A Flexible Datapath Allocation Method for Architectural Synthesis

KYUMYUNG CHOI
Samsung Electronics Co.
and
STEVEN P. LEVITAN
University of Pittsburgh

We present a robust datapath allocation method that is flexible enough to handle constraints imposed by a variety of target architectures. Key features of this method are its ability to handle accurate modeling of datapath units and the simultaneous optimization of direct objective functions. The proposed method consists of a new binding model construction scheme and an optimization technique based on simulated annealing. To illustrate the flexibility of this method, two datapath allocation procedures have been developed for two problem environments: (1) a procedure that incorporates interconnection area and delay estimates, where floor-planning is tightly integrated into datapath allocation; and (2) a procedure that handles registers, register files, and multiport memories for data storage, as well as random and linear topologies for interconnection architectures. Results from these two applications show our method produces competitive designs for benchmark circuits, as well as being flexible enough to be used for a variety of different domains.

Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic synthesis*; *Optimization*; B.7.2 [**Integrated Circuits**]: Design Aids—*Placement and routing*; G.1.6 [**Numerical Analysis**]: Optimization; J.6 [**Computer Applications**]: Computer-Aided Engineering—*Computer-aided design* (CAD)

Additional Key Words and Phrases: Allocation and binding, high-level synthesis

1. INTRODUCTION

Architectural synthesis takes an abstract behavioral specification of a digital system and a set of time and resource constraints and finds a

This work was supported, in part, by the National Science Foundation under grant MIP-9102721. We are grateful for their support.
Authors' addresses: K. Choi, CAE, Semiconductor Business, Samsung Electronics Co., Korea; S. P. Levitan, Department of Electrical Engineering, University of Pittsburgh, 348 Benedum Engineering Hall, 3700 Ohara St., Pittsburgh, PA 15261; email: steve@ee.pitt.edu.

control sequence and a register-transfer level structure that realize the given behavior [McFarland et al. 1990]. Synthesis is composed of two tasks: resource scheduling and datapath allocation. Scheduling is the task of satisfying the temporal constraints in terms of abstract operators, while datapath allocation is the task of assigning real hardware resources and binding operations to functional units, variables to memory units, and data transfers to interconnection units. Therefore, the quality of the result from the allocation task in particular is a function of both the flexibility and accuracy of modeling the complex constraints imposed by real hardware structures, as well as the optimization method chosen to meet those constraints. However, to date, datapath allocators have not been designed in ways that allow them to satisfy the flexibility and performance requirements of real system design.

There are two problems with most current allocation procedures. The first is the sequential solution of subtasks (e.g., functional unit binding, memory unit binding, and interconnection unit binding) that have interdependencies. The second is the use of indirect objective functions as the optimization criteria, followed by postprocessing steps. Both of these problems can lead to less than optimal designs. Unless these problems are addressed, algorithms developed for datapath allocation will not be useful in real design environments.

For an effective datapath allocation tool that can meet the needs of real design, three criteria must be met:

(1) Cost functions must be based on accurate models.

(2) Cost functions must be directly optimized.

(3) All cost functions must be optimized simultaneously.

The majority of current datapath allocation algorithms use a constructive approach based on heuristic algorithms. However, for these approaches, criteria (1) and (2) above are handled correctly only for subtasks, and criterion (3) is not handled at all. The other popular approach for datapath allocation is the ILP (integer linear programming) method. The ILP approach has received much attention, since it meets criterion (3), but the results of research to date show several common disadvantages, including:

—As the number of variables in the ILP formulation increase, the run time increases greatly [Gebotys and Elmasry 1993; Rim et al. 1994].

—It is difficult to formulate the versatile constraints necessary for criteria (1) and (2) [Rim et al. 1994].

—When the ILP approach is used, the ILP solving program (e.g., LINDO Systems) may not converge to a solution on even a moderate number of variables and constraints.

On the other hand, simulated annealing is an optimization technique that is flexible enough to meet all three of the listed criteria. While several datapath allocation algorithms in the past used simulated annealing for

their optimization methods [Devadas and Newton 1989; Krishnamoorthy and Nestor 1992], they used conventional modeling and indirect cost functions, and so failed to meet criteria (1) and (2).

In this paper we address these shortcomings with a new datapath allocation method that consists of a new binding model construction scheme and an optimization technique based on simulated annealing. It meets all three criteria: handling complex models of datapath units, using direct objective functions, and optimizing multiple objective functions simultaneously.

To illustrate the power and flexibility of the method, two datapath allocation procedures for two different problem environments have been developed and tested. The first example incorporates accurate interconnection area and delay estimates, where floor-planning is tightly integrated into datapath allocation. The second example handles registers, register files, and multiport memories for data storage, and random and linear topologies for interconnection architectures using direct binding models and objective functions. Both of these procedures have been incorporated into a datapath allocation tool called *MandM* for "mix and match," developed for datapath allocation problems in a variety of different domains [Choi 1995].

The rest of this paper is organized as follows: Section 2 discusses related work; Section 3 describes the general datapath allocation method; Section 4 presents the application of the method to the problem environment of area and performance optimization in bit-sliced architectures; and Section 5 describes the method applied to the environment of diverse embedded memory architectures; Section 6 presents our experimental results using *MandM* in both datapath allocation environments in comparison with other systems. We conclude our remarks in Section 7.

2. RELATED WORK

The first application environment that we have explored is datapath synthesis incorporating physical layout estimation. The second environment has the memory architecture as an essential constraint on the system. Our optimization approach is based on simulated annealing. Here we briefly review related work; for a complete review, see Choi [1995].

2.1 Area and Performance Optimization

Some researchers have discussed the effects of incorporating physical layout during architectural synthesis [McFarland and Kowalski 1990; Knapp 1990; Hsieh et al. 1993]. Others have proposed modeling techniques that consider the trade-off between accuracy and efficiency for layout area and delay estimation in architectural synthesis [Ramachandran et al. 1992].

The architectural synthesis procedure used by these approaches is (1) synthesize the RTL structure; (2) estimate layout parameters or produce real layout; (3) evaluate the synthesized result in terms of chip area and

performance; (4) accept the design, or redesign from step (1) with different parameters. The possible problems with these approaches comes from the fact that it is not easy to improve the synthesized result by simply modifying the input parameters of architectural synthesis. Therefore, the number of iterations may be large in order to achieve a satisfactory result.

Only a few researchers have proposed methods to efficiently incorporate cost metrics for physical layout into architectural synthesis. 3D scheduling [Weng and Parker 1991] considers scheduling, functional unit binding, and floor-planning simultaneously, and incorporates interconnection delay during architectural synthesis. However, it considers only functional unit binding and floor-planning, but does not consider other elements such as registers, multiplexers, and wiring elements that have a large effect not only on datapath area but also on interconnection delay. GB [Jang and Pangrle 1993] uses a grid-based connectivity binding approach and considers the minimization of interconnection lengths with the assumption of a bit-sliced stack architecture. GB calculates arc lengths between functional units during binding in order to minimize the distance between functional units. Then, registers are inserted by several methods based on a case-by-case analysis. However, arc lengths should really be calculated between functional units and registers. Therefore, an excessive number of registers may be allocated. This increases not only actual layout area but also the longest arc length and delays.

Fang and Wong [1994] consider functional units, register binding, and floor-planning simultaneously. They reduce critical path interconnection delay during binding. The procedure of their simultaneous binding and floor-planning algorithm is (1) find a floorplan, (2) find a binding for the current floorplan, and (3) compute a cost function that considers area, wire length, and interconnection delay of the critical path. This procedure is performed iteratively by their simulated annealing-based floor-planner. A problem in their algorithm is that their binding and floor-planning tasks are not tightly integrated. When a floor-planning result is generated, the previous binding result is not considered. Therefore, they only choose the best solution among many iterations of the above procedure (floor-plan then binding). Also, their binding algorithm is constructive. Therefore, interconnection delay between a functional unit and a register cannot affect the binding correctly. Their binding algorithm will only be valid if execution delays of functional units and registers dominate interconnection delays.

2.2 Embedded Memory Architectures

Several systems have been developed that allocate the datapath directly for the final memory architecture. SPAID [Haroun and Elmasry 1989] and STAR [Tsai and Hsu 1992] consider register files and a linear topology directly. Whereas a unique bus is connected to each register file in SPAID, every register file can be connected to every bus, as required by the allocation results in STAR.

