Algorithms for Hardware Allocation in Data Path Synthesis

SRINIVAS DEVADAS, MEMBER, IEEE AND A. RICHARD NEWTON, FELLOW, IEEE

***Abstract*—The most creative step in synthesizing data paths executing software descriptions is the hardware allocation process. New algorithms for the *simultaneous cost/resource constrained allocation of registers, arithmetic units, and interconnect* in a data path have been developed. The entire allocation process can be formulated as a two-dimensional placement problem of microinstructions in space and time. This formulation readily lends itself to the use of a variety of heuristics for solving the allocation problem. We present simulated-annealing-based algorithms which provide excellent solutions to this formulation of the allocation problem. These algorithms operate under a *variety of user-specifiable constraints on hardware resources and costs*. They also incorporate conditional resource sharing and simultaneously address all aspects of the allocation problem, namely register, arithmetic unit and interconnect allocation, while effectively exploring the existing tradeoffs in the design space.**

I. Introduction

THE GOAL OF the data path synthesis step in a behavioral synthesis system is to produce register-transfer (RT) level hardware designs from an architectural description of a computer or to produce an RT design which implements a given program described in a high-level language in hardware. Significant effort has gone into the development of techniques for automated data path synthesis (e.g. [1]–[5]) in recent years. However, even now, effective and versatile procedures are not available.

Given a fixed amount of hardware resources, global optimizations of microcode can be performed using such techniques as trace scheduling [6]. Microprograms can be made more efficient and parallel. In hardware allocation, both the schedule of operations and the numbers of computational/storage units have to be decided.

Initial work to tackle this problem included the development of a mathematical model for the data path [7] to describe the conditions and relationships to be satisfied. Mixed integer–linear programming techniques were used. Unfortunately, even for very small specifications the cost of generating a design exploded rapidly.

The expert system approach was taken in the DAA [4], [8] system. Design rules were collected, and based on these design rules, a rule-based data memory allocator was developed. As is the case with most rule-based techniques, only local optimization was possible and extensive changes could not be made to the input description to attain a globally optimal solution. Similar problems afflicted the allocators described and implemented in [9] and [10]. Global optimization steps have been introduced into the expert system approach [5], but DAA has been used mainly to synthesize general-purpose computer data paths.

A more global algorithmic approach to the allocation problem was first taken by Tseng and Siewiorek [11], [12]. FACET is an automatic data path synthesis program which minimizes the number storage elements, data operators, and interconnection units. However, FACET performs these steps *sequentially* and independently of the following task(s). The entire design space is thus not explored.

An approach to hardware allocation based on graph grammars and scheduling was taken by Gircyzc [13]. The USC MAHA system [3] uses critical path determination to perform hardware allocation. The heuristics used to guide scheduling are based on the concept of the *freedom* of an operation. A force-directed scheduling approach to hardware allocation has been taken in [14]. The optimization step is global and uses heuristics based on predecessor and successor *forces* on an operation. The different heuristics used in both these scheduling algorithms [3], [14] may result in locally minimum solutions.

Other efforts in this area include Trickey's work [2], [15] and the synthesis of digital signal processor data paths in the CATHEDRAL [16] and SEHWA systems [3], [17].

Trickey [15] addressed the problem of extracting parallelism from a program and maximally scheduling the operations in the program while meeting a user-specified bound on each kind of processing unit(s). This paper is concerned with *hardware allocation*, where the decisions on the number of processing units, storage elements, and their interconnections are made. The scheduling problem is only a small part of the allocation process.

Other high-level systems currently being developed are the CMU System Architect's Workbench [18], Stanford's HERCULES system [19], and the BECOME [20] and BRIDGE [21] systems at AT&T Bell Laboratories.

A recent tutorial [22] classifies scheduling algorithms on the basis of (1) the interaction between scheduling and

Manuscript received December 1, 1987; revised April 29, 1988, October 5, 1988, and January 24, 1989. This work was supported in part by the Semiconductor Research Corporation and in part by the Digital Equipment Corporation. The review of this paper was arranged by Associate Editor M. R. Lightner.

S. Devadas is with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

A. R. Newton is with the Department of Electrical Engineering, University of California, Berkeley, CA 94720.

IEEE Log Number 8927532.

0278-0070/89/0700-0768$01.00 © 1989 IEEE

data path allocation and (2) the type of scheduling algorithm used. For example, in the early DAA system [8] and FLAMEL [15], a limit (or no limit) on the number of functional units available is placed during scheduling. The MAHA system [3] and HAL [14] develop the schedule and resource requirements simultaneously. Scheduling algorithms used in these approaches are iterative/constructive (e.g. [3], [14]) or transformational (e.g. [23]). Our approach performs scheduling and resource allocation simultaneously and uses the iterative optimization technique known as simulated annealing [24].

In this paper, we present new algorithms for the *simultaneous cost/resource constrained allocation of registers, arithmetic units, and interconnect* in a data path. These algorithms operate under a variety of user-specifiable constraints on hardware resources and costs. There are three main differences between this approach and others taken in the past (e.g., [1]–[3]). First, all the allocation subproblems, namely, arithmetic unit, register, and interconnect allocation, are tackled simultaneously, rather than sequentially or iteratively. Second, the optimization is completely global in nature—the entire data path is optimized. Third, we have used a probabilistic hill-climbing algorithm [25], simulated annealing, which can avoid the traps of locally minimum solutions.

As in previous approaches, the hardware allocation problem in automatic data path synthesis has been formulated as a two-dimensional placement problem of microinstructions in space and time. The two dimensions correspond to the hardware (e.g. ALU's) used by the microinstruction and the time of execution of the microinstruction. The problem we solve is to synthesize a data path corresponding to the input data flow specification such that a given arbitrary function of execution time and hardware cost, $f(T, C)$, is minimized. The hardware costs are the sum total of the costs associated with registers, arithmetic units, buses, and links in the data path based on required layout areas for placement and wiring. A given placement of microinstructions corresponds to a unique data path with a certain hardware cost and execution speed. Optimal conditional resource sharing is achieved by solving a constrained two-dimensional placement problem where *disjoint* instructions are allowed to occupy the same spatial and temporal location. Mutually exclusive operations are scheduled so as to use the same hardware at the same time. Given a data flow specification, we present algorithms which find a near-optimal placement of microinstructions, thus determining the spatial and temporal delineation of resources and producing a near-optimal data path configuration.

We present the formulation of the data path synthesis problem as that of two-dimensional placement of microinstructions in Section II and discuss modifications to incorporate conditional resource sharing. Given this formulation, simulated-annealing-based algorithms to solve the allocation problem are presented in Section III. These algorithms are generalized to handle looping constructs present in general software programs in Section IV. Results and illustrative examples including the synthesis of a specialized processor data path for MOSFET model evaluation are presented in Section V. Extensions to synthesize pipelined data paths are discussed in Section VI. Limitations and future work are discussed in Section VII.

II. The Hardware Allocation Problem

A. Introduction

This section describes the algorithms used in the allocation process, which take the architectural description of the machine or a software program and automatically synthesize the data path corresponding to that description under specified hardware constraints and costs. The approach taken here is to produce a data path such that a given arbitrary function of the execution speed of the data path (T) and the total hardware cost of the data path (C), namely $f(T, C)$, is minimized.

B. Input Description

The behavioral description to be synthesized from can be a description of the instruction set of a computer or the description of an algorithm in C. In either case, the description is converted into a code sequence where parallelism, sequentiality, and disjointness (mutually exclusive operations) are explicitly stated. During this transformation, various compilerlike optimization techniques (e.g., dead code elimination, constant folding) are used. This step is performed as in the CMU-DA system [1], DAA [4], and FLAMEL [15]. The code sequence produced has information only about the data transfers required between program values. The control signals which initiate these data transfers are not explicitly stated. This control signal information is used only when the specification of the state machine controller for the data path has to be derived.

The serial blocks are due to the dependences associated with any description. Disjointness is a result of the conditional clauses in the input description. An example of an input sequence is shown in Fig. 1, with *serial*, *parallel*, and *disjoint* blocks, which are the means of representing sequentiality, parallelism, and mutual exclusion, respectively. Each operation is represented in a Lisp-based syntax given by (*op* oper1 oper2 $\cdots$ oper*N* result).

