A Scheduling and Allocation Method to Reduce Data Transfer Time by Dynamic Reconfiguration

Kazuhito Ito
Dept. Elec. Elect. Systems, Saitama University
255 Shimookubo, Urawa, Saitama 338-8570, Japan
kazuhito@ees.saitama-u.ac.jp

Abstract— In the era of deep submicron technology, wire delay on an LSI chip is becoming relatively larger than operation delay. Increase of execution speed by parallel processing may be limited due to the data transfer time between functional units. If we can dynamically reconfigure nearby functional units into desired operation type and execute operations on the reconfigured units, long data transfer is reduced and hence fast processing can be achieved. In this paper we propose a scheduling method to determine static operation execution time and functional unit allocation to achieve fast signal processing by considering dynamic reconfiguration of functional units. Results show the effectiveness of the proposed method.

1 Introduction

It has been reported that although gate delay becomes smaller as the LSI process advances, wire delay is becoming relatively longer than gate delay and will be dominant in total delay [1, 2]. To realize high performance LSIs, it is getting more important to consider not only gate delay but wire delay in high level designs.

Reconfigurable LSIs, such as reprogrammable field programmable gate arrays (FPGA) with rewritable lookup tables and interconnections, can be divided based on the reconfiguration scheme into two categories: statically-reconfigurable and dynamically-reconfigurable [3]. In the case of statically-reconfigurable, the entire operations on the LSI chip are terminated before reconfiguration. Once reconfiguration is completed, operations are initiated. On the other hand, in the case of dynamically-reconfigurable, only the operation executed on the part to be reconfigured is terminated. During the reconfiguration of that part, other part continues operations. Once the reconfiguration is completed, the reconfigured part and the other part together execute some operation.

In dynamically-reconfigurable LSI, it is not necessary to terminate all the operations during partial reconfiguration. Dynamically-reconfigurable LSI can gradually change the components while executing some operations simultaneously. Such dynamically- and partially-reconfigurable hardware has been developed [4, 5] and there have been some reports on applications and design environments for dynamically- and partially-reconfigurable hardware [5, 6, 7]. In the remaining of this paper, we assume that dynamic reconfiguration also implies partial reconfiguration.

When executing a digital signal processing algorithm by parallel processing, usually we need some functional units which are laid out two dimensionally on a LSI chip. Suppose that a result P of a multiplication m is added with another data by addition a. If the multiplication m is allocated to a multiplier M and the addition a to an adder A, the multiplication result P must be transfered from the multiplier M to the adder A. This data transfer is done through a wire on the LSI chip. If the destination adder A is placed far apart from the multiplier M, we need long wire delay from the multiplication to the addition. Consequently, high processing speed will not be achieved because of the wire delay.

In general, the function types and operation parallelism change as the signal processing algorithm goes on. The required type and number of functional units are not always the same. If there is a functional units F which is nearby the multiplier M and becomes idle a little after the multiplication m, we can reconfigure the functional unit F into an adder F_a and execute the addition a on F_a. Hence, we can reduce the long wire delay from a multiplier to an adder.

In this paper, we concentrate on designing dedicated hardware for an iterative processing algorithm like digital signal processing which consists of many primitive operations, such as additions and multiplications. We propose a method for scheduling operation execution and allocating operations to functional units to minimize the iteration period of the processing algorithm by reducing the wire delay by means of dynamic reconfiguration.

It must be noted that although the target hardware is reconfigured dynamically (i.e. run-time reconfiguration), when and which functional units are reconfigured is static and predetermined by the scheduling and allocation.

The paper is organized as follows. In section 2, target dynamically-reconfigurable hardware is modeled. The idea how higher speed is achieved with such hardware is also discussed in this section. The proposed scheduling method is based on range chart guided scheduling method [8], which is briefly reviewed in section 3. In section 4 the proposed scheduling and allocation method is described. The experimental results are shown in section 5.