Balakrishnan et al. [1988] proposed a way of grouping registers to form a multiport memory module. They also offered an approach to minimize the interconnection between ports on a multiport memory and functional units. MAP [Ahmad and Chen 1991] enhanced Balakrishnan's idea, while using the same ILP approach. Kim and Liu [1993] changed the overall procedure of datapath allocation for multiport memories. They first determined the interconnections between memory ports and functional units, and then grouped the variables to form memory modules. They used an ILP approach for the interconnections between memory ports and functional units, and used a heuristic algorithm for grouping the variables to form memory modules.

The common drawbacks of these approaches for datapath allocation of multiport memories are as follows: First, in order to minimize the chip area that consists of functional units, multiport memories, and interconnections, the allocation of each must be improved simultaneously. However, these systems use the results of functional unit allocation as their input, and perform multiport memory and interconnection allocation sequentially (with the opposite order). Second, they minimize the number of connections in order to minimize the interconnection area. But the minimum number of connections does not necessarily guarantee the minimum interconnection area, which consists of multiplexers, buses, and tristate buffers.

2.3 Simulated Annealing Approaches

Devadas and Newton [1989] used simulated annealing for simultaneous scheduling and allocation. That technique handled scheduling and allocation of functional units simultaneously, while minimizing the necessary numbers of registers and buses. However, it handled the most interdependent of all the allocation subtasks, i.e., register binding and interconnection binding, separately.

SALSA also used simulated annealing for scheduling [Nestor and Krishnamoorthy 1993] and for allocation of functional units and registers [Krishnamoorthy and Nestor 1992]. During allocation improvement by simulated annealing, a cost function that is a weighted sum of functional unit, register, and interconnection costs is used. SALSA extended conventional functional unit and register binding models to more flexible binding models for functional units and registers. However, it assumes a register and random topology architecture. Therefore, in order to adapt to diverse memory and interconnection architectures, postprocessing steps are necessary.

The common problems of both of these simulated annealing approaches for datapath allocation are that they lack accurate modeling of the optimization objects and direct cost functions. Rather, they use simulated annealing with conventional modeling and indirect cost functions. Therefore, they cannot achieve significantly improved results over heuristic techniques, even though they incur relatively long computation times.

3. DATAPATH ALLOCATION

Our datapath allocation method is based on the use of new binding models and an optimization technique based on simulated annealing. In this section we present our general approach. In Sections 4 and 5 we refine the method for two specific applications.

3.1 Binding Model Construction

Binding models for datapath allocation are described in several papers [Devadas and Newton 1989; Krishnamoorthy and Nestor 1992]. However, conventional binding models consider only functional unit and register allocation. The datapath allocation procedure used with these binding models may allocate functional units and registers first, then allocate other hardware units or modify hardware units in order to complete datapath allocation. A problem with this approach is that because it is a sequential problem-solving approach and uses indirect objective functions, it cannot produce optimized results for the final target architectures.

In order to solve these problems, we extend our binding model to a larger set of datapath units. First, we classify datapath units as *primary* datapath units and *secondary* datapath units. Primary datapath units are those that have a lower bound set by the scheduling result, where a near-optimum result can be found by the individual binding of each datapath unit. On the other hand, secondary datapath units are dependent on the binding status of the primary datapath units, and their bindings should be optimized during datapath allocation.

Tables I and II summarize primary and secondary datapath units according to the architecture styles supported by our system. As shown, functional units are primary datapath units for any architecture. However, classification as to primary vs. secondary datapath units for memory and interconnection units depends on the target architecture.

As shown in Table I, the memory architectures supported are registers, register files, and multiport memories. Registers are primary datapath units for the register architecture. *Ports* are the primary datapath units for register files and multiport memories because the lower bound on the number of ports is determined during scheduling by the maximum number of simultaneous data accesses through ports. While groups of registers exist in both register files and multiport memories, the difference is in the number shared ports. Register files have only one port, so we just use that as the primary datapath unit. For both, the number of registers is dependent on the port binding that comes from scheduling, so the number of registers within these structures are secondary datapath units.

As shown in Table II, the interconnection architectures supported are either a random topology or a linear topology. In random topology, the number of required multiplexers and wires depends on the functional unit and memory unit binding, so they are secondary datapath units. When we consider physical layout during datapath allocation in Section 4, routing channels and interconnection delay also become secondary datapath units.

Table I. Primary and Secondary Datapath Units for Different Memory Architectures

Target Memory Architectures	Number of Ports	Datapath Units	
		Primary	Secondary
Register	Single	Functional units, registers	(Table II)
Register file	One shared R/W	Functional units, register files	Registers & (Table II)
Multiport memory	Several shared R/W	Functional units, ports	Registers & (Table II)

Table II. Primary and Secondary Datapath Units for Different Interconnection Architectures

Target Interconnection Architectures	Datapath Units	
	Primary	Secondary
Random topology	(Table I)	Multiplexers, wires, routing channels, interconnection delay
Linear topology	Buses & (Table I)	Multiplexers, tristate buffers

In linear topology, several connections between any two ports can share a bus. Because the number of buses necessary is determined by the maximum number of concurrent data transfers in any control step, buses are primary datapath units. Here, multiplexers and tristate buffers become secondary datapath units.

Given these definitions of primary and secondary datapath units, we can proceed to describe our algorithm.

3.2 Datapath Allocation Algorithm

Figure 1 shows the outline of the general datapath allocation algorithm. The algorithm consists of two phases, initial allocation and allocation improvement.

During initial allocation (line 1 in Figure 1), the initial bindings of primary datapath units are performed by efficient graph-based algorithms, as discussed in Sections 4 and 5. These algorithms each produce individual near-optimum solutions. Once these numbers are determined, they are fixed during allocation improvement in order to prohibit the exploration of unnecessary regions in the solution space and reduce the relatively long computation time of the allocation improvement phase.

During allocation improvement (lines 2–13 in Figure 1), new binding states are explored by moving or exchanging each element on two-dimensional (resource vs. control step) matrices for the bindings of all *primary* datapath units. Accurate modeling of datapath units for specific target architectures and direct objective functions enables us to estimate the amount of improvement on *secondary* datapath units accurately during each move or exchange of elements. As a result, the optimization of secondary datapath units is performed. Estimation methods for secondary datapath unit costs are explained in Sections 4 and 5.

```
Datapath_Allocation( )
{
1     S = i = Initial_allocation;
2     while( stopping criteria are not satisfied ) {
3          while( not at equilibrium ) {
4               select_improve_mode;
5               j = generate(i);
6               c = cost(j) - cost(i);
7               y = min( 1, exp(-c/T) );
8               r = random(0,1);
9               if( r < y ) i = j;
10         }
11         if( cost(i) < cost(S) ) S = i;
12         update_temperature;
13    }
14    return(S);
}
```

Fig. 1. Datapath allocation algorithm.

Allocation improvement uses a modification of the general *simulated annealing* technique [Kirkpatrick et al. 1983]. During allocation improvement, our scheme explores various binding alternatives and seeks an optimal solution by probabilistically exploring possible datapath structures. The *select_improve_mode* procedure chooses an allocation improvement mode randomly among possible operations that generate new binding states. In this way, our scheme does not sequentially optimize any of the subtasks during datapath allocation, but performs the optimizations concurrently. As a result, it avoids the sequential nature of other techniques that often fall prey to local optimum solutions.

A notable feature of our datapath allocation method is the efficient modification of the standard simulated annealing algorithm. Line 11 in Figure 1 ensures that "the best result so far" is stored in S. When the inner loop is completed at a given temperature, the result, $cost(i)$, is compared with the best result so far, $cost(S)$. If the new result is better than the best result, the best result is updated. Because the result i is used for the next iteration of the loop, in either case we can always obtain the best explored result while maintaining the ability to escape from local minima.

The long computation times typical of simulated annealing methods are reduced by two schemes: First, we prohibit the system from exploring unnecessary areas of the solution space during allocation improvement by finding the minimum numbers of primary datapath units in the initial allocation and by optimizing secondary datapath units during allocation improvement with these fixed numbers of primary datapath units. Second, we use a *dynamic cooling schedule* for the annealing process.

3.3 Dynamic Cooling Schedule

The conditions determined for the annealing process of simulated annealing are (1) the initial temperature, (2) the equilibrium condition, (3) the temperature decrement, and (4) the stopping condition. Most of the annealing processes in the synthesis literature involve the use of a predefined initial temperature, a fixed method to detect the equilibrium condition, a predefined constant function for temperature decrement, and a predefined stopping condition. This is called a *static cooling schedule*. However, for an annealing process to be not only problem-independent but also efficient, the parameters used in these conditions should be determined by the system itself and should not have any predefined values. That is, we should use a *dynamic cooling schedule*.

