R. W. Hartenstein, et al.: A Datapath Synthesis System for the Reconfigurable Datapath Architecture; Asia and South Pacific Design Automation Conference, ASP-DAC'95, Nippon Convention Center, Makuhari, Chiba, Japan, Aug. 29 - Sept. 1, 1995

A Datapath Synthesis System for the Reconfigurable Datapath Architecture

Reiner W. Hartenstein, Rainer Kress

University of Kaiserslautern

Erwin-Schrödinger-Straße, D-67663 Kaiserslautern, Germany

Fax: ++49 631 205 2640, email: abakus@informatik.uni-kl.de

Abstract — A datapath synthesis system (DPSS) for the reconfigurable datapath architecture (rDPA) is presented. The DPSS allows automatic mapping of high level descriptions onto the rDPA without manual interaction. The required algorithms of this synthesis system are described in detail. Optimization techniques like loop folding or loop unrolling are sketched. The rDPA is scalable to arbitrarily large arrays and reconfigurable to be adaptable to the computational problem. Fine grained parallelism is achieved by using simple reconfigurable processing elements which are called datapath units (DPUs). The rDPA can be used as a reconfigurable ALU for bus oriented systems as well as for rapid prototyping of high speed datapaths.

I. Introduction

Many computation-intensive algorithms take too much execution time, even on a well-equipped modern workstation. This opens a market for hardware accelerators of all kinds. Custom configurable accelerators have the advantage to be adaptable to the computational problem. Such an accelerator should be scalable. This means that it should be extensible to various sizes depending on the computational needs. Custom computing machines (CCMs) [2] provide such extensibilities. CCMs are based on SRAM based field-programmable gate arrays (FPGAs) [1]. Most available FPGAs are configurable only at bit-level to support both random logic as well as datapaths. Consequently they provide only modest performance and capacity with 32-bit datapaths.

The reconfigurable datapath architecture (rDPA) provides higher throughput and more area efficiency for implementation of such wide datapaths than FPGAs available commercially. Therefore it can be used as a basis for building word-oriented CCMs. The rDPA is in-circuit reconfigurable, and it is scalable to nearly arbitrarily large arrays. A controller allows to use the rDPA as a data-driven reconfigurable ALU (rALU). This rALU is intended for the parallel and pipelined evaluation of complete expressions and statement sequences. In scientific computations such a statement sequence or statement block usually occur in loops. The loop is controlled by multiple *for*-statements evaluating a statement block several times (fig. 1). The control of these loops can be performed by a host or a special address generator like in the Xputer [7].

For a high user acceptance, a synthesis system should be available that is able to map the statement blocks onto the rDPA without manual interaction. Such a synthesis system, the datapath synthesis system (DPSS) is presented in this paper. The following section sketches the reconfigurable datapath architecture. Section III explains how this architecture can be used as data-driven rALU on a bus oriented system by describing the rALU controller interface. The datapath synthesis system is introduced in section IV. Finally the paper is concluded.

```
for (j = 0, …, …)                        (1)
    for (i = 0, …, …) {                  (2)
        /* statement block */            (3)
        y[i] = a + b[i] * c[i-1] …;      (4)
        if (a < 3) …;                    (5)
        …                                (6)
    }                                    (7)
```

Fig. 1. Statement block in two nested *for*-loops

II. Reconfigurable Datapath Architecture

The reconfigurable datapath architecture (rDPA) consists of reconfigurable processing elements, a sophisticated kind of configurable logic blocks which we call datapath units (DPUs). Connecting an array of two by four rDPA chips on a PCB board, an array of 128 DPUs can be realized. The DPUs are interconnected by the routing resources (fig. 2). Both are described in the following.

Datapath unit architecture. The rDPA consists of a regular array of identical datapath units (DPUs). Each DPU has two input and two output registers. The operation of the DPUs is data-driven. This means that the operation will be evaluated as soon as all required operands are available. An extensible repertory of operators for each DPU is provided by the datapath synthesis system from a DPU library. This operator repertory includes the operators of the programming language C. The architecture of the DPUs consists of a datapath including an ALU and a microprogramable control unit. Operators such as addition, subtraction or logical operators can be evaluated directly, and larger operators like multiplication or division are implemented by a microprogram sequence. New operators can be added with the aid of a microassembler.