C. Basic Allocation Problems

The hardware allocation process consists of a variety of subproblems. Register allocation deals with allocating variables in the given description to a minimum number of registers. Arithmetic unit allocation entails scheduling operations on a minimum number of ALU's meeting a cost or an execution time constraint. During the allocation, an optimal grouping of arithmetic operators within each ALU is also found. For instance, we might have two ALU's, one performing arithmetic operations and the other performing Boolean operations. Typically, we would like each of the ALU's to perform disjoint sets of operations, but this is not always possible. Lastly, we

```
(serial
  (parallel
    (add x1 y1 z1)
    (add x2 y2 z2)
  )
  (parallel
    (mult z1 y3 z3)
    (minus z2 y4 z4)
  )
  (disjoint
    (divide z3 x3 z5)
    (divide z4 x4 z5)
  )
)
```

Fig. 1. Input description.

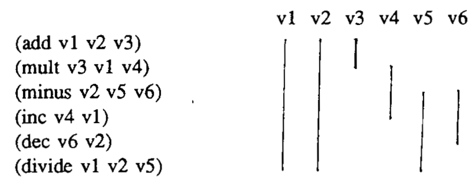


Fig. 2. Densities of variable lifetimes.

```
v1 = v2 + v3
v4 = v2 - v3
v5 = v1 * v2
v6 = v4 and v3
v7 = v5 or v6
```

(a)

```
R1 = R2 + R3
R4 = R2 - R3
R1 = R1 * R2
R4 = R4 and R3
R4 = R1 or R4
```

(b)

```
R1 = R2 + R3
R1 = R1 * R2
R2 = R2 - R3
R2 = R2 and R3
R3 = R1 or R2
```

(c)

Fig. 3. (a) Code sequence. (b) Register allocation without reordering. (c) Register allocation with reordering.

have interconnect allocation, which, given the sets of data transfers required in each time frame, allocates buses and links or multiplexer and demultiplexer connections in the data path.

The basic trade-off in hardware allocation is between serial and parallel implementations of data flow descriptions. Given an input code sequence, one can synthesize a maximally parallel data path which is expensive in terms of hardware resource and cost and uses a large number of registers and arithmetic units. On the other hand, one can synthesize a cheap, serial data path with a single ALU. Hardware resource cost, used in this context, generally represents the layout area required to implement the different modules in the data path after placement and wiring issues have been taken into account. Depending on the user's objective function, the optimal data path configuration will lie somewhere between these two extremes. Thus the allocation process has to tradeoff hardware resource cost against the execution time of the code sequence in an effort to find an optimal solution.

D. A Subproblem

We first define and solve a subproblem in the allocation process which is as follows:

Given a code sequence with singly assigned variables and precedence constraints between operations, assign the code operations to M ALU's so that a given arbitrary function of the number of registers required, N_r, and the execution time, $T, f(N, T)$, is minimized.

Assuming that the synthesized data path is a clocked sequential circuit, a maximally parallel description would use a large number of registers but would execute the fastest. A completely serial description would require a minimal number of registers (if the description had no looping constructs) but would be slow. The algorithm based on clique partitioning which was presented in [11] optimizes the number of registers *with a fixed code sequence.* Our goal is to find the optimal sequence under the given conditions, and this entails **an extra degree of freedom.**

Given a code sequence, the lifetimes of all the variables can be calculated. The lifetime of a singly assigned variable is the duration between its assignment and last use. The number of registers required would be proportional to the overlap of the live periods of the singly assigned variables, or to put it differently, the number of registers required is the *maximal density* of variable lifetimes across the entire sequence. This is illustrated in Fig. 2.

Disjoint variables are those whose lifetimes do not overlap. The allocation of registers to singly assigned variables entails finding the best possible grouping of disjoint variables in sets so as to minimize the number of sets.

However, there is freedom in the ordering of the code operations as long the precedence constraints are not violated and the constraint on the number of processing units is satisfied. A code sequence exploiting this freedom can result in a smaller set of registers being required. This is illustrated in Fig. 3. In Fig. 3(a), an example code sequence being executed on a single ALU is shown. Without changing the order of the operations in the code sequence, the minimum number of registers required is 4, as shown in Fig. 3(b). Allowing reordering of operations within the sequence produces a three-register solution in Fig. 3(c).

Finding the optimal ordering of operations within a sequence so as to allocate a minimum set of registers reduces to the **PLA multiple folding problem.** The goal is to try to find an ordering of the rows (which correspond to the code operations) under certain ordering constraints (constraints due to dependences and processors) such that the maximum number of disjoint columns (each column

corresponds to the lifetime of a variable) can be coalesced (the maximal number of variables can be merged). In the case of minimizing a function of execution time, T, and the number of registers, N_r, i.e., $f(T, N_r)$, what we are trying to find is an *optimal aspect ratio* of the PLA.

The PLA folding problem has been effectively solved using graph heuristics [26], simulated annealing [27], and exact branch-and-bound techniques [28]. These techniques can be used to solve the problem of register allocation as well. However, this formulation is merely representative of one part of the entire data path synthesis process, which will now be discussed.

E. Formulation of the Entire Data Path Synthesis Problem

Our approach to synthesizing a data path is to give a general procedure which minimizes a given arbitrary function of execution time and hardware cost. The entire cost of a data path can be represented as

$$C = p1 * (\#alu) + p2 * (exec_time) + p3 * (\#register) + p4 * (\#bus).$$

The costs of the ALU's, registers, and interconnect should be estimated taking into account layout area, placement, and wiring issues. If C reflects the exact area of a data path, then a procedure which minimizes C under constraints would optimally synthesize a data path. The specification of the parameters, $p1$ through $p4$, is discussed in Section II-F and Section VII.

This can be formulated as a **placement** problem of code operations in two dimensions, those of space and time. A given spatial and temporal placement of code operations represents a data path, and has a unique cost C. We construct a two-dimensional grid where each vertical slice corresponds to a processing unit/ALU and each horizontal slice corresponds to a time slot, as shown in Fig. 4. Code operations are placed in grid locations corresponding to an ALU and a time slot under precedence constraints due to the dependences associated between them. *Nets* connect the occurrences of variables in the code operation and also connect variables to arithmetic units in corresponding slots. The internal position of the variable in the code operation is also specified; for example, in a binary ADD a variable can be in the first or the second position for a given configuration.

The execution time is directly related to the number of occupied horizontal time slots. The horizontal time slots may be of different widths, the widths being proportional to the delays corresponding to the code operations occupying that slot. The issue of operations having different associated delays is discussed in Section IV.

The number of processing units is directly related to the number of occupied vertical space slices. The operations that a given processing unit has to perform depend on the operators occupying the grid locations in its corresponding vertical space slice. A processing unit may be simply an incrementer/counter or may be a complex floating point unit capable of multiply, add, and divide operations. Thus the formulation takes into account *the grouping of arithmetic operators* into processing units.

SPACE/TIME	ALU1	ALU2	ALU3
TIME1	(add x1 y1 z1)	(mult x2 y2 z2)	(equal x3 z3)
TIME2	(minus z1 x2 k1)	(divide z2 x1 k2)	
TIME3	(or k1 z2 11)		(inc k2 12)

Fig. 4. Two-dimensional grid of code operations.

The number of registers required to realize the variables is related to the maximum density of nets across the entire grid. This is because the *extent* of the nets connecting occurrences of a variable is a representation of the lifetime of the variable. Given a maximum density of lifetimes M, using the Left Edge Algorithm (widely used in channel routing [29]), the variables can be coalesced into M registers.

The interconnect relationship to the physical entities of nets and code operations is more difficult to formulate. Obviously the number of registers and ALU's is weakly related to the number of interconnections required. Other measures of interconnect complexity can be obtained—the number of links required can be related to the *stagger* of nets in this formulation.

The stagger of the nets implies the connection of registers to more than one ALU. The more staggered a net, the greater the number of ALU's the variable (and eventually the register) feeds into. The stagger of nets treated as separate entities does not, however, take into account the fact that groups of variables which feed into different ALU's may be coalesced into the same register. This register will then need to feed into many ALU's. Only variables which are disjoint can be coalesced into the same register. However, *the stagger of nets between disjoint variables* is a good indicator of interconnect complexity (number of links) at any stage. The net stagger is further refined by the position information of the variables within the code operation. The position information takes into account the fact that variables may be feeding into one or both ports of the ALU.

