A Scheduling Algorithm for Synthesis of Bus-Partitioned Architectures

Vasily G. Moshnyaga, Fumiaki Ohbayashi and Keikichi Tamaru

Department of Electronics & Communications, Kyoto University
Yoshida-Honmachi, Sakyo-ku, Kyoto 606-01, JAPAN
Phone: +81 75-753-5949 Fax: +81 75-751-1576

***Abstract*--- Due to efficient interconnect structure and internal parallelism bus-partitioned architectures are very beneficial for sub-micron chip design. This paper presents a new approach for integrated scheduling and interconnect binding of bus-segmented data-paths. Experiments show that the approach provides better results than existing methods and is quite flexible.**

I. Introduction

A. Motivation

Due to a simple layout, low area and good extension capability, busses are the most commonly used interconnection units in actual chip and system design. In order to satisfy given timing requirements, the number of busses traditionally is not constrained during data-path synthesis and determined in a post scheduling/allocation phase. For the current $1.0 \div 0.8\mu$m CMOS technology such a strategy is reasonable because busses do not impact heavily on chip area and delay. However, as chip density increases and transistor sizes are scaled below 0.5μm level, the bus characteristics dominate in design. Experts predict that if metal lines and interline spacing are narrowing with the same slope as device feature sizes, data transfers via long lines in chips fabricated in 0.25μm technology will consume 50% of cycle time[1],[2]. Introducing a ''fat'' scheme [3] for global nets reduces this delay but significantly affects the chip area. Figure 1 plots area estimation results for two (8- and 16-bit) implementations of Differential Equation Solver Design[4] obtained for different CMOS technologies. In the experiment, we assumed that both implementations had linear topology (2 multipliers, 3 adders, 8 registers and 4 busses)[5] and the width of bus lines was not less than 1μm. The figure clearly shows the price of the ''fat'' scheme at 0.25μm level: busses occupy approximately half of the total data path area.

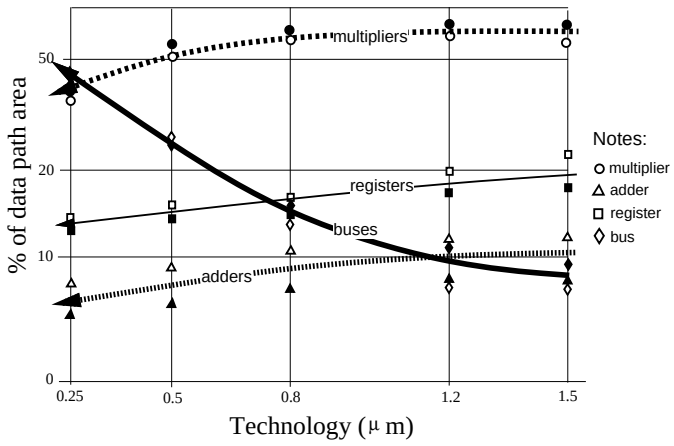

Figure 1: Area distribution between data path components of Differential Equation Solver example vs. technology scaling. (The black marks represent 8-bit implementations, the white marks show 16-bit implementations)

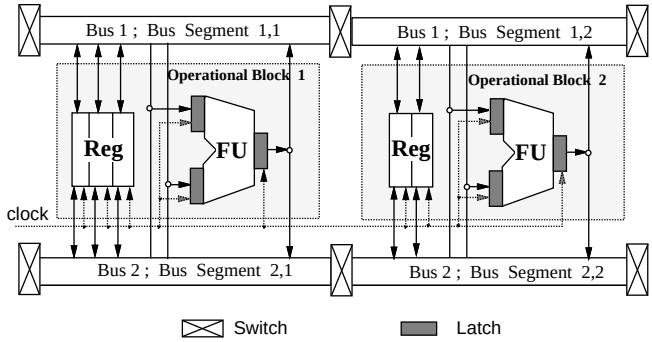

Figure 2: The bus-partitioned architecture

In the deep sub-micron design, bus issues are much important to the chip quality than that of adders or registers, for example, and can not be ignored until the last stage. In contrast, since busses are very expensive, their number might be one of resource constraints in the future chip design.

When the number of busses is limited, bus-partitioning[6] is an efficient way to achieve high throughput. In a bus-partitioned architecture (Fig.2), each bus is split into segments which are functionally connected or disconnected by switches. The disconnected segments can convey different data-transfers simultaneously. So functional units sharing the same bus can operate in parallel. The architecture provides an easier placement, routing and better compaction. Moreover, it is very suitable for power reduction. By enabling the switches to separate electrically the bus segments (with attached hardware units) which are not active in a clock cycle, the total power consumption can be lowered as more as twice[7].

