A Quantitative Analysis for Optimizing Memory Allocation

Youn-Sik Hong, Choong-Hee Cho and Daniel D. Gajski

Department of Computer Science
University of Inchon
177 Dowha-Dong Nam-Gu
Inchon, KOREA 402-749
Tel: 82-32-760-8495
Fax: 82-32-766-6894
e-mail: yshong@inc02.inchon.ac.kr

Department of Information
and Computer Science
University of California
Irvine, USA, CA92717-3425

Abstract - Memory allocation problem has two independent goals: minimization of number of memories and minimization of number of registers in one memory. Our concern is the ordering of bindings during memory allocation. We formulate and analyze three different memory allocation algorithms by changing their binding order. It is shown that when we combine these subtasks and solve them simultaneously by heuristic cost function significant savings (up to 20%) can be obtained in the total area of memories.

1 Introduction

Memory allocation can be defined as a task of mapping scalar or array variables to memory units (**MU**), *i.e.,* single or multiple-port memories and register files, in order to satisfy a set of constraints (*e.g.,* throughput, number of ports, etc) and optimize a cost (*i.e,* area) based on abstract layout model.

The primary focus in this paper, however, is to optimize memory allocation in terms of the area occupied by MUs. Its goals are two fold: to minimize the number of MUs and minimize the number of words (i.e., *registers*) in a MU. This problem has been dealt with by subsequently performing two independent subtasks: *MU allocation* and *register allocation.* Variables can be merged into a MU, if they are not accessed (read or written) at the same time. The MU allocation uses such variables access requirements (*access compatibility*). Similarly variables can share a register, if their lifetimes do not overlap (*lifetime compatibility*).

Our concern is the ordering of binding sequences during memory allocation. That is, we can decide the ordering which subtask to be performed first and the other next. We formulate three different approaches by changing their binding sequences. The first method, we called *Type-I,* is to minimize the number of MUs first and then minimize the number of registers in each MU being allocated. The second one, *Type-II,* maps variables to registers first. Then these registers can be grouped into MUs depending on their read and write times.

These orderings may suffer from the side effect of interaction between subtasks since they are tightly inter-related and their decision made by one subtask affects the other. Therefore, the third one, *Type-III,* combines these subtasks and solve them simultaneously.

Let us illustrate the effects of their binding sequences on synthesis results using a simple example. In **Figure.1**, the lifetime of a variable is represented as interval. It has at least two dots: the dot located at the top of the interval indicates write operation and the rest of them read operation. For example, since the variables, 1,2,3,4 and 5, have access conflicts at step 1, they can not be allocated to a MU. Similarly, the variables, 7 and 10, can not share a register because they have overlapping (lifetime) intervals.

ASP-DAC '97
0-89791-851-7$5.00

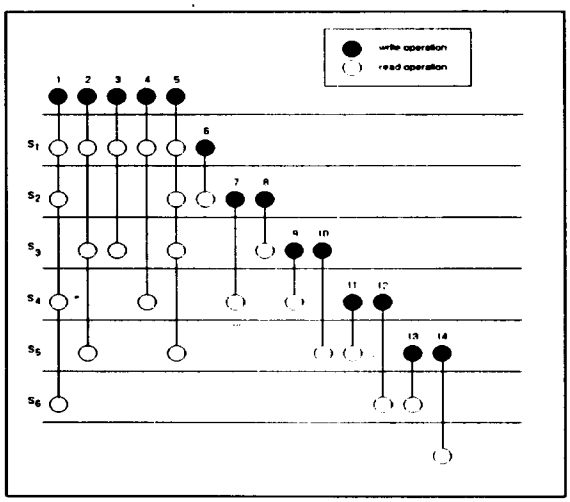


Figure 1 An interval graph representation

These methods mentioned earlier were applied to the interval graph in **Figure 1** and the allocation results are summarized in **Table 1**. With the *Type-II*, 7 registers were allocated after register allocation. However, none of these registers were merged together and thus the number of MUs was still 7. Even though *Type-I* finds the optimal number of MUs, it often fails to minimize the number of words in MUs. Type-III provides the best solution based on the abstract cost measure that will be explained in Section 3.