Other good measures of the number of buses required with a given schedule are the maximum number of distinct sources and the number of sinks in all the time slots (which is an indicator of the number of parallel data transfers required). So, even if all the registers have been previously allocated, the tradeoffs between execution time and interconnections can be made. In the general case, execution time can be traded off against the number of registers, processing units, and interconnections.

The cost function has been defined in terms of the above-mentioned quantities. The problem is, therefore, to find a global placement of code operations in the grid locations under the dependence constraints, and a placement of variables within the code operations which minimizes the cost. Then the variables can be coalesced into registers and the interconnections into buses.

Some variables, for example, arrays, may need to be in memory. If they are, accessing them potentially takes

more cycles. There is a tradeoff between reducing the number of registers by allocating variables to memory locations and increasing the execution time. This tradeoff can be explored if necessary.

To solve the problem, we can use various techniques for solving the placement problem. Our goal is to find a placement which produces a global minimum for the function $f(T, C)$. The use of simulated annealing, a global optimization technique, has produced excellent results for integrated circuit cell placement problems [30]. Hence, we have used simulated annealing to solve our particular placement problem. This simulated-annealing-based algorithm is described in Section III.

F. The Cost Table

The specification of costs is vitally important. Given a cost function, the simulated-annealing-based algorithm can find near-optimal solutions *for that cost function* within reasonable amounts of CPU time. Ideally, the hardware costs should reflect the exact layout area of the data path. While the areas of individual modules (e.g. registers, ALU's) can be estimated exactly or nearly exactly, estimating routing area is much more difficult. In [31], the effects of incorrect estimation were discussed and shown to be significant.

A cost table (Fig. 5) specifies the cost of hardware resources and operators. It also implicitly specifies the parameters $p1$ through $p4$ in the cost function C (Section II-E). The parameters $p1$, $p3$, and $p4$ are *area parameters*, while $p2$ is an *execution time parameter*. The area parameters reflect the layout area of the individual modules. The execution time parameter, $p2$, is a way of specifying whether a fast data path or a relatively slow one is desired. A higher $p2$ implies a greater cost for execution time and will result in a faster data path. These parameters are not necessarily constants; in general, they are functions of the number of ALU's and/or registers and/or buses in the data path.

1) Register Costs: The parameter $p3$ is equal to the area of the library register to be used. It is a multiplying factor for the number of registers in the data path. In the cost table of Fig. 5, $p3$ is a function of the number of registers, in an effort to estimate routing area (Section II-F-1). The first five registers cost ten units each; the next five cost 15 units.

2) Costs of ALU Operations: The cost of each arithmetic or Boolean operator should reflect the layout area to implement that operator. A complication arises when attempting to optimally group operators within ALU's. Given that the ALU is to be implemented using combinational logic, the area required by a set of operators is, generally, *not* equal to the sum of the areas required to implement each operator separately. A case in point is an ALU implementing addition and subtraction. This ALU would be only slightly larger than an ALU implementing only addition or only subtraction, not twice the size. Thus, ALU costs cannot be calculated using simple additive relationships.

```
# cost of different operations in a ALU
ALU
add 50
sub 50
fadd 100
mult 250
add minus 60

# register costs
REGISTER
# starting from register 1, each register has cost 10 units
1 10
# starting from  register 5, each register has cost 15 units
5 15

# execution time
EXECUTION
1 50
50 50

# interconnect, buses and links
BUS
1 100
3 150

LINK
1 5
100 10
```

Fig. 5. Example cost table.

This problem is alleviated by defining costs not only for each operator but also for small sets of operators. A *multiply* operator may have a cost of 100 units, a *divide* a cost of 200 units, and an ALU performing *multiply* and *divide* may be deemed to have a cost of 210 units depending on library-specific information. Given an arbitrary set of operators, the program checks to see if costs have been specified for any subset of operators before adding costs up for the single operators.

3) Estimating Interconnect Area: The areas of the individual modules can be estimated accurately and included in the cost table. The number of links and buses can be estimated closely, as described in the previous section. The area for a link/bus is to be used as parameter $p4$. This area is typically a complex function of the number of registers and ALU's in the data path. Assuming that $p4$ is a constant, i.e., that interconnect area is a linear function of the number of links/buses, can be quite inaccurate [31].

Our approach relies on empirical estimations of routing area. For example, given a layout style, we evaluate the increase in routing area (not total area) due to incremental additions of registers and associated links and add this cost to the link and register costs. The link and register costs then become piecewise-linear functions. Data points over a range of numbers of ALU's and registers in a data path are obtained. The number of data points required to obtain exact accuracy is, unfortunately, infinite. However, with a reasonably small number of data points, one can do better than a linear approximation on the number of links. In the cost table of Fig. 5, register and interconnect costs are modeled as piecewise-linear functions, and ALU costs are modeled as linear functions.

Accurate routing area estimation remains largely an unsolved problem (Section VII). It is clear that the total area of a data path is a nonlinear function of the number of ALU's, links, and registers, even in data paths constructed largely by abutment. Given this complex function, or a good approximation of this function via piecewise-linear functions, the simulated-annealing-based algorithm, described in the next section, obtains high-quality solutions.

G. Conditional Resource Sharing

Conditionals can be introduced into the algorithm. This is done by defining disjointness (mutual exclusion) between statements. For example, the THEN and ELSE clauses in an IF statement are disjoint. *Disjoint statements can exist on top of each other on the same time-space slot.* The algorithm takes into account this disjointness and finds an optimal schedule for the code sequence with an arbitrary number of conditional clauses.

Placing operations on the same time-space slot amounts to conditional resource sharing. Many forms of conditional resource sharing are possible. The coexistence of two ADD operations on the same grid location implies that the two operations are sharing an adder since they are mutually exclusive. If two operations sharing a common variable exist on the same location, a register will be shared by the two disjoint operations, and it will store information dependent on conditional clauses.

We initially had a two-dimensional placement problem, where the two dimensions corresponded to the time of execution and the hardware space (e.g., ALU's) used by an operation. Mutually exclusive operations can be scheduled to occur at the same time on the same ALU. We have now *a set of constraints that limit the freedom in scheduling operations on the same time and space coordinates.*

Disjoint blocks may be arbitrarily nested in the code sequence. Initially, before the optimization, disjointness relationships between each pair of operations in the given code sequence is found, and this information is exploited. For example, given

```
(disjoint
   s_1
   (disjoint
      s_2
      s_3
   )
)
```

s_1 is deemed to be disjoint from both s_2 and s_3 and s_2 is disjoint from s_3.

III. A Simulated-Annealing-Based Solution

A. Introduction

Simulated annealing, proposed by Kirkpatrick *et al.* [24], has proved to be an effective solution to the cell placement problem in LSI layouts [30]. Its basic feature is that it allows *hill climbing moves* [25] in exploring the configuration space of the optimization problem. The probability of accepting these moves is controlled by a parameter analogous to temperature in the physical annealing process and this parameter decreases gradually as the annealing process proceeds. The simulated annealing algorithm can be used for combinatorial optimization problems specified by a finite set of states and a cost function defined on all the states. The algorithm randomly generates a new state or configuration, and the new state is accepted or rejected according to a random acceptance rule governed by the parameter analogous to temperature in the physical annealing process. The basic algorithm proceeds as follows:

```
T = T_0
X = Starting_Configuration;
while("cost is changing"){
      for("a certain number of times"){
      Generate_New_State(j)
      if(accept(c(j), c(X), T)){
         X = j;
      }
   }
   T = update(T);
}
```

Whether or not a new state is accepted is determined by the function **accept**( ):

```
accept(c(j), c(i),T){
   change_in_cost = c(j) - c(i);
   if (change_in_cost < 0) return(1);
   else {
      Y = exp(-change_in_cost/T);
      R = random(0,1);
      if (R < Y) return(1);
      else return(0);
   }
}
```

This basic algorithm forms the core of our approach. The parameter T is analogous to temperature in a physical annealing process. At every temperature point, a number of random moves are generated. The number of moves generated is a parameter that can be controlled by the user; it affects the quality of the solution profoundly. According to existing theoretical results, simulated annealing asymptotically approaches the global optimum of the configuration space [25].