2 Hardware Model and Reconfiguration

2.1 Reconfiguration of functional units

Reconfiguration of functional units is to alter the circuit connections so that operations of different types can be executed on the same hardware before and after the circuit alteration.

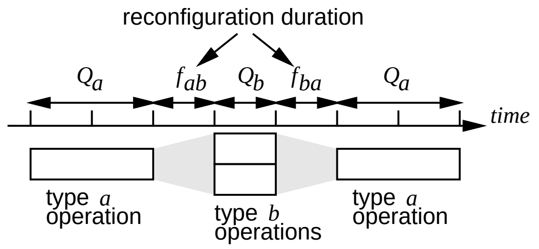


Fig. 1. Reconfiguration of functional units. ($N(O_b^a) = 2$)

In a general parallel multiplier, fast additions of partial products are executed by a carry save adder and a vector merging addition is executed by a carry propagation adder. By slightly modifying the connections, the carry propagation adder can be used to execute ordinary additions. Moreover, if the hardware amount of the carry save adder compares to the hardware amount of a carry propagation adder, it is possible to reconfigure the carry save adder into a carry propagation adder to execute ordinary additions. Consequently, it can be assumed that a parallel multiplier consists of sufficient amount of hardware to be reconfigured into at least two adders. If the reconfigurable multiplier be can reconfigured into two adders, we can execute at most two additions at the same time on the hardware when reconfigured into two adders. When those two adders are reconfigured back into a multiplier, we can execute a multiplication on the same hardware.

Let O_b^a denote a type of functional units which can exclusively execute operations of type a and b by reconfiguration. In addition, let $Q_a(O_b^a)$ and $Q_b(O_b^a)$ denote the execution time of operation type a and b, respectively.

To simplify the problem, we assume that a functional unit of type O_b^a can execute one operation of type a when configured into type a. We also assume that, when configured into type b, the functional units can execute at most $N(O_b^a)$ operations of type b at the same time. For example, if $N(O_A^M) = 2$ for a functional unit of type O_A^M, it can execute one multiplication (type M) when configured into a multiplier or at most two additions (type A) when configured into two adders.

The reconfiguration times are f_{ab} to reconfigure the functional unit from type a to type b and f_{ba} from type b to type a. During these reconfiguration time, the functional unit cannot execute any type of operations. This model is summarized in Fig. 1.

2.2 Data transfer model

In this paper, we assume an architecture as illustrated in Fig. 2 where many functional units are connected through data buses. We can use as many data buses as required. Therefore there are no limit on the number of simultaneous data transfers.

Let τ be defined to be the data transfer time between functional units which are physically adjacent to each other on an LSI chip. Data transfer time between functional units which are not adjacent to each other is relative to the physical distance between these functional units. For example in Fig. 2(a), functional units M_1 and M_2 are adjacent to each other. Therefore data transfer time from M_1 to M_2 and vice versa is τ. Data transfer time from A_1 to M_2 is 2τ because a functional unit M_1 resides between A_1 and M_2 and the distance between A_1 and M_2 is about twice longer than physically adjacent functional units.

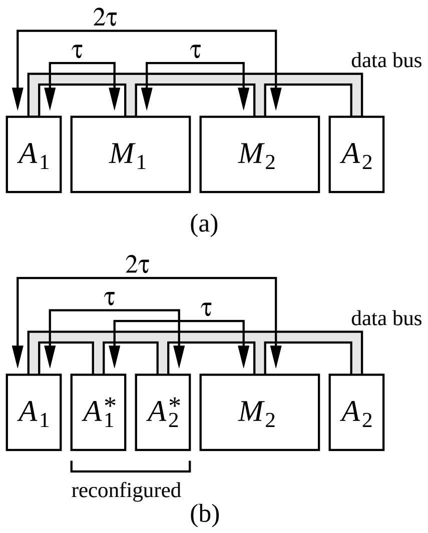


Fig. 2. Data transfer model. (a) when configured into a multiplier. (b) when configured into adders.

