. These optimizations are partially bound to certain assumptions of the KressArray architecture.

The alternative implementations provided by the optimizations allow to trade performance for rDPU usage and vice versa. Thus, the exploration space is extended beyond the design space of the KressArray architecture to the datapath implementation, as the statistics can be generated also for datapaths optimized for time or for rDPU usage. Being adapted from the original scheduler [Kres96], the optimizations supported by the MA-DPSS scheduler are vectorization, loop folding, and loop unrolling. They will be described briefly in the following sections. Except from loop folding, these optimizations require a change to the datapath to actually apply them, and thus a remapping.

7.4.2.1 Vectorization

Vectorization tries to identify several datapaths doing the same computation. These datapaths are then implemented only once, processing the data sets of the original datapaths in a serial way. Thus, the number of rDPUs is reduced by decreasing parallel execution and thus increasing the latency of the alternative datapath. The vectorization optimization is illustrated by a simple example in figure 7-21.

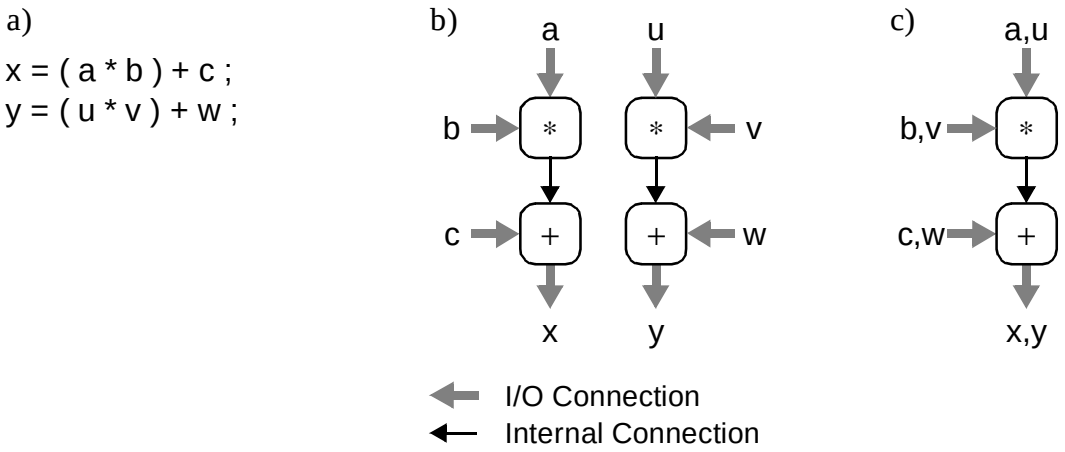

Figure 7-21: Example for vectorization:
a) Computation
b) Implementation with two parallel datapaths
c) Single datapath after vectorization

The vectorization operation changes the paths of the data words to be processed in the array and assumes a possibility to serialize the former parallel data I/O. Thus, in order to enable vectorization and allow a reasonable performance estimation, the I/O connections from and to the datapaths have to be independent of the mapping. This is the case, if all I/O connections are implemented by global bus transfers.

7.4.2.2 Loop Folding

The loop folding optimization improves performance without needing additional rDPUs. It affects only the control code for the data I/O, does not require a remapping, and is thus a popular technique. Assuming the KressArray is used to process data streams, which are fed into the array by an external loop, loop folding implements pipelining over the loop boundaries by starting on the next set of data during the processing of the current set. Loop folding is illustrated by a simple example in figur e7-22.

The figure shows the schedule before and after loop folding have been applied. For the sake of simplicity, the following assumptions are made: All I/O transfers have to be serial due to the use of a common communication resource or memory bus, an I/O transfer takes one time step, an addition two, and a multiplication five time steps.

As can be seen in the figure, one iteration of the computation $x = (a * b) + c$, which is depicted, takes ten time steps. In the original schedule, two consecutive iterations of the datapath will be executed like shown in figure 7-22a, with the second iteration starting after the first one has completely finished, thus taking 20 time steps together, or generally $10 * n$ steps for n iterations. However, part of the second iteration may as well start while the first iteration uses the multiplier rDPU. Thus, the iterations partially overlap like shown in figure 7-22b, yielding 15 steps for two consecutive operations, or $(5 * n) + 5$ time steps for n iterations, with the constant 5 steps being the time to fill the pipeline.

In general, the actual effect of loop folding is dependent on the datapath, as no two computations or I/O operations of different iterations may overlap. Furthermore, no extern dependencies between the data words of consecutive iterations are allowed to exist, as a new data set is read while the result of the previous iteration is not yet issued, as shown in figur e7-22b.

As there is no change in the number of rDPUs, the mapping itself is not affected by loop folding. The schedule for the I/O operations is different than for the unfolded case, including the necessity for extra code to fill and empty the pipeline.