The partitioned busses, however, has a drawback. Since each segment can only communicate with its two neighbor segments, the amount of possible communication between the segments is severely restricted. As result, the transferred data can not be available whenever it is required and, hence, scheduling has to be based on bus binding. An approach to scheduling of the bus-partitioned architectures with limited number of communications paths is presented in this paper.

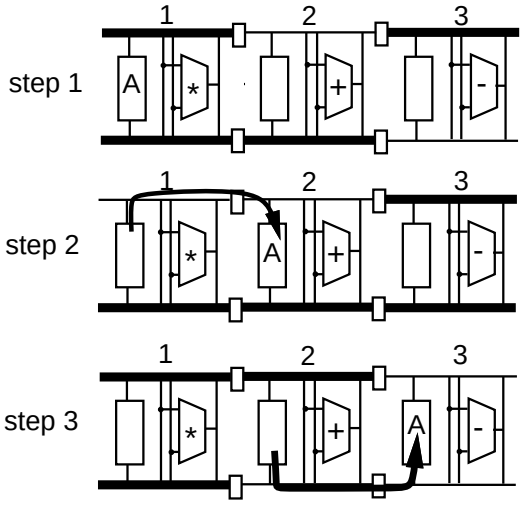

Figure 3: An example

B. Related research

Several high-level synthesis tools support design of bus-partitioned architectures. APPOLON[8] interactively compiles a behavioral specification onto a system of 2 segmented buses and 2 data-paths. ASYL[9] utilizes a rule based approach for operator, register and bus/segment assignment. PARBUS[6] applies an iterative approach consisting of scheduling, partitioning of data-flow operations into clusters, linear ordering of the clusters, register allocation and bus/segment assignment. Although these synthesizers utilize different methods, they have one common feature: they assign operations to clock cycles and functional units assuming no constraints in scheduling on availability of buses and segments. The communication conflicts are relegated to the bus/segment binding phase, which is too far in the design pipeline to meet the design constraints by itself. The problem is that not only a sequence of segments needs to be determined for a data transfer, but also the cycle steps in which it is going to take place needs to be resolved. Consider Figure 3 which shows the availability of bus segments in a data path during 3 cycle steps. Here, bold lines represent busy segments; thin lines represent free segments. Assume that value A produced in cycle step 1 in partition 1 is needed in step 3 in partition 3. Since no path available to convey A to the partition 3 in one cycle step, the only way to perform the data transfer is to transmit A first to partition 2 in step 2, store there temporarily and then transfer it to the destination in step 3. The existing binding models fail to consider this decision and hence prevent tools from finding a high-quality designs. Also no efficient algorithms exist yet which are able to combine the bus-binding with scheduling. As result, either a large number of buses is required to sustain all the parallel data transfers that might occur on any of the clock cycles or extra control steps have to be added to the schedule to satisfy the interconnect requirements. In the best case, several iterations over scheduling and connectivity binding are needed to obtain an acceptable design solution.

C. Contribution

This paper proposes a new model for the data-path synthesis of bus-partitioned architectures and presents algorithms for scheduling and register allocation. We model the design behavior by a stick diagram in which sticks reflect execution of operations on functional units, storing of variables in registers and data transfer through busses. This allows us not only to combine scheduling, with binding and register allocation but apply powerful layout generation techniques to find an efficient solution. The model supports different styles of data-transfers, single- and two-phase clocking schemes, pre-defined constraints on number of busses, segments and registers. The algorithms are based on the model. In contrast to existing approaches, our first algorithm incorporates bus-binding into scheduling; it dynamically binds data-transfers to bus-segments as operations are assigned to control steps. The second algorithm integrates bus binding with register allocation to find the minimal number of registers under the bus constraints. Its main feature is an extra allocation flexibility, by which a value can be assigned to different registers in different control steps such that the total number of registers is minimized.

II. Problem Statement

We suppose that scheduling follows the operation partitioning phase as it done in [12], and the following inputs are known:

(1) a *hypergraph H(V,E,R)*, whose vertex set *V*=$\{v\}$ represents control/data flow graph operations, the directed *edge set E*=$\{e\}$ represents dependences between the operations and the *hyperedge set R*=$\{r\}$ (unordered subsets of V) represent partitioning of V into a number of clusters $r_1, r2, \ldots, r_K$, each of which will be implemented on a separate functional unit. We assume that the feasibility of clusters is determined, and the following is true: $\bigcup r = V$, $r_i \bigcap r_j = 0$, $(i \neq j)$, $| r \geq 1$.

