ALGORITHMS FOR HARDWARE ALLOCATION IN DATA PATH SYNTHESIS

by

Srinivas Devadas and A. Richard Newton

Memorandum No. UCB/ERL M86/92

9 December 1986

COVER PAGE

ALGORITHMS FOR HARDWARE ALLOCATION IN DATA PATH SYNTHESIS

by

Srinivas Devadas and A. Richard Newton

x Memorandum No. UCB/ERL M86/92

9 December 1986

x ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

ALGORITHMS FOR HARDWARE ALLOCATION IN DATA PATH SYNTHESIS

by

Srinivas Devadas and A. Richard Newton

Memorandum No. UCB/ERL M86/92

9 December 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Algorithms for Hardware Allocation in Data Path Synthesis

Srinivas Devadas and A. Richard Newton
Department of Electrical Engineering and Computer Sciences
Cory Hall
University of California, Berkeley, CA. 94720

Abstract

The most creative step in the synthesis of data paths from behavioral 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. These algorithms are based on **novel formulations** of the hardware allocation problem. For example, the register allocation problem is shown to be identical to the PLA multiple folding problem. **The entire allocation process has been formulated as a two-dimensional placement problem of micro-instructions in space and time**. This formulation readily lends itself to the use of a variety of heuristics for actually solving the allocation problem. We present both simulated-annealing-based and exact branch-and-bound algorithms which optimally solve this two-dimensional placement problem thus optimally synthesizing a data path. Unlike previous approaches to automated data path synthesis, these algorithms operate under a **wide variety of user-specifiable constraints on hardware resources and costs, incorporate conditional resource sharing**, 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.

Acknowledgements

This work is supported in part by the Digital Equipment Corporation, the Semiconductor Research Corporation, and the Defense Advanced Research Projects Agency under contract N00039-86-R-0365. Their support is gratefully acknowledged.

1. INTRODUCTION

The goal of behavioral synthesis 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[thom83,tric85] in recent years. However, even now, effective and versatile procedures are not available.

Initial work to tackle this problem included the development of a mathematical model for the data path[hafe81] 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[kowa83,kowa85] 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 [mcfa78,hitc83].

A more global algorithmic approach to the allocation problem was first taken in [tsen83]. FACET is a 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). Design tradeoffs are thus not explored.

The data path synthesis techniques so far published in the literature[thom83,park86] do not incorporate conditional resource sharing[1] during

[1]Conditional resource sharing has been addressed for a pipeline synthesis problem.

the allocation process i.e. sharing hardware resources between disjoint operations. For example, the allocation techniques used in the CMU-DA project[park79], namely EMUCS, DAA and FACET operate on VT[snow78] basic blocks with a single entry and single exit point. Most previous techniques[raja85,hitc83] attempt local modifications of the input description and perform the various subtasks sequentially.

We formulate the hardware allocation problem in automatic data path synthesis as a **two-dimensional placement problem of micro-instructions in space and time.** 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. A given placement of micro-instructions corresponds to a unique data path with a certain hardware cost and execution speed. **Optimal conditional resource sharing is achieved by solving a constrained three-dimensional placement problem** where *disjoint* instructions are allowed to occupy the same spatial and temporal location. Given a data flow specification, we present algorithms which find an optimal placement of micro-instructions, thus determining the spatial and temporal delineation of resources and producing an optimal data path configuration.

We present the formulation of the data path synthesis problem as that of two-dimensional placement of micro-instructions in Section 2 and discuss modifications to incorporate conditional resource sharing. Given this formulation, simulated-annealing-based and exact branch-and-bound algorithms to solve the allocation problem are presented in Sections 3 and 4 respectively. Results and illustrative examples along with possible applications to compiler optimization are presented in Section 5.

2. THE HARDWARE ALLOCATION PROBLEM

2.1. Introduction

The input to our behavioral synthesis system is a purely architectural description of the machine in the specification language. This description contains no information about the implementation. The first step is to synthesize the data path corresponding to that description. The synthesized data path corresponds to the architectural specification and is a skeleton of the final data path which corresponds to the implementation.. After the data path synthesis step, additional hardware for pipelining, clocking etc is added and the control is synthesized.

