Codex-dp: Co-design of Communicating Systems Using Dynamic Programming

Jui-Ming Chang, Massoud Pedram

Abstract— **We present a novel algorithm based on dynamic programming with binning to find, subject to a given deadline, the minimum-cost coarse-grain hardware/software partitioning and mapping of communicating processes in a generalized task graph. The task graph includes computational processes which communicate with each other by means of blocking/nonblocking communication mechanisms at times including, but also other than, the beginning or end of their lifetime. The proposed algorithm has been implemented and experimental results are reported.**

Keywords— **Hardware/Software Co-design, System-Level Partitioning, Coarse-grain Task Graph, Dynamic Programming, Scheduling, Blocking/Non-blocking Communication, Mid-way Communication. Processes.**

I. Introduction

PREVIOUS work on system level synthesis has focused mainly on fine-grain hardware/software partitioning. Examples include VULCAN II [1] and COSYMA [2]. These programs automatically partition the input specification into basic blocks (or fine-grain operations) and move the basic blocks to hardware or software components while satisfying the given constraints. The resulting fine-grain partitioning may, however, move logically coherent blocks across different parts or put logically unrelated blocks in the same part. In addition, the resulting partitioning creates an implementation which is very different from the initial specification, and hence, is not convenient for human designers to debug and/or improve upon.

In contrast, coarse-grain partitioning does not decompose the initial specification into basic blocks; Neither does it assign a process in the initial specification to several processors. It is therefore able to preserve the granularity and modularity of the initial specification. Furthermore, coarse-grain partitioning can exploit the designer expertise more easily and can achieve a partitioning that satisfies macroscopic choices more readily [3]. Finally, the resulting solution has more logical coherence which facilitates the top-down design process and allows for debugging of the hardware/software.

Coarse-grain partitioning algorithms often start from a task graph which consists of a set of communicating processes. In the published literature, task graphs that describe the set of communicating processes (or tasks) (such as the ones shown in [4] [5] [6]) are directed acyclic graphs ($DAGs$) that use nodes to represent processes and arcs to represent precedence relation or communication among the processes. In these task graphs, the communication is assumed to take place from the end of one process (node) to the beginning of another process. We refer to this type of communication as *end/begin* communication. However, the coarse-grain processes may generally communicate with each other at times that are not at the end or beginning of their lifetime. We refer to this type of communication as *mid-way* communication and to the task graph with mid-way communication as a *generalized task graph*. The problem we are trying to solve can then be stated as follows.

Problem I.1: Given a generalized task graph consisting of processes which communicate with each other at arbitrary times by various blocking/nonblocking communication mechanisms and a library containing several possible mappings (or implementations) for each process, simultaneously schedule and map the computational and communication processes to given HW/SW resources so as to minimize the total area cost while satisfying a given deadline.

The hardware components which are available in the library can be classified as computational units and communication units. Both classes can be further divided into programmable or non-programmable. Examples of programmable computational units are CPUs, DSPs and examples of non-programmable computational units are ASICs and custom ICs. Examples of programmable communication units are FIFOs with controllers, bidirectional handshake controllers, direct memory access (DMA) controllers, bus arbiters, or shared memory access and examples of non-programmable communication units are special purpose, customized communication units. All computational and communication units in our library are assumed to be compatible with industry interface standards such as the evolving *Virtual Socket Interface*. As a result, we can mix and match (plug and play) various intellectual property (IP) blocks.

A task graph with mid-way communication becomes a directed *multi-graph*, that is, there may exist more than one arc from one node to another node. The task graph may also be periodic. We can handle the case that the period is greater than or equal to the deadline by using the same schedule for every period. Note that, generally speaking, the processes in the task graph are continuous processes, operating on streams of data as opposed to processes that fire only once. Furthermore, the communication between these processes is non-conditional (i.e., it is data-independent).

In this paper, we only consider a task graph which is composed of computational and communication processes with deterministic characteristics (i.e., no data-dependent

Jui-Ming Chang is with Hewlett-Packard Laboratories, Palo Alto, CA 94304 and Massoud Pedram is with the Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA 90089.

This research was supported by NSF-PECASE under contract MIP-9628999 and SRC under contract 98-DJ-606.

loops are present in the task graph).

The task graph may be periodic (with a loop surrounding the whole task graph). We can handle the case that the period is greater than or equal to the deadline by using the same schedule for every period. Furthermore, we assume that the deterministic microscopic loops in the computational and communication processes are taken care of by the estimators that characterize the performance (area, delay, energy consumption, etc.) of these processes and hence they are not explicitly shown in the generalized task graph which describe the behaviors of the communicating processes in a macroscopic way.

Notice that the cost of mapping a process to a library unit (implementation) cannot be exactly determined because of the possibility of sharing the unit between different processes through time-division multiplexing (TDM). The cost function should account for this possibility and include the area and delay overhead associated with the *context switching*. We assume in this paper that TDM will be used whenever possible, and that the overhead of the context switching is accounted for in the area/delay cost of processes which share the same unit.

We allow the resource sharing of programmable components by different processes according to TDM, even if the process lifetimes overlap. For nonprogrammable resources, the sharing can happen only if the process lifetimes do not overlap or the processes are mutually exclusive.