Table 1 The allocation results

method	no MUs	no words	cost	variables assignment
Type-I	5	9	68.00	({1}, {8,10}), ({2}, {6,9,12}) ({3}),({4}, {11,13}),({5},{7,14})
Type-II	7	7	80.00	({1}, {2,14}),({3,10}),({4,12}) ({5,13}),({6,7,11}), ({8,9})
Type-III	5	7	64.00	({1}, {8,10}),({2,14},{6,7,12}) ({3,9}),({4,11},{5,13})

Type-I approach is most widely used in recent memory synthesis systems. But, it ignores variables lifetime requirements that can be used to reduce the number of words in MUs. It thus fails to optimize the actual silicon area occupied by MUs. Contrary to this, *Type-II* concentrates on the variables with non-overlapping lifetimes. It always discards the variables which are access-compatible but lifetime-incompatible as selecting candidates to be grouped. However, *Type-III* uses cost functions to reflect the correlation between these requirements.

Before we present more detailed discussion about these approaches we will define layout memory model to estimate the memory area. Then a comparison for examples generated randomly is drawn among results achieved by these methods.

2 Previous Works

In <Esc>[9], all variables are first grouped into register files using edge-coloring algorithm. Then left-edge algorithm[6] was used to determine the number of registers in a register file. It is classified into *Type-1*.

Balakrishnan [2] and Ahmed [1] formulate the problem as an 0-1 integer linear programming (ILP) problem. The approach proposed by IMEC[3] has dealt with multi-dimensional signals environment. They also uses ILP approach to allocate memories for such signals. However, they did not address the problem of optimizing memory allocation to minimize total number of words for the memories being allocated. All of them are classified into *Type-I*.

THEDA[5] allocates a set of register files for variables using a branch and bound search technique during constructive binding phase. They use cost functions to predict the impact of interaction among subtasks. However, they take only lifetime compatibility into account in the cost function for storage allocation.

To our knowledge, we have not found the literature related to *Type-III*.

3 Memory Allocation Approaches

We assume that all variables have fixed bit-width w, except some variables may have an integer multiple of w. So our memory model has a fixed bit-width w. The cost function is based on the on-chip memory model similar to [3]:

$$A = T\sum_{i=1}^{M} w^i \cdot (1+\alpha P^i) \cdot (N^i+\beta) \cdot \{1+(P^i+P^i_{rw}-2)/4\}$$

where, M - the number of memory units,
N - the number of words,
F - total number of ports,
P_{rw} - the number of r/w ports,
T - technology scaling factor,
α, β - technology dependent parameters.

Let us assume that the number of ports is 1 and it has read-/write-capability. Then A is proportional to $\sum_{i=1}^{M} N^i$. Thus, the main goal is to minimize the number of MUs and minimize the number of words in MUs simultaneously such that the resulting memory configuration would result in the least area. Now the following subsections will describe each type of the approaches in a concrete level.

3.1 Type-I approach

This method takes variables access requirements into account first to minimize the number of MUs. Then lifetimes of variables are to be considered to reduce the number of registers in a MU. The first problem is modeled as graph-coloring problem and solved using sequential vertex-coloring method. The last one is solved by clique-partitioning method[10]. This approach is somewhat similar to [9], even though he used edge-coloring algorithms to group variables into register files. **Algorithm 1** describes the approach in more detail.

Data flow analysis is used to calculate two sets mainly for each control step s_i. The function READ_ACCESS_CONFLICT (s_i) returns the set V_{reaa} of variables that are read at the same time in s_i. All the variables in V_{reaa} are access-incompatible. Similarly, the function WRITE_ ACCESS_CONFLICT (s_i) returns the set V_{write} of variables that are written simultaneously in s_i. Then we create an access-incompatible graph G_c. In the graph G_c, a variable is represented by a vertex and an edge connects two vertices whose corresponding variables cannot be allocated to a MU.

The function COLOR_INCOMPATIBLE _GRAPH returns the number LL_{MU} of colors and a colored graph. The vertices with the same color correspond to the variables which will be merged into a MU.