Routing architecture. The rDPA provides two interconnection levels: short lines for local interconnect, and long lines for global interconnect. The topology of an interconnection network can be *static* or *dynamic*. Static networks are fixed during run-time. Dynamic networks can be changed during run-time. Normally in commercial FPGAs all routing resources are static, in the rDPA local interconnections are static and global interconnections are dynamic.

The local interconnect of the rDPA is implemented as a mesh. A mesh compared to other array structures is best suited regarding I/O requirements and scalability. Further it allows to execute systolic algorithms efficiently, since these algorithms mainly use the local interconnect. Although bidirectional communication is more flexible in implementing expressions, a unidirectional approach is used for better area efficiency. A problem occurs with the integration of multiple DPUs onto an integrated circuit because of the high I/O requirements of the processing elements. To reduce the number of input and output pins, a serial link is used for data transfer between neighbouring DPUs on different chips. Internally the full datapath width is used. For the user and the software this serial link is completely transparent.

The global interconnect should provide a connection to each datapath unit (DPU). In the rDPA it is used for I/O of

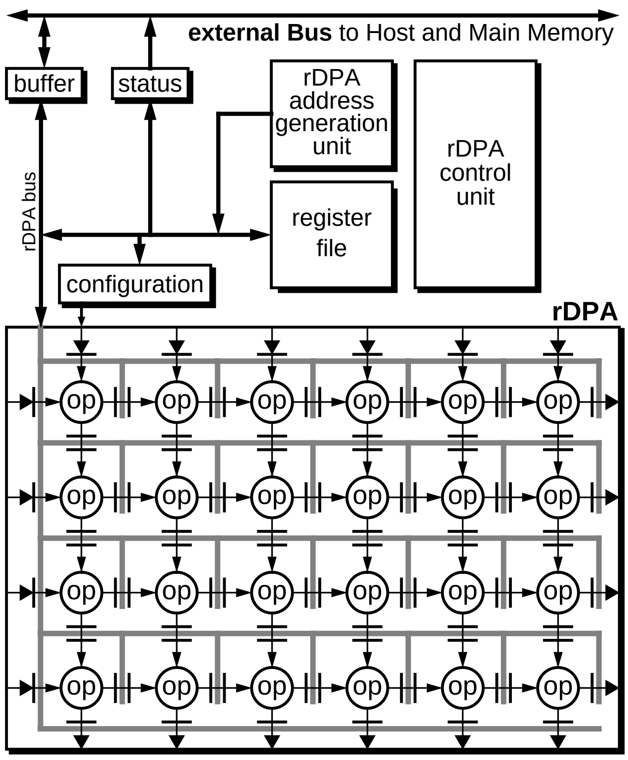

Fig. 2. The reconfigurable datapath architecture (rDPA) with the programmable rALU controller

operands from outside into the array, and for propagation of interim results to other DPUs far away. To save area, a time multiplexing of the global interconnect is considered. A scheduling can determine a good usage of this dynamic network. A bus can be used for such a dynamic interconnect network. A single I/O bus is sufficient to connect all datapath units. Two buses can speed up I/O operations, especially when each DPU has access to both buses. The communication is controlled an external control unit. The data transfers are synchronized data-driven by a handshake like the internal communications.

With the proposed routing architecture, the rDPA can be expanded also across printed circuit board boundaries, e.g. with connectors and flexible cable. Furthermore it is possible to build a torus structure.

Configuration. The array is scalable by using several chips of the same type. The DPU address for register addressing of the bus is configured at the beginning. The communication structure allows dynamic in-circuit reconfiguration of the rDPA. This implies partial reconfigurability during run-time. The rDPA can be configured serially from any serial link at the array boundary. One link is sufficient for the complete array, but multiple ports can be used to save time. With the rALU controller also parallel configuration for bus-oriented systems is possible. The configuration is data-driven, and therefore special timing does not have to be considered.

III. The rALU Controller

With the rDPA, a programmable rALU controller for bus-oriented systems is provided. Both, the rDPA and the rALU controller form a data-driven reconfigurable ALU (rALU). The rALU controller consists of a rDPA control unit, a register file and an address generation unit for addressing the DPUs (fig. 2).