(2) a system architecture with K operational blocks, N parallel busses and K bus segments on each bus. (For simplicity of explanation, we assume that $N = 2$). Each operational block includes one functional unit of a pre-selected type (e.g. multiplier, adder, ALU) and several registers. The components of a block can be connected to corresponding segments of all parallel N buses. Depending on the existence of latches, one- or two-phased clocking is applied to synchronize operations and data transfers. In the one-phase clocking, reading the data out of the registers and writing the data to the registers occur in the same clock phase, while in the two-phase clocking they take place in different phases: ϕ_2 and ϕ_1, respectively. In this case, the system controller outputs a new control word on every control phase.

Since the number of clusters is equal to number of operational blocks, we assume that mapping $\mu : R \rightarrow Z$ of clusters $r \in R$ to operational blocks $z \in Z$ is known or can be easily found by applying the linear ordering algorithm similar to that presented in [11]. Figure 4 illustrates the linearly ordered hypergraph representation for the HAL example[4]. In the figure, the shaded edges represent partitions, the numbered rectangles represent the operational blocks and the edges between the rectangles show the number of data-transfers between the corresponding clusters.

The problem outputs include:

(1) a schedule for the CDFG operations and data-transfers;

(2) assignment of data-transfers to communication buses and bus segments;

(3) register assignment for each of K functional blocks.

III. Approach

The problem has been approached in the following two steps:

1. bus-driven scheduling
2. bus-driven register minimization

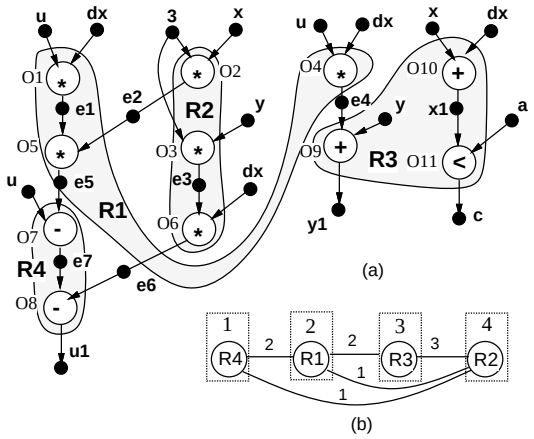

Figure 4: The HAL example: (a) hypergraph; (b) ordering of partitions.