Our algorithm consists of three major phases. First, processes are decomposed into subprocesses which perform parts of the required computation. The correct precedence relationships implied by the specified communication mechanism are then added in by a systematic transformation process. Second, the decomposed subprocesses are scheduled so as to ensure that the subprocesses which belong to the same original process are mapped to the same hardware type (for example, the same CPU with the same utilization factor). [1] We refer to the condition that all of the subprocesses which are obtained from the same original process be mapped to the same hardware type with the same utilization factor as *type consistency constraint*. This constraint is necessary because we assume that the original coarse-grain process has strong internal communication (variable reference, etc.). Based on this assumption, we do not want the subprocesses which are decomposed from the same original coarse grain process to be mapped to different hardware units in the final solution. The scheduling is done by using a *dynamic programming* based algorithm which finds the cost-optimal process mapping while satisfying a given task deadline. The third phase is hardware allocation and binding (sharing) phase which ensures that the decomposed subprocesses will be mapped not only to the same hardware type, but also to the same hardware instance. This phase will also determine the sharing of hardware among all coarse-grain processes in the system. Notice that before the scheduling phase is finished, the actual time span of the processes cannot be completely determined, and the possibility of sharing those processes can be determined only partially. We therefore defer the optimization of hardware sharing to the third phase.

[1] The processor(CPU) utilization factor is the percentage of the CPU time allocated to a process.

We summarize the major contributions of this paper:

- Our paper is the first to solve a time-constrained coarse-grain HW/SW partitioning/mapping problem using dynamic programming in an OPTIMAL fashion. Other existing approaches either address fine-grain HW/SW partitioning/mapping problems, or use ad hoc heuristic techniques for solving the coarse-grain HW/SW partitioning/mapping problems.
- Our paper is the first to handle generalized task graphs consisting of processes which communicate with each other through various blocking or nonblocking communication mechanisms and throughout their active life times (not only at the beginning or the end of their life times).
- Our paper presents a formal method for doing synthesis and mapping of both computational and communication processes in a truly simultaneous and uniform manner under a given global timing constraint.
- Our paper introduces an elaborate scheme for handling type consistency constraints during the application of dynamic programming for general optimization problems. Other applications of this technique can easily be found, for example in technology mapping for dynamic logic circuits. So the solution technique by itself is interesting and has a wider applicability than this particular system-level problem.

The paper is organized as follows. In Section II, we summarize related work for coarse-grain HW/SW partitioning. Section III introduces our transformation rules for process decomposition. In Section IV, we present the mixed integer linear programming (MILP) formulation for the Problem I.1. In Section V, we present our dynamic programming algorithm for solving Problem I.1. In Section VI, we describe the allocation and binding algorithm to be used after the scheduling step and provide some discussions about the cost model used in this paper. Experimental results and conclusion are provided in Sections VII and VIII, respectively.

II. Related work

There are two published works [7] [8] on fine-grain hardware/software partitioning which use dynamic programming. In both of these works, the target architecture contains a single microprocessor and a single hardware chip (ASIC, FPGA, etc.). The authors then try to find the best combination of non-overlapping sequences of fine-grain Basic Scheduling Blocks which fit the available hardware (ASIC or FPGA) and result in maximum speedup (by moving the scheduling blocks from software to hardware). The problem is similar to the knapsack stuffing problem[9], and dynamic programming is performed on organized matrixes or tables to find the optimal solution. Although no global graph traversal is applied in these papers, the authors did try to map the fine-grain task graphs onto given architectures using a dynamic programming approah. Their use of dynamic programming is therefore different from our use of dynamic programming which explicitly traverses the generalized task graph globally.

There have been several research publications on the coarse-grain HW/SW partitioning which handle task graphs with only end/begin type communication [4] [10] [5] [11]. For these task graphs, the total time used by a process is simply the summation of the time used to do the computation and time used to do the communication. These works use greedy heuristic [4], branch and bound [10], or MILP [12] [5] as their optimization techniques. In [13] and [1], the authors allow mid-way communication in a fine-grain HW/SW environment. The form of communication allowed is however not as general as the ones proposed in the present paper and not at the coarse-level.

For the task graphs consisting of only end/begin type of communication, the problem can easily be solved by a dynamic programming algorithm similar to the one used in [14]. For task graphs with mid-way communication, the computation and communication are however, concurrent and the times used in the two parts are not purely additive. For these task graphs, a new method based on a modified dynamic programming algorithm is needed and will be explained later in this paper.

The work reported in [15] for coarse-grain system synthesis, separates the synthesis of computational and communication processes into two distinct stages, one for the computational processes, the other for the communication processes. In this case, it is very difficult to apply a timing constraint (deadline) on the whole system to find a globally optimal solution in this manner because one part of the the critical path is used to do the computation whereas another part is used to do the communication. In [4], a gradient search method is used. In each iteration, the authors perform a *generate and test* operation commonly used in AI. That is, in each iteration, they try to relocate one process from a CPU to another, relocate a message (communication process) from one bus to another, do the rescheduling on the CPUs and buses, and calculate the change on the cost. If the timing constraints on CPUs or buses are violated, they add one more CPU or bus to fix the problem. The synthesis of computational and communication processes can thus be considered to be performed *simultaneously* during each iteration of the search on the solution space. The algorithm is, however, greedy and non-optimal. In our work, the timing constraint is applied to all of the computation and communication subprocesses in all critical paths and thus the synthesis of the two kinds of processes is performed simultaneously. Since our algorithm is based on dynamic programming it produces the optimal solution.