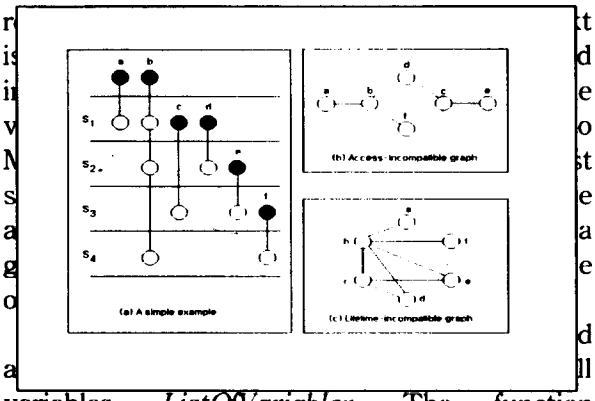
Then a lifetime incompatible graph is built for each MU. The procedure CLIQUE_PARTITIONING performs register allocation for the variables in a MU to allocate the minimum number of registers. Incompatibility arises from overlapping lifetimes of variables in a MU. The function LIFETIME_COMPATIBLE (v_j, v_k) returns *true* if two variables v_j and v_k have disjoint lifetimes. Then a lifetime compatible subgraph $G_c'(V', E')$ is built, where V' is the set of variables that have been grouped into MU and E' the set of edges. Each edge $e'_{j,k} \in E'$ links two different vertices $v_j \in V'$ and $v_k \in V'$ whose lifetimes do not overlap.

Algorithm 1 : Type-I Method

```
/* create an access-incompatible graph   */
/*  Gc(V, E)                             */
V = φ ; E = φ ;
for each control step si ∈ S do
  V_reaa = READ_ACCESS_CONFLICT ( si);
  V = V ∪ V_reaa;
  for v_r1,r2 ∈ V_reaa, r1 ≠ r2 do
    E = E ∪ e_r1r2;
  endfor
  V_write = WRITE_ACCESS_CONFLICT ( si);
  V = V ∪ V_write;
  for v_w1,w2 ∈ V_write, w1 ≠ w2 do
    E = E ∪ e_w1w2;
  endfor
endfor
LL_MU=COLOR_INCOMPATIBLE_GRAPH
Gc(V, E);
index = 1;
for index ≤ LL_MU do
/* create a lifetime-incompatible graph  */
/* Gc'(V', E') for MU_index              */
  V' = φ ; E' = φ ;
  V' = { vi | vi ∈ MU_index};
  for each vj, vk ∈ V', j ≠ k do
    if (LIFETIME_COMPATIBLE ( vj, vk)) then
      E' = E' ∪ e'jk;
    endif
  endfor
  CLIQUE_PARTITIONING ( Gc'(V', E'));
  index = index + 1;
endfor
```

3.2 Type-II Approach

This method takes variables lifetime requirements into account first to solve the r t i d i e v o N st s e a a g e o d a ll



variables, *ListOfVariables*. The function LEFT_EDGE_ALGORITHM returns a set *SetOfRegisters* of registers, where each of them is allocated to the variables whose lifetime do not overlap.

Then a weighted graph $G = (V, E)$ is built to group the registers to MUs using variables requirements. Each vertex $v_i \in V$ is a register in *SetOfRegisters* that can contain a set of variables. Note that all the variables allocated to a register are always access-compatible. There exists an edge $e_{i,j} \in E$ between two vertices, v_i and v_j, if two variables $n_p \in v_i$ and $n_q \in v_j$ are access-compatible. A weight w_{ij} is imposed on its edge e_{ij} to indicate *degree of compatibility* between v_i and v_j. It can be defined as:

$$w_{i,j} = \sum_{n_p \in v_i} \sum_{n_q \in v_j} ACCESS-COMPATIBLE(n_p, n_q)$$

where, $ACCESS-COMPATIBLE(n_p, n_q) = \begin{cases} 1, & \text{if } n_p \text{ and } n_q \text{ are access-compatible} \\ 0, & \text{otherwise} \end{cases}$

Then we can apply a greedy algorithm to find the minimum number of MUs. It finds the vertices v_p and v_q in V such that they are connected by an edge and all the variables of v_p and v_q are access-compatible.

This situation is expressed as a quantity by computing a simple formula. Let N_i and N_j be the number of variables allocated to v_i and v_j, respectively. The *merge-force* associated with v_i and v_j is given by

$$merge-force(v_i, v_j) = \frac{w_{i,j}}{N_i \times N_j}$$

If $merge-force(v_i, v_j)$ equals to 1, the two vertices can be grouped into a single super vertex without causing any access-conflict. We call this *bounding force*. v_p and $v_q$having bounding force are merged into a single super vertex v_{pq}, which contains all the variables v_p and v_q. The registers R_p and R_q corresponding to those vertices can share a MU.