We assume that data transfer time between reconfigured adders and other functional unit is the same as the data transfer time before reconfiguration. For example in Fig. 2(b), A_1^* and A_2^* are adders derived by reconfiguring multiplier M_1. Since data transfer time between M_1 and A_1 is τ, data transfer time between either A_1^* or A_2^* and A_1 is also τ. Similarly, data transfer time between A_2 and M_1 is 2τ, data transfer time between A_2 and either A_1^* or A_2^* is also 2τ. Data transfer time between adders reconfigured from a multiplier is assumed to be τ. For example, data transfer time between A_1^* and A_2^* is 2τ because these adders are derived by reconfiguring an identical multiplier M_1.

For data transfer of long distance which require long data transfer time, we assign as many clock cycles as necessary to accommodate the data transfer time. Thus we can take into account the data transfer time, e.g. clock cycles for data transfer, with some accuracy during high-level design.

2.3 Data transfer time reduction by dynamic reconfiguration

Long data transfer time is necessary for data transfer between distant functional units. When the processing speed is slowed down because of the long data transfer time, we can use dynamic reconfiguration to reduce the data transfer time by reducing the distance between the functional unit which generates the data (data source) and the functional unit which consumes the data (data destination). This is achieved by creating a functional unit of the same type as the data destination nearby the data source and executing the data consuming operation on the created functional unit.

An iterative processing algorithm is given as a data-flow graph (DFG) $G = (N, E)$ as shown in Fig. 3(a). N denotes the set of nodes which represent operations. In the figure, the nodes a_1 from a_5 are additions and the nodes m_1 and m_2 are multiplications. E denotes the set of directed edges between operations. A directed edge represents data dependency from the source to the destination, and therefore the precedence relation between executions of these operations. In this paper, we assume an addition time, a multiplication time, and data transfer time τ between adjacent functional units are 1 unit of time (u.t.), 2 u.t., and 1 u.t., respectively.

Operation schedules determined by taking into account the data transfer time are shown in Fig. 3(b) and (c). If we are not allowed to use dynamic reconfiguration, the processing time becomes shortest when adders A_1, A_2 and multipliers M_1 are placed and operations are allocated to these functional units as shown in Fig. 3(b). Arrows in this figure represent data transfers between functional units. In the schedule shown in Fig. 3(b), 2 u.t. is required for data transfer from a_1 to a_4. Thus the total processing time from the beginning of m_1 to the completion of the final operation needs 8 u.t.

Here we assume the multiplier M_1 can be reconfigured into adders. The reconfigurable functional unit is denoted as M/A_1 in Fig. 3(c). If both the reconfiguration time from a multiplier to adders and from adders to a multiplier is the same and is 1 u.t., by reconfiguring M/A_1 into adders, the operation a_1 can be executed of a reconfigured adder. Since the hardware is dynamically-reconfigurable, the adder A_2 which is not reconfigured can continue additions. Consequently, data transfer time from a_1 to a_4 is reduced to only 1 u.t., and the total processing time is minimized to 7 u.t.

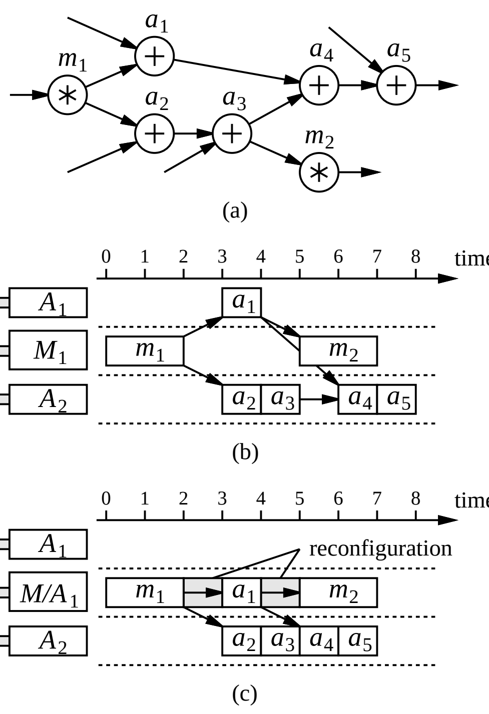


Fig. 3. Fast processing by dynamic reconfiguration. (a) processing algorithm. (b) schedule without dynamic reconfiguration. (c) schedule with dynamic reconfiguration.