This section describes the algorithms used in the allocation process which take the architectural description of the machine and automatically synthesize the data path corresponding to that description under hardware constraints and costs.

Given a programming language like description of a machine and hardware costs, the goal of this step is to synthesize a data path which minimizes a given arbitrary function of execution time, T and total hardware cost, C, namely $f(T, C)$.

2.1.1. Input Description

The architectural description in the specification language is converted into a code sequence where parallelism, sequentiality and disjoincy (mutually exclusive operations) are explicitly stated. The serial blocks are due to the dependences associated with any description. Disjoincy is a result of the conditional clauses in the input description. An example of an input sequence is shown in Figure 1, with *serial*, *parallel*, and *eior* blocks, which are the means of representing sequentiality, parallelism and disjoincy respectively.

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

Fig. 1 Input Description

Data path synthesis involves the specification of data and control flow, register allocation, arithmetic unit allocation and interconnection unit allocation. The constraints imposed on the synthesis are mainly hardware constraints, limitation on the number of ALUs for instance.

Any algorithm can be easily transformed into a representative data flow graph. The nodes in the data flow graph correspond to arithmetic operations and the edges to signals. The problem now is to transform the data flow graph into a functionally equivalent representation but which satisfies the constraints on the hardware. This transformed graph should produce a sequence of operations/blocks which is the optimal sequence under the hardware constraints. The various interconnections in the transformed graph have to be realized in hardware. For example, given an ADD micro-instruction $v1 = v2 +_1 v3$ the interconnections between registers v1, v2 and v3 to the arithmetic unit $+_1$ being used are implicit in the micro-instruction. Buses are vitally important as shared resources. The bus allocation problem thus has to be optimally solved.

2.2. 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 a given arbitrary function of the number of registers required, N_r, and the execution time, T, $f(N_r,T)$, is minimized.

A maximally parallel description would use lots of registers but would execute the fastest. A completely serial description would require a minimal number of registers but would be slow. An algorithm based on clique partitioning was developed[tsen83] 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 life-times of all the variables can be calculated. The life-time 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 life-times across the entire sequence. This is illustrated in Figure 2.

Disjoint variables are those whose life-times do not overlap. The allocation of registers to singly-assigned variables is finding the best possible grouping of disjoint variables in sets so the number of sets is minimized.

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

(add v1 v2 v3)
(mult v3 v1 v4)
(minus v2 v5 v6)
(inc v4 v1)
(dec v6 v2)
(divide v1 v2 v5)

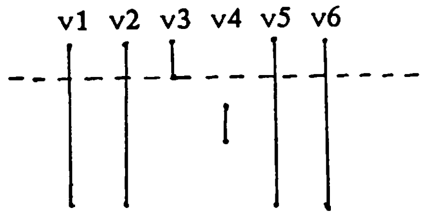


Fig. 2 Densities of Variable Life-times

units is satisfied. **This reduces to the PLA multiple folding problem**, which tries 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 life-time 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 were are trying to find is an *optimal aspect ratio* of the PLA.

The PLA folding problem has been effectively solved using graph heuristics[demi83], simulated annealing[deva86] and exact branch and bound techniques[egan84]. 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.

2.3. Formulation of the Entire Data Path Synthesis problem

Our approach to synthesize 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:

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

A procedure which minimizes C under constraints would optimally synthesize a data path.

This can be formulated as a **placement** problem of code-operations in two dimensions, that 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 Figure 3. Code operations are placed in grid locations corresponding to a ALU and 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 e.g. in a binary ADD a variable can be in the first or second positions 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 would be proportional to the delays corresponding to the code-operations occupying that slot.

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 depends on the operators occupying the grid locations in its corresponding vertical space slice. A processing unit may be simply an incrementer, or may be a complex

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 l1)		(inc k2 l2)

Fig. 3 2-Dimensional Grid of code-operations floating point unit capable of multiply, add and divide operations. Thus the formulation takes into account the grouping of arithmetic operators into processing units.

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 life-time of the variable, and the maximum density of life-times of variables across the schedule is the number of registers required to realize the variables as illustrated in Figure 2.