Register file. It is useful for optimizing memory cycles, e.g. when one data word of a statement will be used later on in another statement. Then the data word does not have to be read again from the main memory. In addition, the register file makes it possible to use each DPU in the rDPA for operations by using the rDPA bus for routing. Currently the register file has 64 34-bit registers (32-bit data, 2-bit status).

Address generation unit. It delivers the address for the DPU registers before each data is written into the rDPA over the bus.

rDPA control unit. It holds a program to control the different parts of the data-driven rALU. The instruction set consists of instructions for loading data into the rDPA array to a special DPU from the external units, for receiving data from a specific DPU, or branches on a special control signal from the host. The control program is loaded during configuration time.

A status can be reported to the host to inform about overflows or to force the host to deliver data dependent addresses. An input and output buffer decouples the rDPA bus from the external bus.

IV. The Datapath Synthesis System

The datapath synthesis system (DPSS) allows to map statements from a high level language description onto the rDPA. The statements may contain arithmetic or logic expressions, conditions, and loops, that evaluate iterative computations on a small number of input data. The input language of the DPSS is called ALE-X (arithmetic and logic expressions for Xputers). It can be edited manually, or it can be generated from the X-C compiler in the Xputer software environment [9].

The task of configuring the rDPA is carried out in the following phases: logic optimization and technology mapping, placement and routing, and I/O scheduling. Partitioning of the statements onto the different rDPA chips is not necessary since the array of rDPA chips appears to be single large array of DPUs with transparent chip boundaries. The DPSS produces an assembler file whereof the configuration files for the rALU and the sequencing file are generated. The sequencing file determines the sequence of input and output data requested or produced by the rALU. The rALU code generation produces a control file for programming the rALU controller and an rDPA code file for the configuration of the reconfigurable datapath architecture. An overview on the datapath synthesis system is given in fig. 3.

A. The ALE-X Programming Language

To simplify the programming by hand, the ALE-X programming language should be easy to read, learn, and understand. For this reason, the language is strongly oriented on the concepts of the programming language C. The general form of a rALU subnet description is listed in fig. 4. It contains a structural part where the interface of the circuit is specified. This means the input and output (I/O) ports, namely the input and output variables as well as their data format is expressed. Then a procedural part describes the sequential code consisting of expressions, condition statements and loop statements.

The *rALUsubnet_name* gives the name of the rALU subnet. Each name of a rALU subnet must be unambiguous. The *variable_declarations* lists the type of the variable and their

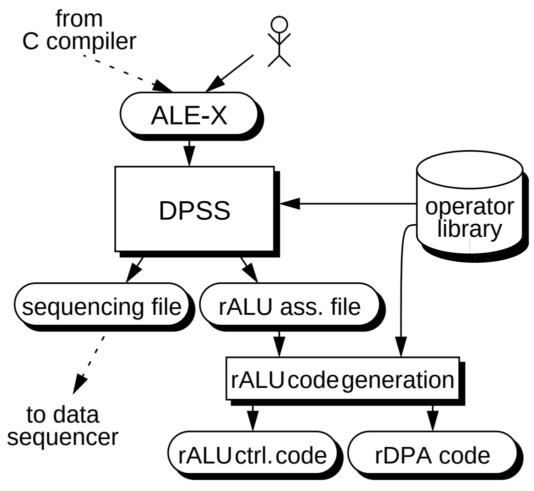

Fig. 3. Overview on the datapath synthesis system (DPSS)

name. The type must be a valid type defined in the ALE-X hardware file. This hardware file lists the limits and restrictions of the hardware. The body of a rALU subnet description has a procedural semantics. First local variables may be declared and constants may be defined. Then statements follow that can consist of assignment statements, conditional statements (*if-else*) and loop statements such as *while-loops* and *do-while*-loops. It is not allowed to use indices for addressing the variables, that means the loops work on local data which is read once. The loops are controlled by the rALU controller, that means only the statements and the condition are implemented onto the rDPA. An example of a body of a rALU subnet description which computes the square root of a variable *x* is given in fig. 5.