We base our dynamic cooling schedule on that of White [1984] and Huang et al. [1986]. We set the following conditions for the annealing process of the dynamic cooling schedule during allocation improvement.

The initial temperature. An initial exploration of the configuration space is performed before allocation improvement by simulated annealing. All the generated states are accepted during this exploration. The initial temperature is set by the standard deviation of costs in this exploration.

The equilibrium condition. We assume equilibrium if the number of accepted states with an energy in a certain range reaches the target value before the number of the accepted states with their costs outside the designated range exceeds the maximum tolerance limit.

Temperature decrement. The temperature decrement is controlled such that the average cost is decreased in a uniform ratio of average cost versus the logarithm of temperature.

Stopping condition. When the average cost is unchanged for several consecutive temperatures, the program terminates.

In order to demonstrate the abilities of this general allocation method, we applied it to two different problem environments, explained in the following sections.

4. DATAPATH ALLOCATION FOR PERFORMANCE AND AREA OPTIMIZATION

Our first application is a procedure for datapath allocation for performance and area optimization. First, we set a restriction for our target architecture to be a bit-sliced stack and a random topology interconnection. We then show how the procedure produces solutions using accurate cost functions for these special architectures.

4.1 Area and Performance Estimation

In bit-sliced stack architectures, a bit slice is constructed by laying out each bit slice unit using standard cells vertically and connecting different units

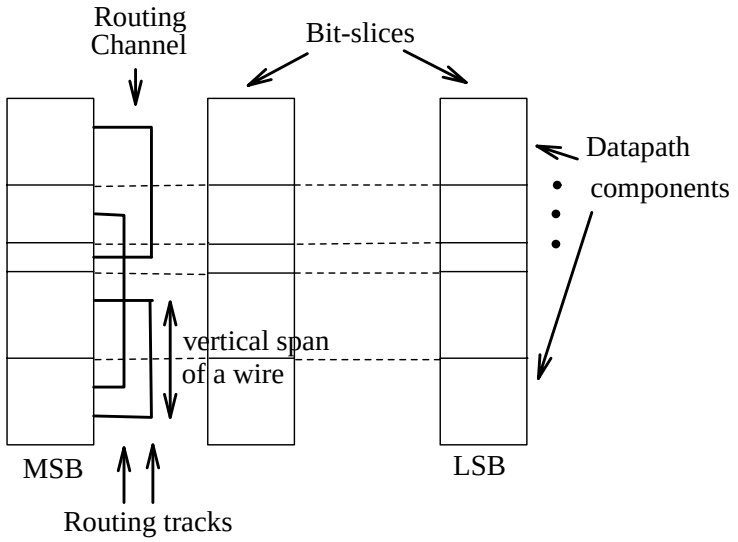

Fig. 2. Bit-sliced stack architecture for datapath.

within the same bit slice using a vertical routing channel [Gajski et al. 1994]. Figure 2 shows this architecture.

To minimize the area of a bit-slice design, we must minimize both the standard cell area and the area of the routing channel. Conventional datapath area metrics, such as numbers of functional units, registers, and multiplexers, can be used for minimizing the area of the standard cells. But in order to minimize the routing channel area, the width of the routing channel must also be minimized. This minimization can be achieved by minimizing the number of routing tracks required to implement the nets between the datapath components.

Performance can be improved by reducing data transfer delay during datapath allocation. To reduce data transfer delay, we can reduce wire lengths during datapath allocation. If we assume the execution delays of functional units are relatively even at each control step, we can improve performance by reducing the longest wire length on any path instead of reducing wire length of the critical path. Multicycling and chaining schemes [Pangrle and Gajski 1987] used in existing schedulers make this assumption reasonable; however, we propose a scheme to remove this assumption later. In bit-sliced stack architectures, the minimization of the longest vertical span (see Figure 2) in any connection can be used as a cost metric for interconnection delay.

Figure 3 illustrates how our algorithm handles these area and performance optimization metrics for datapath allocation of the "differential equation example" [Paulin et al. 1986]. A traditional scheduled data flow graph, functional unit, and register binding model are shown in Figure 3(a), (c), and (d). These functional unit and register binding models are used as *primary* datapath units for this architecture.

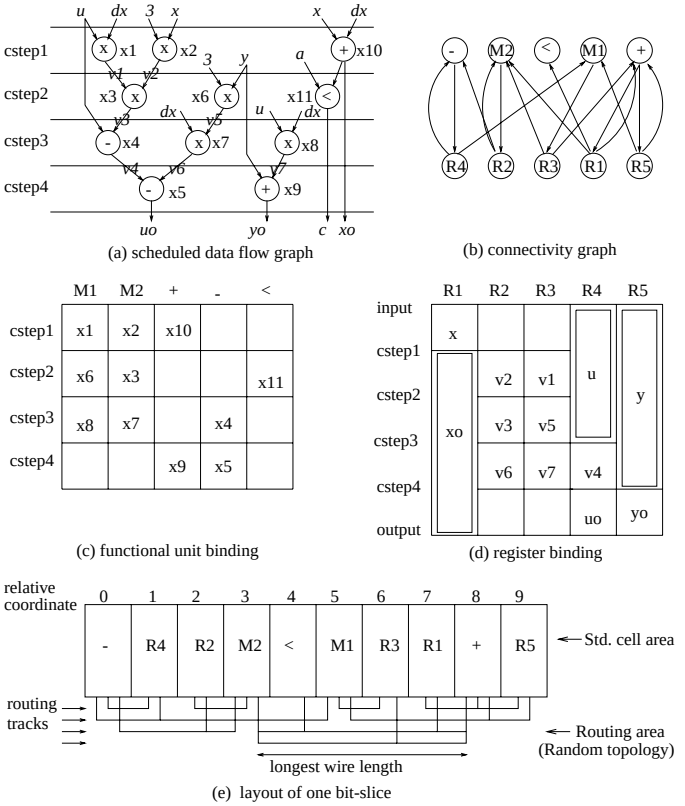

Fig. 3. Binding model for area and performance optimization.

In addition, we include the relative coordinates of each functional unit and register. These values are calculated by floor-planning phases during datapath allocation. Therefore, the gains in the *secondary* datapath units as defined in Table II (i.e., the number of routing tracks, the longest wire length, the number of wires, and multiplexer inputs) can be calculated during each move or exchange operation during binding.

Figure 3(e) shows the routing tracks and the longest wire length calculation based on this relative coordinate scheme for the functional unit and register bindings of Figure 3(c) and (d). If cell sizes are available in a library, they can be used to more accurately calculate these relative coordinates and wire lengths.

The estimation scheme for multiplexer inputs and wires uses the following formulation. Here, both operations and variables in the control data flow graph (CDFG) are defined as *tasks*, and both functional units and registers as *agents*. First, the following terminology is defined:

$agent(i) = j$: task i is bound by agent j;

$taskcon(i_1, i_2, l) = 1$: a data transfer exists from task i_1 to the l^{th} input of task i_2, otherwise 0;

agentcon(j_1, j_2, l): number of data transfers from agent j_1 to the l^{th} input of agent j_2;

in_portcon(j_{in}, l): number of connections on the l^{th} input of agent j_{in}.

Using these definitions, the following relations can be formulated for estimating the number of interconnection units such as wires, multiplexers, and multiplexer inputs.

(1) If *agentcon*(j_1, j_2, l) $\geq$ 1, a wire exists from agent j_1 to the l^{th} input of agent j_2.

(2) If *in_portcon*(j_{in}, l) $\geq$ 2, a multiplexer exists in front of the l^{th} input of agent j_{in}, and *in_portcon*(j_{in}, l) is added to the number of multiplexer inputs.

Figure 4 shows the algorithm for calculating the number of wires, multiplexers, and multiplexer inputs. *Num_wire, Num_mux* and *Num_muxinput* in Figure 4 represent the necessary number of wires, multiplexers, and multiplexer inputs.

An important point to note is that when there is a change in the binding of operations to functional units or the binding of variables to registers, the change is not always reflected in the numbers of routing tracks, wire length, number of wires, or multiplexer inputs. Change in these cost metrics only occurs when there is a change in the interconnection wiring. This interconnection binding is tied to functional unit and register binding by a many-to-one mapping. This fact is more easily understood by introducing the notion of a *connectivity graph*:

$G = \{V, E\}$
$V = \{\text{functional units and registers}\}$
$E = \{V_i \rightarrow V_j \mid \text{if one or more data transfers exist from } V_i \text{ to } V_j\}$

The change of cost metrics follows the change of edges in the connectivity graph. The connectivity graph also serves as a graph model for mincut partitioning-based floor-planning during datapath allocation. Figure 3(b) shows the connectivity graph corresponding to the functional unit and register binding shown in Figure 3(c) and (d) for the the data flow graph of Figure 3(a). Only a change in the edges in this derived connectivity graph would produce a change in the above-mentioned cost metrics. This is the reason we minimize the number of wires in the datapath, even though the number of wires does not directly influence datapath area.

