7

APPENDIX

A1 ILP Formulation for Problem *P1*

The hardware/software mapping and scheduling problem *P1* is formally described by the following integer linear program:

<u>Given</u>:

1. A set of nodes $N = \{1, 2, \ldots, \}$, representing the tasks in the algorithm.
2. Associated with each node i in N, are four non-negative integers: ah_i, as_i, th_i, and ts_i, representing the area and execution time corresponding to the in hardware and software mappings of node i.
3. A directed acyclic graph $G = (N,A)$. A is a set of arcs (i,j), where $i, j \in N$. An arc (i,j) in A implies that task i must complete its execution before task j can start its own execution.
4. Associated with each arc $(i,j) \in A$ is a non-negative number p_{ij}, which represents the number of samples sent from node i to node j.
5. The source and sink nodes in the graph G are identified as S and K respectively.

6. Six non-negative integer-valued parameters AH, AS, T, ah_{comm}, as_{comm}, and t_{comm} are specified. AH and AS are the capacities of the hardware and the software resource respectively. The total area of all nodes mapped to hardware (software) cannot exceed AH (AS). D is the deadline constraint, i.e., the time between the start time of the source S and the finish time of the sink K should be at most D. Parameters ah_{comm}, as_{comm}, and t_{comm} represent the hardware area required, the software area required, and the delay incurred when one sample of data is sent between two nodes mapped to different resources respectively.

Variables:

1. For each $i \in N$, define binary variables h_i and s_i such that $h_i = 1$ if task i is mapped to hardware, 0 otherwise; and $s_i = 1$ if task i is mapped to software, 0 otherwise. Obviously, $h_i + s_i = 1$. We use the two variables to make the formulation easier to read; only h_i contributes to the variable count. (N variables)

2. For each $i \in N$, define a non-negative integer variable t_i, the starting time of task i. (N variables)

3. For each $(i,j) \in A$, define a binary variable b_{ij} such that $b_{ij} = 1$ if both i and j are in hardware, 0 otherwise. (A variables)

4. For each $(i,j) \in A$, define a binary variable c_{ij}, such that $c_{ij} = 1$ if nodes i and j are on different mappings, and 0 if on the same mapping[1].

5. For each $i, j \in N$, define a binary variable d_{ij} such that $d_{ij} = 1$ if $t_i \le t_j$, otherwise $d_{ij} = 0$. (N(N-1) variables)

1. c_{ij} will be defined in terms of b_{ij} and hence it is not included in the variable count.

Objective function F: minimize hardware area

$$\text{minimize}\left(\sum_{i \in N} h_i \times ah_i + \sum_{(i,j) \in A} c_{ij} \times p_{ij} \times ah_{comm}\right)$$

Subject to:

1. Hardware capacity constraint:

$$\sum_{i \in N} h_i \times ah_i + \sum_{(i,j) \in A} c_{ij} \times p_{ij} \times ah_{comm} \le AH \text{ (1 constraint)}$$

2. Software capacity constraint:

$$\sum_{i \in N} s_i \times as_i + \sum_{(i,j) \in A} c_{ij} \times p_{ij} \times as_{comm} \le AS \text{ (1 constraint)}$$

3. Deadline constraint:

$$t_K + (h_K \times th_K) + (s_K \times ts_K) \le D \text{ (1 constraint)}$$

4. Precedence constraints:

Regardless of the mappings, if $(i,j) \in A$, then j starts executing at the end of i.

$$t_i + (h_i \times th_i) + (s_i \times ts_i) + (c_{ij} \times p_{ij} \times t_{comm}) \le t_j,$$

where $(i,j) \in A$ (A constraints)

$d_{ij} = 1$ for $(i,j) \in A$ (A constraints)

5. Processor overlap constraint:

No two jobs can be executed simultaneously on the software processor. We want to express the fact that if i and j are both on the software processor, and j executes before i (i.e., $(j, i) \in A$), then t_j + execution time of j $\le t_i$. The variable d_{ij} is 1 if $t_i \le t_j$, and 0 otherwise. Define $M = \sum_i ts_i$. The meaning of d_{ij} is expressed by the following constraints:

$(d_{ij} + d_{ji}) = 1$ for $i < j$, $i, j = 1, \ldots, N$. (N(N-1)/2 constraints)

$t_j - t_i - (M \cdot d_{ij}) \le 0$ for $i \ne j$, $i, j = 1, \ldots, N$. (N(N-1) constraints)

Since $t_j - t_i < M$ for an optimal solution, the expression is restrictive only for $d_{ij} = 0$. If $d_{ij} = 0$, then j starts no later than i. The overlap constraint can now be specified as:

$$t_j + (h_j \times th_j) + (s_j \times ts_j) + \sum_k c_{jk} \times p_{jk} \times t_{comm} - t_i \le M \cdot (3 - d_{ji} - s_i - s_j)$$

for $i \neq j$, $i,j = 1, \ldots, N$, k such that $(j,k) \in A$ (N(N-1) constraints)

The expression is restrictive only for $d_{ji} = 1$ and $s_i = s_j = 1$, i.e., i and j are both on software, and j starts no later than i. In this case it guarantees that t_j + execution time of i $\le t_i$, as it should be.

6. Additional constraints: To account for costs incurred when two communicating nodes i and j are on different mappings, we use c_{ij}, where c_{ij} is expressed as:

$$c_{ij} = 1 - (h_i \times h_j) - ((1 - h_i) \times (1 - h_j)) \text{, where } (i,j) \in A$$

$$c_{ij} = h_i + h_j - 2h_ih_j$$

Note that this is a quadratic constraint in h_i, and it can be linearized by defining a binary variable b_{ij} for all (i,j) in A such that $b_{ij} = h_ih_j$, where b_{ij} can be expressed by:

$$b_{ij} \le h_i$$

$$b_{ij} \le h_j$$

$$b_{ij} \ge (h_i + h_j - 1)$$

The original expression for c_{ij} can now be replaced by:

$$c_{ij} = h_i + h_j - 2b_{ij}$$

Note that only b_{ij} adds to the variable count (3A constraints); c_{ij} is used only to enhance readability; in the actual formulation c_{ij} is replaced by $(h_i + h_j - 2b_{ij})$. (3A constraints)

A solution to the mapping and scheduling problem (*P1*) is specified completely by h_i (mapping) and t_i (schedule). The formulation has $(N^2 + N + A)$ variables and $(\frac{5}{2} \cdot N^2 - \frac{5}{2} \cdot N + 5A + 3)$ constraints, in addition to the integrality constraints on the variables.