The two most important things in any simulated-based algorithm are the generation of new states (**Generate_New_State**( )) during the annealing process and the cost function (**c**( )) to be optimized for. The generation of states and the cost function together determine the quality of solutions which can be obtained.

These two aspects of the simulated-annealing-based algorithm for the allocation problem are described in detail below. In Section III-G we discuss the advantages of using annealing rather than fast heuristics to solve the placement problem.

B. Generating New States

New states are generated during the annealing process in three different ways:

1) interchanging two code operations;
2) displacing a code operation from one location to another;
3) interchanging the variables in a symmetric operation (e.g., ADD).

Moves 1 and 2 have to satisfy certain constraints, namely, the precedence constraints between operations cannot be violated by such a move, and operations on the same time–space slot have to be disjoint. Examples of interchanges and displacement of operations are illustrated in Fig. 6.

The generation of states proceeds as follows:

1) Two numbers are randomly generated, the first between one and the number of operations, the second between one and the number of operations times a certain quantity (typically 5).
2) If the second number is less than the number of operations, an interchange of the two operations is tried. If the interchange violates any constraint, and either one of the operations happens to have a symmetric operator, the variables in that operation are interchanged.
3) If the second number is greater than the number of operations, a new location for the first operation is randomly generated, and the operation is displaced to the new location if the displacement does not violate the aforementioned constraints.

During the end of the annealing process (at low temperatures), the generation of states takes a different form so as to generate states which are more likely to be accepted:

1) This step is identical to the first step in the previous sequence.
2) If the second number is less than the number of operations, an interchange between the first operation and the operation immediately to the left or right is tried. If one direction fails, the other is tried. If both fail, a variable interchange is tried.
3) If the second number is more than the number of operations, a displacement of the first operation immediately to the left or right in the same time slot and immediately ahead or behind in the same space slot is tried in randomly generated order.

C. The Cost Function

The cost function should be representative of the hardware and execution time cost function C (Section II) to be optimized.

The total execution time required for the entire sequence is one part of the cost function. In the general case, the execution time may be weighted by the frequency of code kernels. Given a large code sequence, parts of the sequence may have higher execution time weights associated with them because they are more frequently used. The weighted *spread* (the time of execution of the last operation in the kernel − the time of execution of the first operation) of kernels can be calculated.

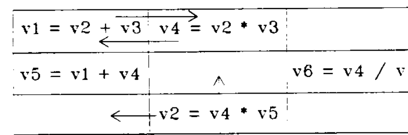


Fig. 6. Interchanges and displacements during annealing.

The number of registers required in hardware is given by the maximum density of nets (which connect occurrences of variables) across all the time slots. The number of registers required is part of the cost function.

For each space slot, the sum of the costs of all the distinct operators (or operator sets) required is found. The sum of all these costs is the processor cost constituent of the cost function.

Interconnect cost is estimated by estimating the number of links and buses required in hardware. The stagger of nets between disjoint variables is a good indicator of link costs. The number of buses required is estimated by calculating the maximum number of distinct sources and the number of sinks in all the time slots, since this is a good indication of the number of parallel data transfers required.

D. Hardware Resource Constraints

Hardware resource constraints (e.g., limits on the number of ALU's or registers) can easily be incorporated into the simulated-annealing-based algorithm by penalizing configurations which violate any of these constraints. A penalty is added to the cost of such an intermediate configuration and is sufficiently high so as to ensure that the final solution satisfies all the constraints.

E. Execution Time Constraints

A bound on the time required by the data path to execute the code sequence, or parts of the code sequence, may be given. This constraint is incorporated using a penalty function approach as in the case of constraints on hardware resources. A discussion of more complex constraints is included in Section VII.

F. Stopping and Inner Loop Criteria

The number of states generated per temperature point is a certain integral multiple of the number of code operations (typically 1–10). The temperature is lowered to a fraction (typically 0.90) of its original value after each temperature point. The annealing process terminates when the cost function has not changed in value for three temperature points.

IV. Further Extensions

For the sake of clarity in presentation it has been assumed thus far that the operations in the input description

have equal delays. However, in general, operations in a software program may have drastically different delays. For example, a 32-bit multiply may take more than ten times the time required by an integer increment.

It is not difficult to generalize the formulation of the data path synthesis problem to handle operations with different delays. A generalized two-dimensional placement of operations is shown in Fig. 7. The height of each operation is proportional to its delay. For example, the MULTIPLY has a delay which is three times the ADD. The placement now resembles a set of linked lists of operations (one for each ALU), rather than the matrix of operations of Fig. 4.

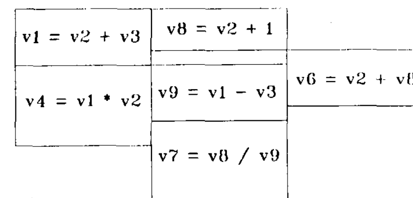


Fig. 7. Generalized two-dimensional placement.

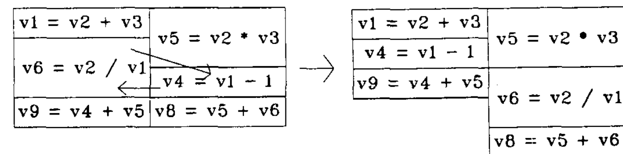


Fig. 8. Placement before (left) and after interchange.

The simulated-annealing-based algorithm for hardware allocation as described in Section III made no assumptions about the relative delays of operations. If operations have different delays, the highest common factor of all the different operation delays in the data flow descriptions is calculated. This becomes the size of one time frame. Operations can occupy more than one time frame. During interchanges and displacements of operations in time or space, the time positions of the successors of the interchanged and displaced operands may also change. This is illustrated in Fig. 8.

Loops are a succinct way of representing iteration in programming languages. It is important that an allocation algorithm be able to provide for loops in the input description.

One method of dealing with loops is to treat each loop as a single operation with delay equal to the number of iterations times the delay of each iteration. This single operation is scheduled just like other basic operations. However, the problem with this approach is that all the iterations of a loop are always scheduled serially on a single ALU. It may be beneficial to schedule iterations in parallel on different ALU's.

Another method of dealing with loops in the input description is *full unwinding* [1]. In this method, all the iterations in a loop are expanded into a number of operations. The number of operations after unwinding will be proportional to the number of iterations in the loop. These operations can be scheduled independently and may be executed in parallel if the precedence constraints between them are not violated. This method exploits all the degrees of freedom present in scheduling iterations of loops separately. However, given a loop with a large number of iterations, full unwinding is not always feasible.

Our solution to this problem is **dynamic partial unwinding** of loops during the optimization process. Initially, all loops are represented as basic operations and their delays computed. However, they are *tagged*. During the annealing, a possible move (other than displacing tagged or untagged operations) is to split a tagged operation into two or more components. For example, a ten-iteration loop may be split (unwound) into two five-iteration components. These components are also tagged. The components are scheduled separately and may be executed in parallel if no precedence constraints exist between them. However, this splitting does not preclude the possibility of all the iterations of the loop being executed on the same ALU if that happens to be the best configuration. A possible scenario of loop splitting during the annealing is shown in Fig. 9.

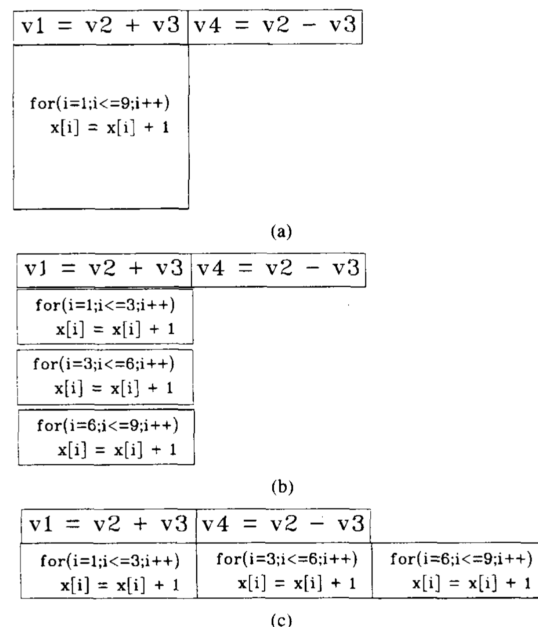


Fig. 9. (a) Initial placement. (b) Loop splitting. (c) Final placement.