4.2 Datapath Allocation With Bit-Slice Floor-Planning

Figure 5 shows the specialization of our general datapath allocation algorithm for this problem environment. During the initial allocation, functional unit binding is performed by a *clique partitioning algorithm* [Tseng and Siewiorek 1986] and register binding is performed by a *left edge algorithm* [Kurdahi and Parker 1987]. As a result, we obtain the minimal

```
Cost_cal_random( )
{
    for( every taskcon(i1,i2,l) ){
        j1 = agent(i1);
        j2 = agent(i2);
        agentcon(j1,j2,l)++;
    }
    Num_wire = 0;
    for( every agentcon(j1,j2,l) )
        if( agentcon(j1,j2,l) >= 1 ) Num_wire++;
    Num_mux = 0;
    Num_muxinput = 0;
    for( each input port l of every agent jin ){
        for( every other-type agent ji of agent jin )
            if( agentcon(ji,jin,l) >= 1 ) in_portcon(jin,l)++;
        if( in_portcon(jin,l) >= 2 ){
            Num_mux++;
            Num_muxinput += in_portcon(jin,l);
        }
    }
}
```

Fig. 4. Cost calculation for random topology.

numbers of functional units and registers required. Next, floor-planning is performed to determine the one-dimensional relative coordinates of each functional unit and register. Minimization of the number of maximum cuts is used as an object function for this one-dimensional floor-planning. The *mincut algorithm* [Kernighan and Lin 1970] with terminal propagation [Dunlop and Kernighan 1985] is recursively used for this purpose.

During allocation improvement (lines 3–16 in Figure 5), datapath area is minimized by reducing the total area for multiplexers and the routing channel area. In order to minimize the multiplexer area, which consists of multiplexers with different numbers of inputs, the total number of multiplexer inputs is minimized. The number of routing tracks is minimized to minimize the routing channel area. Also, in order to improve performance by minimizing the longest wire length, the longest vertical span for any connection is minimized. So the cost function during allocation improvement is

$$cost = P_m \times \#multiplexer\ inputs + P_w \times \#wires + P_{rtrack} \times \#routing\ tracks + P_{lmax} \times longest\ wire\ length \quad (1)$$

All the coefficients, P_m, P_w, P_{rtrack}, P_{lmax}, are weights that indicate the relative importance of each term corresponding to a datapath resource. Here, we define the number of wires and the number of multiplexer inputs as **core cost metrics**, and the number of routing tracks and the longest wire length as **auxiliary cost metrics**. Generally, we set the weights for the core cost metrics greater than the weights of the auxiliary cost metrics. The rationale for this decision is twofold. First, the minimization of the number of wires must be weighted more than the minimization of the auxiliary cost metrics, since the value of the auxiliary cost metrics will be changed only when the value of the number of wires is changed. Second, the number of multiplexer inputs is a more concrete cost metric than the auxiliary cost metrics during architectural synthesis, since it is assumed that detailed routing could affect the values of the auxiliary cost metrics. The recommended values for P_m, P_w, P_{rtrack} and P_{lmax} are discussed in Section 6.

During allocation improvement, a new binding state can be generated by any of the following operations, chosen at random by *select improve mode*:

—move or exchange a functional unit binding;

—move or exchange a register binding;

—swap the input binding of a functional unit in a commutative operation.

One feature of this datapath allocation algorithm is that floor-planning is tightly integrated into datapath allocation. Floor-planning is invoked each time, after completion of the inner loop of allocation improvement. If a new floor-plan produces a better cost for the number of routing tracks or the longest wire length, it replaces the current floor-plan. Note that only these two measures are affected by the floor-plan. According to our experiments, the use of refloor-planning produces far better results than keeping the floor-plan produced after the initial allocation, with only a small increase in computation time.

So far, we have discussed reducing the longest wire length to optimize performance. This scheme makes the assumption that the execution delays of functional units are relatively even at each control step. Multicycling and chaining schemes used by existing schedulers could validate this assumption. On the other hand, in order to remove this assumption we can emphasize the reduction of the critical path's wire length, rather than reduction of the longest wire. This can be done easily by the following technique: First, the critical path on each control step is found before datapath allocation by considering the execution delay of functional units and registers. Then, heavier weights are set for operations on the critical path. These weights are considered for calculating the longest wire length term of our cost function, which will reduce the wire length of the critical path rather than the overall longest wire length.

Finally, we note that for bit-sliced architectures it is relatively simple to model channel area and wire length to gain performance estimates. How-

```
Datapath_Allocation( )
{
1     S = i = Initial_allocation;
2     FP(best) = Floorplanning;
3     while( stopping criteria are not satisfied ) {
4          while( not at equilibrium ) {
5               select_improve_mode;
6               j = generate(i);
7               c = cost(j) - cost(i);
8               y = min( 1, exp(-c/T) );
9               r = random(0,1);
10              if( r < y ) i = j;
11         }
12         FP(i) = Floorplanning;
13         if( FP(i) is better than FP(best) )
                FP(best) = FP(i);
14         if( cost(i) < cost(S) ) S = i;
15         update_temperature;
16    }
17    return(S, FP(best));
}
```

Fig. 5. Algorithm for datapath allocation for bit-sliced architectures.

ever, the procedure shown in this section would be equally effective in any other environment where estimation techniques for area and wire length are available.

5. DATAPATH ALLOCATION FOR DIVERSE TARGET ARCHITECTURES

A second application of our general datapath allocation method is an approach that supports diverse target architectures using direct binding models and cost functions. Several specialized algorithms based on the proposed datapath allocation method explained in Section 3 have been developed for each of the supported target architectures, that is, combinations of memory architectures and interconnection architectures [Choi 1995]. Since this procedure emphasizes different capabilities of the general method, we do not consider physical layout information.

5.1 Target Architectures and Binding Models

The target architectures for this approach consist of multiport memories, register files, and registers for data storage. For multiport memory architectures, a unique bus is connected to each port. In Figure 6(a), we show two multiport memory modules that have one read/write port, one read-

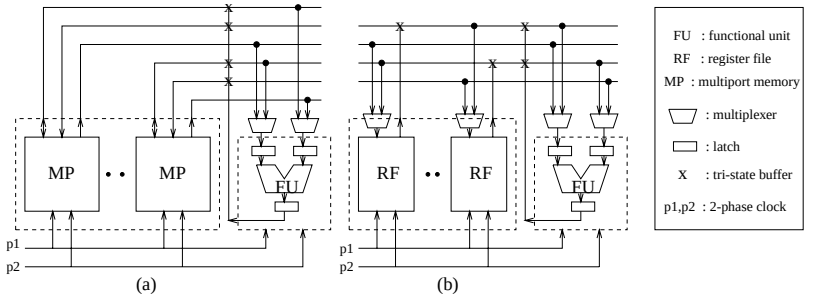

Fig. 6. Target architectures: (a) multiport memory architecture, (b) register file architecture.

only port, and one write-only port. For register file and register architectures, both linear and random topologies are supported. For linear/register file-based architectures, any register file can be connected to any bus as required by the allocation results, as shown in Figure 6(b). Figure 6 shows a two-phase clocking scheme for data transfers. Our system also supports a single-phase clocking scheme.

For these architectures, as in Table I, we add binding models for buses, register files, and multiport memories to the binding models for functional units and registers. Therefore, all hardware units in the datapath are treated as objects, and considered together in the search for the optimum final target architecture. Figure 7 shows our new binding models for datapath allocation of the same scheduled data flow graph shown in Figure 3.

Figure 7(a) shows the bus binding model for the interconnection allocation of a linear architecture using two-phase clocks. The left-top operator (variable or constant) and right-bottom variable (operator) pairs in each square of a two-dimensional matrix indicate that the data transfer between them is implemented using the bus specified by the corresponding column.

Figure 7(b) shows the memory binding model for a register file (RF) architecture. Variables are bound to register files specified by the corresponding columns of a two-dimensional matrix. The differences between this binding model and the register binding model in Figure 3(d) arise from the fact that we must consider a variable's *access time* for the binding of register files, whereas we must consider a variable's *life time* for the binding of registers. The life time of a variable is the time interval from its definition to its last use. The access time of a variable is the time when a variable is entered into or taken from a register file.