Algorithm 2 : Type-II method

```
SetOfRegisters=LEFT_EDGE_ALGORITHM(ListOfVariables);
/* create a weighted graph G(V, E) */
V = SetOfRegisters; E = φ;
/* SetOfRegisters is an orderer set of registers */
for Ri∈SetOfRegisters do
  j = i + 1;
  for Rj∈SetOfRegisters do
    E = E ∪ eij;
    for np∈Ri and nq∈Rj, p≠q do
      if (ACCESS_COMPATIBLE ( np, nq)) then
        wij = wij + 1;
      endif
    endfor
    j = j + 1;
  endfor
  i = i + 1;
endfor
MERGE_REGISTERS_INTO_MU G(V, E);
```

3.3 Type-III Approach

Having introduced the basic concept by presenting a walkthrough example and terminology we proceed to describe *Type-III* algorithm. Let N^i_{ovl} be the number of variables whose lifetimes overlap with that of variable v_i. Similarly, let N^i_{inc} be the number of variables which have access-conflict with v_i. N^i_{ovl} and N^i_{inc} reflects the degree of access compatibility and the degree of lifetime compatibility among the rest of variables for v_i, respectively. N^i_{ovl} is the degree of corresponding vertex of v_i in a lifetime

incompatible graph. Similarly, N^i_{inc} is the degree of v_i in an access-incompatible graph.

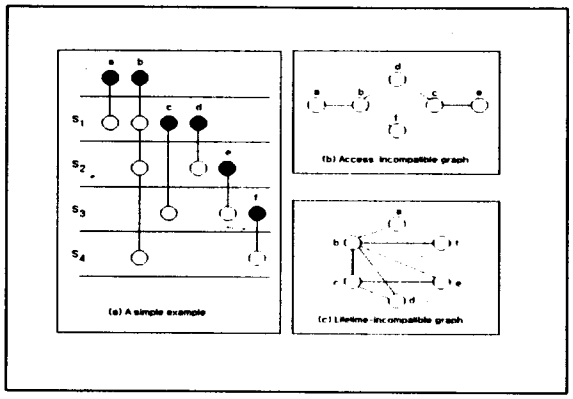


Figure 2. An illustrative example

The access-incompatible graph and the lifetime incompatible graph for an example in **Figure 2(a)** is depicted in **Figure 2(b)** and **Figure 2(c)**, respectively. Thus, we compute $N_{ovl}$and N_{inc} as shown in **Table 2(a)** for the variables in **Figure 2(a)**.

Let us define the degree of hardness h_i of v_i as the binding difficulty.

$$h_i = w_1 \times N^i_{ovl} + w_2 \times N^i_{inc} \qquad (1)$$

where, w_1 and w_2 are weighting factors.

Let v_{seea} be the variable with the highest degree of hardness. By eliminating v_{seea} at earlier step, we can increase the possibility that merging will occur among the rest of variables. Variables with higher degree of hardness have less chance to be grouped than variables with lower degree at earlier step. Eventually when they can be merged with those variables, their lifetimes must be overlapped and it increases the number of registers required. So v_{seea} will be allocated to a MU first.

In **Table 2(a)**, b with the highest degree, 8, when $w_1 = w_2 = 1$, is selected and allocated to a memory unit M_1. **Table 2(b)** shows the results by deleting b from the **Table 2(a)**. At next, c is chosen. But, c will not be allocated to new M_2 since b and c can share M_1. Then there are four variables (a, d, e, f) with sum '0' in **Table 2(c)**. It means they can be merged into M_2 with single register(word). Note none of them can be merged into M_1.

Table 2(a)

	a	b	c	d	e	f
N^i_{ovl}	1	5	3	2	2	1
N^i_{inc}	1	3	2	2	1	1
h_i	2	8	5	4	3	2

Table 2(b)

	a	c	d	e	f
N^i_{ovl}	0	2	1	1	0
N^i_{inc}	0	2	1	1	0
h_i	0	4	2	2	0

Table 2(c)

	a	d	e	f
N^i_{ovl}	0	0	0	0
N^i_{inc}	0	0	0	0
h_i	0	0	0	0

Before we map v_{seea} to a new MU, we do check if there exists M_j that is either access- or lifetime-compatible with v_{seea}. If there are more than one M_j, a heuristic cost function will be used to determine in which M_j the variable v_{seea} should be merged. This reduces the number of MUs needed, while it tends to increase the number of registers in a MU. The merging cost C^i_i of assigning a variable v_i to a memory unit M_j is calculated according to:

$$C^i_i = \alpha \times D_{ij} + \beta \times E_{ij} - \gamma \times S_{ij} \qquad (2)$$

where, D_{ij} - Increments in the lifetime density of resulting from assigning v_i to M_j.