The components after splitting are tagged and may be further split up into subcomponents. The number of components a loop is split into (the degree of unwinding) and the level of splitting is specified initially by the user. If the number of components equals the number of loop iterations, then we are effectively performing full unwinding. If splitting is not allowed, then the loop is being treated as a basic operation. Other allocation algorithms (e.g., [1], [4], [14]) also provide for loop unwinding and scheduling in different ways. Data-dependent loop exits

are a major problem with all loop unwinding approaches (including ours). The number of iterations in the loop has to be known in advance.

Another extension is trading off delay and cost for single operations. For example, different adders may exist in the library with varying area costs and delays. A fast adder performing 32-bit addition in 25 ns may cost ten units; a slower 40-ns adder may cost only five units. The choice of the adder which minimizes the objective function, f, can be made during the annealing. A move during the annealing would be to change a fast adder into a slow one or vice versa. In general, more than two implementations with different cost–delay tradeoffs can exist for an operator.

V. Examples and Results

We use the code sequence in [11] as our first example. The input file is shown in Fig. 10. The entire sequence consists of an *implic* block, which implies that data dependences are derived by the program and have not been explicitly stated. Each operation is written in a Lisp-based syntax with the operator as the first argument and the result the last, as described in Section II. The INITIAL and FINAL declarations imply that the following variables are live in the beginning and the end of the sequence, respectively. The SYMMETRIC declaration enumerates all the operations whose operands are interchangeable.

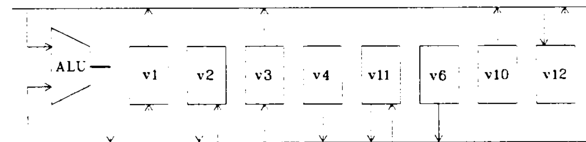
```
(implic
  ( add v1 v2 v3 )
  ( minus v3 v4 v5 )
  ( mult v3 v6 v7 )
  ( add v3 v5 v8 )
  ( add v1 v7 v9 )
  ( divide v10 v5 v11 )
  ( equal v3 v13 )
  ( equal v1 v12 )
  ( and v11 v8 v14 )
  ( or v12 v9 v15 )
  ( equal v14 v1 )
  ( equal v15 v2 )
)
INITIAL v1 v2 v4 v6 v10
FINAL v1 v2 v4 v6 v10
SYMMETRIC add mult or and
```

Fig. 10. Input file for example from [11].

In the first run (using the simulated-annealing-based algorithm) the costs of arithmetic operations were ≥ 50 units, each register cost was ten units, each link ten units, and execution cost per time slot was fixed at five units. Execution speed was thus given a low priority in this run. The optimization produced a serial sequence, shown in Fig. 11(a), which needs eight cycles to execute. The CPU time required for the simulated annealing run was 30 s on a VAX 11/8650. The data path synthesized after bus allocation is shown in Fig. 11(b). The minimal numbers of registers and interconnections have been used.

(add v1 v2 v3)	(equal v1 v12)
(minus v3 v4 v11)	
(mult v3 v6 v2)	
(add v3 v11 v3)	
(add v1 v2 v2)	
(divide v10 v11 v11)	
(and v3 v11 v1)	
(or v12 v2 v2)	

(a)



(b)

Fig. 11. (a) Code sequence after two-dimensional placement. (b) Synthesized bus-style data path.

Bus allocation is done after the code operation placement using algorithms similar to [32]. However, *during the placement* the amount of interconnect required is calculated at every stage and minimized as described earlier. We have assumed, while performing bus allocation, that the data transfers for every microinstruction $(opV_aV_bV_c)$ look as follows:

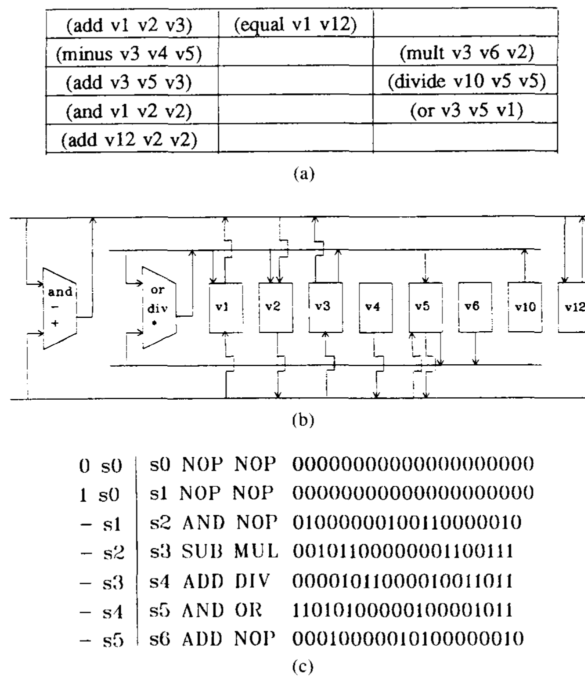
$$V_a \rightarrow \text{link} \rightarrow \text{bus} \rightarrow \text{link}\,ALUin1$$
$$V_b \rightarrow \text{link} \rightarrow \text{bus} \rightarrow \text{link}\,ALUin2$$
$$ALUout \rightarrow \text{link} \rightarrow \text{bus} \rightarrow \text{link} \rightarrow V_c$$

The two input transfers to the ALU are required to occur in parallel. If in fact we are allowed to make the two input transfers to an ALU in sequence, one can synthesize a data path for this example with only one bus.

The freedom in being able to arrange symmetric operands in order to minimize interconnect has been exploited by the program. If that had not been done, more links would have been required.

The placement of code operations produced by the program given a higher execution time cost than in the previous case, that of 50 units, is shown in Fig. 12(a). The register/ALU/interconnect cost was unaltered from the previous run. Note that the placement is such that operations in the two ALU's have *no* operators in common—an optimal grouping. The data path corresponding to the code sequence in Fig. 12(a) is shown in Fig. 12(b), again with a bus-style design. The CPU time required for synthesis was 40 s on a VAX 11/8650. For two microinstructions in the same time slot, all the *ALUin* transfers are assumed to occur simultaneously, and all the *ALUout* transfers together. In the data path shown, four buses are required. If the constraint of simultaneous input/output transfers to all ALU's is relaxed, fewer buses will suffice. The finite state machine controller specification for the data path is shown in Fig. 12(c). A single input is required to start computations. The outputs are the load signals to the different links in the data path. Some links are controlled by the same output.

Another small example, this time with conditional clauses in the input description, is shown in Fig. 13. The



(add v1 v2 v3)	(equal v1 v12)	
(minus v3 v4 v5)		(mult v3 v6 v2)
(add v3 v5 v3)		(divide v10 v5 v5)
(and v1 v2 v2)		(or v3 v5 v1)
(add v12 v2 v2)		

(a)

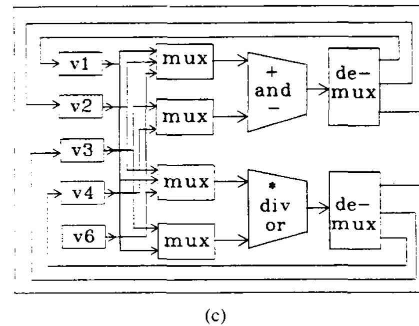
Fig. 12. (a) Code sequence after two-dimensional placement. (b) Synthesized bus-style data path. (c) Finite state machine controller.

```
(serial
  (parallel
    (add v2 v3 v1) (divide v2 v3 v4)
  )
  (disjoint
    (add v1 v4 v6) (minus v1 v4 v6)
  )
  (disjoint
    (mult v6 v3 v7)
    (serial (divide v6 v3 v8) (mult v8 v2 v7))
  )
  (parallel
    (and v7 v4 v9) (or v7 v1 v10)
  ))
```

(a)

(add v2 v3 v1)	(divide v2 v3 v4)
[(add v1 v4 v6) (minus v1 v4 v6)]	
	[(mult v6 v3 v6) (divide v3 v6 v3)]
	(mult v2 v3 v6)
(and v6 v4 v2)	(or v6 v1 v3)

(b)



(c)

Fig. 13. (a) Input description. (b) Two-dimensional placement. (c) Multiplexer-style data path.

input description is shown in Fig. 13(a); the two-dimensional placement is shown in Fig. 13(b), and a multiplexer-style data path, which takes five or six cycles to execute the description depending on what conditions are asserted, is shown in Fig. 13(c).