The ALE-X Hardware File. This file allows to use the datapath synthesis system in a very flexible way. If the hardware of the rALU changes, e.g. new operators are available, a new hardware file has to be specified only. Such a file consists of the following parts: The limits of the hardware resources (e.g. number of datapath units available in the rALU subnet) are used to check a given ALE-X file if it can be implemented on the current available hardware. The data types and the operators available are specified. The operators are listed with their delay times. All listed operators must be accessible in the operator repertory of the assembler for the code generation. User defined functions can extend the available operator library. Such functions can be scan primitives [5], trigonometric functions, square root, etc.

B. *Logic Optimization and Technology Mapping*

The condition statements are converted into single assignment code. The same is done for the loop conditions. The loop itself is controlled by the rALU controller. Loops and sequences of assignments are considered as basic blocks. Directed acyclic graphs (DAGs) are constructed from the basic blocks. Herewith common subexpressions, identical

```
rALUsubnet rALUsubnet_name ()              (1)
   variable_declarations                   (2)
{                                          (3)
   body of the rALU subnet description     (4)
}                                          (5)
```

Fig. 4. General form of a rALU subnet description

```
int   i;                                  (1)
y = 0.22 + 0.89 * x;                      (2)
i = 0;                                    (3)
while ( i < 4 ) {                         (4)
  y = 0.5 * ( y + x / y);                 (5)
  i = i + 1; }                            (6)
```

Fig. 5. Example of a body of a rALU subnet description

assignments, local variables, and dead code are removed. Further constant folding and reduction in strength is used. Unary operators are combined with the next operator if the operator library provides this new merged operator. This step reduces the number of required DPUs in the rDPA array. Further parallelism of single expressions is increased by tree-height reduction [3]. This is done to a level such that an expression can be placed with at most a single routing operator per local interconnection. A simple algorithm is performed which uses the commutativity and the associativity of some operators. If expressions can be vectorized, it is sufficient to implement one of these expressions, thus saving area. The required results can be computed by pipelining this single expression.

C. *Placement and Routing*

A poor placement degrades the performance since some internal variables have to be routed via the internal rDPA bus. During that time the bus is blocked for other I/O operations. A simulated annealing algorithm is chosen which gives better results than a simple constructive cluster growth algorithm, especially when the rDPA is connected as a torus. Different torus structures can be considered. The simulated annealing algorithm belongs to probablistic algorithms which improve iteratively [8]. One additional routing operation per local connection is considered in each iteration step by the simulated annealing algorithm. This is usually sufficient since then expression statements that represent a fully balanced tree up to seven operators can be mapped onto the rDPA. If the tree is not balanced much more operators can be implemented. Larger connections are made via the rDPA bus. Thus the routing is considered during the placement. Unlike other simulated annealing algorithms, the minimum placement is always saved in each iteration step. The cost function of the annealing algorithm considers the chip boundaries, the required routing operators and with a high cost the connections via the rDPA bus. Fig. 6 shows the implemented algorithm.

First, an initial placement for the simulated annealing process is computed by placing the operators randomly onto the rDPA. Then the *COST*-function gives the complete cost of this placement. The cooling schedule is controlled by a linear function f(temp) = 0.95 * temp. Two positions for exchange are searched by the *SELECT_POSITION()*-function. Without argument, this function gives a random position of an operator in the rDPA array. With arguments (for the second position), the function results a random position of an operator in the neighbourhood of the position of the first argument. The second argument determines the range of this neighbourhood. A high argument allows to search for the new position in the whole array, a low argument allows to search only in a small neighbourhood. Usually the temperature of the cooling schedule is used for this argument. Further the number of iterations of the inner loop is decreased with the temperature. This can be done because only a local neighbourhood for exchange is

```
/* Algorithm Simulated Annealing for the DPSS */
placement = INITIAL_PLACEMENT();
actual_cost = COST(placement);
minimum_cost = actual_cost;
for (temp = START_TEMP; temp > FINAL_TEMP && actual_cost != 0;
SCHEDULE(temp)) {
  /* inner loop */
  for (i = 1; i <= MAX_ITERATION(temp) && actual_cost != 0; i++) {
    /* select two positions for exchange */
    position1 = SELECT_POSITION();
    position2 = SELECT_POSITION(position1, temp);
    /* change positions and compute cost that results from change */
    delta_cost = EXCHANGE(position1, position2);
    /* accept new placement if … */
    if (delta_cost < 0 || RANDOM(0, 1) < exp (-delta_cost / temp )) {
      /* new minimum cost, retain minimal placement */
      if ( actual_cost < minimum_cost ) {
        minimum_placement = placement;
        minimum_cost = actual_cost;
      }}
    else {
      /* return to previous placement */
      EXCHANGE(position1, position2);
    }}}
placement = minimum_placement;
```

