ALGORITHMS FOR PIPELINE SCHEDULING AND SYNTHESIS

by

Srinivas Devadas and A. Richard Newton

Memorandum No. UCB/ERL M86/91

9 December 1986

COVER

ALGORITHMS FOR PIPELINE SCHEDULING AND SYNTHESIS

by

Srinivas Devadas and A. Richard Newton

Memorandum No. UCB/ERL M86/91

9 December 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

ALGORITHMS FOR PIPELINE SCHEDULING AND SYNTHESIS

by

Srinivas Devadas and A. Richard Newton

Memorandum No. UCB/ERL M86/91

9 December 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Algorithms for Pipeline Scheduling and Synthesis

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

Abstract

Very little work has gone into the area of synthesizing pipelined data paths from behavioral descriptions. This paper addresses the problems of pipeline scheduling and synthesis.

Given a data flow specification, in contrast to previous approaches, we define a **more general pipeline scheduling problem** which involves optimally synthesising a pipeline schedule under the given cost/performance constraints while simultaneously performing **micro-instruction timing synthesis** i.e. deciding on the costs and delays of the micro-instructions, so a given arbitrary function of pipeline execution time and hardware cost is minimized. The micro-instructions can have discretized or continuous cost-delay tradeoffs and the algorithm finds the best possible schedule simultaneously fixing the cost-delay tradeoff point for each micro-instruction.

We present algorithms to pipeline an already existing skeleton data path, adding the minimal amount of links, buses, registers and arithmetic units for the first time. Pipeline synthesis on skeleton data paths entails **partitioning the micro-instruction sequence corresponding to the data path** into phases. We have developed simulated-annealing-based and modified Kernighan-Lin partitioning algorithms to solve this pipeline synthesis problem, given hardware costs and execution time constraints.

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

Pipelining is an essential feature of the computers being designed today. Pipelining implies overlapping of multiple tasks - each computation task is partitioned into subtasks and each subtask is executed in a clock phase.

Given an input data flow specification, pipeline synthesis involves splitting the data flow graph into stages, with constraints on the number of stages and stage delays, so as to optimize for execution time and/or hardware cost. Engineering solutions to pipeline scheduling given fixed hardware resources have been published[dave75,pate76]. A pipeline synthesis procedure based on scheduling algorithms was first published in [park86]. SEHWA[park86] generates data paths from data flow graphs along with a clocking scheme which overlaps execution of tasks. SEHWA assumes that micro-instructions have fixed delays and costs. It 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.

In contrast to SEHWA's approach we **define a more general pipeline scheduling problem incorporating micro-instruction timing synthesis** i.e. finding a schedule of operations simultaneously deciding on the costs and delays of the micro-instructions, so a given arbitrary function of pipeline execution time and hardware cost is minimized. The micro-instructions can have discretized or continuous cost-delay tradeoffs and the algorithm finds the best possible schedule, fixing the cost-delay tradeoff point for each micro-instruction.

Many applications may require pipelining an already existing data path into a certain number of pipeline phases. Given a skeleton data path, **pipeline synthesis involves partitioning the micro-instruction sequence into phases adding the minimal amount of hardware**. The extra hardware required could be not only pipeline latches or processing units but also buses, links and registers. Previous work in this area[park85] concentrated on minimizing execution time of the pipelined data paths without regard to hardware costs. Given hardware costs and/or execution time constraints, we present, for the first time, simulated-annealing-based and modified Kernighan-Lin partitioning algorithms for optimally pipeline skeleton data paths.

The general pipeline synthesis problem given an input data flow specification is described in Section 2 along with a heuristic algorithm to solve it. Results using the algorithm on digital signal processor examples are given in Section 3. Pipeline synthesis starting from general purpose skeleton data paths is described and solved using simulated annealing and modified Kernighan-Lin algorithms in Section 4.