Figures 7(c) and (d) show memory binding models for a multiport memory architecture for the same system. Because a register can be accessed through different ports in different control steps and a port can be shared by different registers in different control steps, variables are bound to read and write *ports* specified by the corresponding columns of two-dimensional matrices. Because our system considers read and write ports separately and a variable must reside in one memory module, we must

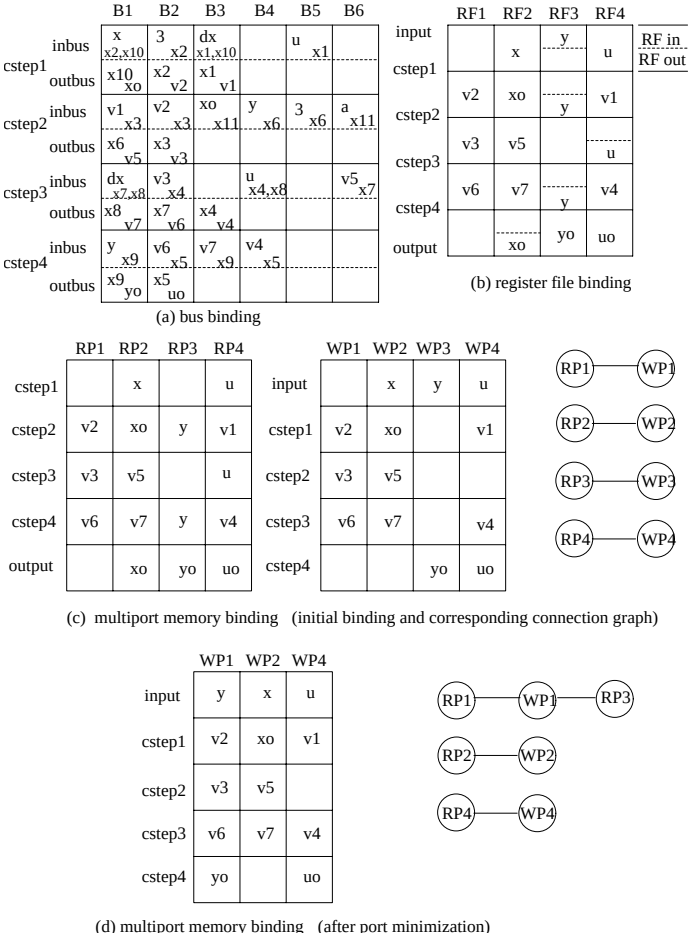

Fig. 7. Binding models for bus and different memory architectures.

maintain the relationship of which ports must exist in the same memory module. This information is maintained in a **connection graph:**

$G = \{V, E\}$

$V =$ {read ports and write ports}

$E =$ {ports represented by vertices on an edge must reside in one memory module}

When read-write ports are used for a target multiport memory, a read port and the corresponding write port are merged at the final step of datapath allocation. We use this binding model because sharing ports by registers gives a reduction in the final interconnection area.

The following formulation is added to theone in Section 4 for estimating multiplexer inputs and tristate buffers in a linear topology architecture. Register files or ports in multiport memories are defined as *agents*:

taskcon(i_1, i_2, l) = b: a data transfer exists from task i_1 to the l^{th} input of task i_2, and is connected by bus b in a linear topology architecture, otherwise 0;

in_buscon(j_{in}, l, b): number of data transfers from bus b to the l^{th} input of agent j_{in};

out_portcon(j_{out}, b): number of data transfers from the output of agent j_{out} to bus b.

Using these definitions, the following relations can be used to estimate the number of *secondary* datapath interconnection units such as connections to buses, multiplexers, multiplexer inputs, and tristate buffers:

(1) If *in_buscon*(j_{in}, l, b) $\geq$ 1, a connection exists between bus b and the l^{th} input of agent j_{in}.

(2) If *out_portcon*(j_{out}, b) $\geq$ 1, a tristate buffer exists between the output port of agent j_{out} and bus b.

Figure 8 shows the algorithm for calculating the number of tristate buffers, multiplexers, and multiplexer inputs for a linear topology. *Num_mux, Num_muxinput,* and *Num_tsbuffer* in Figure 8 represent the necessary number of multiplexers, multiplexer inputs, and tristate buffers.

The other *secondary* datapath units, the numbers of registers in a register file or in a multiport memory, are calculated by their density. The register density of a register file is calculated by the following procedure. The register density of a multiport memory is calculated similarly.

(1) Initialize a two-dimensional matrix(D) by 0:
D[#of *control_step*][#of *register_file*]

(2) For all variables assigned on a register file (RF_i)
$D[j][RF_i]$ is increased by 1, $\forall j \in$ {control steps in a variable's lifetime}

(3) Density of $RF_i = MAX_{j=1}^{control\ steps} D[j][RF_i]$

5.2 Datapath Allocation for Bus-Based Multiport Memory Architectures

Given the binding models above for various target architectures, *MandM* uses refinements of the general allocation algorithm in Section 3 to achieve datapaths with the minimum total memory and interconnection area. We explain in detail the refinements for bus-based multiport memory architectures as a representative case.

As given in Tables I and II, *Primary* datapath units for this architecture are the number of functional units, buses, and ports because the minimum number of ports is the maximum number of concurrently accessed variables in any control step, and we assume a unique bus is connected to each port. Following Figure 1 we perform initial allocation, followed by allocation improvement.

```
Cost_cal_linear( )
{
    for( every taskcon(i1,i2,l) ){
        j1 = agent(i1);
        j2 = agent(i2);
        b = taskcon(i1,i2,l);
        in_buscon(j2,l,b)++;
        out_portcon(j1,b)++;
    }
    Num_mux = 0;
    Num_muxinput = 0;
    for( each input port l of every agent jin ){
        for( each bus connection )
            if(in_buscon(jin,l,b) >= 1) in_portcon(jin,l)++;
        if(in_portcon(jin,l) >= 2){
            Num_mux++;
            Num_muxinput += in_portcon(jin,l);
        }
    }
    Num_tsbuffer = 0;
    for( each possible bus connection of every agent jout )
        if( out_portcon(jout,b) >= 1 ) Num_tsbuffer++;
}
```

Fig. 8. Cost calculation for linear topology.

5.2.1 *Initial Allocation*. The initial assignment of variables to ports is performed by the following procedure:

(1) Assignment of variables to read and write ports.

(2) Minimization of read and write ports.

(3) Partitioning of ports to different multiport memories.

In step 1, we set the restriction that a write access and following read accesses of a variable are assigned to a write port and a read port. Then, we can generate the compatibility graph using the variable's access time. In this compatibility graph, each vertex represents a variable and an edge exists between two vertices if the two variables do not overlap their access times. The assignment of variables to ports is performed by the *clique partitioning algorithm* for this compatibility graph. Also, the connection graph is initially generated where an edge exists between a write port and the corresponding read port, as shown in Figure 7(c), in order to keep the information that those ports must reside in one memory module.

In step 2, the number of read ports and write ports are minimized separately by merging ports when there is no access conflict. For example, ports WP1 and WP3 in Figure 7(c) are merged to port WP1 in Figure 7(d). The connection graph is updated when ports are merged.

In step 3, the maximum number of permitted ports for target multiport memories is examined, and ports are partitioned to different multiport memories. For example, if multiport memories allow 4R-3W (4 Read Ports and 3 Write Ports), Figure 7(c) and (d) are synthesized by one multiport memory. If they allow 2R-2W (2 Read Ports and 2 Write Ports), it may be synthesized by two multiport memories, and {RP1, RP3, WP1}{RP2, RP4, WP2, WP4}is a possible partition.

5.2.2 *Allocation Improvement*. During allocation improvement, a new binding state can be generated by any of these operations:

—move or exchange a functional unit binding;

—swap the input binding of a functional unit in a commutative operation;

—move or exchange a port assignment of a multiport memory.

After each step the number of *secondary* datapath units (i.e., multiplexers, tristate buffers, and registers) are recalculated as described above. The cost function is based on the interconnection cost and the memory unit cost. The interconnection cost is

$$interconnection\ cost = P_m \times \#multiplexer\ input + P_b \times \#tristate\ buffer \quad (2)$$

where, P_m, P_b are weights that indicate the relative importance of each objective.