Fig. 6. Simulated annealing algorithm used

considered at low temperatures. The *EXCHANGE*-function exchanges two positions of the current placement and results the difference of the costs between the old and new placement. Only the cost increase or decrease due to this exchange is computed. The rest of the operators in the rDPA array are not considered, for reasons of speed of the implementation. The *RANDOM()*-function results a random number in the range between the first and the second argument. An example of an expression statement sequence mapped onto the rDPA is given in fig. 7.

D. I/O Scheduling

Scheduling determines a precise start time of each operation. The start time must satisfy the dependencies between the operations which limit the amount of parallelization. Since scheduling determines the concurrency of the resulting implementation, it affects its performance.

The placement and routing step of the design implementation takes care that for each operation one datapath unit (DPU) for evaluation of the operation is available. The static local interconnect between the DPUs is responsible for transferring intermediate results to the following operation. That

```
x1 = x + dx;
u1 = u - 3 * x * u * dx - 3 * y * dx;
y1 = y + u * dx;
c = x1 < a;
```

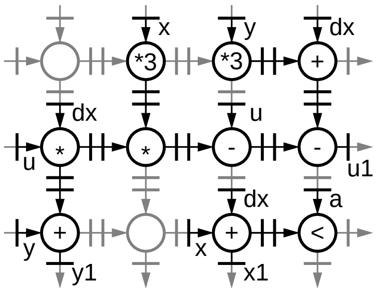

Fig. 7. Example of the placement of four expression statements onto the reconfigurable datapath architecture

means, the arithmetic and logic operations are not constraint by the resources. But two other resources are constraint: the rDPA bus and the external bus providing the data from the main memory.

- The external bus provides the rDPA with a regular data stream. This bus is limited to a single I/O operation per time.
- The rDPA bus is limited to a single operation per time too, though as a local bus, it can operate faster than the external bus.

Constraints can be classified into two major groups: *interface constraints* and *implementation constraints*. Interface constraints are additional specifications to ensure that the circuits can be embedded in a given environment. In our case, these are the timing constraints of the data stream via the external bus. Considering the complete rALU, the I/O constraints of the rDPA bus can be seen as implementation constraints. The rDPA bus is used for internal global routing via the register file, additionally to the I/O operations. These transfers to the register file and to the buffer register can be considered as operations in the schedule. The external bus adds timing constraints to these operations. Further, the sequence of the data words via the external bus has to be the same as the sequence of data words from the buffer register to the rDPA or vice versa. But in-between this sequence via the rDPA bus, getting and putting data words to the register file is allowed. Due to the data-driven synchronization concept of the rALU, the time steps for the scheduling algorithm are very short and most operations use multiple cycles for evaluation. The time step should be the largest common divisor of the delay times of all used operations.

To determine deadlock free schedule and an optimal sequence of the I/O operations, the following strategy is used:

- Perform a resource constraint scheduling of the *sequencing graph*, including the *get* and *put* operations for I/O, but without considering the timing restrictions of the external data stream.
- Since the sequence of operations is fixed now after the first scheduling, the timing constraints of the external data stream are included.
- Perform a timing constraint schedule including the requirements of the external data stream.

The goal of this scheduling is to find a deadlock free schedule and an optimal sequence of the I/O operations. First a minimum-latency resource-constraint scheduling is used. The sequencing graph consists of a set of operations including the I/O operations of the rDPA bus. The vertex set $V = \{v_i;\ i = 0, 1, 2, \ldots, n\}$ of a sequencing graph $G_S(V, E)$ is in one-to-one correspondence to a set of operations, and the edge set $E = \{(v_i, v_j);\ i, j = 0, 1, 2, \ldots, n\}$ represent dependencies. The source vertex v_0 and the sink vertex v_n with $n = n_{ops} + 1$ are both no-operations (NOP). The set of execution delays is $D = \{d_i;\ i = 0, 1, 2, \ldots, n\}$ with $d_0 = d_n = 0$. It is assumed that the delays are data independent and known. Further it is assumed, that the delays are integers and a multiple of the time step.The external bus is not considered at this time. Scheduling is done on basic blocks only. This means, that there is no difference between expression statements and conditions. The same is valid for *while* and *do-while* loops. They are considered by the code generation step. An example (fig. 7) of a sequencing graph of a basic block is listed in fig. 8.