3 Range Chart Guided Scheduling Method

The proposed scheduling method for dynamically-reconfigurable hardware is based on the range chart guided scheduling (RCGS) method [8]. In this section we briefly review the range chart guided scheduling method.

The basic idea of RCGS method is as follows: from unscheduled operations of the given iterative processing algorithm, we choose an operation and assign the operation an execution staring time within the *scheduling range* so as to minimize the maximum number of operations executed in parallel. When all operations are scheduled, the number of functional units necessary to execute the processing algorithm is minimized. *Scheduling range* of an operation is the set of time at which the execution of the operation could be initiated without violating any precedence relations. The larger the scheduling range is, the more probable to find the execution start time to minimize the number of functional units.

The precedence of operations only defines the relative relations among operation execution times. Scheduling ranges of nodes can be determined uniquely by fixing the execution time of one of the operations in the processing algorithm. This is called *a reference operation* and should be carefully chosen and fixed at time 0. Let R_i denote the scheduling range of operation i. The earliest and the latest time within R_i are respectively called the *lower bound* LB_i and the *upper bound* UB_i of the operation i. When an operation is scheduled (assigned the execution start time), the scheduling ranges of unscheduled operations will be changed. Therefore, each time an operation is scheduled, we must reevaluate the scheduling range of unscheduled operations.

The first priority to choose one of unscheduled operations is the tightness of the scheduling range. If operation j uses a result of operation i, the execution start time of j must be later than the execution completion of i. When the execution start time of i is set to UB_i, the lower bound of j (LB_j) is pushed toward the upper bound UB_j and hence R_j becomes smallest. In general, for an operation with small scheduling range, whichever time it is scheduled, the reduction of scheduling ranges of other operations would be small. Hence, it is preferable to choose the operation with the smallest scheduling range and schedule the operation first in order to minimize reduction of the schedule ranges of other operation.

The second priority to choose one operation among unscheduled operations is the existence of *fixed bounds*. The lower (upper) bound of the operation is said *fixed* if all the immediate predecessors (successors) of an operation are already scheduled. When LB_i is fixed, setting the execution start time of i to LB_i does not change the upper bounds of other operations since all the immediate predecessors are already scheduled. Hence, it is preferable to choose the operation with the fixed bound and schedule the operation at or close to the fixed bound in order to minimize reduction of the schedule ranges of other operation.

4 Scheduling and Allocation Algorithm

In order to minimize total time of operation execution, reconfiguration, and data transfer, we must allocate functional units to satisfy data transfer time requirements which should be obtained from scheduling ranges of operations. On the other hand, scheduling ranges of operations depends on the data transfer time between operations. Thus scheduling operations and allocating functional units are closely related. In this paper we propose a method to determine schedule and allocation in parallel.

4.1 Basic idea

From the set of operations of the given iterative processing algorithm, we choose an operation, i, among unscheduled operations. Then i is allocated to an appropriate functional unit and assign an execution start time by taking into account the reconfiguration time and data transfer time. This is repeated until all the operations are scheduled.

4.2 Choice of operation to schedule

The preference of choosing an operation among the set of unscheduled operations is as follows.

1. scheduling range is smallest
2. the lower bound or the upper bound is fixed
3. the operation time is longest

In the case a single operation cannot be identified by the step 3, we arbitrarily choose one operation among the candidates.

Let i denote the operation chosen. For each functional unit x of the set of already allocated functional units X, it is checked if i can be scheduled at time $t \in R_i$ on x. In the case that functional unit x is used by other operation or during reconfiguration at time t, operation i cannot be allocated to x. Moreover, if scheduling i at time t on x violates some precedence to/from i, then i cannot be allocated to x at time t.