A2 Proof of NP-Completeness for Problem *P1*

Let $\overline{P1}$ be the decision (or recognition) version of *P1*, i.e., $\overline{P1}$ is the problem of finding a *feasible* mapping and schedule. We will first show that $\overline{P1}$ is NP-complete. Since *P1* is at least as hard as $\overline{P1}$, it follows that *P1* is NP-hard.

Claim: $\overline{P1}$ is NP-complete.

Proof:

1. $\overline{P1}$ is in NP

   A given solution (schedule and mapping) can be verified in polynomial time.

2. $\overline{P1}$ can be restricted to the Precedence Constrained Scheduling problem (*PCS*)

   The precedence constrained scheduling problem [Garey79] is as follows.

   Instance: Set Y of tasks, each having length $l(y) = 1$, m processors, a partial order $<$ on Y, and a deadline D.

   Question: Is there an m-processor schedule s for Y that meets the overall deadline D and obeys the precedence constraints, i.e., such that $y1 < y2$ implies $s(y2) \geq s(y1) + l(y1) = s(y1) + 1$?

   $\overline{P1}$ can be restricted to *PCS,* by setting the execution times in hardware and software to be equal (i.e., the tasks take the same execution time on all processors — hardware and software in this case).

   The precedence constrained scheduling problem is NP-complete

([Garey79], pg. 239) for two processors and task lengths not all equal to 1[2]. Thus, a known NP-complete problem is a special case of $\overline{P1}$, and hence, by restriction ([Garey79], pg 63), $\overline{P1}$ is NP-complete.

Obviously, *P1*, the optimization version of $\overline{P1}$ (find a feasible mapping and schedule such that the hardware area is minimum), is at least as hard as the decision version[3]. Thus, *P1* is NP-hard.

A3 ILP formulation for Problem *P2*

The extended partitioning problem can be formulated as an integer linear problem. The formulation is similar to that of the binary partitioning problem.

<u>Given</u>:

1. A set of nodes $N = \{1, 2, \ldots,\}$, representing the tasks in the algorithm.

2. Associated with each node $i \in N$, are non-negative integers, ah_{ij}, th_{ij}, where $j = 1, \ldots, |NH_i|$, and as_{ij}, ts_{ij}, where $j = 1, \ldots, |NS_i|$, representing the hardware and software implementation curves.

3. A directed acyclic graph $G = (N,A)$. A is a set of arcs (i,j), where $i, j \in N$. An arc (i,j) in A implies that task i must complete its execution before task j can start its own execution.

4. Associated with each arc $(i,j) \in A$ is a non-negative number p_{ij}, which represents the number of samples sent from node i to node j.

5. The source and sink nodes in the graph G are identified as S and K respectively.

---

2. For the class of heterogeneous task-level graphs that we are interested in, the execution times are obviously greater than 1.
3. *P1* is not in NP, since a given solution for *P1* cannot be verified in polynomial time to be the minimum solution.

6. Six non-negative integer valued parameters AH, AS, D, ah_{comm}, as_{comm}, and t_{comm} are specified. AH and AS are the capacities of the hardware and the software resource respectively. The total area of all nodes mapped to hardware (software) cannot exceed AH (AS). D is the deadline constraint, i.e., the time between the start time of the source S and the finish time of the sink K should be at most D. Parameters ah_{comm}, as_{comm}, and t_{comm} represent the hardware area required, the software area required, and the delay incurred when one sample of data is sent between two nodes mapped to different resources.

Variables:

1. For each i in N, define binary variables h_i and s_i such that $h_i = 1$ if task i is mapped to hardware, 0 otherwise; and $s_i = 1$ if task i is mapped to software, 0 otherwise. Obviously, $h_i + s_i = 1$. We use the two variables to make the formulation easier to read; only h_i contributes to the variable count. (N variables)

2. In addition to the variables indicating the mapping, additional variables for selecting the optimal implementation bin are needed. Define a set of binary variables bh_{ij} (and bs_{ij}), where $i \in N$, and $j \in NH_i$ (and $j \in NS_i$). In particular, $bh_{ij} = 1$, if node i is mapped to hardware implemented bin j. In general, let B be the number of implementation bins per node per mapping. ($2BN$ variables)

3. For each $i \in N$, define a non-negative integer variable t_i, the starting time of task i. (N variables)

4. For each $(i,j) \in A$, define a binary variable b_{ij} such that $b_{ij} = 1$ if both i and j are in hardware, 0 otherwise. (A variables)

5. For each $(i,j) \in A$, define a binary variable c_{ij}, such that $c_{ij} = 1$ if nodes i and j are on different mappings, and 0 if on the same mapping. c_{ij} will be defined in terms of b_{ij} and is not included in the variable count.

6. For each $i, j \in N$, define a binary variable d_{ij} such that $d_{ij} = 1$ if $t_i \leq t_j$, otherwise $d_{ij} = 0$.
(N(N-1) variables)

Objective function F: minimize hardware area.

$$\text{minimize}\left(\sum_{i \in N} \sum_{j \in NH_i} bh_{ij} \times ah_{ij} + \sum_{(i,j) \in A} c_{ij} \times p_{ij} \times ah_{comm}\right)$$

Subject to:

1. To ensure that only one implementation bin is selected for every node:

$$\sum_{j \in NH_i} bh_{ij} + \sum_{j \in NS_i} bs_{ij} = 1\,.$$ (N constraints)

Define $h_i = \sum_{j \in NH_i} bh_{ij}$, $s_i = \sum_{j \in NS_i} bs_{ij}$. where h_i (s_i) is 1 if i is in hardware (software).

2. Hardware capacity constraint:

$$\sum_{i \in N} \sum_{j \in NH_i} bh_{ij} \times ah_{ij} + \sum_{(i,j) \in A} c_{ij} \times p_{ij} \times ah_{comm} \leq AH$$

(1 constraint)

3. Software capacity constraint:

$$\sum_{i \in N} \sum_{j \in NS_i} bs_{ij} \times as_{ij} + \sum_{(i,j) \in A} c_{ij} \times p_{ij} \times as_{comm} \leq AS$$

(1 constraint)

4. Deadline constraint:

$$t_K + \sum_{j \in NH_K} (bh_{Kj} \times th_{Kj}) + \sum_{j \in NH_K} (bs_{Kj} \times ts_{Kj}) \le D$$

(1 constraint)

5. Precedence constraints:

$$t_i + \sum_{k \in NH_i} (bh_{ik} \times th_{ik}) + \sum_{k \in NH_i} (bs_{ik} \times ts_{ik}) + (c_{ij} \times p_{ij} \times t_{comm}) \le t_j$$

where $(i,j) \in A$ (A constraints)

$d_{ij} = 1$ for $(i,j) \in A$ (A constraints)

6. Processor overlap constraint:

No two jobs can be executed simultaneously on the software processor. We want to express the fact that if i and j are both on the software processor, and j executes before i (i.e., $(j, i) \in A$), then t_j + execution time of j $\le t_i$.