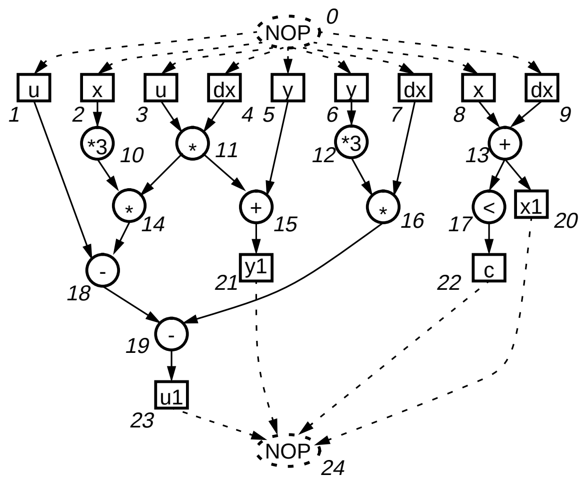

Fig. 8. Example of a sequencing graph

For the scheduling algorithm, a list schedule with improvement of the priority list in each iteration step like in the force-directed scheduling is used. The mobility determines the ranking of the operations. The resource constraints are represented by the vector **a**, in our case, only the I/O operations are required for this priority list ($a_{I/O} = 1$), since all other operations are not resource constraint. The operations whose predecessors have already been scheduled early enough, so that the corresponding operations are completed at time step *l* are called candidate operations $U_{l,k}$, with

$$U_{l,k} = \{ v_i \in V : type(v_i) = k \text{ and } t_j + d_j \; l \; \forall j : (v_j, v_i) \in E \} \qquad (1)$$

for any resource type k = 1, 2, …, n_{res}. Fig. 9 shows the scheduling algorithm used.

For simplicity it is assumed that each I/O operation has a delay of three time steps, a multiplication 22, and an ALU operation four. A multiplication with three is implemented as a shift followed by an addition, which requires six time steps in total. A first schedule determines the sequence of I/O operations via the rDPA. Thus the sequence of data words via the external bus can be fixed.

Each multiply used variable is read once from the main memory and then transferred into the rDPA array and concurrently to the register file. This transfer to both locations does not increase the delay time of the I/O operation. Supposing

```
I/O ($G_S$(V, E), $a_{I/O}$) {
  l = 1;
  do {
    /* schedule the I/O operations (k = 1) */
    compute the mobility of the I/O operations $_1 = t_1^L - t_1^S$
      from ASAP and ALAP schedule;
    determine canidate operations $U_{l,1}$;
    schedule the $U_{l,1}$ with lowest mobility;
    /* schedule the rest of operations (k   1) */
    for (resource type k = 2; k   $n_{res}$; k++) {
      determine canidate operations $U_{l,k}$;
      schedule the $U_{l,k}$ requiring no additional resources;
    }
    l = l +1;
  } while ($v_n$ is not scheduled);
  return (t);
}
```

Fig. 9. I/O scheduling algorithm minimizing latency under resource constraints

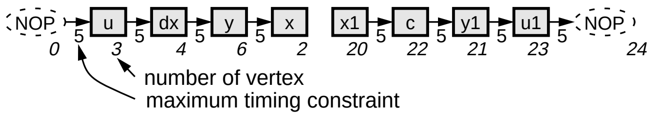

Fig. 10. Additional minimum timing constraints from the data stream via the external bus

that every five time steps a new variable can be transferred via the external bus gives the minimum timing constraint for the final scheduling. Fig. 10 shows the additional timing constraints introduced by the restrictions of the external bus.

Now a schedule which uses the same algorithm as before leads to the final I/O schedule. Due to the timing restrictions on the external bus, the free time slots on the rDPA bus are used for transfers from the register file to the rDPA registers. Fig. 11 shows the final I/O schedule which has a latency of λ = 73 time steps. From this schedule, the sequencing file can be determined. Further the schedule is deadlock free, that means the data-driven synchronization concept of the rALU will work with no problems.