4.3 Operation allocation

Let a functional unit x be of type O_b^a. When x is configured into a number of $N(O_b^a)$ functional units of type b, each functional unit is denoted as x_k $(k = 0, 1, \ldots, N(O_b^a) - 1)$. The set of functional units X is the collection of functional units to be used to execute operations. The contents of X varies time to time by dynamic reconfiguration. Namely, when functional unit x is configured into type a, X does not contain functional units x_k $(k = 0, 1, \ldots, N(O_b^a) - 1)$. On the other hand, when x is configured into type b, x is removed from X and x_k $(k = 0, 1, \ldots, N(O_b^a) - 1)$ are included instead.

Let i denote an operation of either type a which can be executed by x or type b which can be executed by x_k.

When i is type a, the conditions where the functional unit x can execute i at time t are: no operation of type a is scheduled to x at any time $t1$ which satisfies $t - Q_a(O_b^a) + 1 \leq t1 \leq t + Q_a(O_b^a) - 1$; and x is not reconfiguring at any time $t2$ which satisfies $t - f_{ba} + 1 \leq t2 \leq t + f_{ab} - 1$.

When i is type b, the conditions where the functional unit x_k can execute i at time t are: no operation of type b is scheduled to x_k at any time $t1$ which satisfies $t - Q_b(O_b^a) + 1 \leq t1 \leq t + Q_b(O_b^a) - 1$; and x_k is not reconfiguring at any time $t2$ which satisfies $t - f_{ab} + 1 \leq t2 \leq t + f_{ba} - 1$.

4.4 Scheduling range

Suppose a directed edge (i, j) heading from operation i to operation j is included in the edge set E. In the case that the operation type p_i of i is different from the operation type p_j of j, we need either reconfiguration of a functional unit executable of p_i into a functional unit executable of p_j, or transferring data from a functional unit executing i to another functional unit executing j. Let γ_{ij} denote the required time for either reconfiguration or data transfer between i and j. In this case, γ_{ij} is the minimum of the time to reconfigure the functional unit from type p_i to p_j and the time to transfer data from one functional unit to another. At this point we have no information about functional unit allocation. Therefore, the time to transfer data is assumed to be the smallest, i.e. τ. In the case that p_i and p_j are the same, we do not need reconfiguration. If i and j can be allocated to the identical functional unit, then the data transfer time is 0. Hence γ_{ij} is set to 0.

By using γ_{ij}, the lower bound LB_i and the upper bound UB_i of operation i satisfy the following equations

$$LB_i = \max_{(h,i)\in E} \{LB_h + Q_h + \gamma_{hi} - d_{hi}Tr\} \tag{1}$$

$$UB_i = \min_{(i,j)\in E} \{UB_j - Q_i - \gamma_{ij} + d_{ij}Tr\} \tag{2}$$

where Q_i is the operation execution duration of i, d_{ij} is the number of delays associated with the edge (i, j), Tr is the iteration period of the processing algorithm. We must choose one operation as a reference and schedule it at time 0. Then LB_i and UB_i are respectively identical to the as soon as possible schedule and the as late as possible schedule of operation i. The scheduling range R_i is the set of time t which satisfies $LB_i \leq t \leq UB_i$.

By assuming the reference operation is the start node and LB_i is the longest path to operation i, Equation (1) forms a longest path problem. The smallest Tr so as to make all the directed loops nonpositive and make the longest path problem solvable is the feasible minimum iteration period of the given iterative processing algorithm.