E_{ij} - The number of access-incompatible variables with v_i to be excluded resulting from assigning v_i to M_j.

S_{ij} - The number of common variables of v_i and M_j. They are either access or lifetime compatible with both v_i and M_j.

α, β, γ - weighting factors.

Algorithm 3 shows the overall steps in more detail. This algorithm groups variables into clusters such that each cluster can be realized using a single MU. The all information about access- and lifetime compatibility for given a set of variables are

transformed in a table ATTRIBUTE_TABLE. The algorithm will first try to find the v_{seed} with the highest degree of hardness. The function COMPUTE_degree_of_hardness (v_i) returns the degree of hardness of v_i by (1).

Then we map v_{seed} to M_{index} with the lowest merging cost. The merge is only legal if v_{seed} and M_{index} are either access or lifetime compatible. If merge is illegal, say, the merging cost is infinite, then new $M_{|M|+1}$ is built. In this case, the set of variables which have non-overlapping lifetimes with v_{seed} are collected into the set LL to share the same $M_{|M|+1}$. The variable in LL with the highest degree of hardness is chosen. The variables selected so far are deleted from the set V of variables and ATTRIBUTE_TABLE is updated. The above steps are repeated until there are no variables left in the set V.

Algorithm 3 : Type-III Method

```
/* V : a set of variables          */
/* M : a set of MUs                */
/* ATTRIBUTE_TABLE : memory allocation table */
M = φ ;
while V ≠ φ do
  binding_difficulty = 0;
  for each variable v_i ∈ V do
    h_i = COMPUTE_degree_of_hardness ( v_i);
    if ( h_i > binding_difficulty) then
      binding_difficulty = h_i; seed = i;
    endif
    MergeCost^seed_index = min_{1≤ j ≤|M|}
                MERGE_COST ( v_seed, M_j);
    if MergeCost^seed_index ≪ ∞ then
      M_index = M_index ∪ {v_seed} ;
      V = V - { v_seed};
    else
      M_{|M|+1} = ALLOCATE_MEMORY (M);
      Group v_k into the set LT ;
      if(LIFETIME_COMPATIBLE( v_k, v_seed)),
where, v_k ∈ V, k ≠ seed.
        Select v_p ∈ LT with the maximum
degree of hardness, h_p;
    if h_p ≪ ∞ then
      M_{|M|+1} = M_{|M|+1} ∪ {v_p};
    endif
  endif
  UPDATE_ATTRIBUTE_VALUE;
endwhile
```

4 Experimental Results

First we have tested *Type-III* approach with varying selection strategies. These strategies limit the search space efficiently, i. e., the number of variables that can be grouped into a MU M_i, to reduce running time extensively.

longest length of lifetime first (LF) The variable with the *longest length of lifetime* is selected first. By eliminating them it increases lifetime compatibility among the rest of variables.

shortest length of lifetime first (SF) Variables with lifetime compatibility are always access compatible. So the variable with the *shortest length of lifetime* has the least cost even though by doing it dost not guarantee a global optimum solution.

most access-compatible first (AF) This uses the cost function in Eq.(2) without any acceleration technique.

Table 3 summarizes some experimental results for randomly generated examples to compare their performance and execution time. In all cases, the number of MUs are equal for all of these techniques. However, both the number of words needed for the MUs allocated and the CPU time (seconds) elapsed varies with the strategies. As we expected, the AF strategy is most time-consuming.

For the rest of two methods, the LF technique produces less number of words by 32% to 18% than the SF. Besides, the former is three times faster than the latter. Since the SF technique only emphasizes lifetime compatibility among variables, it tends to cluster as many variables as possible into a MU, as long as they still preserve lifetime compatibility relations. However, this prevents variables with longer lifetime but most access-compatible with other variables from being selected as candidates. This is the reason why its performance is not good as LF. This is almost true for the rest of data. Therefore, Type-III employs LF selection strategy to accelerate its running time.

Now we present the results of running the three approaches on a number of randomly generated examples. The **Figure 3**

shows the costs of memory area for the these approaches.