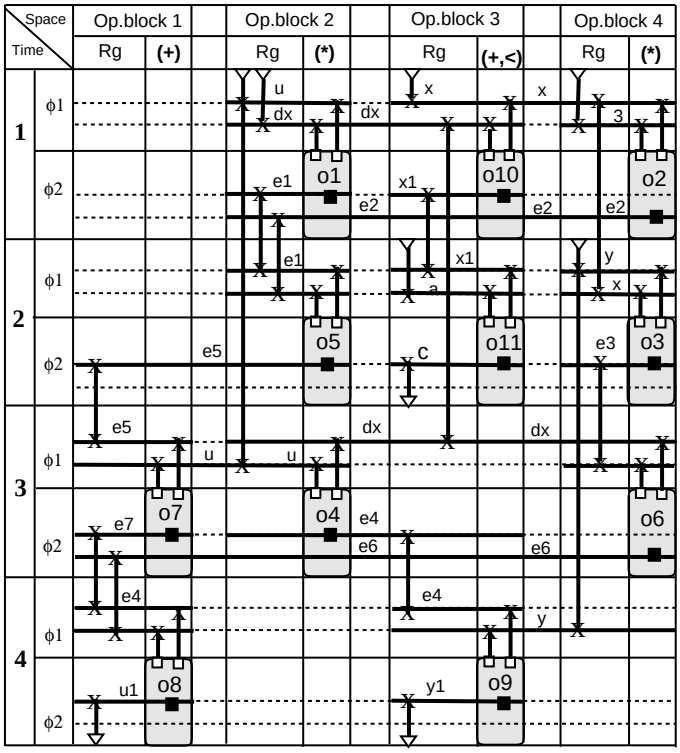

Figure 5: A binding model for the HAL example

A. Bus-Driven Scheduling

A.1. Binding Model

Our scheduling is based on two dimensional binding model (Z, T) (similarly to [10], where each vertical slice $z \in Z$ corresponds to an operating block and each horizontal slice $t \in T$ corresponds to a time slot. (The latter is related to the phase interval). The model, however, has one main departure as can be seen from Fig.5. Additionally to modeling of code operations, we explicitly determine storing of variables on registers and latches and transferring the variables via busses. The key idea behind the model is to represent routes of data processing in time/space domains by abstracting from details of how the data is transformed and transferred. Each data processing route is defined by a collection of $operational$ and $transportational$ sticks. The $operational$ sticks model execution of data-flow operations on functional units. (We depict them by gray patterns in the figure). The $transportational$ sticks (depicted by bold lines) model data transfers through time and space domains without any transformation. The length of a vertical transportational stick corresponds to time during which a value is stored in a register or latch, whereas length of a horizontal stick indicates the number of bus-segments required to convey a value along operational blocks (i.a space). The symbol (**X**) at the intersection of horizontal and vertical sticks reflects a data transfer between the corresponding bus and register (latch) pair. The square symbol shows data transfers at inputs (outputs) of functional units. (In order to distinguish data consumption from data production, the data transfers on inputs and outputs of functional units are shown by white and black patterns, respectively).

Thus the Figure 5 shows that value $e1$ generated by operation $o1$ in time step 1, phase (ϕ_2) is transferred to a register in the same operational block 1, stored their and then in step 2, phase ϕ_1, it is transmitted back from the register to the ALU of the same block. The symbols $\triangle$ and $\triangledown$ mark the border registers.

The user can restrict the maximal number of registers in each operational block or the number of busses in the design by specifying the number of available tracks (dotted lines in the figure) in corresponding hardware or time slots, respectively. In this case, sticks are located on the available t racks only. Consequently, a code operation can be assigned to a time/space slot if and only if sticks, which model the propagation of its input data, can be routed to the slot. The model has several advantages. (1) It clearly shows what bus segments are busy in each clock phase, how many registers are required to support the behavior and where the registers are located. The maximum density of the vertical transportational sticks across all time steps defines the number of registers required. A register belongs to that operational block whose vertical slice contains the register's stick. By extending the horizontal sticks which convey a value from one block to another, we can easy reassign the value to a vacant register such that the total number of registers in the design is reduced. Such ability of values to be stored on different registers during their lifetimes introduces a new degree of freedom which has not been exploited before in bus-oriented synthesis.

(2) It explicitely models utilization of i/o ports at each clock phase. Hence, constraints on i/o ports can be easily considered.

(3) The model unifies scheduling, register allocation and connectivity binding tasks and allows to solve them concurrently by moving from timing domain to space domain and one task to another while preserving the integrity of the tasks.

(4) The model allows us to formulate the high-level synthesis as a layout design problem and thus apply a well elaborated theory of layout design to find effective solutions.

Based on the model, the scheduling problem may be stated as follows: *Given a linearly ordered hypergraph H(V,E,Z), define locations and lengths of sticks S_v, S_e that map the nodes $v \in V$ and the edges $e \in E$ into two-dimensional grid (Z, T) such that the stick area is minimized under the constraints:*

- $\forall v_i \in V \rightarrow S_v(v_i) \bigcap S_v(v_j) = \emptyset$;
  $\forall e_i \in E \rightarrow S_e(e_i) \bigcap S_e(e_j) = \emptyset$ (no stick overlapping);
- $\forall v_j \in z_x \rightarrow S_v(v_j) \in z_x$ (assignment of operational sticks is given);
- $\forall S_e^{horiz} \in t, (t \in T) \rightarrow \bigcup S_e \leq N$ (the number of horizontal tracks is limited);

One approach to this problem is to use simulated annealing when operations are randomly moved to slots such that a cost function with factors of module cost, execution time and interconnection is minimized[10]. While the approach is able to find good solutions, the considerable execution time makes a direct solution more desirable. Therefore a heuristic algorithm was developed.

A.2. Scheduling Algorithm

The backbone of our algorithm is a list scheduling technique[15]. Unlike others, our list scheduler determines data-transfer routes and binds them to buses simultaneously as scheduling proceeds. The process is simplified by the fact that assignment operations to functional units (or blocks) is done before scheduling, therefore the source of a signal and its destination are known. However, such a scheduling becomes more complicated in that an unfortunate decision can lead to a deadlock, when the transferring of data to FU's inputs is impossible. Therefore a check for the deadlocks must precede assignment of operations to control steps. The following *ALGORITHM 1* outlines our scheduling method.