When we specify a particular functional unit to execute operation i, we can take into account more accurate time for reconfiguration or data transfer. Suppose operation i is allocated to functional unit x. Also suppose an operation which has been scheduled and decides either the lower bound or the upper bound of i is allocated to a functional unit y. In the case that $x \neq y$, we need data transfer time τ_{xy} associated with the displacement of x and y. For example in the DFG shown in Fig. 4(a), the operations h and j are already scheduled. If the operation i is allocated to functional unit x as illustrated in Fig. 4(b), the lower bound of i is determined by the execution start time of h, execution duration of operation n, and the data transfer time τ. On the other hand, the upper bound is determined by the execution start time of j and the necessary data transfer time of 2τ. Hence, the scheduling range of operation i is the area bounded by the bold lines in Fig. 4(b). Consequently, the lower bound LB_i^x and the upper bound UB_i^x when i is allocated to the functional unit x are given by the equations

$$LB_i^x = \max_{p(h,i)} \{LB_h + Q_h^i + \tau_{xy} - D_{hi}Tr\} \tag{3}$$

$$UB_i^x = \min_{p(i,j)} \left\{UB_j - Q_i^j - \tau_{xy} + D_{ij}Tr\right\} \tag{4}$$

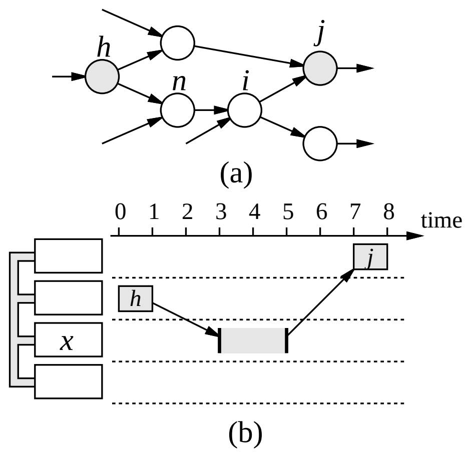


Fig. 4. Scheduling range by precise calculation of data transfer time.

where $p(i,j)$ denotes the directed path from i to j, Q_i^j is the total sum of execution durations of operations on $p(i,j)$, D_{ij} is the total sum of the number of delays on $p(i,j)$.

The set of t which satisfies $LB_i^x \le t \le UB_i^x$ is the scheduling range obtained by precisely calculating data transfer time by assuming that i is allocated to x. It is denoted as R_i^x.

4.5 Proposed scheduling and allocation algorithm

The scheduling and allocation algorithm for dynamically-reconfigurable hardware is summarized as follows. The major difference to the original RCGS method is the addition of the search to evaluate the allocation of operations to functional units.

1. For a given iterative processing algorithm, choose one operation as the reference operation and set its execution start time to 0. Prepare a reconfigurable functional unit x and allocate the reference operation to it. Let the set of functional units $X = \{x\}$. Calculate the minimum iteration period of the given algorithm and set it to Tr.

2. Evaluate scheduling range of each operation for the specified iteration period Tr.

3. Among unscheduled operations, choose one operation i as mentioned in section 4.2. When all the operations are scheduled, then exit.

4. Determine the execution start time of i.

   If the lower bound of i is fixed, let the fixed bound be denoted as t. If the upper bound of i is fixed, let the upper bound be denoted as t. Otherwise, if both the lower and upper bounds are not fixed, then choose the lower bound and set t to the lower bound.

   Search for a time to start execution of i by increasing t (if t is the lower bound) or decreasing (if t is the upper bound) t one by one so that the execution start time will be determined nearby the fixed bound.

   4.1 For each of already existent functional units $x \in X$ (or x_k which can be obtained by reconfiguring x), calculate the scheduling range R_i^x where data transfer time is precisely taken into account.

   Then, for each functional unit $x \in X$, check if operation i can start execution at time t on x. The conditions are, (a) t is contained in R_i^x, (b) functional unit x is idle at time t, and (c) at time t, x is configured or can be configured into the same operation type as i.

   Let t_i^x denote the time at which x can execute i and closest to the fixed bound.

   4.2 If there exist functional units $x \in X$ where $t_i^x \in R_i$, then choose one functional unit x' with $t_i^{x'}$ being closest to the fixed bound of i. Allocate i to x' and fix the execution start time of i to $t_i^{x'}$.