2. A GENERAL PIPELINE SYNTHESIS PROBLEM

2.1. Problem Definition

The input description is a data flow graph G, which describes the computations tasks to be performed. Define $M = m_1, m_2 .. m_N$ as the set of micro-instructions ϵG and $O = o_1, o_2 .. o_Z$ as the set of distinct operators in G. Each o_i has a set of costs c_{ij}, and delays d_{ij}, with $j = 1, T_i$, where $T_i \geqslant 1$ is the number of tradeoff points for each o_i. The c_{ij} and d_{ij} values for each o_i are constrained to be monotonic i.e.

$$c_{ik} > c_{il} \;=>\; d_{ik} < d_{il}$$

and vice versa. For a given pipeline schedule with micro-instruction cost-delay points fixed, we can the total hardware cost C is given by

$$C = \sum_{k=1}^{N} \sum_{l=1}^{T_k} c_{kl} \cdot n_{kl}$$

where n_{kl} gives the number of units required of each kind (tradeoff point) and each type of operator. The execution time E for N_I instructions in the ideal case neglecting resynchronization would be

$$E = (N_I - 1 + K)^* \; max(S_i \; , i = 1 \; , K)$$

where K is the number of stages in the pipeline and S_i is the delay of the i th stage. The S_i can be calculated given the schedule and d_{kl}.

Three possible optimization problems can be defined.

(1) Cost constrained synthesis: minimize E subject to $C \leqslant C_0$.

(2) Performance constrained synthesis: minimize C subject to $E \geqslant E_0$

(3) Function minimization: minimize $f(C, E)$

2.2. Global Strategy

Most scheduling problems are NP-complete. The pipeline scheduling problem addressed in [rama75] was also shown to be NP-complete. The general pipeline scheduling problem described in the previous section has an extra degree of freedom and is thus more difficult to solve. Therefore, heuristic techniques must be used.

In addition to the input data flow description and micro-instruction cost-delay information, the number of stages in the pipeline may be constrained to lie between certain values, and a limit can be placed on the maximum stage delay. The maximum stage delay can be derived from the constraint on E in the case of performance constrained synthesis if $N_I >> K$.

The global strategy to the solve the problem is as follows: First, given the mode of optimization (e.g. cost constrained, performance constrained), the operation delays and costs are fixed at different sets of values and a number of feasible schedules are generated. These schedules are passed through a micro-instruction timing optimization phase where the overall structure of each schedule is not changed but the cost-delay points for each micro-instruction are massaged for best possible results.

2.3. The First Stage - Generating Feasible Schedules

Given the mode of optimization. Stage 1 proceeds as follows:

Cost constrained synthesis: The micro-instruction cost-delay points are fixed at the minimum delay, maximum cost point. A maximally parallel schedule is synthesized. If the schedule does not violate the cost constraint, the problem has been solved, since this is the fastest possible schedule. However, in most cases, the cost of this schedule will be prohibitively high. Therefore, the schedule is serialized in four ways:

(a) The delays of micro-instructions in each stage beginning from the first stage are increased to the maximum without violating the maximum stage delay limit. The process ends when $C_{schedule} \leqslant C_0$.

(b) Same as (a) but beginning from the last stage and moving upward.

(c) Starting from the first stage, the micro-instructions are serialized till the maximum stage delay limit is reached, the remaining operations are pushed to the next stage and the process continues till $C_{schedule} \leqslant C_0$.

(d) Same as (c) except starting from the last stage and pushing micro-instructions upward.

If a feasible schedule has not been found after (a), (b), (c) and (d), the lowest cost results from (a) and (b) are processed through (c) and (d), and vice versa.

Performance constrained synthesis: The process is identical to the cost constrained synthesis except the procedures (a), (b), (c) and (d) are terminated when $E_{schedule} \leqslant E_0$. That is, the cost of the design is reduced to the point where further reduction violates the performance constraint. If the maximally parallel initial schedule does not satisfy the performance constraint, no feasible schedule is possible.

$f\ (C\ ,E)$ *minimization during synthesis*: each operator cost-delay point in each micro-instruction $m_{i}(c_{kM})$, $m_{i}(d_{kM})$ (where o_k is the operator in m_i) is fixed such that $j = M$ is

$$MIN(\ f\ (c_{kj}\ ,d_{kj}\ )\ )\ \text{over all } j .$$

A maximally parallel schedule is created with these micro-instruction delays. The maximally parallel schedule is drawn out using procedures (c) and (d) until further serialization violates the limit on the maximum number of stages. The feasible schedules selected are the maximally parallel schedule itself, and serialized outputs from procedures (c) and (d).