The variable d_{ij} is 1 if $t_i \le t_j$, and 0 otherwise. Define $M = \sum_i ts_i$. The meaning of d_{ij} is expressed by the following constraints:

$(d_{ij} + d_{ji}) = 1$ for $i < j$, $i, j = 1, \ldots, N$. (N(N-1)/2 constraints)

$t_j - t_i - (M \cdot d_{ij}) \le 0$ for $i \ne j$, $i, j = 1, \ldots, N$ (N(N-1) constraints)

Since $t_j - t_i < M$ for an optimal solution, the expression is restrictive only for $d_{ij} = 0$. If $d_{ij} = 0$, then j starts no later than i. The overlap constraint can now be specified as:

$$t_j + time_h + time_s + \sum_k c_{jk} \times p_{jk} \times t_{comm} - t_i \le M \cdot (3 - d_{ji} - s_i - s_j),$$

where, $time_h = \sum_{k \in NH_j} (bh_{jk} \times th_{jk})$ $time_s = \sum_{k \in NH_j} (bs_{jk} \times ts_{jk})$,

$$s_i = \sum_{k \in NS_i} bs_{ik}, \text{ and } s_j = \sum_{k \in NS_j} bs_{jk}.$$

for $i \neq j$, $i,j = 1, \ldots, N$, k such that $(j,k) \in A$ (N(N-1) constraints)

The expression is restrictive only for $d_{ji} = 1$ and $s_i = s_j = 1$, i.e., i and j are both on software, and j starts no later than i. In this case it guarantees that t_j + execution time of i $\leq t_i$, as it should be.

7. Additional constraints:

To account for costs incurred when two communicating nodes i and j are on different mappings, we use c_{ij}. c_{ij} is expressed as:

$c_{ij} = 1 - (h_i \times h_j) - ((1 - h_i) \times (1 - h_j))$, where $(i,j) \in A$

$c_{ij} = h_i + h_j - 2h_ih_j$

Note that this is a quadratic constraint in h_i, and it can be linearized by defining a binary variable b_{ij} for all (i,j) in A. such that $b_{ij} = h_ih_j$. This can be expressed by:

$b_{ij} \leq h_i$

$b_{ij} \leq h_j$

$b_{ij} \geq (h_i + h_j - 1)$

The original expression for c_{ij} can now be replaced by:

$c_{ij} = h_i + h_j - 2b_{ij}$

Note that only b_{ij} adds to the variable count. c_{ij} is used only to enhance readability; in the actual formulation c_{ij} is replaced by $c_{ij} = h_i + h_j - 2b_{ij}$. The three expressions defining b_{ij} add 3A constraints. (3A constraints)

A solution to the extended partitioning problem is specified completely by h_i (mapping), bh_{ij} or bs_{ij} (implementation bin depending on the mapping selected), and t_i (schedule). The formulation has $(N^2 + N \cdot (2B + 1) + A)$ variables and

$(\frac{5}{2} \cdot N^2 - \frac{3}{2} \cdot N + 5A + 3)$ constraints, in addition to the integrality constraints on the variables.

A4 Complexity Analysis of the GCLP Algorithm

The *run-time* complexity for the GCLP algorithm is computed. Estimations of area and time, and computations of the extremity and repeller measures are outside of the main loop and are not included in the complexity calculation. The complexity of each sub-step of the GCLP algorithm (described in Section 3.3) is shown below:

Complexity(GCLP)

S1. Compute GC: $O(|A| + |N|)$

S1.1.Estimate the set of nodes to move to hardware: $O(1)$

S1.2.Compute the actual finish time: $O(|A| + |N|)$

S2. Determine the set of *ready* nodes: $O(|N|)$

S3. Compute the effective execution time: $O(|N|)$

S4. Compute the longest paths: $O(|A|)$

S5. Select a *ready* node with maximum longest path: $O(|N|)$

S6. Determine the mapping and schedule: $O(1)$

The GC computation in S1 involves the estimation of a mapping, followed by the computation of the actual finish time for this mapping. The estimation in S1.1 is done by selecting nodes from an ordered list and is done in constant time. The actual finish time is computed in S1.2 by ignoring communication costs, using the following procedure:

Procedure: **Compute_actualFinishTime**

Input: DAG with predetermined mapping and corresponding execution time ($t_{exec}(i)$) for each node i

Output: T_{finish} = finish time of the DAG

Initialization: $dsp\_Finish\_Time = 0$, $T_{finish} = 0$:

S1. Label all nodes with their indegree

S2. Mark all nodes with 0 indegree as *ready* nodes

S3. while(*ready* nodes exist){

S3.1. Select ready node i: O(1)

S3.2. Find $t_{start}(i)$: O(1)

S3.3. if(i in software) $t_{start}(i) = \max(\max_j(t_{avail}(j)), dsp\_Finish\_Time)$

(j: inputs of node i)

if(i in hardware) $t_{start}(i) = \max_j(t_{avail}(j))$

S3.4. Update $t_{finish}(i) = t_{start}(i) + t_{exec}(i)$: O(1)

S3.5. If i is in software:

$dsp\_Finish\_Time = (dsp\_Finish\_Time > t_{finish}(i))$?

$dsp\_Finish\_Time : t_{finish}(i)$

S3.6. For each output k of node i set $t_{avail}(k) = t_{finish}(i)$

S3.7. For each output k of node i, access node p connected to it

S3.7.1. Decrease indegree of node p

S3.7.2. If indegree of p is 0, add p to list of *ready* nodes

S3.8. $T_{finish} = (T_{finish} > t_{finish}(i))$? $T_{finish} : t_{finish}(i)$

} : (O($|A| + |N|$)

Since each node and arc is traversed once, the complexity of the algorithm is O($|N| + |A|$). The longest path calculation (S4 of GCLP) is described with the help of the following procedure. Note that as the graph is acyclic, a label setting algorithm can be used.