   4.3 If there does not exist any functional unit $x \in X$ where $t_i^x \in R_i$, then prepare a new functional unit $y \notin X$ and assume that y is placed at one of the edges of already placed function units. For y, calculate scheduling range R_i^y by precisely considering data transfer time and check if i can be executed on y. If true, let t_i^y denote the time to start i which is closest to the fixed bound. Repeat this by assuming that y is placed at the other edge of already placed functional units.

   If y can execute i, then let $X = X \cup \{y\}$, allocate i to y and fix the execution start time of i to t_i^y.

   4.4 In the case that the execution start time of i cannot be determined by above steps, we try extending the scheduling range R_i of i.

   The execution start time of already scheduled operation is determined as close as possible to the fixed bound. In the case that both the lower and upper bounds are fixed, we choose the lower bound and the operation is scheduled closest to the lower bound. However this does not always make the scheduling ranges of other operations largest. Thus we check if the scheduling range of i can be widened by slightly shifting the execution start time of operations which immediately preceding or succeeding i

   If R_i can be widen, then go back to step 3. Otherwise, it means that we can not schedule i with the given iteration period Tr. Increase Tr by one, unschedule all the operations, and go back to step 1.

5. If there is unscheduled operation, reevaluate scheduling ranges and go to step 3.

The computational complexity of the algorithm for a value of Tr is $O(n^2e+npTr)$ where n is the number of operations, e the number of edges, p the number of functional units. This will be repeated with Tr increased one by one until every precedence among operations is satisfied. The upper bound of Tr is $O(n)$ where all the operations are executed sequentially on a functional unit. Thus the total complexity is $O(n^3(e+p))$.

5 Experimental Results

The proposed scheduling method is implemented with C programming language. CPU time for each of the experiments is within one second on a 75 MHz Sparc computer.

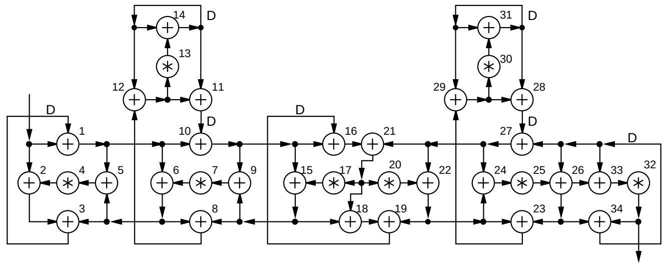
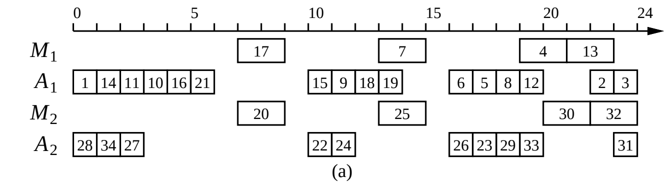


Fig. 5. The 5th order wave filter algorithm.



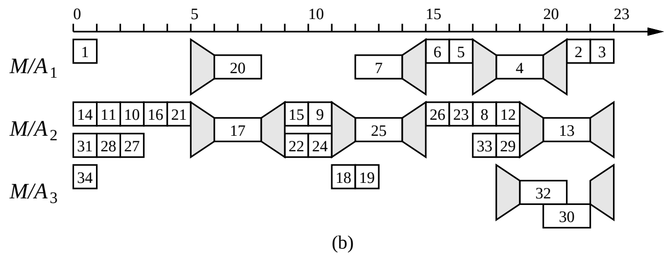


Fig. 6. Schedules of the wave filter with the minimum iteration period. (a) without dynamic reconfiguration. (b) with dynamic reconfiguration.

The scheduling result of the 5th order wave filter (Fig. 5) is shown in Fig. 6(a). In this case, a multiplication time is 2 units of time (u.t.), an addition 1 u.t., and data transfer time between adjacent functional units $\tau = 1$ u.t. The multiplier is pipelined so that we can start a new multiplication every unit of time. In Fig. 6, a small square implies an addition and a small rectangle implies a multiplication. In scheduling operations, iterations are allowed to overlap with each other as long as precedence among operations is maintained (i.e., functional pipelining). Operations 28 and 34 scheduled in the beginning of the iteration period may be considers as the executions of the previous iteration. When dynamic reconfiguration is not in use, we need two adders and two multipliers, and the minimum iteration period is 24 u.t.