2.4. The Second Stage - Optimizing Micro-instruction Timing

The schedules generated by the first step are optimized for micro-instruction timing using critical path analysis as described below.

Each stage in the schedule is optimized separately. In the cost constrained mode, each stage is optimized for performance without exceeding its initial cost. In the performance constrained mode, each stage is optimized for minimum cost, without violating the maximum stage delay limit or the performance limit. In the $f\ (C\ ,E)$ minimization mode, each stage i is minimized for $f\ (SC_i\ ,SE_i)$, where SC_i and SE_i are the stage costs and stage execution times respectively.

The algorithm if no precedence constraints exist between operations in a given stage is as follows:

```
if (stage $S$ is such that no precedence constraints
                                    exist between operations) {
   for ( $o$ = each operator in $S$ ) {

     $N_o$ = no. of micro-instructions with operator $o$
      for ( $i$ = each operator tradeoff point ) {

   case A: if ( MODE is performance constrained ) {

           $n_i$ = number of processing units required
            without violating performance constraints.

           $OC_i = n_i * c_{oi}$
         }
   case B: if ( MODE is cost constrained ) {

           $n_i$ = maximum number of processing units possible
            without violating stage cost constraints.

           $OE_i = \left\lceil \frac{N_o}{n_i} \right\rceil$
         }
   case C: if ( MODE is $f\,(C ,E)$ minimization ) {

            for ( each $n_i$ which does not violate
               maximum stage limit AND $n_i \leqslant N_o$ ) {

               calculate $OC_i$ and $OE_i$ as before.
            }
            pick $n_i$ which minimizes $f\,(\,OC_i\ ,OE_i\,)$.
         }
      }
      case A: Select $j$ so $OC_j$ is min. over tradeoffs.
      case B: Select $j$ so $OE_j$ is min. over tradeoffs.
      case C: Select $j$ so $f\,(\,OC_j\ ,OE_j\,)$ is min. over tradeoffs.
   }
}
```

However, in the general case, precedence constraints exist. Critical path analysis is then required for optimization.

```
if ( precedence constraints exist ) {

 OPTLOOP:
     Find critical path $P = ( m_{i1} , m_{i2..} )$ through $S$

     optimizePath( $P$, MODE )
     Tag all ( $m_{i1} , m_{i2..}$ ) as optimized.

     Recompute critical path $P$
     if ( $P$ unchanged ) {

       Increase delay of each untagged $m_j$ in $S$ for
       minimum cost without exceeding delay of $P$.
     }
     else {
       goto OPTLOOP
     }
}
```

The procedure optimizePath() takes a given micro-instruction stream and optimizes it for performance, cost or $f\ (C\ ,E)$. It is described below.

```
procedure optimizePath( $P$, MODE ) {

  if ( MODE is $f\ (C\ ,E)$ minimization ) {
    for ( each $m_i$ in $P$ ) {
      Fix point $j$ so $f\ (\ c_{ij}\ ,d_{ij}\ )$ is minimum.
    }
  }
  else if ( MODE is performance constrained ) {

    $sum\_median\_delay = 0$
    for ( each $m_i$ in $P$ ) {

      find median delay points $d_{im}$
      $sum\_median\_delay\ =\ sum\_median\_delay\ +\ d_{im}$
    }
    for ( each $m_i$ in $P$ ) {
      $Ldelay_i\ =\ d_{im}\ *\ max\_allowed\_delay\ /\ sum\_median\_delay$
    }

    Sort $m_i$ in $P$ according to increasing
    cost slope around median delay point $d_{im}$.

    $slack = 0$
    for ( $m_i$ in sorted order ) {
      Pick $r$ so $d_{ir}\ \leqslant\ slack\ +\ Ldelay_i$
                   AND
             $d_{ir+1}\ >\ slack\ +\ Ldelay_i$
      $slack\ =\ slack\ +\ Ldelay_i\ - d_{ir}$
    }

    for ( $m_i$ in reverse sorted order ) {
      while ( $slack > 0$ ) {
        if ( $d_{ir+1}\ \leqslant\ d_{ir}\ +\ slack$ ) {
          $slack = slack\ - d_{ir+1} + d_{ir}$
          Update $r$
        }
      }
    }
  }
  else if ( MODE is cost constrained ) {

    /* same as performance constrained except the role
      of delay and cost is interchanged. */
  }
}
```

In most cases. procedure optimizePath() is called only once. since there is usually one path in the stage which is much more critical than the others. nevertheless to avoid looping indefinitely around OPTLOOP. if a path P repeats itself the process is terminated. After all the feasible schedules have been optimized. the best among them is selected.