An important point to note is that we must maintain the rule that a variable resides in one memory module when an exchange has happened between different memory modules. Therefore, if a move happens between different multiport memories, the following memory unit cost must be updated as well:

$$memory\ unit\ cost = P_r \times \#register \quad (3)$$

where P_r is a weight that indicates the relative importance of this objective.

6. EXPERIMENTAL RESULTS

Both of the datapath allocation procedures presented above have been implemented in a system called *MandM*. These particular specializations of the general allocation method were chosen in order to compare our results to published results in datapath allocation. Also, a scheduler system has been implemented in order to realize the potential of the proposed datapath allocation method, to evaluate it for a larger set of examples, and to establish a complete stand-alone architectural synthesis system for diverse

Table III. Comparing Allocation EWF Results for Register-Random Topology Architecture

System	Steps	FU *	FU +	Reg	Mux	Mux input	Mux 2-1	Wire
MandM	17	3	3	11	10	26	16	37
SALSA				11	n/a	n/a	18	n/a
MandM	17	2P	3	11	10	26	16	37
HAL				12	n/a	31	n/a	n/a
SE				11	12	31	19	n/a
SALSA				11	n/a	n/a	17	n/a
MandM	19	2	2	11	6	21	15	31
HAL				12	n/a	29	n/a	n/a
EMUCS				12	n/a	34	n/a	n/a
ELF				11	10	30	20	n/a
SE				10	11	31	20	n/a
SALSA				11	n/a	n/a	16	n/a
STAR				10	6	27	21	43
MandM	19	1P	2	10	8	23	15	30
HAL				12	n/a	26	20	n/a
ELF				11	11	30	19	n/a
SE				11	9	25	16	n/a
SALSA				11	n/a	n/a	16	n/a
MandM	21	1	2	11	6	19	13	28
HAL				12	n/a	31	n/a	n/a
SE				11	8	24	16	n/a
SALSA				11	n/a	n/a	16	n/a

system design environments [Choi 1995]. *MandM* was implemented in C under the UNIX operating system on a SUN Sparcstation.

6.1 Allocation Results for Area and Performance Optimization

This section analyzes allocation results for register-random topology architectures. First, our results are compared with those of other systems in terms of conventional cost metrics. Then, the additional ability of our system in performing area and performance optimization are examined.

Table III shows a comparison of allocation results for the fifth-order elliptical wave filter (EWF) example [Paulin and Knight 1989]. We used our scheduler to generate scheduling results from the VHDL behavioral description. For allocation, the weighting values of P_{rtrack} and P_{lmax} of Eq. (1) were set to 0 because we considered only conventional cost metrics for these runs. As shown in Table III, *MandM* produced far better results than most of the representative datapath allocators. Also, *MandM* run times for the EWF ranged between 3 and 6 minutes. In comparison, *ELF* [Ly et al.1990] took 60 minutes; *SE* [Ly and Mowchenko 1993] 30 to 35 minutes; *SALSA* [Krishnamoorthy and Nestor 1992] 8 to 10 minutes; and *STAR* [Tsai and Hsu 1992] took several minutes. So the run times of *MandM* are quite competitive with those systems.

Table IV shows the comparison of allocation results for the discrete cosine transform (DCT) example [Krishnamoorthy and Nestor 1992]. We

Table IV. Comparing Allocation DCT Results for Register-Random Topology Architecture

System	Steps	FU *	FU +/-	Reg	Mux	Mux input	Mux 2-1	Wire
MandM	10	4	4	15	15	44	29	60
SALSA	10	4	4	15	n/a	n/a	30	n/a
MandM	14	3	3	14	14	42	28	54
SALSA	14	3	3	14	n/a	n/a	29	n/a
MandM	18	2	2	15	9	39	30	53
SALSA	19	3	2	14	n/a	n/a	30	n/a

followed the same procedure as above for the EWF example. As shown in Table IV, our results are slightly better than those of *SALSA*. *MandM* run times for the DCT ranged between 6 and 14 minutes, whereas *SALSA* run times ranged between 10 and 17 minutes [Krishnamoorthy and Nestor 1992].

Using conventional datapath cost metrics we have shown that *MandM* produces very competitive results for register and random topology architectures. However, there are many near-optimum solutions in terms of conventional datapath cost metrics that are far from optimum when we examine them in terms of cost metrics based on physical layout. The advantages of *MandM* are based on its flexibility to easily incorporate cost metrics such as area and delay during datapath allocation. When the target architectures are restricted to bit-sliced stacks, we can get good estimates of area and delay from floor-planning. In order to run under this mode, we activate the floor-planning stages within the datapath allocation. We then increase the weights of the *auxiliary cost metrics*, P_{rtrack} and P_{lmax} of Eq. (1) from 0 to 5, while we fix the weights of the *core cost metrics*, P_w and P_m of Eq. (1), to 5.

Table V shows the allocation results of *MandM* under these execution conditions. Table V(a) shows design space exploration for the EWF example by changing the auxiliary cost metrics, while the scheduling and the primary datapath units are maintained at 19 control steps, 2 adders, 2 multipliers, and 11 registers. Table V(b) shows design space exploration for the EWF example by changing the auxiliary cost metrics, while the scheduling and the primary datapath units are maintained at 19 control steps, 2 adders, 1 pipelined multiplier, and 10 registers. R_{track} is the number of routing tracks and L_{max} is the longest wire length.

We can see the following trends from these tables: The number of routing tracks is reduced greatly as P_{rtrack} increases from zero. The longest wire length is reduced as P_{lmax} increases from zero. In addition, the longest wire length generally explores a better solution space when P_{rtrack} increases and the number of routing tracks generally explores a better solution space when P_{lmax} increases. For the latter case, improvement is not due to the weight of the respective metric but due to interdependency with the other

Table V. Allocation Results for Register-Random Topology Architecture

P_{rtrack}	Mux input	Wire	R_{track}	L_{max}	P_{lmax}	Mux input	Wire	R_{track}	L_{max}
0	21	31	18	11	0	21	31	18	11
1	20	30	10	12	1	20	31	12	8
2	20	30	10	13	2	20	31	12	9
3	21	33	12	11	3	19	31	15	9
4	20	32	10	8	4	21	33	16	9
5	20	32	10	8	5	21	32	11	8
(a) EWF example (2 adders, 2 multipliers)									
0	23	30	16	9	0	23	30	16	9
1	23	31	10	7	1	22	31	14	7
2	22	30	9	6	2	23	31	12	7
3	24	33	13	9	3	23	31	12	7
4	23	31	11	9	4	23	32	15	6
5	23	31	10	7	5	25	32	11	7
(b) EWF example (2 adders, 1 pipelined multiplier)									
0	44	57	26	18	0	44	57	26	18
1	45	59	23	20	1	44	57	20	16
2	44	58	17	19	2	43	57	19	12
3	44	57	16	15	3	45	58	22	15
4	48	60	15	15	4	48	61	22	14
5	44	58	16	17	5	48	62	29	16
(c) DCT example (2 adders, 2 subtracters, 3 multipliers)									
0	48	57	30	18	0	48	57	30	18
1	47	56	20	17	1	47	55	18	12
2	47	56	19	17	2	46	55	15	12
3	46	55	18	16	3	46	55	18	12
4	48	55	17	15	4	47	55	18	12
5	48	56	15	13	5	49	57	20	15
(d) DCT example (2 adders, 2 subtracters, 2 pipelined multipliers)									

weight. Also, the number of multiplexer inputs and the number of wires generally maintain their values as P_{rtrack} and P_{lmax} are increased from 0 to 5.

From these examples we can conclude that if we set the weights for the auxiliary cost metrics to nonzero but less than the core cost metrics, *MandM* produces results that are all nearly optimum solutions in terms of conventional datapath metrics. In addition, by varying these auxiliary cost metrics on the basis of user requirements *MandM* can explore diverse area-optimized and performance-optimized datapath structures while maintaining nearly optimum values of conventional datapath allocation metrics.

Table V(c) shows design space exploration for the DCT example by changing the auxiliary cost metrics, while scheduling and the primary datapath units are maintained at 14 control steps, 2 adders, 2 subtractors, 3 multipliers, and 15 registers. Table V(d) shows design space exploration for the DCT example by changing the auxiliary cost metrics, while schedul-

ing and the primary datapath units are maintained at 13 control steps, 2 adders, 2 subtractors, 2 pipelined multipliers, and 13 registers. We can see almost the same improvement trends as in the EWF example. However, Table V(c) shows that the core cost metrics become worse when P_{rtrack} and P_{lmax} approach the values of P_m and P_w.