Procedure: **Compute_longestPath**

Input: $G = (N,A)$, $t_{eff}(i)$ for $i \in N$

Output: longest path $d(i)$ for $i \in N$ ($d(i)$ is the longest path from i to sink)

Initialization: counter $c = 0$, $d(i) = 0$ for $i \in N$

S1. Reverse the directions of all the arcs in the graph: (O($|A|$))

S2. Topologically order the graph (an ordering where arc (i,j) means node i is a predecessor of node j)

S2.1. Label all nodes with their indegree: $(O(|N|))$

S2.2. while $(c < |N|)$ {

S2.2.1. Identify a node i with indegree = 0.

S2.2.2. Label i with c.

S2.2.3. Decrease indegree of all nodes connected to i.

S2.2.4. Delete node i and all arcs emanating from i.

S2.2.5. $c = c+1$

} : $(O(|A|))$

S3. Negate effective execution times for all $i \in N$. $(t_{eff}(i) = (-1) \cdot t_{eff}(i))$: $(O(|N|))$

S4. Traverse graph in topological order: $(O(|A|))$

S4.1. $d(i) = \min(d(j)) + t_{eff}(i)$, where $j \in$ {predecessors of i}

S4.2. Negate $d(i)$ to give longest path for each i, $i \in N$: $(O(|N|))$

Note that the longest path is computed as the shortest path on the graph with negated execution times. The complexity of the algorithm is $O(|A|)$, since each edge is visited once.

The actual mapping and schedule (S6 of GCLP) are computed in constant time. Thus, the total complexity for one step of the GCLP algorithm is $O(|N| + |A|)$. The GCLP algorithm runs $|N|$ times, hence total complexity is $O(|N| \cdot (|N| + |A|))$. Typically, for DSP applications, $|A| \approx |N|$, hence the worst case complexity of the GCLP algorithm is $O(|N|^2)$.

A5 Complexity Analysis of the Bin Selection Procedure

The bin selection procedure is applied on a tagged node to select its implementation bin. In general, let B be the number of implementation bins for each node. The complexity of the bin selection procedure is computed as follows:

Complexity(bin selection)

S1. Compute BFC: $O(B \times (|N| + |A|))$

S1.1.For each bin, compute the bin fraction: $O(|N| + |A|)$

S1.1.1.Estimate the nodes to move to L bins

S1.1.2.Compute actual finish time assuming this mapping and bin selection

S2. Compute bin sensitivity: $O(B)$

S3. Compute weighted bin sensitivity: $O(B)$

S4. Select bin: $O(B)$

The bin fraction computation in S1.1 is similar to that used in the computation of *GC*. Hence, the complexity for computation of the bin fraction for a particular bin is $O(|N| + |A|)$ (see Section 4). The complexity of S1 is hence $O(B \cdot (|N| + |A|))$ for B implementation bins. The complexity of other steps is simply in the order of the number of bins B. The complexity of the bin selection procedure hence $O(B \cdot (|N| + |A|))$.

A6 Complexity Analysis of the MIBS Algorithm

The MIBS algorithm applies the GCLP-bin selection sequence $|N|$ times, for all the nodes in the graph. The complexity for each sub-step in the MIBS algorithm is computed as follows:

Complexity(MIBS)

S1. Determine the mapping for all free nodes by running GCLP: $O(|N|^2)$

S2. Determine ready nodes: $O(|N|)$

S3. Select tagged node: $O(|N|)$

S4. Compute implementation bin for tagged node using the bin selection procedure: $O(B \cdot (|N| + |A|))$

The complexity of one step of the MIBS algorithm is $O(|N|^2 + B \cdot |N|)$. Each step is repeated $|N|$ times for the $|N|$ nodes, hence the complexity of the MIBS algorithm is $O(|N|^3 + B \cdot |N|^2)$.

A7 Algorithm for Random Graph Generation

To get an estimate on the behavior of an algorithm in *practice*, empirical studies can be performed. This involves devising a set of supposedly "typical" instances, running the algorithm and its competitors on them, and comparing the results. Of course, the usefulness of this technique depends on how "typical" the sample instances are. Ideally we would like to run our algorithm on practical examples. However, due to the absence of system-level benchmarks, and the lack of time to generate a significant number of examples ourselves, we resort to generating random examples. The following procedure presents the method used to generate random DAGs.

A7.1 Random DAG Generation

Procedure: **Generate_Random_Graph**
Inputs: size of graph N
Output: Directed, acyclic graph, without parallel edges.

S1. $A = random\_int(N, N^2)$[4] (number of arcs)

S2. Generate a random permutation of 1, …, N in the array "*perm*"
    S2.1. for(i=0; i<N; i++) {$perm[i] = i$;}

4. The random number generator $random\_int(l,u)$ is used to generate random integers within the range (l,u) and is adapted from [Press88].

S2.2. for(i=0; i<N; i++) {

S2.2.1. j = *random_int*(0,N-1);

S2.2.2. *tmp* = *perm*[i];

S2.2.3. *perm*[i] = *perm*[j];

S2.2.4. *perm*[j] = *tmp*;}

S3. Generate "A" random edges of the form (*perm*[i], *perm*[j]), for $i < j$

S3.1. for(i=0; i<A; i++) {

S3.2. *src* = *random_int*(0, N-2);

S3.3. *dest* = *random_int*(*src*+1, N-1);

S3.4. if(!*existsArc*(*src*, *dest*)) *add_arc*(*src*, *dest*)}

S4. Generate nodal information for each node i: $size_i$, ts_i, th_i, as_i, ah_i.

S4.1. Determine if node i is an extremity, repeller, or normal node

S4.1.1. y = *ran1*()[5]

S4.1.2. if((y >= 0) && (y < 0.33)) i: extremity

S4.1.3. if((y >= 0.33) && (y < 0.66)) i: repeller

S4.1.4. if((y >= 0.66) && (y <= 1)) i: normal

S4.2. Set area and time values based on the nature of node i

S5. Generate constraints: T, AH, AS.

S5.1. T = *random_int*(*sum_th*, *sum_ts*)

S5.2. AH = *random_int*(*ah_low*, *ah_high*)

S5.3. AS = *random_int*(*as_low*, *as_high*)

The number of arcs is first determined in S1. To generate a directed acyclic graph [Johnsonbaugh91] with N nodes, a random permutation perm[0], …, *perm*[N-1] is generated in S2. This random permutation serves as a topological ordering for the graph. In S3, a set of random edges is generated of the form (*perm*[i], *perm*[j]) with $i < j$. This generates a directed acyclic graph.