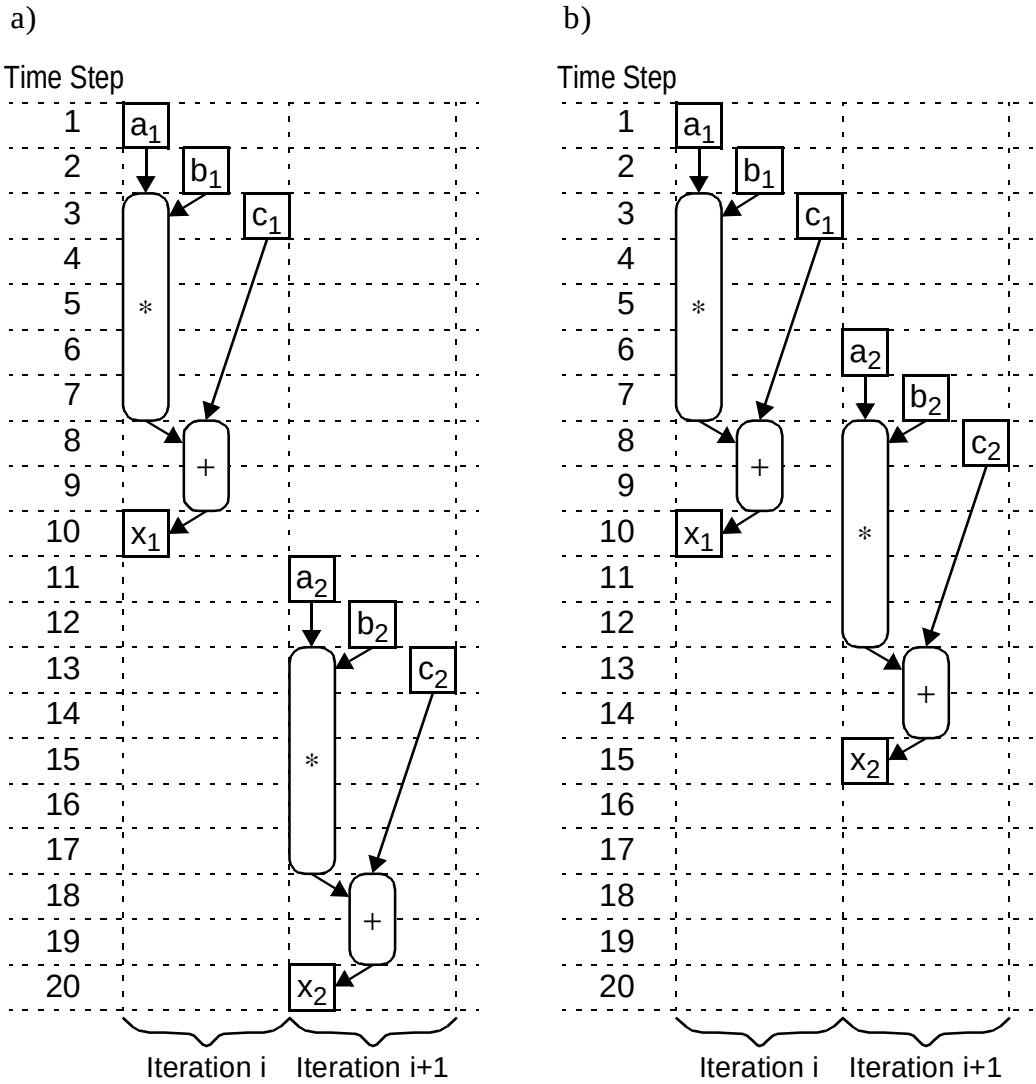

Figure 7-22: Example for loop folding:
a) Original schedule
b) Schedule with loop folding applied

7.4.2.3 Loop Unrolling

While loop folding requires that the operations of consecutive iterations do not overlap, loop unrolling relieves this restriction by providing multiple copies of the same datapath, to which the data words of consecutive iterations are fed. Thus, if the KressArray is again assumed to be provided with data by a loop, several loop iterations are executed in parallel, reducing the latency of the schedule. Loop unrolling is illustrated in figure 7-23.