Fig. 6(b) shows the scheduling result by using dynamically reconfigurable functional units of type O_A^M. We assume that a multiplication time $Q_M(O_A^M) = 2$, an addition time $Q_A(O_A^M) = 1$, the number of adders obtained by reconfiguration is $N(O_A^M) = 2$, and reconfiguration times are $f_{MA} = f_{AM} = 1$. A gray trapezoid means that the functional unit is being reconfigured. In Fig. 6(b) multiplications 32 and 30 are executed in pipelined fashion on the functional unit M/A_3. The iteration period of 23 u.t. which is smaller than the case of no dynamic reconfiguration is obtained by using three dynamically-reconfigurable functional units of type O_A^M.

Table I shows the scheduling results for one dimensional 8-point DCT [9] which consists of 29 additions, 11 multiplications, and 69 directed edges. The table shows that faster processing can be achieved by dynamic reconfiguration also in the case of DCT.

TABLE I Scheduling Results

processing algorithm	5th order wave filter	8-point DCT
# mul	8	11
# add	26	29
# edge	58	69
w/o reconfig	24 u.t.	14 u.t.
# FUs	2 adds, 2 muls	3 adds, 3 muls
w/ reconfig	23 u.t.	13 u.t.
# FUs	3 reconfigurable	4 reconfigurable, 1 mul

6 Conclusions

We proposed a scheduling and allocation method for operation executions to achieve fast processing by reducing data transfer time between functional units by means of dynamic reconfiguration. Experimental results show that the proposed method minimizes the iteration periods of the given processing algorithms.

In this paper, we assume that a multiplier is reconfigured into some adders and these adders would be reconfigured back into a multiplier. By removing such boundaries of reconfiguration and assuming hardware resource can be freely configured into adders and/or multipliers, the solution space becomes larger and better solution could be found. It remains as a future work.

References

[1] M. Yamashina, "Prospect of Sub-Quarter Micron LSI Design," in *IEICE Tech. Report*, vol. VLD95-136, pp. 53–60, 1996.

[2] T. Sakurai, "Outline of System LSI: An Introduction to System LSI's — Applications and Issues," *Journal of IEICE*, vol. 81, pp. 1083–1086, Nov. 1998.

[3] Toshinori Sueyoshi, "Reconfiurable Logic," *Journal of IEICE*, vol. 81, pp. 1100–1106, 1998.

[4] XILINX, *XC6200 Field Programmable Gate Arrays Data Sheet*, 1997. http://www.xilinx.com.

[5] Masayuki Ito and Junji Kitamichi and Nobuo Funabiki, "A Design of Dynamically Reconfigurable FPGA and An Implementation of Parallel Algorithm on it," *IPSJ SIG Notes*, vol. 98DA88, pp. 1–8, 1998.

[6] P. Lysaght and J. Stockwood, "A Simulation Tool for Dynamically Reconfigurable Field-Programmable Gate Arrays," *IEEE Trans. VLSI Systems*, vol. 4, pp. 381–390, Sept. 1996.

[7] M. J. Wirthlin and B. L. Hutchings, "Improving Functional Density Using Run-Time Circuit Reconfiguration," *IEEE Trans. VLSI Systems*, vol. 6, pp. 247–256, June 1998.

[8] S. M. Heemstra de Groot, S. H. Gerez, and O. E. Herrmann, "Range-Chart-Guided Iterative Data-Flow Graph Scheduling," *IEEE Trans. Circuits Syst.-I: Fund. Theory & Appl.*, vol. CAS-39, pp. 351–364, May 1992.

[9] C. Loeffler, A. Ligtenberg, and G. S. Moschytz, "Practical Fast 1-D DCT Algorithms with 11 Multiplications," in *Proc. IEEE ICASSP*, pp. 988–991, 1989.