The topology or architecture of the system is not determined initially; rather it is determined by the final result of the instantiated communication units (and the network formed by these communication units). These communication units are in turn instantiated as a direct result of the DP approach and the sharing step that follows it. Our approach in determining the topology or architecture is similar to that of [15], where the topology or architecture of the system is the result of their communication synthesis, not a result of their initial assumptions. This is in sharp contrast to most other HW/SW co-design systems where the topology or architecture is determined initially. Our approach is superior to that of [15] because computational and communication synthesis steps are done simultaneously with respect to the same global timing constraint whereas in [15], the two kinds of synthesis are done in separate phases.

III. Process Decomposition in a Task Graph

This phase decomposes the communicating processes into some smaller computational subprocesses and communication processes. The decomposition step ensures that all of the precedence relationships imposed by the required blocking/nonblocking communication mechanisms are accounted for. Transformation of the communicating processes into computational subprocesses and communication processes for blocking send/blocking receive, nonblocking send/blocking receive, blocking send/nonblocking receive and nonblocking send/nonblocking receive are shown in Fig. 1(a), (b), (c) and (d), respectively. In Fig. 1, S represents the subprocess which sends the data from the sending process whereas R represents the subprocess which sends the *reply* or *acknowledgment* from the receiving process. The arcs with single tail denote the precedence relationships between the nodes. The arcs with double tails denotes the precedence relationship between the two subprocesses that are decomposed from the same original coarse-grain process. Note that there is strong internal communication (variable accesses) and logical coherence between two such subprocesses and hence they should be finally mapped to the same hardware or software instance.

For a task graph with complex communications among processes, we follow the transformation rules shown in Fig. 1 to create to the decomposed task graph.

When there are more than one mid-way communications within a given process (cf. process A in Fig. 2(a)), the decomposition depends on whether the given process is single threaded or multi-threaded. For a single threaded process, the mid-way communication is referenced to the same time line as that of the thread. In this case, the appropriate transformation rules are applied to all mid-way communications at different points on the time line (cf. Fig. 2(b)). For a process with multi-threads, the mid-way communications may be referenced to different time lines for different threads. In this case, we add two dummy nodes Y_1 and Y_2 (with zero cost and zero delay) at the beginning and the end of that process to synchronize the multiple threads. The appropriate transformation rule is then applied on each thread that serves the time line for the corresponding mid-way communication (cf. Fig. 2(c)).

IV. MILP Formulation for the Scheduling

After the decomposition of the original task graph, some (computational) process P_i are decomposed into a number

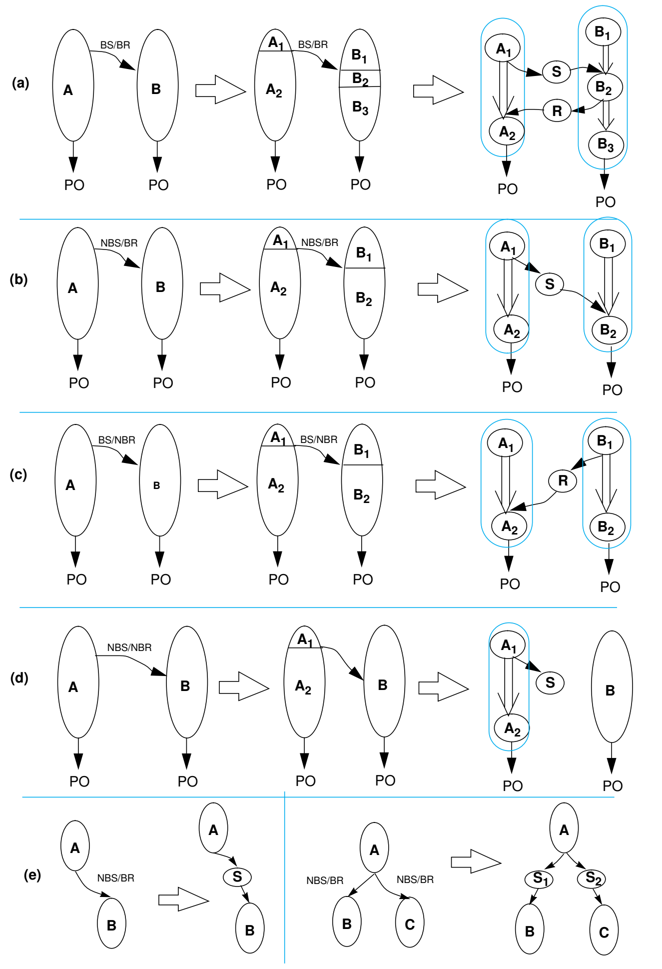

Fig. 1. Decomposition of communicating processes with different type of mid-way communication