2.5. Resynchronization and Conditional Resource Sharing

Events which break a pipeline are called *resynchronization* events. Resynchronization hurts longer pipelines (pipelines with larger number of stages) more than shorter ones since the time required to start the pipeline up again i.e the set up time $(K - 1) * T_{clock}$ is proportional to the number of stages K. The equation for the execution time E has to be modified to take resynchronization into account. We refer the reader to [park86] where a general equation for the execution time of a sequence of tasks. a subset causing resynchronization. has been derived. This equation is used to calculate E in our procedures given a resynchronization rate.

Conditional clauses are common in data flow specifications. Sharing processors between *disjoint* (mutually exclusive) operations is essential for cost-effective pipelines. Conditional resource sharing is incorporated into the scheduling algorithms by constructing a $N * N$ disjoincy matrix for N micro-instructions after analyzing the conditional clauses in the input data flow. This matrix is used to test for disjoincy during the scheduling. if two micro-instructions are disjoint and possess the same operator. they can be placed on the same time-space point (share the same processor in the same time zone).

3. ILLUSTRATIVE EXAMPLES

Digital signal processors are good candidates for pipelining. The first example is a textual data flow specification of the processor given in [park86] and is shown in Figure 1a. This data flow specification was pipelined assuming fixed micro-instruction (add and multiply) costs/delays in [park86]. Micro-instruction timing synthesis during pipeline scheduling, is illustrated by allowing for two adders - a fast adder which costs 1.5 units and performs additions in 25 ns and a slower adder with a cost of 1.0 units and delay of 40 ns. Only one kind of multiplier is assumed to exist with a cost of 2.0 units and a 80 ns delay.

Given latch delays of 20 ns, latency 1, and a maximum stage limit of 100 ns, the fastest possible schedule given a cost constraint of 25.0 units is shown in Figure 1b. This schedule uses 5 stages, costs 23.5 units and is optimal. The [] around a set of operations e.g. stages 1 and 2, implies that all the operations are being performed in parallel. The two adder units are denoted by $+_s$ (slow adder) and $+_f$ (fast adder). both kinds of adders have been used to maximum advantage.

A second example of pipelining a data flow specification with more complicated precedence constraints and tradeoffs is illustrated in Figure 2. Figure 2a 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 6 stages (performance constrained synthesis). The schedule synthesized is shown in Figure 2b. $+_f$ denotes a fast adder and $+_s$ a slow adder (similarly for multiply). Both kinds of adders and multipliers have been used, again to maximum advantage. Since the latency is 2, resources can be shared across stages 1 and 2, 3 and 4, 5 and 6 so one $+_s$, two $+_f$, two $*_s$ and two $*_f$

$w1 = v1 + v2$ $w2 = v3 + v4$ $w3 = v5 + v6$ $w4 = v7 + v8$
$w5 = v9 + v10$ $w6 = v11 + v12$ $w7 = v13 + v14$ $w8 = v15 + v16$
$x1 = v17 * w1$ $x2 = v18 * w2$ $x3 = v19 * w3$ $x4 = v20 * w4$
$x5 = v21 * w5$ $x6 = v22 * w6$ $x7 = v23 * w7$ $x8 = v24 * w8$
$y1 = x1 + x2$
$y2 = y1 + x3$
$y3 = y2 + x4$
$y4 = y3 + x5$
$y5 = y4 + x6$
$y6 = y5 + x7$
$y7 = y6 + x8$

(a) DSP data flow specification

[$w1 = v1 +_s v2$ $w3 = v5 +_s v6$ $w5 = v9 +_s v10$ $w7 = v13 +_s v14$]
[$w2 = v3 +_s v4$ $w4 = v7 +_s v8$ $w6 = v11 +_s v12$ $w8 = v15 +_s v16$]