MandM run times for the EWF with floor-planning ranged between 180 and 370 seconds. Also, *MandM* run times for DCT runs with floor-planning ranged between 8 and 12 minutes. For reference, run times for datapath allocation without floor-planning ranged between 170 and 320 seconds for the EWF runs, and between 6 and 14 minutes for the DCT runs. We can confirm that our scheme for tight integration of floor-planning with datapath allocation does not seriously impact computation times.

6.2 Allocation Results for Diverse Embedded Memory Architectures

This section analyzes the allocation results for multiport memory architectures. First, we compare our results with those of other systems. We then examine the advantages of the specific multiport memories and also analyze allocation results for register-file and linear topology architectures.

6.2.1 *Allocation Results for Multiport Memories*. Table VI(a) shows the comparison of allocation results of the EWF example for multiport memory architectures. The 19 control steps scheduling result was borrowed from Paulin and Knight [1989] and used as input. Two adders, one pipelined multiplier, and a two-phase clocking scheme were used, which were the same conditions used by the other systems under comparison. R/W notation indicates the number of read-write ports in a multiport memory. Table VI(a) indicates that *MandM* produced better results than other systems for almost all hardware resources except the number of tristate buffers.

Another comparison of allocation results for multiport memory architectures is the example of the sixth-order elliptic bandpass filter from Papachristou and Konuk [1990]. Our scheduler was used to generate scheduling results. We made the same assumption as Papachristou and Konuk [1990], which is that multiplications, additions, and subtractions each consume one control step. Table VI(b) shows our scheduling and allocation results with two other existing results that consider multiport memory architectures. Our scheduler ran under two different input conditions: First, multiplications are implemented by multipliers, and addition and subtraction are implemented by ALUs. Second, multiplications, additions, and subtractions are implemented by ALUs. Thirteen control steps are needed under the first input condition, and ten control steps are needed under the second input condition. One more port is needed under the second input condition, and as a result one more bus is needed. Both of our runs produced better results than the other systems in terms of the necessary numbers of registers and tristate buffers. Our result under the first input condition produced better results than other systems for the number of multiplexer inputs. Our results under the second input condition were similar to the other systems.

Table VI. Comparison of Allocation Results for Multiport Memory Architecture

System	*MandM*	*Kim&Liu's*	*MAP*
Multiport module	2	2	2
	(1 3R/W, 1 2R/W)	(1 3R/W, 1 2R/W)	(not specified)
Port	5	5	5
Bus	5	5	5
Register	10	12	14
Mux input	7	9	10
Tristate buffer	7	5	5

(a) EWF example

System	*MandM*	*MandM*	*Kim&Liu's*	*MAP*
Steps	13	10	11	11
FU *	1		1	1
FU ALU	2	3	2	2
Multiport module	2	2	2	2
	(1 3R/W, 1 2R/W)	(2 3R/W)	(1 3R/W, 1 2R/W)	(not specified)
Port	5	6	5	5
Bus	5	6	5	5
Register	9	9	11	11
Mux input	8	12	10	12
Tristate buffer	6	6	6	7

(b) bandpass filter example

Having shown that *MandM* produces competitive results for multiport memory architectures, we now show the improvements possible when the cost metrics of multiport memories are considered directly and simultaneously with other cost metrics. Table VII shows how the difference in port restrictions affects the allocation results, especially the interconnection cost metrics. For the EWF example, the same scheduling result was used as in Table VI(a). For the DCT example, a 14 control step scheduling result generated by our scheduler was used. As *primary* datapath units, 3 multipliers, 2 adders, 2 subtractors, and a single-phase clocking scheme were used. Also, 16 ports and 16 buses were used due to the single-phase clocking scheme. The iRjW notation indicates the number of read-only ports and write-only ports in a multiport memory.

From this result, we can see that the necessary number of registers, multiplexers, multiplexer inputs, and tristate buffers, which are all *secondary* datapath units, are reduced when the multiport memory modules permit more ports. However, if datapath allocation was performed sequentially—that is, registers and interconnection units are allocated, then registers are grouped into multiport memories (or these tasks are performed in the reverse order)—these improvements would not be possible.

6.2.2 *Allocation Results for Register File-Based Architectures*. Table VIII shows the comparison of allocation results of the EWF example for

Table VII. Allocation Results for Multiport Memory Architecture

Port restriction	Multiport module	Reg	Mux	Mux input	Tristate buffer
1R/W	5 (5 1R/W)	13	4	13	6
2R/W	3 (2 2R/W, 1 1R/W)	10	4	11	6
3R/W	2 (1 3R/W, 1 2R/W)	10	3	7	7
5R/W	1 (1 5R/W)	9	3	6	6
		(a) EWF example			
1R,1W	8 (8 1R1W)	17	10	31	19
2R,2W	4 (4 2R2W)	15	9	25	13
3R,3W	3 (2 3R3W, 1 2R2W)	15	7	21	12
8R,8W	1 (1 8R8W)	15	7	18	9
		(b) DCT example			

Table VIII. Comparison of Allocation Results for Register File-Linear Topology Architecture

System	Steps	FU *	FU +	RF	Bus	Reg	Mux	Mux input	Tri-state buffer
MandM	17	3	3	7	4	10	6	17	11
SPAID		2	3	6	6	17	n/a	26	n/a
MandM	17	2P	3	7	5	11	7	21	13
SPAID		2P	3	6	6	17	n/a	26	n/a
MandM	19	2	2	5	4	10	4	11	11
MandM	19	1P	2	5	3	10	5	12	8
SPAID		1P	2	5	5	19	n/a	17	n/a
MandM	21	1	2	5	5	10	5	12	12
SPAID		1	2	6	6	19	n/a	19	n/a
		(a) Comparison with *SPAID* using EWF example							
MandM	17	2	2	5	4	9	4	12	13
STAR	17	2	2	3	5	11	n/a	16	13
		(b) Comparison with *STAR* using EWF example							

register file and linear topology architectures. A two-phase clocking scheme is assumed in these allocations.

Table VIII(a) compares our allocation results with the results of *SPAID* [Haroun and Elmasry 1989]. We used a result of our scheduler as input for this allocation. *MandM* produced better results than *SPAID* for each of the different designs. Notably, *MandM* needed far fewer buses, registers, and multiplexer inputs. This is due to the following features of *MandM*: (1) any register file can be connected to any bus as required in *MandM*, whereas each register file has a fixed bus in *SPAID*; (2) *MandM* does bus minimization [Choi 1995]; (3) because all allocation subtasks are performed simultaneously in *MandM*, a higher level of optimization is possible for the interconnection units, since they are highly interdependent with the functional units and memory units.

Table VIII(b) compares our allocation results with the results of *STAR* [Tsai and Hsu 1992]. The 17 control steps scheduling result is borrowed from Tsai and Hsu [1992] and used as input. *MandM* produced better results than *STAR* for most hardware resources except the necessary number of register files.

7. SUMMARY AND CONCLUSIONS

In this paper we present a robust method for datapath allocation, which is based on the need for handling the complex models of datapath units, the direct consideration of objective functions, and simultaneous optimization of multiple objective functions. It consists of a new binding model construction scheme based on primary and secondary datapath units and a flexible optimization technique based on simulated annealing. We illustrate the powerful features of the method by formulating two different allocation procedures for two different datapath allocation problems.

First, we developed an approach that incorporates more accurate interconnection area and delay estimates than previous approaches for a bit-sliced stack and random topology architectures. To minimize chip area, we considered the number of routing tracks as well as the number of functional units, registers, and multiplexers. To optimize execution time, we reduced the longest wire length used in any control step. In order to estimate the number of routing tracks and the longest wire length accurately, floor-planning phases were tightly integrated into the datapath allocation algorithm. Experimental results show our algorithm can explore diverse area-optimized and performance-optimized datapaths to meet user requirements, while maintaining the optimal values of conventional datapath area metrics.

Second, we developed another allocation procedure that can handle registers, register files, and multiport memories for data storage, and random and linear topologies for interconnection architectures using direct binding models and objective functions. The unique features of this approach include: (1) We use more extensive binding models than those previously considered. In particular, data transfers are bound directly to buses, and variables are bound directly to register files or multiport memories according to the target architecture. (2) Our cost functions consider all hardware units directly, and their costs are reevaluated during the optimization phase of allocation. Experimental results show our algorithm is not only competitive for embedded memory architectures, but also flexible enough to efficiently handle a large variety of other target architectures.

As future work, we are exploring how the datapath allocation approach developed for a bit-sliced stack and random topology architectures can be extended to more general target architectures. We are also examining how the effect of control logic on the total chip area and system performance can be considered during datapath allocation.