A7.2 Generation of Hardware-Software Area-Time Values

The nodal information is next generated in S4. Traditionally, random graphs have been used for testing out scheduling heuristics, where the only param-

5. *ran1*() uses a multiplicative congruential method to generate a uniformly distributed random number in the range (0,1) [Press88].

eter associated with a node is its execution time. In that case, the execution time is generated from a uniform distribution (see for example, [Sih91]). In our specific problem, estimates for *area and time* on *hardware and software* are required. It is obvious that there is a correlation between the area and time values for a particular realization. In addition, there is a correlation between the values across hardware and software. Hence generating independent random values for all the four quantities will not model a realistic example. To overcome this problem, we have devised a methodology for generating these values, based on practical observations and realistic correlations. First, to model the nodal heterogeneity, a probabilistic measure is used (S4.1) to mark a node as an extremity node, repeller node, or normal node. Secondly, the area and time values for the node are then set depending on the nature of the node (S4.2). The following procedure illustrates the general methodology for generating the area and time estimates for an extremity node.

Procedure: **Generate_Extremity_Node**
Input: ts_{min}, as_{min}, th_{min}, ah_{min}, ts_{max}, as_{max}, th_{max}, ah_{max}, cutoff threshold
Output: ts, th, ah, as

S1. Compute threshold values ts_{th}, as_{th}, th_{th}, ah_{th} (for ex: $ts_{th} = ts_{min}$ + cutoff * $(ts_{max} - ts_{min})$)

S2. Compute maximum and minimum extremity values

S2.1. $xs_{max} = ts_{max} / ah_{min}$

S2.2. $xs_{min} = ts_{th} / ah_{th}$

S2.3. $xh_{max} = ah_{max} / ts_{min}$

S2.4. $xh_{min} = ah_{th} / ts_{th}$

S2.5. $delta_h = xh_{max} - xh_{min}$

S2.6. $delta_s = xs_{max} - xs_{min}$

S3. Determine if the node is a hardware or software extremity

S3.1. $z = ran1()$

S3.2. if($z < 0.5$) hardware_extremity; else software_extremity;

S4. Generate an extremity measure in the range (0, 0.5) ($ex = ran(0, 0.5)$)

S5. If hardware extremity:

S5.1. $ah = random\_int(ah_{th}, ah_{max})$

S5.2. $ts = ah / ((2 * delta_h * ex) + xh_{min})$

S5.3. $as = random\_int(as_{min}, as_{th})$

S5.4. $th = random\_int(th_{min}, th_{th})$

S6. If software extremity:

S6.1. $ts = random\_int(ts_{th}, ts_{max})$

S6.2. $ah = ts / ((2 * delta_s * ex) + xs_{min})$

S6.3. as = random_int(as_{min}, as_{th})

S6.4. $th = random\_int(th_{min}, th_{th})$

The mechanism for the calculation of the area and time values for extremities *reverse engineers* the mechanism used for extremity calculation described in Section 3.3.3.2. The range of area and time values in hardware and software is specified by the input parameters. In S1, the cut-off threshold is used to compute the threshold values of area and time in hardware and software, similar to that described in Section 3.3.3.2 for extremity identification. The maximum and minimum extremity values are computed in S2, using the definition of extremities. In S3, the node is probabilistically set to be either a hardware or a software extremity, and in S4 its extremity measure is generated as a random number in the range (0, 0.5). The area and time values are next generated in S5 and S6. If a node is a hardware extremity, its *ah* is set to a random number between (ah_{th}, ah_{max}), and its extremity measure is then used set the *ts* value. The *th* and *as* are set to random numbers in the ranges (th_{min}, th_{th}) and (as_{min}, as_{th}) respectively. The area and time values for a software extremity are similarly generated, as shown in S6.

The procedure for generating area and time estimates for repellers is as follows:

Procedure: **Generate_Repeller_Node**
Output: *ts*, *th*, *as*, *ah*

S1. Generate *TS* and *TH* array values

S2. Determine if the node is a hardware or software repeller

  S2.1. $z = ran1()$

  S2.2. if($z < 0.5$) hardware_repeller; else software_repeller;

S3. Generate a repeller measure in the range (0, 0.5) ($r = ran(0, 0.5)$)

S4. If software repeller:

  S4.1. $ts = TS[random\_int(0, 5)]$

  S4.2. $as = ts / 2$

  S4.3. $ah = (1 - r) * as$

  S4.4. $th = (1 + ran1()) * ah$

S5. If hardware repeller:

  S5.1. $th = TH[random\_int(0, 5)]$

  S5.2. $ah = th / 2$

  S5.3. $as = (1 - r) * ah$

  S5.4. $ts = (1 + ran1()) * th$

Recall that repellers identify *relative* preferences, i.e., give two *similar* software nodes, the software repeller measure is the relative area gain in hardware. To take the similarity of nodes into account, in S1, a set of possible *ts* and *th* values is generated. The *ts* (*th*) values for software (hardware) repellers are then selected from this set *TS* (*TH*) so as to ensure that *similar* nodes are considered. In S2, a node is probabilistically assigned to be either a hardware or a software repeller, and in S3 its repeller measure is generated. If the node is a software repeller, its *ts* is selected to be one of the set of possible values from the set *TS*. The *as* is related to this *ts*. The value of *ah* is then generated from the *as* value using the repeller measure. The area and time values for hardware repellers are similarly generated in S5.

Procedure: **Generate_Normal_Node**
Output: *ts*, *th*, *ah*, *as*

S1. $ts = random\_int(ts_{min}, ts_{th})$

S2. $as = ts / 2$

S3. $th = ts$ / $random\_int(1, 4)$
S4. $ah = th / 2$

For a normal node, *ts* is generated as a random number between ts_{min} and ts_{th}. The value of *th* is a fraction of the software time; this fraction is a probabilistic number from 1 to 4. The areas are computed as a fraction of the times so as to correlate the area and time values on a given mapping.

Finally, in S5 of the graph generation algorithm, the constraints are generated. The deadline is set to a random value between the sum of hardware and software execution times. The hardware capacity is set to a random number between ah_{low} and ah_{high}, where ah_{low} and ah_{high} are fractions of the sum of the hardware sizes of all the nodes. The software capacity constraint is similarly generated.

A7.3 Implementation Curve Generation

To generate the random graphs for the extended partitioning problem, hardware and software implementation curves are required for each node. A *generic* implementation curve was obtained from practical observations of a number of implementation curves. This curve gives the area and time values for various bins, normalized with respect to the L bin values. The generic curve is of the form (tf^k, af^k), where $tf^k > 1$, and $af^k < 1$, for the *k* bins.