[$x1 = v17 * w1$ $x2 = v18 * w2$ $x3 = v19 * w3$ $x4 = v20 * w4$
$x5 = v21 * w5$ $x6 = v22 * w6$ $x7 = v23 * w7$ $x8 = v24 * w8$]

$y1 = x1 +_f x2$
$y2 = y1 +_f x3$
$y3 = y2 +_f x4$

$y4 = y3 +_s x5$
$y5 = y4 +_s x6$

$y6 = y5 +_s x7$
$y7 = y6 +_s x8$

(b) Cost constrained pipelining
Fig. 1

units are required adding up to a total cost of 14.0 units. The multiplier in stages 5-6

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) Input specification with cost-delay tradeoffs

v1 = x1 $+_s$ x2 v3 = x5 $*_s$ x6
v1 = x3 $+_s$ x4

w1 = v1 $+_s$ x3 v4 = x7 $*_s$ x8
w2 = v2 $+_s$ x2

w3 = v3 $*_f$ x7 w4 = v4 $*_s$ x6
y3 = w3 $+_f$ v1

y1 = w1 $*_f$ v3 y2 = w2 $*_s$ v4
y4 = w4 $+_f$ v2

z1 = y1 $+_f$ y3 z2 = y1 $*_f$ y3
z3 = y2 $+_f$ y4
a1 = z1 $+_f$ x5

a2 = z2 $+_f$ x6 z4 = y2 $*_f$ y4
a3 = z3 $+_f$ x7
a4 = z4 $+_f$ x8

(b) Performance constrained pipelining
Fig. 2

has to be a $*_f$ unit since a4 has to be computed after computing z4 in stage 6.

4. PIPELINE SYNTHESIS FROM SKELETON DATA PATHS

4.1. Problem Definition

Pipelining a general purpose data path is a different problem from synthesizing a pipeline schedule for a data flow specification which is primarily concerned with the number of processing units and intermediate latches required.

Given a skeleton data path which is made up of arithmetic units and registers interconnected by buses and links and representative of a fixed micro-instruction sequence, pipeline synthesis would involve partitioning the micro-instruction sequence into a given number of phases with the minimal addition of hardware.

Hardware resources have to be added during the pipelining step since resources no longer can be shared across the entire sequence, only across the individual phases (assuming single micro-cycle phases - resource sharing constraints are different when the individual phases themselves are made up of multiple micro-cycles as explained in Section 4.4). Hardware costs and the minimum and maximum limits on the delays of each of the phases have to be taken into account during the pipeline synthesis step.

The initial micro-instruction sequence representing the skeleton data path possesses more information than in the data flow case. For example, V1 = V3 + V4 in the data flow case may be expanded to a form

```
V3 -> link1 -> bus1 -> link3 -> Alu1_in1
V4 -> link2 -> bus2 -> link4 -> Alu1_in2
Alu1_out -> link5 -> bus1 -> V1
```

representing the skeleton data path. Thus shifting the micro-instruction to another ALU changes interconnect requirements as well and may have far reaching effects on the number of buses and links required.

We assume hardware costs for links, buses, registers and operators. Using modified Kernighan-Lin and simulated-annealing-based techniques, we develop algorithms for the partitioning of the micro-instruction sequence into a given number of phases, with limits on the delays of each individual phase, and with minimal increase in hardware cost.

4.2. Initial Partition

An initial partition of micro-instructions into the given number of phases is constructed. This initial partition satisfies the constraints on the delays on the different phases.

Starting from the first phase, micro-instructions are packed into a phase till the total phase delay exceeds or is equal to the mean delay (mean of the minimum and maximum limits). During this packing the sequencing of micro-instructions is not disrupted. Even if precedence constraints do not exist between two micro-instructions, they are placed in the same order in the phase as in the initial sequence to keep resource sharing across time zones intact. Then, a new phase is begun and the process repeated. This is done for the all the stages.

The cost of this partition is found by enumerating all the distinct arithmetic operators, registers, links and buses required *for each phase* and summed over all the phases.

4.3. Modified Kernighan-Lin Algorithm

Given the initial partition, a number of micro-instruction moves are made across phases in an effort to minimize hardware cost. The Kernighan-Lin algorithm[kern70] was originally proposed for partitioning graphs.