REFERENCES

AHMAD, I. AND CHEN, C. Y. R. 1991. Post-processor for data path synthesis using multiport memories. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design* (ICCAD '91, Santa Clara, CA, Nov. 11-14) IEEE Computer Society Press, Los Alamitos, CA, 276–279.

BALAKRISHNAN, M., BANERJI, D. K., MAJUMDAR, A. K., LINDERS, J. G., AND MAJITHIA, J. C. 1990. Allocation of multiport memories in data path synthesis. *IEEE Trans. Comput.-Aided Des. 7*, 4 (Apr. 1990), 536–540.

CHOI, K. 1995. A robust architectural synthesis method for realistic system design. Ph.D. Dissertation. Dept. of Electrical Engineering, University of Pittsburgh.

DEVADAS, S. AND NEWTON, R. 1989. Algorithms for hardware allocation in data path Synthesis. *IEEE Trans. Comput.-Aided Des. 8*, 7 (Jul. 1989), 768–781.

DUNLOP, A. E. AND KERNIGHAN, B. W. 1985. A procedure for placement of standard-cell VLSI circuits. *IEEE Trans. Comput.-Aided Des. 4*, 1 (Jan. 1985), 92–98.

FANG, Y.-M. AND WONG, D. F. 1994. Simultaneous functional-unit binding and floorplanning. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD '94, San Jose, CA, Nov. 6–10, 1994), J. A. G. Jess and R. Rudell, Eds. IEEE Computer Society Press, Los Alamitos, CA, 317–321.

GAJSKI, D. D., VAHID, F., NARAYAN, S., AND GONG, J. 1994. Design quality estimation. In *Specification and Design of Embedded Systems* Prentice-Hall, Inc., Upper Saddle River, NJ, 233–307.

GEBOTYS, C. H. AND ELMASRY, M. I. 1993. Global optimization approach for architectural synthesis. *IEEE Trans. Comput.-Aided Des. 12*, 9, 1266–1278.

HAROUN, B. S. AND ELMASRY, M. I. 1989. Architectural synthesis for DSP silicon compilers. *IEEE Trans. Comput.-Aided Des. 8*, 4 (Apr. 1989), 431–447.

HSIEH, Y.-W., LEVITAN, S. P, AND PANGRLE, B. M. 1993. Incorporating interconnection delays in VHDL behavioral synthesis. In *Proceedings of the 4th on ACM/SIGDA Physical Design Workshop* 175–186.

HUANG, M. D., ROMEO, F., AND SANGIOVANNI-VINCENTELLI, A. 1986. An efficient general cooling schedule for simulated annealing. In *Proceedings of the International Conference on Computer-Aided Design* 381–384.

JANG, H.-J. AND PANGRLE, B. M. 1993. A grid-based approach for connectivity binding with geometric costs. In *Proceedings of the International Conference on Computer-Aided Design* (ICCAD '93, Santa Clara, CA, Nov. 7–11, 1993), M. Lightner and J. A. G. Jess, Eds. IEEE Computer Society Press, Los Alamitos, CA, 94–99.

KERNIGHAN, B. AND LIN, S. 1970. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.* (Feb.).

KIM, T. AND LIU, C. L. 1993. Utilization of multiport memories in data path synthesis. In *Proceedings of the 30th International Conference on Design Automation* (DAC'93, Dallas, TX, June 14–18), A. E. Dunlop, Ed. ACM Press, New York, NY, 298–302.

KIRKPATRICK, S., GELATT, C. D., JR., AND VECCHI, M. P. 1983. Optimization by simulated annealing. *Science 220*, 4598 (May), 671–680.

KNAPP, D. W. 1990. Feedback-driven datapath optimization in Fasolt. In *Proceedings of the International Conference on Computer-Aided Design* (ICCAD'90) IEEE Computer Society Press, Los Alamitos, CA, 300–303.

KRISHNAMOORTHY, G. AND NESTOR, J. A. 1992. Data path allocation using an extended binding model. In *Proceedings of the 29th ACM/IEEE Conference on Design Automation* (DAC '92, Anaheim, CA, June 8-12), D. G. Schweikert, Ed. IEEE Computer Society Press, Los Alamitos, CA, 279–284.

KURDAHI, F. J. AND PARKER, A. C. 1987. REAL: A program for REgister ALlocation. In *Proceedings of the 24th ACM/IEEE Conference on Design Automation* (DAC '87, Miami Beach, FL, June 28-July 1, 1987), A. O'Neill and D. Thomas, Eds. ACM Press, New York, NY, 210–215.

LINDO, 19XX. Linear interactive and discrete optimizer for linear, integer, and quadratic programming problems. LINDO Systems, Inc..

LY, T. A., ELWOOD, W. L., AND GIRCZYC, E. F. 1990. A generalized interconnect model for data path synthesis. In *Proceedings of the ACM/IEEE Conference on Design Automation* (DAC '90, Orlando, FL, June 24-28), R. C. Smith, Ed. ACM Press, New York, NY, 168–173.

LY, T. AND MOWCHENKO, J. 1993. Applying simulated evolution to high level synthesis. *IEEE Trans. Comput.-Aided Des. 12*, 3 (Mar. 1993), 389–409.

MCFARLAND, M. C. AND KOWALSKI, T. J. 1990. Incorporating bottom-up design into hardware synthesis. *IEEE Trans. Comput.-Aided Des. 9*, 9 (Sept. 1990), 938–950.

MCFARLAND, M. C. AND PARKER, A. C. 1990. The high-level synthesis of digital systems. *IEEE Computer 78*, 2 (Feb. 1990), 301–317.

NESTOR, J. A. AND KRISHNAMOORTHY, G. 1992. SALSA: A new approach to scheduling with timing constraints. *IEEE Trans. Comput.-Aided Des. 12*, 8 (Aug.), 1107–1122.

PANGRLE, B. M. AND GAJSKI, D. D. 1987. Slicer: A state synthesizer for intelligent silicon compilation. In *Proceedings of the International Conference on Computer-Aided Design* (ICCAD) 42–45.

PAPACHRISTOU, C. A. AND KONUK, H. 1990. A linear program driven scheduling and allocation method followed by an interconnect optimization algorithm. In *Proceedings of the ACM/IEEE Conference on Design Automation* (DAC '90, Orlando, FL, June 24-28), R. C. Smith, Ed. ACM Press, New York, NY, 77–83.

PAULIN, P., KNIGHT, J., AND GIRZYC, E. 1986. HAL: A multi-paradigm approach to datapath synthesis. In *Proceedings of the Conference on Design Automation*

PAULIN, P. G. AND KNIGHT, J. P. 1989. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Trans. CAD 8*, 6 (June 1989), 661–679.

PAULIN, P. G. AND KNIGHT, J. P. 1989. High-level synthesis benchmark results using a global scheduling algorithm. In *Logic and Architecture Synthesis for Silicon Compilers* North-Holland Publishing Co., Amsterdam, The Netherlands, 211–228.

RAMACHANDRAN, C., KURDAHI, F. J., GAJSKI, D. D., WU, A. C.-H., AND CHAIYAKUL, V. 1992. Accurate layout area and delay modeling for system level design. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD '92, Santa Clara, CA, Nov. 8–12), L. Trevillyan, Ed. IEEE Computer Society Press, Los Alamitos, CA, 355–361.

RIM, M., MUJUMDAR, A., JAIN, R., AND DE LEONE, R. 1994. Optimal and heuristic algorithms for solving the binding problem. *IEEE Trans. Very Large Scale Integr. Syst. 2*, 2 (June 1994), 211–225.

TSAI, F. S. AND HSU, Y. C. 1992. An automatic data path allocator. *IEEE Trans. Comput.-Aided Des. 11*, 9 (Sep. 1992), 1053–1064.

TSENG, C. AND SIEWIOREK, D. P. 1986. Automated synthesis of data paths in digital systems. *IEEE Trans. Comput.-Aided Des. 5*, 3 (July 1986), 379–395.

WENG, J.-P. AND PARKER, A. C. 1991. 3D scheduling: High-level synthesis with floorplanning. In *Proceedings of the 28th ACM/IEEE Conference on Design Automation* (DAC '91, San Francisco, CA, June 17–21), A. R. Newton, Ed. ACM Press, New York, NY, 668–673.

WHITE, S. R. 1984. Concepts of scale in simulated annealing. In *Proceedings of the International Conference on Computer Design* (ICCD '84) 646–651.

Received: December 1995; revised: July 1996; accepted: March 1998