The area and time values for a single implementation are generated for a node, as discussed in Section 7.2. The generated values of *ah* and *th* are assumed to correspond to the *L* bin (ah^L, th^L). The generic implementation curve is then used to compute the area and time values for other bins. The generated curve is of the form (th^k, ah^k), where $th^k = tf^k * th^L$, $ah^k = af^k * ah^L$.

A8 Estimation of Area and Execution Time Parameters

The partitioning process relies heavily on the estimates of the area and execution time on different hardware and software implementations. The usefulness of the generated partition depends on how accurate the estimates are; if the actual values obtained after synthesis are very different from the estimates, then the resultant solution could be either infeasible or have an unnecessarily large hardware area. Accuracy of the estimated area and time values is directly proportional to the time spent on generating the estimates; the more accurate the estimate, the longer it takes to derive it. In the limiting case, the best estimate is obtained after actually synthesizing the implementation. Our philosophy in generating the estimates was to get fairly accurate estimates from behavioral specifications, without going through the entire synthesis process.

In Section A8.1, some of the related techniques in hardware estimation are first briefly mentioned, followed by a discussion of the estimation mechanism that we use. In Section A8.2, the software estimation techniques are discussed.

A8.1 Hardware Estimation

The hardware area required to implement a node consists of the active area of the datapath[6] (i.e., execution units, registers, multiplexors, and buffers) as well as the interconnect area. Several methods to estimate the active area have been proposed [Chaudhari94][Kurdahi89][Rabaey94]. Since the interconnect area is usually a significant fraction of the active area, considering only the active area is inaccurate, the interconnect area also needs to be estimated. BUD [McFarland90b], developed by McFarland et al., was one of the first highlevel syn-

---

6. We assume a standard-cell type hardware architecture; the extension to FPGAs is quite straightforward.

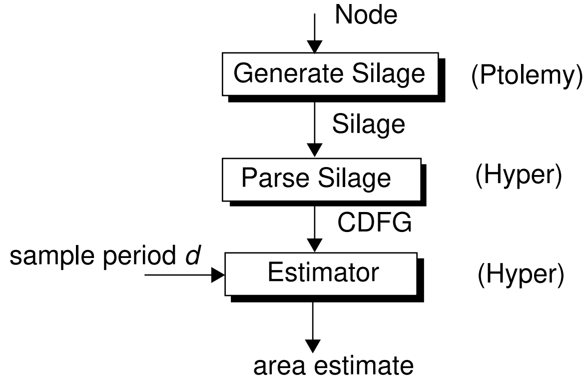

Figure A.1. Methodology for the estimation of hardware area required for a node when implemented at a sample period *d*.

thesis tools that took the interconnect area into account. The approach, however, is based on doing an actual floorplanning of the generated modules to get an estimate of the total area of the design. Zimmerman [Zimmerman88] describes slicing-structure based mechanisms for estimating the floorplan areas. In the approach proposed by Sharma et al. [Sharma93], the number of busses required in the design is first estimated. This number is correlated to the interconnect area to estimate the total area.

Mehra et al. [Mehra94] propose a statistical approach for computing the total chip area (active plus interconnect) for a given sample period d. In this scheme, the minimum bounds on the active area (A_{act}) and the number and size of buses (N_{bus}, N_{bits}) are first estimated using techniques proposed by Rabaey *et al.* [Rabaey94]. The total area is then given by a statistical model that uses these values: $A_{total} = \alpha 1 + \alpha 2 \cdot A_{act} + \alpha 3 \cdot N_{bus} \cdot N_{bits} \cdot A_{act}$, where the parameters $\alpha 1$, $\alpha 2$, and $\alpha 3$ have been determined by experimental analysis. These techniques have been implemented in the Hyper highlevel synthesis system [Rabaey91]. We use the estimation techniques implemented in Hyper to generate the hardware area estimates.

The overall methodology for obtaining the hardware area estimate for a node, for a given sample period d, is summarized in Figure 8.1. Silage

[Hilfinger85] code is first generated for the node, using the approach described in Chapter 4. This Silage code is then parsed to get a control-dataflow graph (CDFG) [Verbauwhede94]. The estimator, using the estimation techniques implemented in Hyper, operates on this CDFG to give the area estimate for this particular sample period.

A8.2 Software Estimation

Software performance estimation is usually a very difficult problem. Compiler optimizations and operating system interaction make the area and time requirements difficult to predict. Tools such as Quantify [Quantify] can be used to get accurate estimates of the actual time spent in running the code. However, since this requires running the software, code has to be generated, compiled, and run for each node; the estimation times may be unacceptable. Besides, this gives only a single trace, making it hard to estimate execution times when there is data dependency within a node. Other approaches for estimation have also been reported [Gong94] [Tiwari94].

In our approach, the Motorola DSP 56000 is used as the target software processor. Since the software model assumes no operating system, the estimates generated by static profiling of the generated assembly code are fairly accurate. Assembly code is generated for each node. A simple C program parses the generated assembly code to compute the estimates of the software size (program and data memory required) and software execution time.

Herrmann et al. [Herrmann94] describe techniques to move the estimation process into the partitioning loop in order to improve the quality of the solution in the cases where the estimates are inaccurate.

A9 Automated Flow Execution in the DMM Domain

In this section, the details of the flow execution mechanism are provided. Some of the terms used in this section are first defined in A9.1. The scheduling mechanisms used in the "*Run All*", "*Run Upto*", and "*Run This*" modes are described in A9.2, A9.3, and A9.4 respectively.

A9.1 Preliminaries

We define some of the terms used in the rest of this section:

$G_{flow}(N,A)$	The design flow specified as a graph $G_{flow}(N,A)$, where N is the set of nodes representing tools and A is the set of arcs representing the connectivity between the tools.
<u>Required Input Port</u>	The tool cannot run unless data is present on a required input port.
<u>Optional Input Port</u>	The presence of data on an optional input port is not essential for execution of the tool. The behavior of the tool could depend on whether or not data is present on an optional input port.
<u>Required Output Port</u>	Data is always generated on a required output port when the tool executes.
<u>Optional Output Port</u>	Data may or maynot be generated on an optional output port.
<u>Path</u>	A path from tool i to tool j is a sequence of arcs from tool i to tool j.
<u>Predecessor</u>	$i \in P(j)$ (i is a predecessor of j) if there is a path from i to j.
<u>Ordered predecessor</u>	$i \in OP(j)$ (i is an ordered predecessor of j) if there is atleast one path from i to j that traverses all *required* input ports. That is, $i \in OP(j)$ implies that j cannot run without i.
<u>Source</u>	A tool with no inputs.
<u>Nondeterminacy Condition</u>	The generated data is possibly not independent of the sequence used to invoke the tools.