ALGORITHM 1
INPUT: $H(V, E, Z)$
OUTPUT: $\tau : V \rightarrow t, \pi : E \rightarrow g$
begin
for all $z \in Z$ **do**
GET_READY_OPS($V_z, LIST_z$), $FU_z \rightarrow$ free;
endfor
$t = 0$; $Tlist = \emptyset$; $Dlist = \emptyset$;
while ($\exists LIST_l \neq \emptyset$) **do**
$t = t + 1$;
for all $z \in Z$ **do**
if $LIST_z \neq \emptyset$**then**
begin
if (FU_z is free) **then**
select v with the highest priority;
$Tlist = Tlist + (v, z)$;
endif
end
endfor
Dlist=CHECK_DEADLOCK($G_{current}, Tlist, t$);
if ($Dlist \neq \emptyset$)
then *Tlist=DELETE(Tlist,Dlist)*;
endif
if ($Tlist \neq \emptyset$) **then**
{ *SCHEDULE_OP(Tlist,t)*;
DELETE($\{LIST_z, z = 1, ldots, \mid Z \mid\}$,*Tlist)*;
$G_{current} = G_{current}$+*OUT_TRACE(Tlist)*;
SEGMENT_ALLOCATOR($G_{current}, t$);
}; **endif**
for all $z \in Z$ **do**
GET_READY_OPS($V_z, LIST_z$);
endfor
endwhile;
end

The algorithm uses a priority list $LIST_z$ for each cluster $z \in Z$. The function *GET_READY_OPS* scans the set of nodes, V_z, determines if any of unscheduled operations in the set are $ready$ (i.e. all its predecessors are scheduled), deletes each ready node from the set V_z and appends it to $LIST_z$. Initially, all nodes that do not have predecessors are inserted into the appropriate $LIST_z$. The **while** loop extracts the highest priority operation from each list $LIST_z$ whose corresponding functional unit is free, and maps the operation to the slot (z, t) if there is no deadlock in data transfers. The priority is given to an operation which has:

(i) *minimum* $\{mobility(v) - urgency(v)\}$,
where $mobility(v) = ALAP(v) - ASAP(v)$,
$urgency(v) = ASAP(u) - \{ASAP(v) + delay(v)\}$ and u is a direct successor of v;
(ii) the longest path to the output;
(iii) the large number of successors.

The set of highest priority operations is stored in list $Tlist$. The function *CHECK_DEADLOCK* determines nearest locations (z_a, t_a) and (z_b, t_b) of the sticks (S_e^a, S_e^b) which have been assigned to input data-transfers of the operations $v \in Tlist$, and tries to find traces from them to the slot (z, t) without deadlocks. The trace is built first down till horizontal slice t and then in left (or right) direction till vertical slice z. The *LEFT_EDGE* algorithm[13] is used to map horizontal traces to N busses in each slice t. The sticks which already are assigned to the slice t are also considered. A deadlock exists when the number of horizontal traces to be assigned to the slice t exceeds the number of tracks N. In this case, the function *CHECK_DEADLOCK* tries to map the untraced data transfers to the upper slices $t-1, t-2, \ldots, t_{source}$. If it succeeds, the tracing down is applied again to connect the obtained horizontal traces with free segments of the destination slice t. The function returns the set $G_{current}$ of traces constructed and the list, $Dlist$, of deadlocked operations $v \in Tlist$ whose input data transfers were not traced at the current step t. These operations are deleted from the $Tlist$ by function *DELETE(Tlist,Dlist)*. The remaining in $Tlist$ operations are then sticked (i.a. scheduled) to time step t and deleted from the corresponding priority list $LIST_z$. The length of an operational stick equals the sum: $t + delay_z$, where $delay_z$ is the delay of a functional unit in block z. For all this time, the functional unit,z is considered $busy$. The function *OUT_TRACE(Tlist)* routes the data from the scheduled operations to the first free bus. The function *SEGMENT_ALLOCATOR* maps the constructed traces $G_{current}$ to sticks. Figure 5 shows the results of applying the algorithm to the HAL example. (Here, a 2-phase clocking scheme with the input latching has been considered).