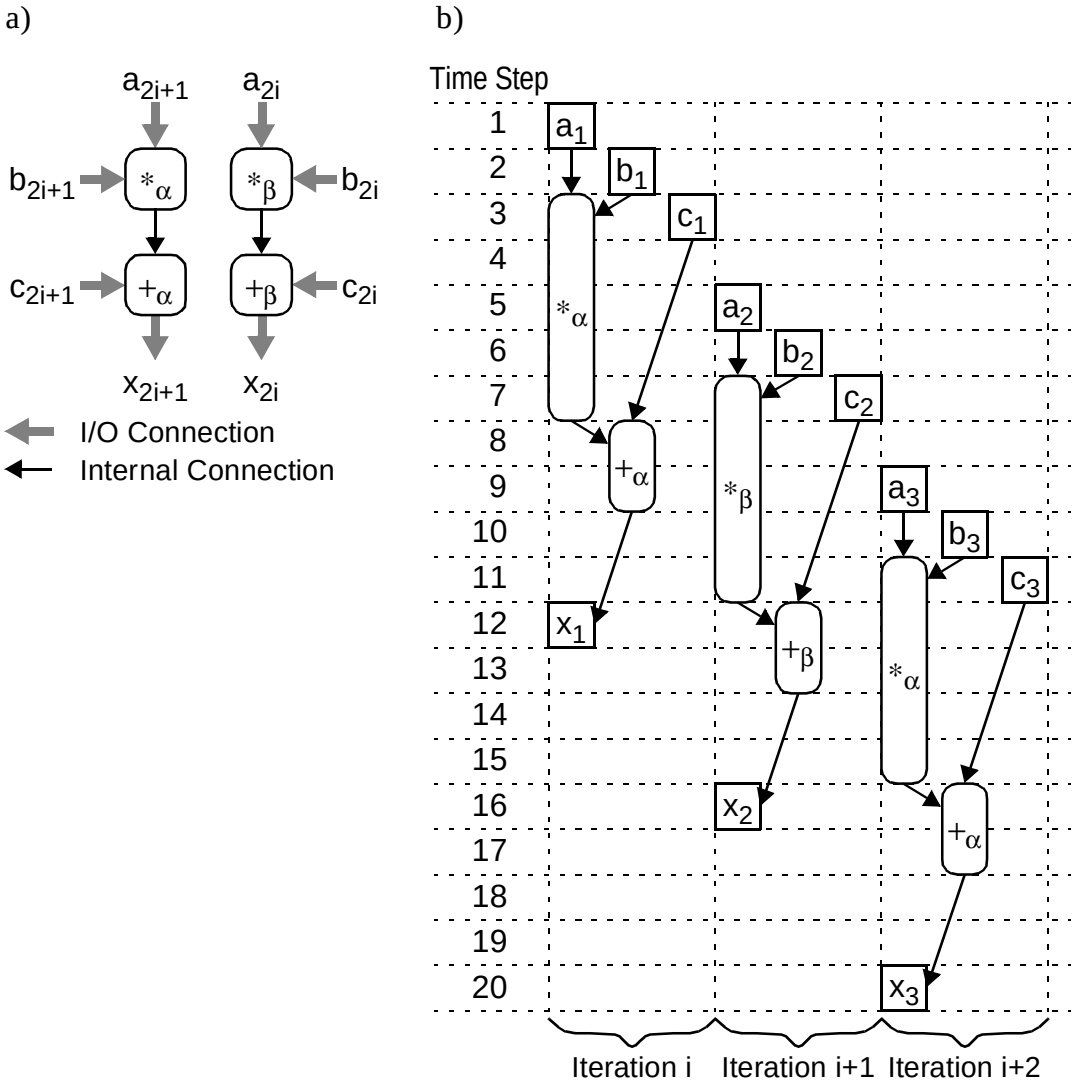

Figure 7-23: Example for loop unrolling:
a) Two identical datapaths for the computation
b) Schedule showing three iterations with alternate use of the datapaths

The example shown in the figure is the same as in figure 7-22, and the same assumptions about timing and I/O restrictions are made. The datapath is duplicated, using twice the number of rDPUs than the original implementation, as depicted in figure 7-23a. The data words are fed to alternate datapaths as shown in the schedule in figure 7-23b. Due to the parallel use of the datapaths, three iterations take 20 time steps compared to 30 for the non-optimized datapath shown in figure 7-22a. For the shown three iterations, both loop folding and loop unrolling yield the same latency of 25 time steps, according to figure 7-23b and the formula for the latency of the loop folding in section 7.4.2.2. However, the general expression for the delay of the solution with loop unrolling applied is

obviously (4 * *n*) + 8 for *n* iterations, as can be derived from figure 7-23b. Thus, for greater numbers of iterations than three, the solution with loop unrolling features better performance than the one with loop folding.

Like vectorization, loop unrolling affects the mapping in a significant way, so the I/O operations have to be independent of the mapping in order to get reasonable performance estimations. Thus, it is assumed, that the I/O runs over the global bus. Like loop folding, the loop unrolling operation requires extra code to fill and empty the pipeline. Also, there must be no extern data dependencies between the data words.

While the example in figure 7-23 shows loop unrolling with one copy of the datapath, it is also possible to unroll more than two iterations, resulting in more parallel datapaths. However, the amount of the possible performance increase is dependent on the computation to be implemented. In the example, no further performance improvement can be gained, if more than two instances of the datapath are used, as the I/O is occupied in every time step when the pipeline is filled once.

7.4.2.4 Memory Cycle Optimization

Another type of optimization is restricted to MoM-like architectures [Rein99] with a memory interface attached to a global bus controller, as shown in figure 7-16. The memory cycle optimization can be applied together with loop folding or loop unrolling to save memory accesses. The basic principle lies in identifying data words of consecutive iterations, which are located on the same physical memory address. Such data words have to be read from memory only in the first iteration and are then stored in a register file contained in the controller unit. For future iterations, the data words are read from the register file, which acts as a deterministic cache. Thus, the according memory cycles are saved. This optimization relies on the scanwindow and handle offset information provided in the ALE-X file (see section 7.1.1). The memory cycle optimization is described in more detail in [Kres96] and [Haas99].