Given a starting configuration of N objects each in Partition A and in Partition B, the Kernighan-Lin algorithm performs N interchanges across the partition selecting the two objects (one from each partition) every time which maximize the gain accrued by the interchange. This gain can be negative however. The cost function is evaluated at the end of each move and stored. After all the moves have been made, the best configuration seen across the entire set of moves is chosen as the starting point for the next iteration. The process is terminated if an iteration fails to improve the cost function.

For the pipeline synthesis problem, we propose a modified version of the algorithm described above. Given an initial two-way partition of micro-instructions, N_A and N_B, a subset of micro-instructions M_A and M_B are chosen from N_A and N_B.

Let M_A be the set of all micro-instructions which have no children in the data flow graph (DAG) representing the micro-instructions in partition A i.e. N_A. Similarly let M_B be the set of all micro-instructions which have no ancestors in the DAG representing the micro-instructions in partition B alone i.e. N_B. Precedence constraints may exist between micro-instructions in M_A and M_B. Starting from the initial configuration, the gains accrued due to all possible displacements of $m \in M_A$ from A to B which do not violate the precedence or delay constraints are calculated, and alternating with each A to B displacement, the maximal gain micro-instruction displacement of $m \in M_B$ is tried from B to A. The hardware cost is evaluated after each move and the best possible configuration stored. This serves as the starting point for the next iteration. The number of trials in each iteration is bounded by $M_A + M_B$, but is typically less. The process terminates when an iteration fails to improve the cost.

The process just described is done for all sets of consecutive phases i.e. 1 and 2, 2 and 3 and so on.

4.4. Simulated Annealing Based Algorithm

Simulated annealing[kirk83] is a general combinatorial optimization technique which draws an analogy to the physical cooling (annealing) process in materials. It has proved to be an effective solution for a variety of optimization problems involving a large number of degrees of freedom and a large number of variables. The LSI cell placement problem has been effectively solved using simulated annealing[sech85] as has the PLA folding problem[deva86].

Simulated annealing is characterized by a random generation of new states: it allows hill-climbing moves during the optimization process. i.e. a move may be accepted with a finite probability even if increases the cost of the configuration. A parameter analogous to temperature in the physical annealing process governs the acceptance of these moves. which is less and less likely at low temperatures.

The main features of a simulated-annealing-based algorithm are the generation of states and the cost function. In this case. the cost function is the hardware cost required for a given configuration.

The generation of states proceeds as follows:

(1) A phase is randomly picked. If it is not the first or last phase a random number between 0 and 1 is generated. If the number is < 0.5 the phase before it is picked as the second phase. else the phase after it is picked. In the case of the first and last phases a single choice exists.

(2) Given the two consecutive phases. with the micro-instructions in them. a randomly picked micro-instruction in the no-descendant list of the phase ahead in time is displaced to the following phase if it is still placed in the original phase and if precedence constraints and delay constraints are not violated. else a randomly picked micro-instruction in the no-ancestor list of the latter phase is displaced to the former. again only if precedence and delay constraints allow the displacement.

At each temperature point. moves equal to an integer multiple of the sum total of the no-ancestor and no-descendant micro-instructions for each phase (typically 1-10) are generated. after which the temperature is decreased to a fraction (typically 0.90) of its original value. The annealing process is terminated when the cost does not change for four temperature points.

4.5. Micro-cycles Within Phases

In the general case, an individual phase may be made up of several micro-cycles. Assume a data path whose clocking scheme comprises of N phases with M micro-cycles in each phase. Given a hardware resource (bus, link, register, arithmetic unit) in $[\, phase_i \,, micro\text{-}cycle_k \,]$ the same cannot be re-used in $[\, phase_j \,, micro\text{-}cycle_k \,]$, $j \neq i$. However, it can be re-used in a different micro-cycle in a different phase i.e. $[\, phase_j \,, micro\text{-}cycle_l \,]$, $j \neq i$ and $l \neq k$.

The overall structure of the algorithms described in Sections 4.2 and 4.3 remains the same for the case of multiple micro-cycles as well, however the hardware cost calculations should take into account the new set of constraints on resource sharing.

4.6. An Example

We now give an example which illustrates how a skeleton unpipelined data path can be pipelined with the minimal addition of hardware resources using the algorithms described in this section.