The interconnect/bus 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. However, other measures of interconnect complexity can be obtained from the *stagger* of the 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 more the number of ALU's the variable (and eventually the register) that it feeds into. The stagger of nets treated as separate entities does not take into account however 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 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.

Aside from interconnect cost, a good measure of the number of buses required given a schedule is the maximum number of distinct sources and number of sinks in all the time slots (which is an indicator to 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 like arrays for instance may need to be in memory. If they are accessing them potentially takes longer. There is a tradeoff between reducing the number of registers by allocating variables to memory locations and increasing the execution time. This tradeoff too can be explored if necessary.

To actually solve the problem, we can use various techniques for solving the placement problem. Two approaches have been taken. A single simulated annealing phase which produces excellent results is described in the Section 3. The optimal global placement of code-operations can also be found using an exact branch and bound scheme which is described in Section 4.

2.4. Hardware Costs

A cost file specifies the cost of hardware resources and operators. It is very general. Different costs can be specified for each succeeding register or sets of succeeding registers, e.g. first four registers cost x units, next three y units and so on. Similarly the execution time cost and the interconnect costs can be specified as a piece-wise linear or a non-linear function.

ALU operations have costs associated with them, so the algorithm can take care of grouping of operations as well during the optimization step. A floating point multiply may cost 250 units as compared with a mere 20 units for a increment operation. An example cost file is shown in Figure 4.

2.5. Conditional Resource Sharing

Conditionals can be introduced into the algorithm, unlike other allocation algorithms. This is done by defining disjoincy between statements. For example, the THEN and ELSE clauses in a 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 disjoincy and finds a optimal schedule for the code sequence with an arbitrary number of conditional clauses.

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

# 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
INTERCONNECT
1 5
10 10
100 10
```

Fig. 4 Example Cost File

Placing operations on the same time-space slot amounts to conditional resource sharing. Many forms of conditional resource sharing are possible. The co-existence 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 which share a common variable exist on the same location, a register is being shared by the two disjoint operations, and it will store information dependent on conditional clauses.

The problem thus becomes more like a **3-Dimensional placement problem with constraints in the third dimension** as to what statements can exist on the same time and space coordinates.

3. A SIMULATED ANNEALING BASED SOLUTION

3.1. Introduction

Simulated annealing[kirk83] is general combinatorial optimization technique which has been used on a variety of NP-complete problems involving a large number of variables and degrees of freedom. It belongs to a general class of algorithms called *Probabilistic Hill Climbing* (PHC) algorithms[rome85].

Theoretical results exist[rome85] that simulated annealing asymptotically approaches the global optimum of the configuration space. It has proved to be an effective solution to the cell placement problem in LSI layouts[sech85], to the generalized array optimization problem[deva86] and the global routing problem[vecc83].

The two most important things in any simulated-annealing-based algorithm are the generation of new states during the annealing process and the cost function 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 in the rest of the section.

3.2. 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.

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 before-mentioned constraints.

During the end of the annealing process i.e. at low temperatures to generate states which are more likely to be accepted, the generation of states takes a different form.

(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 to the immediate left or right in the same time slot, immediately ahead or behind in the same space slot is tried in randomly generated order.

3.3. The Cost Function

The cost function should be representative of the hardware and execution time cost function C (Section 2) which is to be optimized for.

The total execution time required for the entire sequence is one part of the cost function.

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 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 good indicator of link costs. The number of buses required is estimated by calculating the maximum number of distinct sources and number of sinks in all the time slots, since this is a good indication of the number of parallel data transfers required.

3.4. 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 a intermediate configuration and is sufficiently high so as to ensure that the final solution satisfies all the constraints.

3.5. Stopping And Inner Loop Criteria

The number of states generated per temperature point is a certain integer 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 value for three temperature points.

4. A SOLUTION BASED ON BRANCH AND BOUND

4.1. Introduction

An exact algorithm for the solution of this problem using a branch and bound technique is described. Various heuristics have to embellish the basic technique. so as to minimize cpu time requirements.

Scheduling algorithms for 'N' processors using branch and bound exist. but this problem is much more difficult because (1) the number of processors is variable and (2) the cost function is not merely the execution time. but also the number of registers, interconnections, alu's etc.

4.2. Generating All Configurations

Generating all the possible configurations recursively is accomplished using the procedure described below. A row corresponds to a time slot and a column corresponds to an arithmetic unit.