A larger example is a MOSFET model evaluation routine implementing the Schichman–Hodges [33] level-1 MOSFET model. Our goal, as before, was to synthesize the data path of a specialized processor executing the software description optimally under different cost constraints. The inputs to the processor are the MOSFET voltages and device parameters, and the outputs are the currents, conductances, and their derivatives. The data paths can be used as coprocessors for model evaluation in a hardware simulation engine.

The software description initially consisted of about 150 lines of C code. This was converted into about 300 lines of input to the synthesis program. A total of 228 possible operations existed in the input description (some of them mutually exclusive). The operators used were all floating point—add, minus, divide, multiply, minimum, maximum, etc. Using different hardware and execution time costs, three different data paths were synthesized.

The first was a serial data path with a single ALU; the second and third have two ALU's. The execution speeds of the data paths (normalized to the serial data path), the number of registers, buses, and links in the data path, the estimated areas of the data paths (normalized to the serial data path), and the CPU times, in minutes, for synthesis on a VAX 11/8650 are summarized in Table I. The ALU's in data paths 2 and 3 execute different sets of operations.

TABLE I
MOSFET Model Data Path Statistics

DP	execution time	#reg	#bus	#link	estimated area	CPU time
1	1.0	21	2 + 1*	54	1.0	10.1m
3	0.54	21	4 + 1*	77	2.5	11.2m
4	0.50**	24	4 + 1*	70	2.6	13.1m

* memory bus
** signifies throughput rather than execution time

In data path 3, both ALU's perform multiplication/division in addition to addition and subtraction. In data path 2, only *ALU*1 performs multiplication/division. The data paths are shown in Fig. 14. This large example illustrates how the algorithms described in this paper can be used to effectively explore trade-offs in the design space.

MOSFET evaluation entails filling in a matrix of currents and conductances—the matrix is assumed to be stored in memory. This would be the case if the data paths were to be used as coprocessors for a hardware simulation engine.

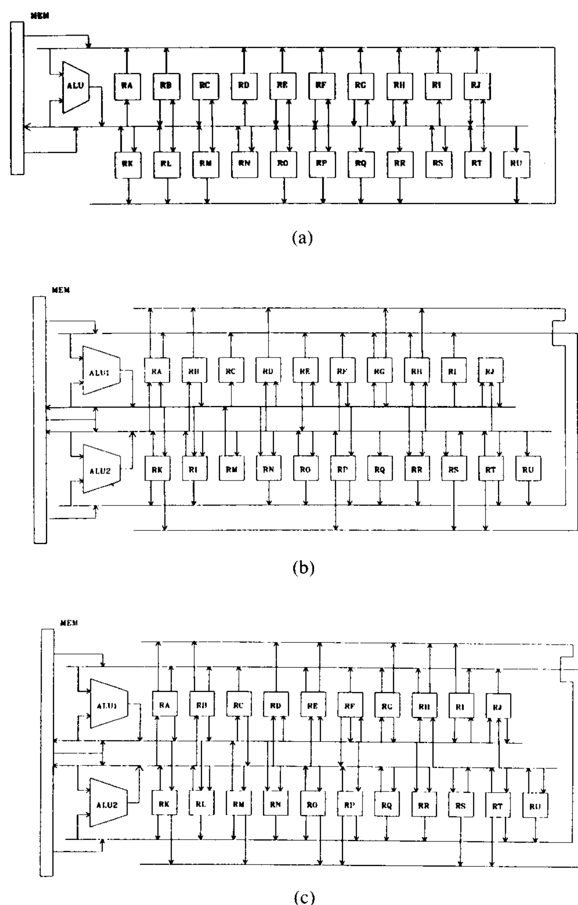


Fig. 14. (a) Data path 1. (b) Data path 2. (c) Data path 3.

$$
\begin{aligned}
&I = In\\
&a = i + t2\\
&b = a + t13\\
&g = t33 + t39\\
&e = g + t26 + b\\
&d = (m21 * e) + b\\
&f = (m24 * e) + g\\
&t26 = f + d + e\\
&c = m9 * (b + d) + a\\
&h = m30 * (f + g) + t39\\
&j = t18 + c + d\\
&k = t38 + f + h\\
&t39 = o + h\\
&t38 = t38 + (m36 * k)\\
&t33 = t38 + k\\
&t18 = t18 + (m16 * j)\\
&t13 = t18 + j\\
&t2 = c = i + m6 * (a + c)\\
&Out = o
\end{aligned}
$$

(a)

ADDER	ADDER	ADDER	MULTIPLIER	MULTIPLIER
i = In				
a = i + t2	g = t33 + t39			
b = a + t13	e' = g + t26			
e = e' + b				
			d' = m21 * e	f' = m24 * e
d = d' + b	f = f' + g			
t26' = f + d	c" = b + d	h" = f + g		
t26 = t26' + e	j' = t18 + d	k' = t38 + f	c' = m9 * c"	h' = m30 * h"
c = c' + a	h = h' + t39			
k = k' + h	j = j' + c	t2" = a + c		
t39 = o + h			t38' = m36 * k	t18' = m16 * j
t18 = t18' + t18	t38 = t38' + t38		t2' = m6 * t2"	
t13 = t18 + j	t33 = t38 + k			
Out = o	c = i + t2'	t2 = i + t2'		

(b)

Fig. 15. (a) Elliptic filter. (b) Synthesized schedule.

Our final example is the well-known elliptic filter example, originally used in [14]. In Fig. 15(a), the input description is shown, taken from [34]. The fastest realization obtained by HAL [14] took 17 cycles to execute the description and required three multipliers and three adders (an adder is assumed to execute in one cycle and the multiplier in two cycles). The fastest possible realization using the simulated-annealing-based algorithms also takes 17 cycles but requires one less multiplier. The schedule obtained is shown in Fig. 15(b). The CPU time required was 4 min on a VAX 11/8650. Tradeoffs can be made for this example as well. Slower realization requiring less hardware can easily be derived from the schedule shown. These realizations are identical to those produced by HAL in terms of processor utilization. Our algorithms thus compare favorably with those proposed in the past on this benchmark example.

The time required by the annealing algorithm grows approximately quadratically with the size of the data path being optimized. In our implementation, the calculation of the cost function is incremental and the complexity of cost evaluation per move grows linearly with problem size. Empirical evidence has demonstrated that the total number of moves required to obtain high-quality solutions, minimal with respect to the evaluation function, grows approximately linearly with problem size. This results in an overall approximate quadratic complexity. This behavior is quite reasonable—large examples like the MOSFET model evaluator terminate expending acceptable run times.

Heuristic algorithms work as well as, if not better than, simulated annealing for problems that have relatively simple analytical formulations (e.g., graph partitioning). Our placement problem has a large number of variables (microinstructions which have to be placed), an associated set of constraints (hardware, execution speed, and disjointness constraints), and, most importantly, a complex (possibly nonanalytical) cost function. Simulated annealing works best for these kinds of problems relative to heuristic algorithms. Also, incorporating additional constraints and complications in the cost function is easier in a simulated-annealing-based algorithm than in a heuristic algorithm.

The run time of our algorithm relies on quick evaluation of cost functions. A good cost function is required to produce near-optimal data paths. However, a good cost function may not be amenable to a quick evaluation. Run time can be kept down to a manageable level by using a quickly

evaluated cost function at high temperatures, and the optimality of the solution can be maintained by using the good cost function at low temperatures.

Our placement problem is conceptually similar to the problem of floorplanning in VLSI layout. It is interesting to note that simulated annealing is extensively used in industry to solve the floor-planning problem.

VI. Synthesizing Pipelined Data Paths

A. Introduction

Pipelining is an essential feature of the computers being designed today [35]. Pipelining implies overlapping of multiple tasks—each computation task is partitioned into subtasks and each subtask is executed in a clock cycle. Consecutive tasks are initiated at certain intervals, called the *latency* of the pipeline, which are integral multiples of a clock cycle.