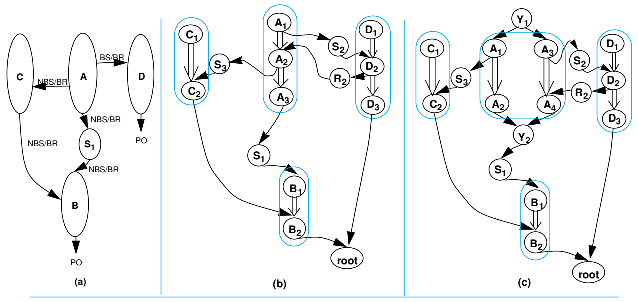

Fig. 2. Example to illustrate decomposition of single and multi-threaded processes

of subprocesses $P_{i,j}$, $1 \leq j \leq U_i$ where U_i denotes the number of subprocesses decomposed from P_i. The new labeling is also applied to existing and/or newly added communication processes (communication processes however are not decomposed and will simply be labeled as $P_{k,1}$ $(U_k = 1)$).

For each (sub)process, there may be more than one mappings or implementations. We classify two mappings to the same programmable processor type with different processor utilization factors as different implementations of that (sub)process. $P_{i,j,k}$ denotes the k-th mapping of subprocess $P_{i,j}$. Note that $P_{i,j_1,k} = P_{i,j_2,k}$ for all i, k but $P_{i_1,j,k}$ and $P_{i_2,j,k}$ are in general unrelated. We allow sharing of same processor instance for subprocesses whose lifetimes overlap only if the total processor utilization for that processor instance does not exceed 100%.

The following variables are needed for the MILP formulation:

$$x_{i,j,k} = \begin{cases} 1 & \text{if (sub)process } P_{i,j} \text{ uses the } k\text{-th} \\ & \quad \textit{implementation} \\ 0 & \text{otherwise} \end{cases}$$

$S_{i,j} \in Z^+ \cup \{0\}$ is the start time for (sub)process $P_{i,j}$. $d_{i,j,k}$ and $c_{i,j,k}$ denote the delay and the cost of (sub)process $P_{i,j}$ when mapped to the k-th implementation. Values of $d_{i,j,k}$ and $c_{i,j,k}$ are known before the start of the MILP. T_{comp} is the deadline for the task graph.

The MILP formulation is written as:

$$min \sum_i \sum_j \sum_k x_{i,j,k} \cdot c_{i,j,k}$$

subject to:

Mapping Constraint: $\sum_k x_{i,j,k} = 1, \ \forall (i,j)$

Global timing constraint:

$S_{i,U_i} + \sum_k x_{i,U_i,k} \cdot d_{i,U_i,k} \leq T_{comp}$, $\forall\ P_{i,Ui}$ whose successor is a primary output.

$S_{i,1} = 0 \ \ \forall\ P_{i,1}$ whose predecessor is a primary input.

Precedence Constraint:

$S_{i,j} + \sum_k x_{i,j,k} \cdot d_{i,j,k} \leq S_{m,n} \ \ \exists\, e \in E$, $e =< P_{i,j}, P_{m,n} >$ for a decomposed task graph $G = (V, E)$.

Type Consistency Constraint:

$x_{i,j,k} = x_{i,1,k}, \ \ \forall k, \ \forall i, \ 1 < j \leq U_i$

The MILP formulation of the mapping/scheduling is simple to write, but contains many integer variables, equations and inequalities. This makes the formulation impractical for large problem sizes. Instead, in the next section, we propose a dynamic programming solution for the same problem which is in practice more useful.

V. Scheduling Using Dynamic Programming

For simple task graphs, the scheduling algorithm is based on dynamic programming. For more complex task graphs, the scheduling is based on dynamic programming with binning.

A. Area vs. delay curves

Before the scheduling, all processes are assigned an area vs. delay curve which represents the area cost and delay for mapping the process to different types of processors. Typical cost vs. delay curves are shown in Fig. 7. The corner points on those curves are non-inferior points. A point is inferior to another point if both its cost and delay are equal or higher. The area cost of a process mapped to a processor type X is the chip area of the hardware realization of processor X. In case the utilization factor is less than 100%, then the area cost is multiplied by the utilization factor. Similarly, the delay cost of a process mapped to this processor is the total computation time for the process running on that type of hardware. In case the processor is shared among multiple processes, the delay cost of each process accounts for the overhead of context switching.

In this paper, we only consider a task graph which is composed of computational and communication processes with deterministic characteristics. The data size for each communication process is assumed to be known as part of the input specification (a priori), and the corresponding delay for mapping to different communication units is estimated by behavioral simulation and profiling. For communication processes, the area estimate includes the area used by communication controller, buses, and local buffers for both the sender and the receiver. The area of a communication process that uses programmable communication controller with some utilization factor $< 100\%$ is estimated as the total cost described above times the utilization factor. For communication units, which may be shared by several communication processes in a TDM manner, the cost and delay should include the overhead of context switching.

B. Simple task graphs

For a task graph without re-convergent fanout and with only end/begin type communications, the algorithm used in [14] can be directly used without going through the process decomposition phase. This algorithm would then produce the optimal hardware/software mapping for a tree-structured task graph (and a good solution for a DAG-structured task graph) under a given timing constraint (deadline) in *pseudo-polynomial* time.