```
search()
{
  for(all instructions which can be scheduled) {
    if (curr_inst disjoint to inst[curr_row,curr_col] AND
      curr_inst satisfies precedence constraints for all inst[curr_row]) {
      schedule curr_inst at curr_row,curr_col
      continue search;
    }
    if (curr_inst satisfies precedence constraints for inst[curr_row]) {
      schedule curr_inst at curr_row, curr_col + 1
      continue search;
    }
    schedule curr_inst at curr_row+1,1
    continue search;
  }
```

A branch and bound technique is viable since the precedence constraints are typically tight, and thus the search space can be restricted. The technique implemented finds the optimal configuration, which minimizes the total cost of registers, alu operations/ALU's, and buses/interconnections. The number of required registers is calculated as in the simulated annealing case. The cost of ALU operations, is found by summing up the distinct operations for each ALU, and then summing the ALU costs. The interconnection requirement for a given configuration is calculated identically to the simulated-annealing-based solution.

Lower bounds on costs are calculated at each step. The lower bound on the maximal density of registers is estimated over the entire time slots even for incomplete configurations by assuming the following sequence to be maximally serial. The lower bound on execution time corresponds to the addition of the present execution time plus the execution time of a maximally parallel following sequence. The lower bound on processor cost is calculated by assuming a maximally serial following sequence, with the most optimistic grouping of operators. If at any time the present cost of the configuration exceeds the cost of best solution found thus far, the search is terminated.

Hardware resource constraints are incorporated into this branch and bound based algorithm by terminating the search for a solution along any branch which violates any of these constraints.

The freedom offered by symmetric binary operations is exploited after this step.

by a second optimization step.

5. EXAMPLES, RESULTS AND APPLICATIONS

We use the code-sequence in [tsen83] as our first example. The input file is shown in Figure 5. 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. 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.

In the first run, (using the simulated-annealing-based algorithm) the costs of arithmetic operations were $\geqslant 50$ units, each register cost was 10 units, each link 10 units and execution cost per time slot was fixed at 5 units. Execution speed was thus given a low priority in this run. The optimization produced a serial sequence shown in Figure 6a, which needs 8 cycles to execute. Cpu time required for the simulated annealing run was 2 minutes on a VAXstation-II. The exact branch and bound algorithm verified that this was the optimal sequence given the hardware costs in about 10 cpu minutes. The data path synthesized after bus allocation is shown in Figure 6b. It consists of 8 registers, 1 arithmetic unit, 15 links and 2 buses. The minimal number of registers and interconnections have been used.

Bus allocation is done after the code-operation placement using algorithms similar to [tsen81]. However *during the placement* the amount of interconnect required is calculated at every stage and minimized as described earlier. We have assumed that the data transfers for every micro-instruction ($op\ V_a\ V_b\ V_c$) look as follows while performing bus allocation:

V_a -> link-> bus-> link-> *ALUin* 1 V_b -> link-> bus-> link-> *ALUin* 2

ALUout -> link-> bus-> link-> V_c

```
(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. 5 Input File for example from [tsen83]