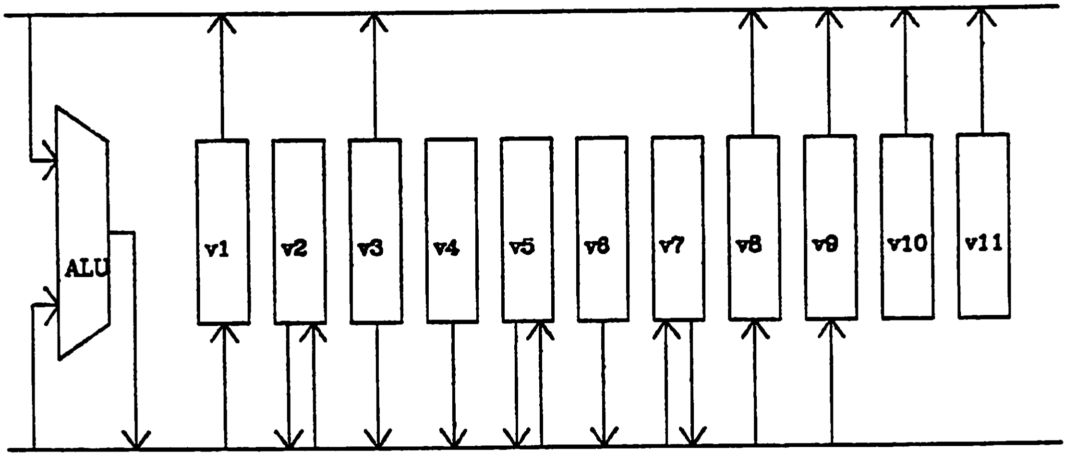
Figure 3a shows a optimized code-sequence and the corresponding unpipelined data path is shown in Figure 3b. Suppose we want to pipeline this data path into two phases so as to increase throughput by a factor of 2. An initial partition is chosen near the middle of the sequence between micro-instructions 6 and 7. Given that resources cannot be used across phases, the pipelined datapath corresponding to this description requires an ALU with the arithmetic operations and, +, -, and * for the first phase and an ALU with the operations *or*, -, + and / for the second phase. With this partition, 8 registers are required for the first phase, namely, v1, v2, v3, v4, v5, v6, v7 and v11. 9 registers are required for the second phase, v1', v2', v3', v5', v7', v8, v9, v10 and v11'.

In all 17 registers are required as compared to 11 in the unpipelined data path. Values have to be passed across phases between the 6 registers pairs (v1, v1'), (v2, v2'), (v3, v3'), (v5, v5'), (v7, v7') and (v11, v11').

This initial partition can be improved so as to minimize the amount of hardware required. Micro-instructions 5 and 6 in phase 1, and micro-instructions 7, 8, 9, 10 and 11 in phase 2 are identified as being those with no descendants and ancestors respectively in their corresponding phases. Applying the modified Kernighan-Lin algorithm

```
(1) v3 = v1 + v2     (2) v11 = v1
(3) v5 = v3 - v4
(4) v2 = v3 * v6
(5) v3 = v3 + v5
(6) v7 = v1 and v7
(7) v8 = v8 - v5
(8) v9 = v9 or v7
(9) v2 = v1 + v2
(10) v5 = v10 / v5
(11) v1 = v3 and v5
(12) v2 = v11 or v2
```

(a) Code Sequence



(b) Unpipelined Data Path
Fig. 3

produces a new partition of micro-instructions into two phases which is shown in Figure 4a. The corresponding data path is shown in Figure 4b. The improved partition requires an ALU in the first phase for +, - and * operators, and an ALU in the second with and, *or* and / operators, a total of 6 operators as compared to 7 in the initial partition. A total of 16 registers are required (as compared to 17 in the initial partition) in the two phases. Values have to passed across phases between 5 register pairs instead of 6 as in the initial partition. Thus the number of required arithmetic operators, registers and links has been minimized.

5. CONCLUSIONS

We have presented new algorithms for pipeline scheduling and synthesis from behavioral descriptions. These problems have been proven to be NP-complete and we have developed heuristic algorithms which achieve near optimal results.

Given a data flow specification, in contrast to previous approaches, we have defined a more general pipeline synthesis problem involving micro-instruction timing synthesis and presented a heuristic solution which achieves excellent results.