The only modification is to replace the end/begin type communication with a sending process S and add the required arcs to the task graph as shown in Fig. 1(e).

The algorithm assumes that we are given the area vs. delay curves for different module alternatives (implementations) which match each node of the task graph. Then the algorithm perform a post-order traversal which adds the area vs. delay curves of the children of a node and the module alternatives for the node to build the area vs. delay curve of this node. This step will also use the lower bound merge to delete all inferior points. The post-order traversal will continue until the graph roots are reached. Then a pre-order traversal will commence at the roots using user specified arrival time constraint. The minimum area point on the area vs. delay curve of the root which satisfies the arrival time constraint will determine the module alternative to be used at the root. The pre-order then traverses the children of the root with the new arrival time constraint calculated as the arrival time at the root minus the delay of the module used at root. The recursive procedure will continue until all leaves have been visited.

B.1 Post-order traversal

A post-order traversal of the tree is performed, where for each node n and for each module alternative at n, a new delay function is produced by appropriately *adding* the delay functions at the children of node n. Adding must occur in the common region among all delay functions in order to ensure that the resulting merged function reflects feasible matches at the children of n (cf. Fig. 4). The delay function for successive module alternatives at the same node n are then merged by applying a *lower-bound merge* operation on the corresponding delay functions. The procedure is repeat until all combinations of points on curves A and B are exhausted. To illustrate the lower-bound merge operation, see Fig. 3.

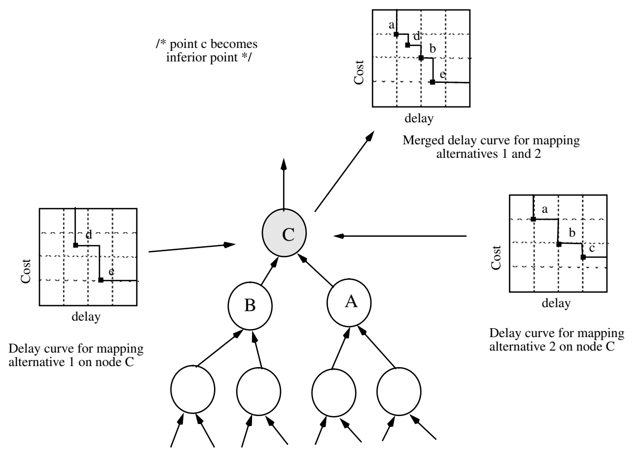

Fig. 3. Lower bound merging of delay curves

The delay function addition and merging are performed recursively until the root of the tree is reached. The resulting function is saved in the tree at its corresponding node. Thus each node of the tree will have an associated delay function. The set of (t, e) pairs corresponding to the composite delay function at the root node defines a set of arrival time-cost trade-offs for the user to choose from.

To illustrate the delay function addition, consider the example in Fig. 4. It shows the addition of the children's curve to its parent for a module alternative m match at node C. The children of this match are nodes A and B. The delay functions for A and B are known at this time. To compute a point on the delay function for node C, we select a point from delay function of the children, i.e. point a on delay curve of node A. The delay of point a is 3 units. So, we look for a point on the delay function of node B with delay less than 3 which has the minimum cost. In this example, d is the desired point. We therefore combine points a and d to generate point a' on delay-curve(C), with

$$
\begin{aligned}
arrival(a') &= t^s_{a'} + delay(m) \\
t^s_{a'} &= \lceil arrival(a)/t_c \rceil \cdot t_c \\
cost(a') &= cost(a) + cost(d) + cost(m)
\end{aligned}
$$

where $arrival(x)$ and t^s_x denote the output arrival time and the starting time for x respectively; t_c denotes the basic unit of time used in the system.

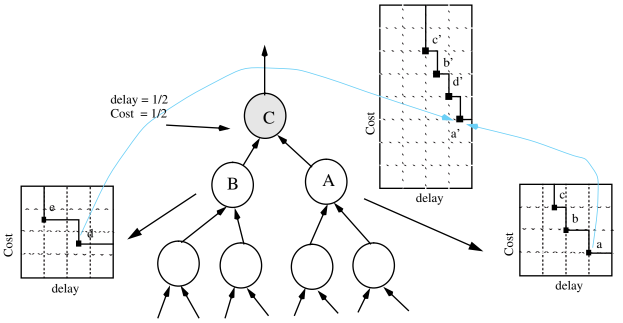

Fig. 4. An example of adding two curves to obtain the parent curve

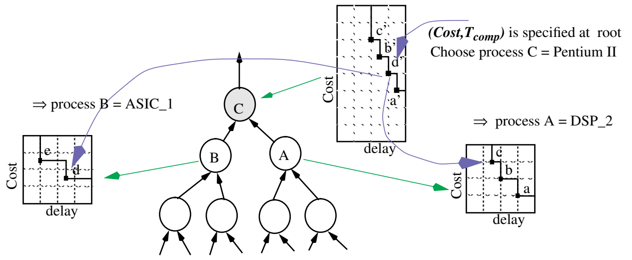

Fig. 5. Pre-Order traversal

B.2 Pre-order traversal

The user can now use the total computation time constraint $\mathbf{T_{comp}}$ on the root of the tree and perform a pre-order traversal to determine the specific point on each curve associated with each node of the tree. The timing constraints of children at the root is computed as $\mathbf{T_{comp}} - \mathbf{t_{delay}}$, where $\mathbf{t_{delay}}$ is the delay of the module alternative of the root that makes the root satisfy arrival time $\leq$ $\mathbf{T_{comp}}$ and has the minimum cost (cf. Fig. 5). This module selection and timing constraint propagation technique is applied recursively at all internal nodes during the pre-order traversal.