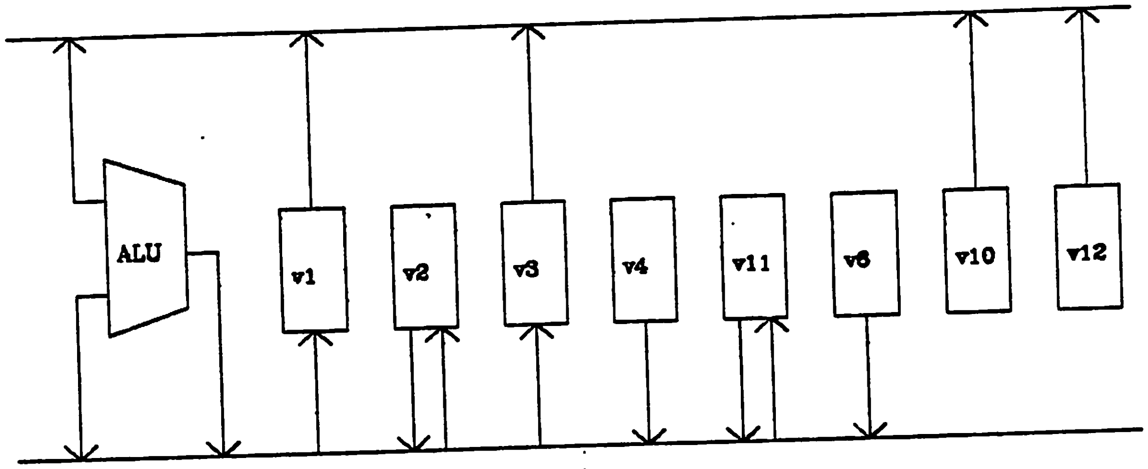
The two input transfers to the ALU are required to occur in parallel. If in fact, we are allowed to make the two input transfers to an ALU in sequence one can synthesize a data path for this example with only one bus.

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 than two buses would have been required.

Figure 7a shows the placement of code-operations produced by the program given a higher execution time cost than in the previous case, that of 50 units, with the register/ALU/interconnect cost unaltered. Note that the placement is such that operations in the two ALU's have *no* operators in common - an optimal grouping. Figure 7b shows a data path corresponding to the code-sequence in Figure 7a again with a bus-style design. The cpu time required for synthesis was about 3 minutes on a VAXstation-II. For two micro-instructions in the same time slot, all the ALUin

(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) Code-sequence after 2-D placement



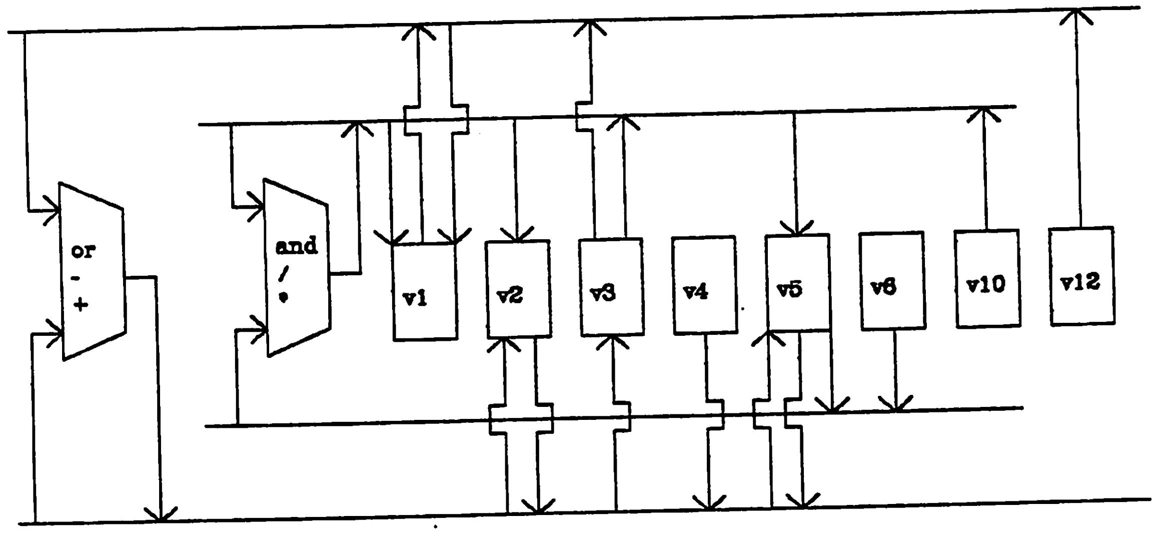
(b) Synthesised Bus-style Data-Path.
Fig. 6

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.

Unlike previous approaches to data path synthesis, our approach incorporates conditional resource sharing during the allocation process, treating the problem as one of 3-D placement with constraints in third dimension as to what code-operations can exist in the same time-space slot. We now give an example with conditional clauses in the input description.