Extensions of the algorithm

Operator chaining: For the chained operations, specific edges are needed in the hypergraph representation. The function *GET_READY_OPS* includes both chained operations in the specific $LIST_0$ which has the highest priority among the other lists. Among the operations in the chain, the first operation has the highest priority. All the chained operations are sticked to the same horizontal slice t as others. The binding however differs in that the function OUT_TRACE traces the chained data transfers only in horizontal direction, because no registers and latches are allowed. If both chained operations belong to the same cluster, the OUT_TRACE assigns a stick to a free track segment in the same vertical slice. If they belong to different clusters (e.g. $z1, z2$), the stick is assigned to free segments in $z1, z2$ as well as in between, respectively. A deadlock in this case is solved by adding an extra track between the corresponding blocks.

Multicycle operations: For these operations we use corresponding multicycled $delay$ values.

Pipelining: The pipelined functional units are considered $busy$ for a specified number of phases which correspond to the delay of the pipelined stage. Operations can be assigned to the different pipelined and thus share a functional unit in one clock cycle.

Conditional operations: We assume that $disjointness$ of operations is determined before scheduling. Mutually exclusive operations and data transfers can occupy the same spatial and temporal location. Therefore, they are scheduled so as to use the same hardware (functional units, registers and bus/segments) at the same time.

B. Bus-Driven Register Minimization

Research on register minimization in a post-scheduling phase has already resulted in many effective techniques which employ the Left-Edge Algorithm, clique partitioning algorithm, bipartite edge coloring algorithm, scanline sweep algorithm, simulated annealing, etc. Good surveys of the previous efforts in this field can be found in [14],[15]. Comparing to related techniques, our algorithm has two main features: (1) It combines register assignment with bus-binding to obtain a minimal number of registers under the bus constraints. (2) It utilizes an extra allocation flexibility by breaking the lifetime of each value into segments of one control step and allows different segments of a value to be assigned to different registers.

The algorithm can be outlined as follows.

ALGORITHM 2
INPUT: $H(V, G, Z, T)$
OUTPUT: $H(V, G^*, Z, T)$
begin
$G^* = G$;
GET_LIFETIMES(G,TG)*;
CList=LEFT_EDGE(G,TG)*;
for all $Chnl \in CList$ **do**
Clist=DELETE(CList,Chnl);
for all *intrv=EMPTY_INTRVL(Chnl)* **do**
search=.true.;
z_i*=LOCATION(intrv)*;
t_s*=START(intrv)*;
t_e*=END(intrv)*;
OvList=OVERLAP(intrv,Clist) **do**
while (*OvList*$\neq \emptyset \wedge$*search*) **do**
z_j*=LOCATION(l)*;
Glist=TRACE($z_i \rightarrow z_j, t_s, t_e$);
if (*Glist*$\neq \emptyset$) **then**
G=SEGMENT_ALLOCATOR(G*, Glist)*;
search=.false.;
endif
endwhile
endfor
endfor
end

The function *GET_LIFETIMES* extracts vertical nets from the set of all nets G and determines their lifetimes (set TG). The function *LEFT_EDGE* runs the Left-Edge algorithm that allocates the vertical nets to a set of channels $CList$. (The allocation is done for each vertical slice separately). If there is an empty interval $intrv$ in these channels, the algorithm determines its start and end time t_s, t_e, respectively, the vertical grid slice z, where the interval is located and the list $OvList$ of nets which overlap the interval. The $while$ loop determine the location of the first-left overlapping net, and exams the possibilities to move its cut $< t_s, t_e >$ to vertical slice z_i. The function *TRACE* searches for two deadlock-free paths in horizontal slices t_s, t_e, respectively, to convey data from z_j to z_i and back. If both the data-transfers possible, the function *SEGMENT_ALLOCATOR* maps the set of routed segments ($Glist$) to nets. These iterations continue for all empty intervals.

Figure 6 illustrates how the ALGORITHM 2 works on an example. Applying the Left-Edge algorithm, we can assign the 7 vertical nets $g_1 \div g_7$.(shown by the bold lines) to 4 channels $Ch_1 \div Ch_4$. Since the second channel has an empty space in the slice t3, the algorithm will try to fill it by the cut of the net $g6$ (Fig.6 (b). If the tracing of the corresponding data transfers is possible (nets $n1, n2$), the algorithm will allocate the nets to 3 channels only and hence will save 1 register. Figure 7 shows how the algorithm reduces the number of registers from

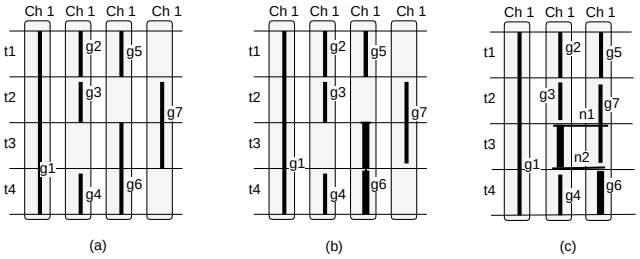

Figure 6: An illustration of the Algorithm 2.