C. Complex task graphs

Handling task graphs with processes that have reconvergent fanout and use mid-way communication during their lifetime is a much more difficult task. This is because processes in the task graph have to be decomposed into subprocesses, and the communication processes which reflect the blocking/nonblocking communication mechanism have to be inserted. Furthermore, after the decomposition phase, the dynamic programming paradigm must be modified to ensure that the subprocesses which belong to the same original process are mapped to the same hardware or software component instance in order to maintain the logical coherence and performance. This is achieved in two steps; during scheduling, we ensure that the decomposed subprocesses which correspond to the same original process are mapped to the same HW or SW type with the same utilization factor. During the allocation and binding, we ensure that these subprocesses are further mapped to the same HW or SW component instance.

We first show that this problem is *NP-complete.*

Theorem V.1: Problem I.1 is *NP-complete.*

Proof: The scheduling of generalized task graph with blocking/nonblocking communication mechanism is defined as follows: given a generalized task graph (each process in the task graph has several possible implementations) with blocking/nonblocking communication mechanism, find the optimal cost scheduling of that task graph satisfying the given deadline (total computation time of the task graph) and thus every subprocess decomposed from the same original coarse grain process is mapped to the same type of hardware with the same utilization factor.

By restricting the generalized task graph to be a chain, restricting all of the communications to be of nonblocking send/blocking receive types and allowing only two implementations for each process in the chain, our problem becomes identical to the circuit implementation problem which is known to be NP-complete [16]. Hence, our problem is proven to be NP-complete by restriction [9]. ■

The mid-way communication among coarse-grain processes is possible and occurs frequently. Using the original dynamic programming in [14], we may get some point on the curve of a node with re-converging inputs. This point may result in inconsistent type assignments for the multiple fanout node that gave rise to the re-converging inputs of the node in question. This is obviously wrong (cf. Fig. 10). In addition, using the original algorithm in [14], during the post-order graph traversal, we may drop some points that actually lead to the optimal solution (cf. Example V.1).

We use type defined (tagged) bins on each node in the decomposed task graph to ensure that the above mentioned situation does not arise.

Example V.1: Here we use an example to show why binning is necessary to obtain the correct solution to our problem.

Suppose in Fig. 7(a), we want to get the minimal cost solution on the primary output (PO). A_1 and A_2 are two subprocesses decomposed from an original process A, and C is another process. The area vs. delay curves associated with different matchings on the (sub)processes are also shown in Fig. 7(a). Each subprocess A_1, A_2 and process C can be mapped to either implementation E or F.

Using dynamic programming without binning, we obtain a final accumulated curve at the PO with each point annotated with the implementation type of A_1 and A_2 as shown in 7(b). We can see that some points (solutions) are composed of different types of mappings used for both A_1 and A_2. Such points do not represent valid solutions to our problem.

Using dynamic programming with binning, we obtain two curves at the PO as shown in Fig. 7(c), each curve is tagged with the implementation used for both A_1 and A_2. There is no type inconsistent solution here. Notice that point (9,46) in the curve tagged by $A = E$ in Fig. 7(c) is not present in the solution curve obtained in Fig. 7(b). The reason is that this point was inferior to point (9, 45) and hence was dropped from Fig. 7(b).

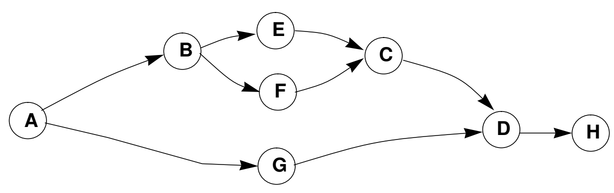

Fig. 6. Example to show the definition of re-convergence

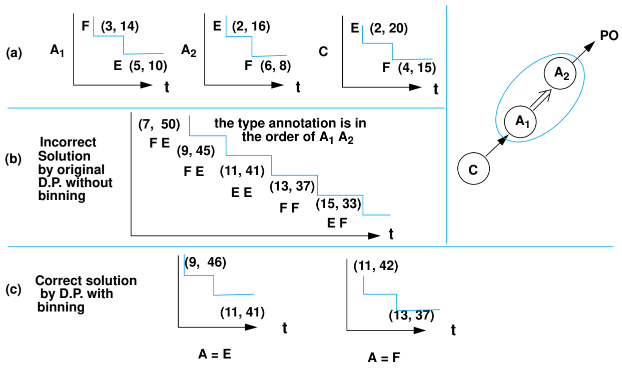

Fig. 7. Example to show why binning is required during D.P.

C.1 Creating the binning strings

We first provide the definition of re-convergent nodes which will be needed in this paper.