Given a skeleton unpipelined data path, we have presented, for the first time, simulated-annealing-based and modified Kernighan-Lin partitioning algorithms to pipeline the data path adding the minimal amount of hardware, namely registers, arithmetic units and interconnect.

6. REFERENCES

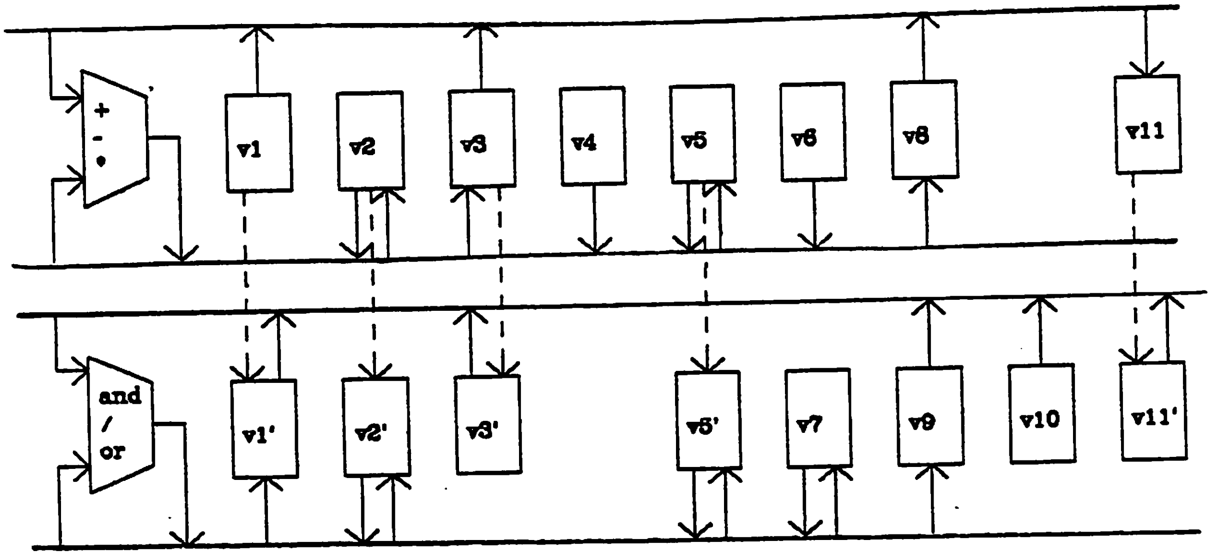
[dave75]
E. Davidson et. al, "Effective Control for Pipelined Computers", COMPCON Digest, pp 181-184, 1975.

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

[kern70]
B. W. Kernighan and S. Lin, "An efficient heuristic procedure for Partitioning graphs" *The Bell Syst. Tech. journal* 49:2, pp 291-307.

```
(1) v3 = v1 + v2     (2) v11 = v1
(3) v5 = v3 - v4
(4) v2 = v3 * v6
(5) v3 = v3 + v5
(7) v8 = v8 - v5
(9) v2 = v1 + v2
--------------------------------
(6) v7 = v1 and v7
(8) v9 = v9 or v7
(10) v5 = v10 / v5
(11) v1 = v3 and v5
(12) v2 = v11 or v2
```

(a) Partitioned Code Sequence



(b)'Pipelined Data Path
Fig. 4

[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

[park85]
N. Park and A. C. Parker, "Synthesis of Optimal Clocking Schemes", Proceedings of the 22nd Design Automation Conference, June 1986.

[park86]
N. Park and A. C. Parker, "SEHWA: A Program for the synthesis of pipelines", Proceedings of the 23rd Design Automation Conference, June 1986.

[pate76]
J. H. Patel and E. S. Davidson. "Improving the throughput of a Pipeline by the insertion of delays", IEEE/ACM 3rd Annual Symposium on Computer Architecture, pp 159-163, 1976.

[rama75]
C. V. Ramamoorthy and H. F. Li. "Some Problems in Parallel and Pipeline Processing", Proceedings of COMPCON, IEEE, pp 177-180, 1975.

[sech85]
C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package", IEEE Transactions on Circuits and Systems, April 1985.