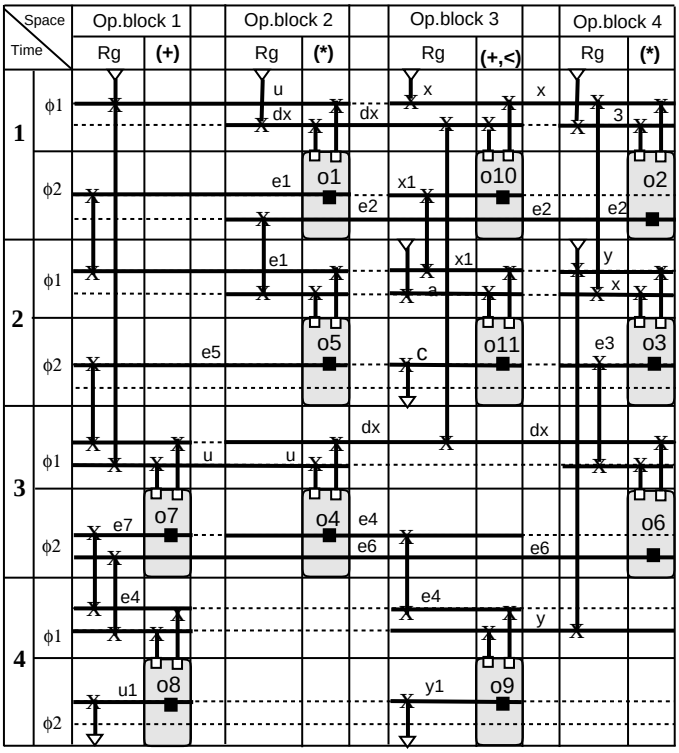

Figure 7: The final schedule of the HAL example.

10 (Fig.5) to 8 in the HAL example.

IV. Experimental Results

The proposed scheduling approach was implemented in C language on SUN 3 system. The scheduling algorithm has a complexity of $O(|V| * |E| \log |E|)$ where $|V|$ is the number of nodes in the CDFG, $|E|$ is the number of edges. The register minimization algorithm has a complexity $O(|E| \log |E|)$. To demonstrate the efficiency of the proposed approach, two examples were used. The first one is Differential Equation Solver or HAL example (Fig.6) and the second is the 5-th order elliptic wave filter [16]. The filter contains 26 additions and 8 multiplications. Table 1 compares our scheduling results for these examples with the results of PARBUS system which implements the existing technique for synthesis of bus-partitioned architectures[6]. In both examples, latches have been inserted at the input and output ports of the functional units and the two-stages pipelined multiplier delay was used. The HAL examples was designed by 1 multiplier and 1 adder, while in Elliptic Filter design 2 adders and 1 multiplier were

Table 2: Scheduling results of the Elliptic Filter

Program	3*,3+			2*,2+			1*,2+			1*(pipelined),2+		
	cs	bus	rg	cs	bus	rg	cs	bus	rg	cs	bus	rg
FDS[4]	19	6	12	21	-	-	19	5	-	17	6	12
ALPS[18]	19	5	19	21	6	19	18	5	15	17	6	16
FAMOS[20]	-	-	-	21	4	13	18	5	15	17	6	16
InSyn[19]	23	3	8	22	4	8	18	6	8	18	6	8
SPAID[17]	18	6	21	21	6	19	18	6	16	-	-	-
Ours	19	2	11	21	2	10	18	2	11	17	2	11

Table 1: Comparison with PARBUS

	HAL (1*(p),1+)		ElF (1*(p),2+)	
	PARBUS	Ours	PARBUS	Ours
c.steps	4	4	19	17
registers	5	5	12	11
bus/segm.	2/4	2/4	2/6	2/6

used. As can be seen, the schedule generated by our algorithm is superior to PARBUS not only in the number of control steps but also in less register count.