Table 3 Experimental results for the Type-III method

data		AF strategy		
control steps	no of vars	no of MUs	no of words	CPU time
24	70	24	37	9.71
27	75	24	42	17.09
31	80	24	45	22.02
35	90	24	42	29.22
37	95	24	40	42.51
39	100	24	43	52.41
44	110	24	45	69.52
49	120	24	44	178.08
57	135	24	44	248.64
62	145	24	48	516.51
67	155	24	45	620.97
72	165	24	48	929.86
77	175	24	47	1154.51

data		with acceleration					
		SF strategy			LF strategy		
control steps	no of vars	no of MUs	no of words	CPU time	no of MUs	no of words	CPU time
24	70	24	35	0.50	24	28	0.19
27	75	24	39	0.73	24	28	0.21
31	80	24	39	1.16	24	29	0.27
35	90	24	37	1.00	24	29	0.31
37	95	24	37	1.40	24	29	0.37
39	100	24	37	1.52	24	30	0.40
44	110	24	40	2.22	24	31	0.49
49	120	24	40	3.29	24	30	0.70
57	135	24	44	4.91	24	31	1.01
62	145	24	46	4.60	24	32	1.25
67	155	24	47	4.71	24	32	1.51
72	165	24	47	5.45	24	32	1.83
77	175	24	46	6.89	24	32	1.97

5 Conclusion

In this paper, we have addressed the problem of optimizing memory allocation on the estimated silicon area. There are two important requirements, access-time and lifetime compatibility among variables, which allows to address the problem of register and memory unit allocation. In most previous approaches, they divided the problem into two independent subproblems and solve each of them separately. However, they suffer from the negative interaction between subproblems. Consequently, a complete approach based on the heuristic cost function is proposed. From the experiments with the set of randomly generated examples, it is shown that the proposed approach results in a significant area savings (up tp 15%) in the total area of memory units. The area has been calculated based on the abstract layout model. Actually, the method also incorporated seed-selection techniques with the cost function to improve its performance. These are substantiated with results for randomly generated examples. The result presented in this paper constitute the foundation for out future work on solving the memory allocation to a cost-efficient memory synthesis problem.

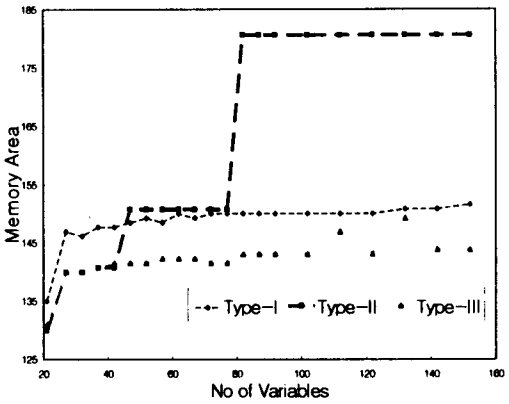


Figure 3

References

[1] Imtiaz Ahmad and C.Y. Roger Chen, *"Post Processor for Data Path Synthesis Using Multiport Memories,"* **Proceedings of the ICCAD**, pp276-279, 1991.

[2] M. Balakrishnan et. al, *"Allocation of Multiport Memories in Data Path Synthesis,"* **IEEE Trans. on CAD**, vol.7, no.4, pp536-540, April, 1988.

[3] F. Balasa, F. Catthoor, and H. De Man, *"Dataflow-Driven Memory Allocation for Multi-Dimensional Signal Processing Systems,"* **Proceedings of the ICCAD**, pp31-34, 1994.

[4] D.D. Gajski and Y.L. Lin, *Module Generation and Silicon Compilation*, in **Physical Design Automation of VLSI Systems**, B. Preas and M. Lorenzetti Eds., The Benjamin/Cummings Pub. Co., 1991.

[5] Y.C. Hsu and Y.L. Lin, *High-Level Synthesis in the THEDA System*, in **High-Level Synthesis**, P. Composano and W. Wolf, Eds., Kluwer Academic, 1991.

[6] F.J. Kurdahi and A.C. Parker, *"REAL: A Program for Register Allocation,"* **Proceedings of 24th DAC**, pp210-215, 1987.

[7] P.E.R. Lippens, J.L. van Meerbergen, and W.F.J. Verhaegh, *"Allocation of Multiport Memories for Hierarchical Data Streams,"* **Proceedings of the ICCAD**, pp728-735, 1993.

[8] B.M. Pangrle, *"Splicer: A Heuristic Approach to Connectivity Binding,"* **Proceedings of the 25th DAC**, 1988.

[9] L. Stok, *"Interconnect Optimization During Data Path Allocation,"* **Proceedings of the EDAC**, pp141-145, 1990.

[10] C.J. Tseng and D.P. Siewiorek, *"Automated Synthesis of Data Paths in Digital Systems,"* **IEEE Trans. on CAD**, vol.5, no.3, pp379-395, July, 1986.