E. Code Generation

The rDPA configuration file is computed from the placement information of the processing elements and a library with the microprogram code of the operators. The required code is found in the operator library. The sequencing file is computed from the sequence of data words via the external bus. The configuration file for the rALU controller is extracted from the final schedule of the I/O operators. It consists of move operations from and to the rDPA, buffer register and register file. An example schedule (the same as in fig. 11) of the I/O operations via the external bus and the rDPA bus is shown in fig. 12.

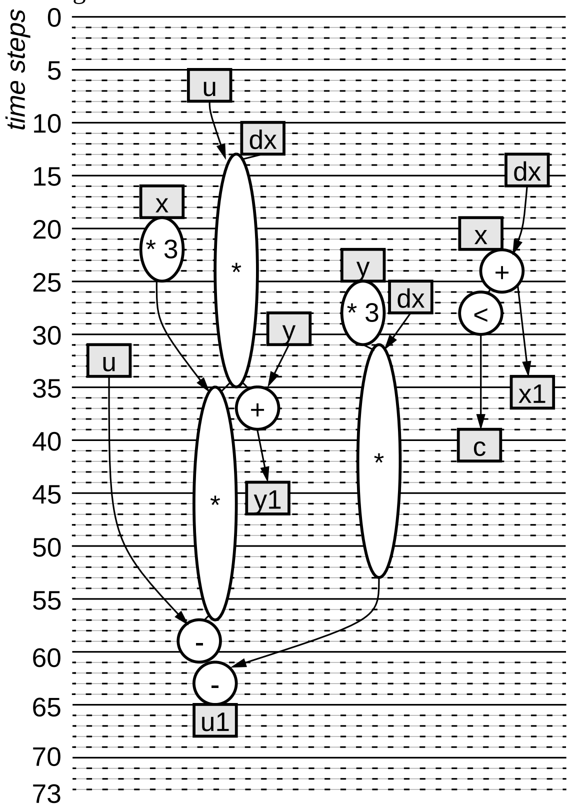

Fig. 11. Example of a final schedule for the rDPA taking care of the data stream via the external bus

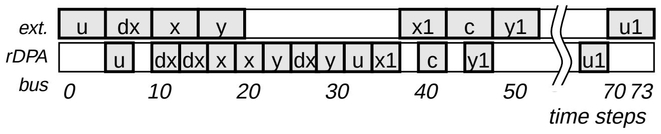

Fig. 12. I/O operations of the external bus and the rDPA bus

F. *Optimization*

Optimizations can be made in terms of area or speed. Area improvements can be achieved by pipelining vectorized statements. Speed improvements can be achieved by loop folding or loop unrolling [4] if the statement block in the rDPA is evaluated several times. This requires that these loops are controlled by an external address generator like in the Xputer [7] or that these loops are controlled by the host.

Vectorized Statements. Instead of mapping the vectorized statement a few times onto the rDPA, it can be pipelined. Adding timing constraints to force the sequence of vector operations in the sequencing graph leads to a constraint sequencing graph which can be scheduled as before.

Loop Folding. This the most popular technique for improving the speed of statement blocks that occur in inner loops since no additional hardware is required. If the implementation in the rDPA is not I/O bound, the operations can be pipelined across loop boundaries. Dependencies between two iterations are considered by the mapping and the scheduling algorithm. Two iterations are scheduled at once with additional timing constraints of the sequencing graph like in the vectorization example. Loop folding requires a special control signal of the address generator or the host to signal the end of the loop.

Loop Unrolling. With this technique a certain number of loop iterations is unrolled. This action results in a loop with a larger body but with fewer iterations. The larger loop body provides a greater flexibility for improving the speed of loop. Loop unrolling requires at least twice as much datapath units as without using this technique. Improvement against loop folding can be achieved in designs that are not bound by the I/O. Loop unrolling requires additional control especially if the number of operations is not a multiple of the loop iterations that are unrolled.

Fig. 13 shows a small example of loop folding and loop unrolling.The loop folding is bounded by the multiplication, whereas the loop unrolling is bounded by the hardware resources. Using more hardware, or combining with loop folding can further increase the performance.