Table 2 shows the results of Elliptic Filter scheduling with various resource restrictions. Here *cs, bus, rg* represent the number of control cycles, busses and registers, respectively. The results of several programs are summarized for comparison. As in other papers, the functional unit types used are adders with 1 cycle delay and multipliers with 2 cycle delay. The last column makes use of the two-stage pipelined multiplier. Each schedule take less than 5 seconds to compute. As a whole, our approach was able to equal the best results published hitherto.

V. Conclusions and Future Work

The main objective of our research is to reduce impact of wiring in the high-performance VLSI circuits to be produced in sub-micron technologies. The bus-partitioned architectures are very promising for applications where high-speed and compact circuits are needed. Their synthesis, however, requires bus features to be considered as soon as possible in the design trajectory. In this paper, we have presented a new binding model and two new algorithms for scheduling and register allocation suitable for synthesis of bus-partitioned structures. The algorithms are actually one of the first attempts which incorporate bus-binding into these synthesis procedures. The experimental results demonstrated the reasoning of such approach and its efficiency.

In the current formulation we have not considered scheduling with loops. This will be investigated in the future. The application of the approach on large designs will also be examed.

Acknowledgements

The authors would like to acknowledge the support they received for this research from the Ministry of Education, Science and Culture of Japan under Grant No. A06780255.

References

[1] H.Schelttler, ''Processor chip design on submicron ASICs'', *Euro-ASIC'91*, pp.58-62.

[2] L.Yamada, ''Deep sub-micron IC design'', Cadence seminar, Tokyo, May 1994, unpublished.

[3] G. Sai-Halasz, ''Performance trends in high-end processors'', *Proceedings IEEE*, Vol.83, No.1, Jan. 1995, pp.20-36,

[4] P.Paulin and J.Knight, ''Force-directed scheduling in automated data-path synthesis of ASICs'', *IEEE Trans.on CAD*, Vol.8, No.6, 1989, pp.661-679.

[5] B.Pangrle, et al., ''Relevant issues in high-level connectivity synthesis'', *Proc. 28-th DAC*, 1991, pp.607-610.

[6] C.Ewering, ''Automatic high-level synthesis of partitioned buses'', *Proc. ICCAD'90*, 1990, pp.304-307.

[7] K.Asada, M.Ikeda,''Design of general purpose microprocessor using partitioned bus architectures,'' *Proc. 1995 IEICE General Conf.*, Fukuoka, 1995, pp.114-115 (in Japanese).

[8] R.Jamier, A.Jerraya, ''Appolon, a datapath silicon compiler'', *Proc. ICCD'85*, pp.308-311.

[9] A.Mignotte, ''Resource assignment with different target architectures'', *Proc. Euro-ASIC'91*, pp.172-177.

[10] S.Devadas and R.Newton, ''Algorithms for Hardware Allocation in Data-Path Synthesis'', *IEEE Trans.on CAD*, Vol.8, No.7, July 1989, pp.768-781.

[11] S.Kang, ''Linear Ordering and Application to Placement'', *Proc. 20-th DAC*, 1983, pp.447-464.

[12] J.Scheichenzuber, et al., ''Global Hardware Synthesis from Behavioral Dataflow Descriptions'', *Proc. 27-th DAC*, 1990, pp.456-461.

[13] A.Hashimoto, J.Stevens, ''Wire Routing by Optimizing Channel assignment within Large Apertures'', *Proc. 8th DAC*,1971, pp.155-169.

[14] L.Stock, ''Data Path Synthesis'', *Integration, the VLSI journal*, Vol.18, No.1, Dec. 1994, pp. 1-71.

[15] D.Gajski, etc., *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer A.P., 1992.

[16] S.Y.Kung, et al., ''VLSI modern signal processing'', *Prentice Hall*, pp.258-264, 1985.

[17] B. Haroun and M.Elmasry, ''Architectural Synthesis for DSP Silicon Compilers'', *IEEE Trans.on CAD*, Vol.8, No.4, 1989, pp.431-447.

[18] J.Lee, et al., ''A new integer linear programming formulation for the scheduling program in the data path synthesis'', *Proc. ICCAD-89*, pp.20-23.

[19] A.Sharma and R.Jain, ''InSyn: Integrated Scheduling for DSP Application'', *Proc. 30th ACM/IEEE DAC*, 1993, pp.349-354.

[20] I.Park, et al., ''Fast and Near Optimal Scheduling in Automatic Data Path Synthesis'', *Proc. 28-th DAC*, 1991, pp.680-685.