A9.2 Run All mode

In this mode, the graph is traversed, starting with the source tool, executing tools as needed. The following algorithm describes the mechanism.

```
runAll(G_flow(N,A)) {
    if (number_of_sources > 1)
        Error("Multiple sources not allowed");
    for each tool i in N {
        compute_predecessors(i);            /* set P(i) */
        compute_ordered_predecessors(i);    /* set OP(i) */
    }
    /* check for possible nondeterminacy condition */
    for each tool i in N {
        for each tool j in P(i) {
            if (i ∈ P(j)) {
                if((i ∈ OP(j)) && (j ∈ OP(i))
                    Error("Deadlock");              /* Fig. 8.2-a*/
                if((i ∉ OP(j)) && (j ∉ OP(i))
                    Error("Possible Nondeterminacy");
                                                    /* Fig. 8.2-b*/
            }
            else if (i ∉ P(j)) {
                if(j ∉ OP(i))
                    Error("Possible Nondeterminacy");
                                                    /* Fig. 8.2-c*/
            }

        }
    }
    if (nondeterminacy)
        Query("continue", "run tool by tool", "abort");
    enabledToolsList.add(source);
    for each tool i
        initialize(i);    /* retrieve last_file_name and last_time_stamp
                          from Oct. Set param_changed_flag if parameters
                          were changed */
    run_enabled_tools();  /* the main procedure */
}
```

As discussed in Section 5.3, to avoid possible nondeterminacy, design flows with multiple sources are not supported for automated scheduling. The pre-

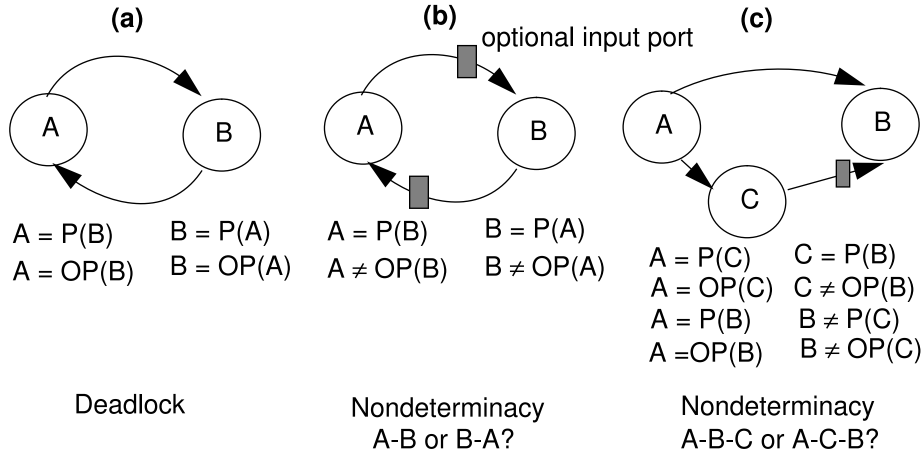

Figure A.2. Illustration of possible nondeterminacy.

decessors and ordered predecessors for all the nodes are first computed. The algorithms for computing the predecessors and ordered predecessors of a node are outlined below and are self-explanatory. The predecessors and ordered predecessors are used to detect possible nondeterminacy and deadlock. Figure 8.2 illustrates some of the cases implying possible nondeterminacy. If the scheduler detects possible nondeterminacy, the user has the option to either abort the run, or control the flow execution by running a tool at a time. If the user has designed the tools so as to avoid nondeterminate behavior, the user can ask the system to run anyway and automated scheduling is used (procedure *run_enabled_tools*()).

Before calling the procedure *run_enabled_tools*(), the tools are first initialized. The filename and timestamp attributes for each port are loaded in from the Oct database. If the tool has never run before, the last filename attribute is initialized to "DUMMY". The system also checks if a parameter has changed since the previous invocation of this tool. The main routine for running the tools is described in the procedure *run_enabled_tools*().

---

compute_predecessors(*i*) {
    for all tools { *clear_tag*(); *clean_mark_on_all_ports*(); }
    tag_predecessors(*i*);

```
    for each tool j in N
        if(is_tagged(j)) add(P(i),j);
}
```

```
tag_predecessors(i) {
    /* recursively tag predecessors */
    tag(i);
    for each input port p of i {
        if(!is_marked(p)) {
            mark(p);
            j = connected_tool(p);
            tag_predecessors(j);
        }
    }
}
```

```
compute_ordered_predecessors(i) {
    for all tools { clear_tag(); clear_mark_on_all_ports(); }
    tag_required_predecessors(i);
    for each tool j in N
        if(is_tagged(j)) add(OP(i),j);
}
```

```
tag_required_predecessors(i) {
    /* recursively tag required predecessors, i.e., predecessors connected to
required input ports*/
    tag(i);
    for each required input port p of i {
        if( !is_marked(p)) {
            mark(p);
            j = connected_tool(p);
            tag_required_predecessors(j);
        }
    }
}
```

The following procedure shows the mechanism to run enabled tools. A tool *i* is first removed from the head of the list of enabled tools. If there is a tool *j* on the list of enabled tools that is a predecessor to *i*, then *i* is not run, but simply moved to a slot after *j*. This tries to enforce an ordering where a predecessor is always run first. This rule will enforce the A-C-B ordering for the graph shown in Figure 8.2-c. If there is no predecessor on the list of enabled tools, the selected tool is examined for live dependencies and executed if needed.

```
run_enabled_tools() {
        while(enabledToolsList != empty) {
                i = enabledToolsList.head();
                nextTool = 0;
                for each j in enabledToolsList {
                        if((j ∈ P(i)) && (i ∉ P(j)) {
                                enabledToolsList.move(i,j); /* move i to after j */
                                /* try to force a behavior by running j, which is a
                                 predecessor  to i, before running i. If there is a loop
                                 between i and j, then we just run them in the order
                                 on the list; we have already flagged to the user that
                                 this could be a nondeterminate case. */
                                nextTool = 1;
                                break;
                        }
                }
                if(!nextTool) {
                        examine_dependency_and_run_if_needed(i);
                        enabledToolsList.remove(i);
                }
        }
}
```

The following procedure examines a tool for live dependencies. If a dependency is alive, the tool is executed, the generated data is passed on to its descendents, and the descendents are enabled, otherwise the old data is propogated to its descendents and the descendents are enabled.

```
examine_dependency_and_run_if_needed(i) {
        if(dependency_alive(i)) {
                execute_tool(i);
                send_new_output_data(i);
```

```
            enable_descendents(i);
            update_status(i);
      }
      else {
            send_old_output_data(i);
            enable_descendents(i);
      }
}
```

The following procedure checks for live dependencies. A parametric dependency is alive if *param_changed_flag* is set. A source needs to run if it has never run before, i.e., a required output port has attribute "DUMMY".