Given an input data flow specification, pipeline synthesis involves splitting the data flow graph into stages (phases or partitions), with constraints on the number of stages and stage delays, so as to optimize for execution time and/or hardware cost. The number of stages, the schedule of operations, and the hardware resources required in each stage are determined in pipeline synthesis. Engineering solutions to pipeline scheduling given fixed hardware resources have been published [35]–[37]. A pipeline synthesis procedure based on scheduling algorithms was first published in [17].

SEHWA [17] generates data paths from data flow graphs along with a clocking scheme which overlaps execution of tasks. SEHWA estimates the cost of a pipeline based on the number of processing units of each type and the number of latches required in the hardware implementation. It has been used to synthesize clocking schemes for general-purpose computers with fetch–decode–execute pipelines [38] and pipelined digital signal processors.

B. The Pipeline Synthesis Problem

Hardware resources cannot be shared across pipeline stages. For example, given a two-stage pipeline, after pipeline setup, the micro-operations in both stages will have to be simultaneously performed on each clock cycle (albeit on different input streams) and will, therefore, need distinct computational units.

Our goal is to solve a more general pipeline synthesis problem than that in [17], where register/latch, arithmetic operator, and interconnect costs are taken into account during the pipelining. To this end, we have extended the hardware allocation algorithms presented in Sections II–IV to be able to synthesize pipelines.

Pipeline synthesis involves *partitioning* the input data flow description into a number of pipeline stages and *finding a placement* of micro-operations within each stage so as to meet a cost or an execution time constraint. The problem we solve is to synthesize a pipelined data path, given a constraint on the maximum delay for each stage, while minimizing a user-specified function of hardware resource cost, C, and throughput of the pipeline, E, namely, $f(C, E)$. C is the total hardware cost summed over all the partitions (stages). E depends on the number of stages in the pipeline and the maximum delay of any stage.

C. Extensions for Pipeline Synthesis

The following modifications were made to the simulated-annealing-based hardware allocation algorithm to synthesize pipelined data paths.

1) The algorithm begins with a serial pipeline schedule which does not violate the maximum stage delay constraint. This serial schedule is constructed by scheduling operations serially in a given stage and beginning another stage when the stage delay exceeds the maximum allowed value. Given a partition, hardware costs are calculated as before, treating every partition as a separate two-dimensional placement, and adding up all the costs of each partition. This is done because hardware resources cannot be shared across the phases.
2) Moves are generated during the annealing as before, interchanging and displacing operations both within a stage as well as across adjacent stages. The moves are such that the precedence constraints between operations are not violated. However, the maximum stage delay limit may be violated by a move. These violations are allowed in intermediate solutions but are penalized so they do not appear in the final result. Operations in the last phase may be displaced to a previously empty following phase, increasing the number of phases. The number of phases may also decrease during the annealing.
3) The throughput, E, of the pipeline is measured using the number of stages, k, the delay of the stages, d_i, and the expected resynchronization rate, ρ, using the equation shown below, which is similar to those derived in [17]:

$$E \propto 1/\left(1 + \left(\mathrm{MAX}_i (d_i) \,.\, k - 1\right)\rho\right).$$

The tradeoff between delay and cost for single operations (Section IV) can also be made while synthesizing pipelined data paths.

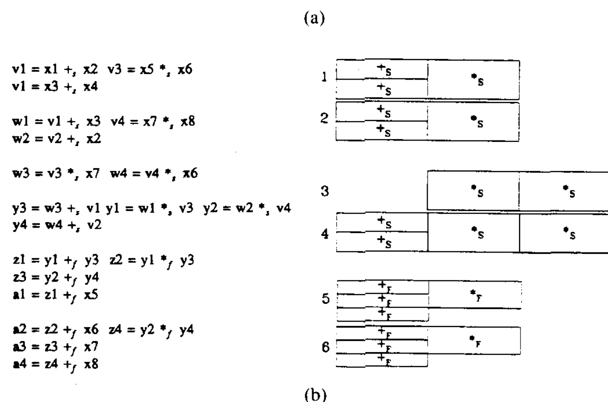
D. Examples

An example of pipelining a data flow specification is illustrated in Fig. 16. Fig. 16(a) gives the unpipelined data flow specification, with the tradeoffs for the adders and multipliers specified as (cost, delay) number sets. Given these tradeoffs, a maximum stage limit of 100 ns, 20-ns latch delay, and a latency of 2, the program was asked to find the cheapest possible schedule with a maximum of six stages. The schedule synthesized is shown in Fig. 16(b). The symbol $+_f$ denotes a fast adder and $+_s$ a slow adder (similarly for multiply). Both kinds of adders

```
v1 = x1 + x2  v2 = x3 + x4  v3 = x5 * x6  v4 = x7 * x8
w1 = v1 + x3  w2 = v2 + x2  w3 = v3 + x7  w4 := v4 + x6
y1 = w1 + v3  y2 = w2 + v4  y3 = w3 + v1  y4 = w4 + v2
z1 = y1 + y3  z2 = y1 * y3  z3 = y2 + y4  z4 = y2 * y4
a1 = z1 + x5  a2 = z2 + x6  a3 = z3 + x7  a4 = z4 + x8
```

$+_s$ (1.0,40) $+_f$ (1.5,25) /* cost delay tradeoff for + */
$*_s$ (2.0,80) $*_f$ (3.0,50) /* cost delay tradeoff for * */

(a)



(b)

Fig. 16. (a) Input specification with cost-delay tradeoffs. (b) Synthesized pipeline schedule.

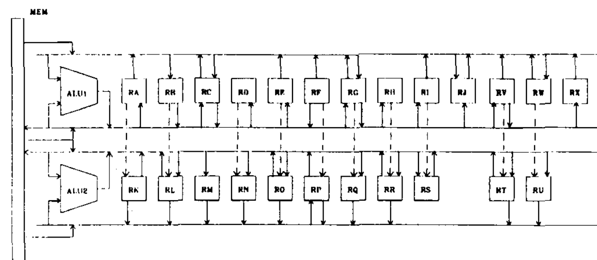


Fig. 17. Data path 4.

TABLE II
SERIAL, PARALLEL, AND PIPELINED DATA PATH STATISTICS

DP	execution time	#reg	#bus	#link	estimated area	CPU time
1	1.0	21	2 + 1*	54	1.0	10.1m
2	0.65	21	4 + 1*	66	1.7	9.2m
3	0.54	21	4 + 1*	77	2.5	11.2m

* memory bus

and multipliers have been used to maximum advantage. Since the latency is 2, resources can be shared across stages 1 and 2, 3 and 4, 5 and 6; so two $+_s$, one $+_f$, three $*_s$, and one $*_f$ unit(s) are required, adding up to a total cost of 12.5 units. The multiplier in stages 5–6 has to be a $*_f$ unit, since a4 has to be computed after computing z4 in stage 6. The CPU time required to synthesize this pipeline schedule was 2 min on a VAX 11/8650.

Our second example is the MOSFET model evaluator of Section V. The data path synthesized for a two-stage pipeline with latency 1 is shown in Fig. 17. The statistics of this data path are compared wtih those of data paths 1 and 3 (Fig. 14) in Table II. Data path 3 is a parallel implementation of the MOSFET model routine with two ALU's, whereas data path 4 is a pipelined implementation with two stages (each with a single ALU). Data path 4 has higher throughput (assuming no resynchronization) but is slightly larger in area. The links shown in dotted lines in Fig. 17 correspond to data transfers occurring from the registers in the first pipeline stage to registers in the second pipeline stage.

VII. Limitations and Future Work

In our approach, the quality of the synthesized data path depends on how accurately the evaluation (or cost) function of the placement problem predicts the resulting data path configuration. Given sufficient time, the minimality of the cost function is guaranteed, via the use of simulated annealing [25].

There are two problems with the evaluation function that we use. One problem, as mentioned in Section II-F, has to do with interconnect area estimation [31]. More accurate estimations of routing area, given the ALU's, registers, and their connectivity, can improve the quality of results. Better estimations will typically take longer to evaluate. As mentioned in Section III-G, the run time of the annealing algorithm can be reduced, while maintaining solution quality, by using a quick, relatively less accurate evaluation function at high temperatures and the more accurate evaluation function at lower temperatures.