Definition V.1: In a directed graph if there are two or more *vertex disjoint directed paths* from node s to node t, s is the re-convergent fanout stem (node), and t is a primary re-convergent node of node s.

If there are no re-convergent fanout nodes on the paths between a re-convergent fanout node A and its primary re-convergent nodes, then A is a simple re-convergent fanout node. Otherwise, A is a complex re-convergent fanout node.

Definition V.2: [17] Let A be a simple re-convergent fanout node.

a) if A is located on a path between a re-convergent fanout node B and a primary re-convergent node of B, then all of the primary re-convergent nodes of A which are not primary re-convergent nodes of B are secondary re-convergent nodes of B.

b) if node B is located on a path between a re-convergent fanout node C and a primary re-convergent node of C, then all the primary and secondary re-convergent nodes of B which are not primary re-convergent nodes of C are secondary re-convergent nodes for C.

For example, in Fig. 6, node B is a re-convergent fanout node, and C is a primary re-convergent node of B. Node C is also a secondary re-convergent node for node A. The primary re-convergent nodes of node A is D. Node H is not a re-convergent node of node A, because all paths from A to H are not vertex disjoint. In the rest of this paper, we will refer to the re-convergent nodes of certain node as primary and/or secondary re-convergent nodes of that node.

To satisfy the type consistency constraint, we modify the dynamic programming algorithm as follows. First, in the solution of [14], the post-order and pre-order traversal can be performed on the individual PO's sequentially for the min-cost solution under timing constraint. However, this approach can lead to a type inconsistent solution. For this reason, we add some dummy nodes and a root with zero cost and zero delay to merge different PO's into a single root. (cf. Example V.3 for more details). Second, we add binning strings to each node as detailed next.

The pseudo code for *create_binning_strings* is shown in Fig. 8(a). Note that in the last part of the pseudo code, the function *search_reconvergent_nodes*(z) will return both primary and secondary re-convergent nodes of z. The primary and secondary re-convergent nodes of any given multiple fanout node can be easily found by conducting a search which is a modified recursive pre-order traversal on the decomposed task graph starting from the given multiple fanout node. When we visit a node for the second time, that node is one of the re-convergent nodes, and we immediately return from this re-convergent node without traversing its subtree further. The pre-order traversal continues to visit other nodes until it terminates. In the create_binning_strings, to check whether or not a node t is in a path from node z to node s, we can first use Floyd-Warshall algorithm [18] to find the transitive closure of the graph and then check the reachability from z to t and from t to s. An example of using the algorithm is shown in Fig. 8(b). The following two theorems tell us how to calculate the binning strings in a graph.

Theorem V.1.1: Suppose a process B is decomposed into subprocesses $B_1, B_2, B_3, \ldots, B_n$, with precedence relationships $B_1 \prec B_2 \prec B_3 \prec \ldots, \prec B_n$. We denote the subset of those subprocesses decomposed from process B with fanout count greater than or equal to two as nodes $C_1, C_2, C_3, \ldots, C_m$ with the precedence relationships $C_1 \prec C_2 \prec C_3 \prec \ldots, \prec C_m$. Let $U_1, U_2, \ldots, U_m$ denote the re-convergent nodes of $C_1, C_2, \ldots, C_m$, respectively. Note that each U_i is in general a collection of nodes. We also denote the set of all nodes which lie on the re-convergent fanout paths from C_i to U_i as T_i. Then we have $T_1 \supseteq T_2 \supseteq T_3 \supseteq, \ldots, \supseteq T_m$.

Proof: We denote the set of re-convergent nodes for node C_i as U_i. $C_1 \prec C_2$ and both C_1 and C_2 are decomposed from the original process B, therefore C_2 is reachable from C_1. Because we merge all primary outputs into a single root, for any node x in U_2, there must exist a node y in U_1 such that x lies in the path from C_1 through C_2 to x to y. This can be easily shown as follows. In Fig 9(a), there exists a node y in U_1 such that there is a path from $C_1 \rightsquigarrow C_2 \rightsquigarrow x \rightsquigarrow y$. The desired result obviously holds here. In Fig 9(b), suppose we have a node y in U_1, but there is no path from x to y. Because we merge multiple primary outputs into a single root, there must exist a node m which is the intersection of the path from $x \rightsquigarrow$ root and the path from $y \rightsquigarrow$ root (in the extreme case, m may be the same as x). Node m will serve as the re-convergent node in U_1 and this fact makes x (re-convergent node of C_2) also the re-convergent node of C_1. That is, every node in U_2 will be

(a)

```
create_binning_strings (graph)
{
 for (each node z in graph) z.binning_string = ∅ ;
 for (each node z in graph) z.binning_string = w.process_ID where
   (z is a decomposed subprocess of w) or (z is the same as w and fanout(z) ≥ 2);
 for (each node z in graph)
   {
    w = z.original_process;
    if (( (z is the 1st decomposed subprocess of w) or (z is the same as w)) and
                                                                   (fanout(z) ≥2))
      z.marked = true;
   }
 for (each marked node z in graph)
   {
    for (each node r in the transitive fan-in cone of z)
      {
       r.binning_string = r.binning_string ∪ process_ID(r.original_process);
       z.binning_string = z.binning_string ∪ process_ID(r.original_process);

       for (each node k in graph that lie in a path from r to z)
         k.binning_string = k.binning_string ∪ process_ID(r.original_process);
      }

    S(z) = search_reconvergent_nodes(z);
    for (each node s in S(z))
      {
       s.binning_string = s.binning_string ∪ z.binning_string;
       for (each node t in graph that lie in a path from z to s)
         t.binning_string = t.binning_string ∪ z.binning_string;
      }
```