In general, after a tool finishes execution, the newly generated file name is copied over to the descendent ports. If a tool does not generate data on a certain output (because it is optional), it sends an explicit "DUMMY" to the descendent port. As a result, for non-source tools, a new file name of "DUMMY" on an input port indicates that no new data has been propogated by the predecessor tool to this port. This is the reason for the first check comparing the new filename to DUMMY. If this check holds, then the data and temporal dependencies are evaluated.

```
dependency_alive(i) {
      if(param_changed(i))
            return 1; /* parametric dependency */
      if(is_source(i)) {
            for each required output port p
                  if(strcmp(last_file_name(p),"DUMMY")==0)
                        return 1;
      }
      else {
            for each input port p of i {
                  if(strcmp(new_file_name(p),"DUMMY")!=0) {
                        if(strcmp(new_file_name(p),last_file_name(p)) != 0)
                              return 1; /* data dependency */
                        if(last_time_stamp(p) < new_time_stamp(p))
                              return 1; /* temporal dependency */
                  }
```

```
        }
    }
}
```

The following procedure calls the user-defined code associated with a tool. This operates on the new filenames associated with the input ports. A *data_changed* flag gets set for all the ports for which new data is generated during this execution. The generated data is available in the new filename attribute.

```
execute_tool(i) {
    execute the user-defined code associated with the tool;
        for each output port p of i
        if (data is generated on p)
            set_data_changed_flag(p);
}
```

After a tool finishes execution, the next step in the procedure *examine_dependency_and_run_if_needed* is to send the generated data to the connected ports, as shown in the following procedure *send_new_output_data*. For required output ports, the corresponding descendent port is *marked* (as shown shortly, this mark is used to identify whether a tool is enabled), while for an optional output port *p*, the corresponding descendent port is marked only if *p* generated new data. If the optional output port did not generate data, an explicit "DUMMY" is sent to the descendent port.

```
send_new_output_data(i) {
    /* If new data is generated, send it to the connected input port, otherwise
     send explicit DUMMY.  Mark far port only if new data is sent. */
    for each output port p of i {
        farPort = input port at the other end of the arc originating from p;
        if(has_data_changed(p)) {
            set_new_file_name(farPort, new_file_name(p));
            mark(farPort);
        }
        else {
        /* p is an optional port that did not generate data in this iteration */
            set_new_file_name(farPort, "DUMMY");
```

```
                    set_new_file_name(p, "DUMMY");
                    clear_mark(farPort);
               }
          }
}
```

The following procedure examines all the descendents of a tool to check if they are enabled. A tool is enabled only if all its required input ports are marked.

```
enable_descendents(i) {
     for each output port p {
          farPort = input port at the other end of the arc originating from p;
          descendent = tool containing the farPort;
          if(each required input of descendent is marked)
               enabledToolsList.put(descendent);
               /* put only if not already there */
     }
}
```

The following procedure is called to update the filename and timestamp attributes in the Oct database in preparation for the next run.

```
update_status(i) {
     for all ports {
          last_file_name = new_file_name;
          last_time_stamp = new_time_stamp;
     }
     store in Oct database;
}
```

The following procedure is executed by a tool if it has no live dependencies. This procedure propagates the filenames to the descendent ports.

```
send_old_output_data(i) {
     /* Send data generated on the previous invocation.
      Mark only if not DUMMY */
     for each output port p {
          farPort = input port at the other end of the arc originating from p;
          set_new_file_name(farPort, last_file_name(p));
          if(strcmp(last_file_name(p),"DUMMY") != 0)
               mark(farPort);
          else
```

```
                clear_mark(farPort);
        }
}
```

A9.3 Run Upto mode

In this mode, the user selects a certain tool upto which the flow should be executed. All the predecessors of the selected tool are identified and tagged. The design flow is then traversed as in the *Run All* mode; only the tagged predecessors are executed. In the current implementation, we support this mode for acyclic flows only.

```
runUpto(G_flow, i) {
        /* run upto a tool i in flow G_flow */
        if(!is_acyclic(G_flow))
                Error("runUpto supported only on acyclic flows");
        if (number_of_sources > 1)
                Error("Multiple sources not allowed");
        for each tool j in G_flow
                initialize(i);    /* retrieve last_file_name and last_time_stamp from
                                     Oct. Set param_changed_flag if parameters were
                                     changed */
        compute_predecessors(i);
        for each j in P(i)
                tag(j);
        enabledToolsList.add(source);
        while(enabledToolsList != empty) {
                k = enabledToolsList.head();
                if(is_tagged(k)) {
                        examine_dependency_and_run_if_needed();
                }
        }
}
```

A9.4 Run This mode

In this mode, a single user-selected tool is run. The selected tool is run if it has valid input data. The tool operates on the data generated by its predecessors in their latest invocation.

```
runTool(i) {
        /* Run one iteration of a selected tool.  Operates on the last data generated
         by predecessor tools. Doesn't run other tools */
        for each tool i initialize(i);
        /* read in last_file_name and last_time_stamp for tool i from Oct */
        for each input port p of i {
                farPort = output port at the other end of the arc terminating in p;
                /* read in last_file_name and last_time_stamp of ancestor */
                set_new_file_name(p) = last_file_name(farPort);
                set_new_time_stamp(p) = last_time_stamp(farPort);
        }
        for each required input port p of i {
                if(strcmp(new_file_name(p),"DUMMY")==0)
                        Error("Predecessors never run before, can't run i.");
                else
                        mark(p);
        }
        for each optional input port p of i {
                if(strcmp(new_file_name(p),"DUMMY")!=0)
                        mark(p);
                else
                        clear_mark(p);
        }
        if(dependency_alive(i)) {
                execute_tool(i);
                update_status(i);
        }
}
```