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

(a) Code-sequence after 2-D placement



(b) Synthesised Bus-Style Data-Path.
Fig. 7

The input code-sequence is shown in Figure 8a. The *eior* (either-or) block implies that only one the blocks or operations within it is executed depending on the branching condition. Dependences have been explicitly stated using the *serial* and *parallel* blocks in this example. The hardware costs were unaltered from the previous example, but the execution time cost was fixed at 25 units to see if a good tradeoff between speed and required hardware could be made. The program took 30 cpu seconds to produce a placement of the code-sequence which is shown in Figure 8b. The code-operations within [ ] co-exist in the same time-space slot. The following points should be noted:

(1) An optimal assignment of operations to ALU's has been achieved, with the two ALU's sharing no operators in common.

(2) The minimum number of registers (5) have been used.

(3) Depending on the conditional clauses execution time is 4 or 5 cycles.

It is not necessary to constrain the data paths to be bus-style designs, though in many cases it is desirable that they are such. In the case of highly parallel sequences, many interconnections between registers and ALU's are required and buses and links look alike, due to much less sharing. Figure 8c shows the data path corresponding to the placement of micro-instructions in Figure 8b using multiplexors rather than buses.

The algorithms described in this paper can be used to perform compiler optimizations. For example, in the register-memory allocation phase of compilation, the algorithms can do the following.

(1) Optimal allocation of registers using the freedom in being able to re-order operations which don't have dependence constraints between them.

(2) Deciding which variables should reside in memory and which in a register, taking into account constraints on the number of registers and optimizing for execution time (memory access being slower than register access, heavily used variables should be in registers).

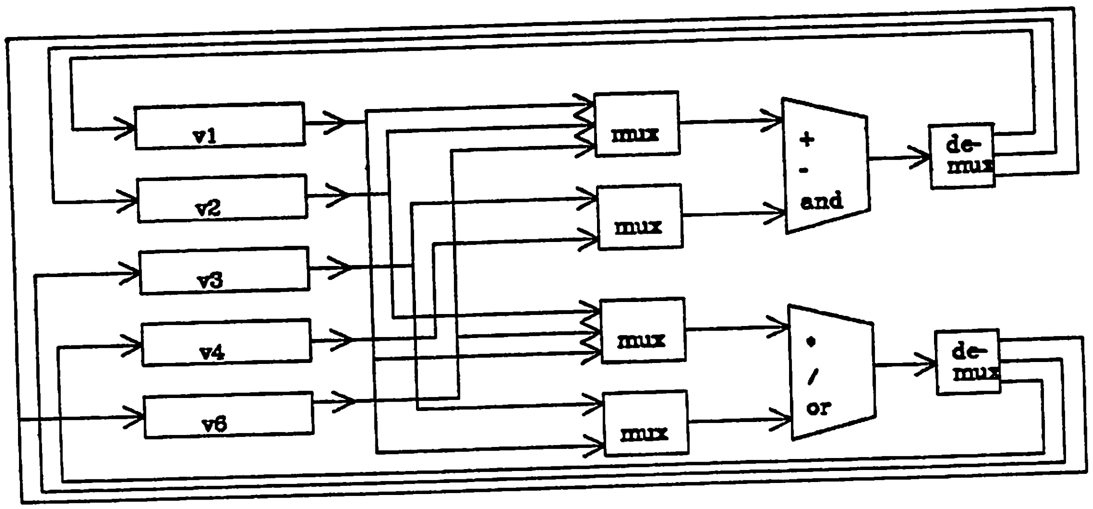
(3) Certain variables can be constrained to be in registers only.

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

(a) Input Description

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

(b) 3-Dimensional Placement



(c) Multiplexor-Style Data Path
Fig. 8

6. CONCLUSIONS

We have described a novel method for synthesising data paths from behavioral descriptions.

The entire allocation process in data path synthesis has been formulated as a two-dimensional placement problem of micro-instructions in space and time. This formulation allows simultaneous cost-constrained allocation of registers, arithmetic units, interconnect (buses and links) while trading off hardware cost against execution speed. Conditional resource sharing is incorporated by defining disjoincy between operations and formulating a three-dimensional placement problem with positional constraints.