The algorithm can incorporate constraints on the total execution time of the code sequence and optimize execution time for specific code kernels. More complex constraints between the time of execution of different sets of operations can be incorporated but would require a complicated analysis during each move of the annealing, thereby decreasing efficiency. Future work will address these limitations.

VII. Conclusions

In this paper, we have described a novel method for synthesizing data paths from behavioral descriptions. The entire allocation process in data path synthesis can be formulated as a two-dimensional placement problem of microinstructions in space and time. This formulation allows simultaneous cost-constrained allocation of registers, arithmetic units, and interconnect (buses and links) while trading off hardware cost against execution speed. We have presented a new, simulated-annealing-based solution to the data path synthesis problem which has achieved excellent results. Finally, this simulated-annealing-based approach has been extended to synthesize pipelined data paths.

Acknowledgment

The authors would like to thank T. Ma for several interesting discussions on data path synthesis.

References

[1] D. E. Thomas, C. Y. Hitchcock, T. J. Kowalski, J. V. Rajan, and R. A. Walker, "Automated data path synthesis," *IEEE Computer*, vol. 21, pp. 59–70, Dec. 1983.

[2] H. Trickey, "Compiling Pascal programs into silicon," Stanford Computer Science Rep. STAN-CS-85-1059, Stanford Univ., Stanford, CA, July 1985.

[3] A. C. Parker, M. Mlinar and J. Pizarro, "MAHA: A program for data path synthesis," in *Proc. 23rd Design Automat. Conf.* (Las Vegas), June 1986, pp. 461–466.

[4] T. J. Kowalski and D. E. Thomas, "The VLSI design automation assistant: What's in a knowledge base," in *Proc. 22nd Design Automat. Conf.* (Las Vegas), June 1985, pp. 252–258.

[5] M. C. McFarland and T. J. Kowalski, "Assisting DAA: The use of global analysis in an expert system," in *Proc. Int. Conf. Comput. Design* (New York), Oct. 1986, pp. 482–485.

[6] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, pp. 478–490, July 1981.

[7] L. J. Hafer, "Automated data memory synthesis: A formal method for the specification, analysis and design of register transfer level design logic," Ph.D. dissertation,Carnegie Mellon Univ., Pittsburgh, PA, June 1981.

[8] T. J. Kowalski and D. E. Thomas, "The VLSI design automation assistant: Prototype System," in *Proc. 20th Design Automat. Conf.* (Miami Beach), June 1983, pp. 479–483.

[9] J. V. Rajan and D. E. Thomas, "Synthesis by delayed binding of decisions," in *Proc. 22nd Design Automat. Conf.* (Las Vegas), June 1985, pp. 367–373.

[10] C. Y. Hitchcock and D. E. Thomas, "A method for automatic data path synthesis," in *Proc. 20th Design Automat. Conf.* (Miami Beach), June 1983, pp. 484–489.

[11] C. Tseng and D. P. Siewiorek, "Facet: A procedure for the automated synthesis of digital systems," in *Proc. 20th Design Automat. Conf.* (Miami Beach), June 1983, pp. 490–496.

[12] ——, "Emerald: A bus style designer," in *Proc. 21st Design Automat. Conf.* Las Vegas, June 1984, pp. 315–321.

[13] E. F. Girczyc, "Automatic generation of microsequenced datapaths to realize ADA circuit descriptions," Ph.D. dissertation Carleton Univ., July 1984.

[14] P. G. Paulin anad J. P. Knight, "Force-directed scheduling in automatic datapath synthesis," in *Proc. 24th Design Automat. Conf.* (Miami Beach), July 1987, pp. 195–202.

[15] H. Trickey, "Flamel: A high-level hardware compiler," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 259–269, Mar. 1987.

[16] H. D. Man, "CATHEDRAL-II: A silicon compiler for digital signal processing," *IEEE Design and Test*, pp. 13–25, Dec. 1986.

[17] N. Park and A. C. Parker, "SEHWA: A program for the synthesis of pipelines," in *Proc. 23rd Design Automat. Conf.* (Las Vegas), July 1986, pp. 454–460.

[18] D. E. Thomas *et al.*, "The system architect's workbench," in *Proc. 25th Design Automat. Conf.* (Anaheim), June 1988, pp. 337–343.

[19] G. DeMicheli and D. Ku, "HERCULES: A system for high-level synthesis," in *Proc. 25th Design Automat. Conf.* (Anaheim), June 1988, pp. 483–488.

[20] R. Wei, S. Rothweiler, and J. Jou, "BECOME: Behavior level circuit synthesis based on structure mapping," in *Proc. 25th Design Automat. Conf.* (Anaheim), June 1988, pp. 409–414.

[21] C. Tseng *et al.*, "BRIDGE: A versatile behavioral synthesis system," in *Proc. 25th Design Automat. Conf.* (Anaheim), June 1988, pp. 415–420.

[22] M. C. McFarland, A. C. Parker, and R. Campasano, "Tutorial on high-level synthesis," in *Proc. 25th Design Automat. Conf.* (Anaheim), June 1988, pp. 330–336.

[23] R. K. Brayton *et al.*, "The Yorktown Silicon Compiler," in *Silicon Compilation.* Reading, MA: Addison Wesley, 1988, pp. 204–311.

[24] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[25] F. Romeo and A. Sangiovanni-Vincentelli, "Probabilistic hill climbing algorithms: Properties and applications," in *Proc. 1985 Chapel Hill Conf. VLSI* (Chapel Hill), Dec. 1985, pp. 393–417.

[26] G. D. Micheli and A. Sangiovanni-Vincentelli, "Multiple constrained folding of programmable logic arrays: Theory and applications," *IEEE Trans. Computer-Aided Design*, vol. CAD-2, pp. 151–167, July 1983.

[27] S. Devadas and A. R. Newton, "GENIE: A generalized array optimizer for VLSI synthesis," in *Proc. 23rd Design Automat. Conf.* (Las Vegas), July 1986, pp. 631–637.

[28] P. Egan and C. L. Liu, "Bipartite folding and partitioning of a PLA," *IEEE Trans. Computer-Aided Design*, vol. CAD-3, pp. 191–198, July 1984.

[29] A. Hashimoto and J. Stevens, "Wire routing by optimizing channel assignment within large apertures," in *Proc. 8th D. A. Workshop* (Las Vegas), 1971, pp. 155–169.

[30] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf placement and routing package," *IEEE J. Solid-State Circuits*, vol. SC-20, pp. 510–522, Apr. 1985.

[31] M. C. McFarland, "Reevaluating the design space for register-transfer hardware synthesis," in *Proc. ICCAD-87* (Santa Clara), Nov. 1987, pp. 262–265.

[32] C. Tseng and D. P. Siewiorek, "The modeling and synthesis of bus systems," in *Proc. 18th Design Automat. Conf.* (Nashville), June 1981, pp. 471–478.

[33] D. A. Hodges and H. G. Jackson, *Analysis and Design of Digital Integrated Circuits.* New York: McGraw-Hill, 1983.

[34] B. M. Pangrle, "Splicer: A heuristic approach to connectivity binding," in *Proc. 25th Design Automat. Conf.* (Anaheim), June 1988, pp. 536–541.

[35] P. M. Kogge, *The Architecture of Pipelined Computers.* New York: McGraw-Hill, 1981.

[36] E. Davidson, "Effective control for pipelined computers," in *COMPCON Dig.* (San Francisco), 1975, pp. 181–184.

[37] J. H. Patel and F. S. Davidson, "Improving the throughput of a pipeline by the insertion of delays," in *Proc. IEEE/ACM 3rd Annual Symp. Comput. Architecture* (Rochester), 1976, pp. 159–163.

[38] N. Park and A. C. Parker, "Synthesis of optimal clocking schemes," in *Proc. 22nd Design Automat. Conf.* (Las Vegas), June 1985, pp. 489–495.

*

Srinivas Devadas (S'87–M'88), for a photograph and biography, please see page 188 of the February 1989 issue of this TRANSACTIONS.

*

A. Richard Newton (S'73–M'78–SM'86–F'88), for a photograph and biography, please see page 22 of the January 1989 issue of this TRANSACTIONS.