V. Conclusions

A datapath synthesis system (DPSS) for the reconfigurable datapath architecture (rDPA) has been presented. The DPSS allows automatic mapping of high level descriptions onto the rDPA without manual interaction. The required algorithms of this synthesis system have been described in detail. Some techniques for optimization such as vectorization for area improvement, and loop folding or loop unrolling for speed improvement have been outlined. The rDPA is scalable to arbitrarily large arrays and reconfigurable to be adaptable to the computational problem. Fine grained parallelism is achieved by using simple reconfigurable processing elements which are called datapath units (DPUs). The rDPA can be

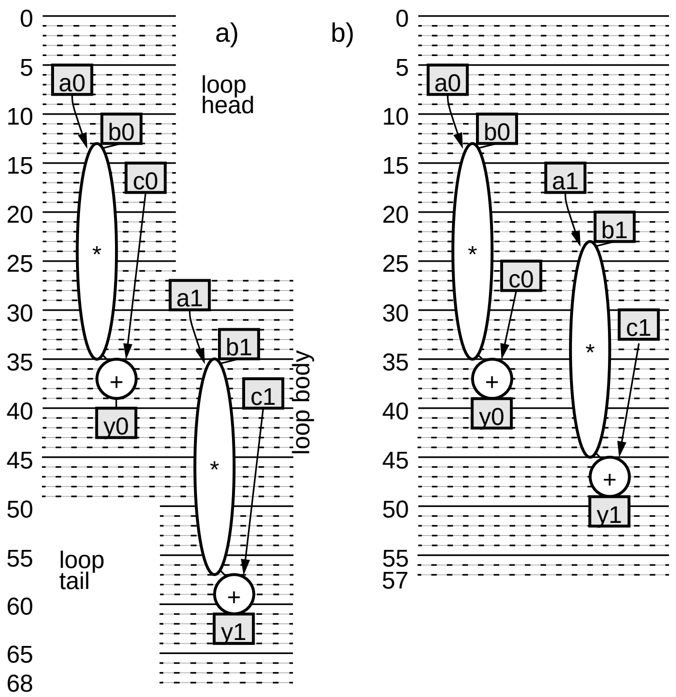

Fig. 13. Loop folding (a) and loop unrolling (b) of a small expression statement example

used as a reconfigurable ALU for bus oriented systems as well as for rapid prototyping of high speed datapaths.

A second implementation of the rDPA using standard cells with a datapath compiler will soon be submitted for fabrication. It provides 32 bit datapaths and arithmetic resources for integer and fixed-point numbers. The datapath synthesis system is completely specified. The optimization, the placement, as well as the scheduling have been implemented. The code generation is currently being implemented.

References

[1] S. D. Brown, R. J. Francis, J. Rose, Z. G. Vranesic: Field-Programmable Gate Arrays; Kluwer Academic Publishers, 1992

[2] D. A. Buell, K. E. Pocek: Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, IEEE Computer Society Press, April 1994

[3] G. De Micheli: Synthesis and Optimization of Digital Circuits; McGraw-Hill, Inc., New York, 1994

[4] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, S. Y.-L. Lin: High-Level Synthesis, Introduction to Chip and System Design; Kluwer Academic Publishers, Boston, Dordrecht, London, 1992

[5] S. A. Guccione, M. J. Gonzalez: A Data-Parallel Programming Model for Reconfigurable Architectures; IEEE Workshop on FPGAs for Custom Computing Machines, FCCM'93, IEEE Computer Society Press, Napa, CA, pp. 79-87, April 1993

[6] R. W. Hartenstein, J. Becker, R. Kress, H. Reinig, K. Schmidt: A Reconfigurable Machine for Applications in Image and Video Compression; European Symposium on Advanced Services and Networks / Conference on Compression Techniques and Standards for Image and Video Communications, Amsterdam, March 1995

[7] R. W. Hartenstein, A. G. Hirschbiel, M. Riedmüller, K. Schmidt, M. Weber: A Novel ASIC Design Approach Based on a New Machine Paradigm; IEEE Journal of Solid-State Circuits, Vol. 26, No. 7, July 1991

[8] N. A. Sherwani: Algorithms for Physical Design Automation; Kluwer Academic Publishers, Boston 1993

[9] K. Schmidt: A Program Partitioning, Restructuring, and Mapping Method for Xputers; Ph.D. Thesis, University of Kaiserslautern, 1994