We have presented both simulated-annealing-based and exact branch-and-bound algorithms solutions to this micro-instruction placement problem and achieved excellent results, thereby successfully demonstrating that data paths can be synthesized optimally from behavioral descriptions using this technique.

7. REFERENCES

[barb79]
M. Barbacci, G. Barnes, R. Cattell and D. P. Siewiorek, "The Symbolic Manipulation of Computer Descriptions: ISPS Computer Description Language", Carnegie-Mellon University 1979.

[demi83]
G. De Micheli and A. Sangiovanni-Vincentelli, "Multiple Constrained Folding of Programmable Logic Arrays: Theory and Applications", IEEE Transactions on CAD, July 1983.

[demi85]
G. De Micheli, R. K. Brayton and A. Sangiovanni-Vincentelli, "Optimal State Assignment for Finite State Machines", IEEE Transactions on CAD, July 1985.

[deva86]
S. Devadas and A. R. Newton, "GENIE: A Generalized Array Optimizer for VLSI Synthesis", Proceedings of the 23rd Design Automation Conference, July 1986.

[egan84]
P. Egan and C. L. Liu, "Optimal Bipartite Folding of a PLA", IEEE Transactions on CAD, July 1984.

[hafe81]
L. J. Hafer, "Automated Data Memory Synthesis: A Formal Method for the Specification, Analysis and Design of Register Transfer Level Design Logic", PhD Thesis, Carnegie-Mellon University, June 1981.

[hitc83]
C. Y. Hitchcock III and D. E. Thomas, "A Method for Automatic Data Path Synthesis", "Proceedings of the 20th Design Automation Conference, June 1983.

[kirk83]
S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing",Science, Vol.220,N. 4598, pp 671-680, 13 May 1983

[kowa83]
T. J. Kowalski and D. E. Thomas, "The VLSI Design Automation Assistant: Prototype System", Proceedings of the 20th Design Automation Conference, June 1983.

[kowa85]
T. J. Kowalski and D. E. Thomas, "The VLSI Design Automation Assistant: What's in a Knowledge Base", Proceedings of the 22nd Design Automation Conference, June 1985.

[mcfa78]
M. C. McFarland, "The VT: A Database for Automated Digital Design", Technical Report DRC-01-4-80, Design Research Center, Cernegie-Mellon University, December 1978.

[park79]
A. C. Parker, D. E. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Lieve and J. Kim, "The CMU Design Automation System", in ACM IEEE 16th Design Automation Conference Proceedings 1979.

[park86]
A. C. Parker, M. Mlinar and J. Pizarro, "MAHA: A Program for Data Path Synthesis", Proceedings of the 23rd Design Automation Conference, June 1986.

[raja85]
J. V. Rajan and D. E. Thomas, "Synthesis by Delayed Binding of Decisions", Proceedings of the 23rd Design Automation Conference, June 1985.

[rome85]
F. Romeo, and A. Sangiovanni-Vincentelli, "Probabilistic Hill Climbing Algorithms: Properties and Applications*H. Fuchs ed., 1985 Chapel Hill Conference on VLSI,* May 1985.

[snow78]
E. A. Snow, "Automation of Module Set Independent Register-transfer Level Design", PhD Thesis, Carnegie-Mellon University, April 1978.

[thom83]
D. E. Thomas, C. Y. Hitchcock III, T. J. Kowalski, J. V. Rajan and R. A. Walker, "Automatic Data Path Synthesis", IEEE Computer, December 1983.

[tric85]
H. Trickey, "Compiling Pascal Programs into Silicon", PhD Thesis, Stanford University, Jlu 1985. Stanford Computer Science Report STAN-CS-85-1059.

[tsen81]
C-J Tseng and D. P. Siewiorek, "The Modeling and Synthesis of Bus Systems", Proceedings of the 18th Design Automation Conference, June 1981.

[tsen83]
C-J Tseng and D. P. Siewiorek, "Facet: A Procedure for the Automated Synthesis of Digital Systems", Proceedings of the 20th Design Automation Conference, June 1983.

[tsen84]
C-J Tseng and D. P. Siewiorek, "Emerald: A Bus Style Designer", Proceedings of the 20th Design Automation Conference, June 1983.