(b)

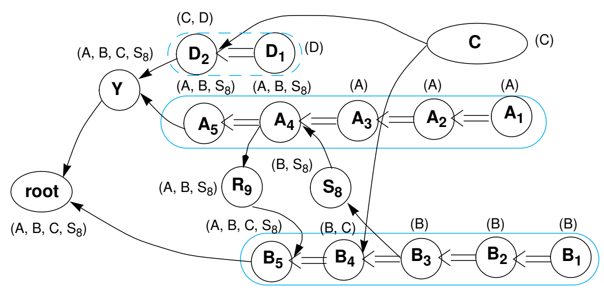

Fig. 8. Pseudo code for creating binning strings

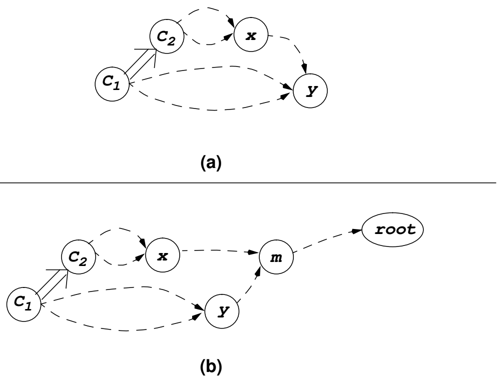

Fig. 9. Fig. to be used in Theorem V.1.1

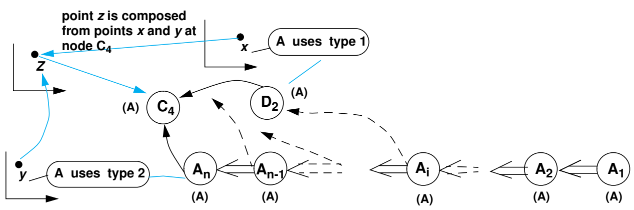

Fig. 10. Fig. to be used in Theorem V.1.2

either a primary or a secondary re-convergent node of C_1. This implies $U_1 \supseteq U_2$. For any node w in T_2, suppose the corresponding re-convergent node in U_2 is node z. Then w must also lie on a path from C_1 to C_2 and ending at z, which is also a node in U_1. This implies w is in T_1, i.e., $T_1 \supseteq T_2$. Proof for other T_i's is the same. ■

From the above theorem, we can save some computation as follows. For a process decomposed into several subprocesses with fanout count greater or equal than two, only the first node (the one which precedes all other nodes) with fanout count greater or equal than two need to be involved in the binning string calculation.

Theorem V.1.2: Suppose a process A is decomposed into $A_1, A_2, \ldots, A_n$ with precedence constraint $A_1 \prec A_2 \prec A_3 \prec \ldots, \prec A_n$. Let A_i be the first decomposed subprocess with fanout > 1. Then we must include the process ID of A in all of the nodes that lie on any path from A_i to any re-convergent node of A_i.

Proof: The situation is illustrated in Fig. 10. It is obvious that subprocesses $A_1, A_2, \ldots, A_n$ must have the ID of A in their binning strings. A_i is the first decomposed subprocess with fanout > 1. For a re-convergent node of A_i (say C_4), there may be some solution point z at the curve of C_4 which is composed of point x of solution curve of D_2 and point y of solution curve of some subprocess decomposed from A (say A_n). In general, the mapping types on subprocesses of A used to generate points x and y may be different (inconsistent). We however want to enforce the type consistency for all subprocesses decomposed from A, therefore we need to put the ID of A in the binning string of C_4, the re-convergent node of A_i. Furthermore, for all nodes which lie on any path connecting A_i and C_4 (including the end points A_i and C_4), their binning strings must also contain the ID of A in order to propagate forward the type information for the mapping of A. For the subprocesses $A_1, A_2, \ldots, A_{i-1}$, each of the nodes has a fanout count of one and hence does not have any re-convergent nodes associated with it. Consequently, there will be no possibility of creating type inconsistent solutions from these subprocesses. For any subprocesses A_j with $j > i$, the set of all nodes which lie on the paths from A_j to its re-convergent nodes is a subset of the corresponding set of A_i according to Theorem V.1.1. Therefore, for the purpose of calculating the binning strings, it is sufficient to include the process ID of A in all of the nodes that lie on any path from A_i to any re-convergent node of A_i. ■

Example V.2: Suppose in Fig. 11, A_1 is a subprocess decomposed from process A. In Fig. 11(a), node Y does not lie on any of the paths from node A_1 to B_2, therefore, Y does not need the ID of A in its binning string. In contrast, in Fig. 11(b), Y lies some path from A_1 to C, therefore, Y needs the ID of A in its binning string.

Each node (subprocess) will have several bins, and each bin will have an associated **tag** which describes the implementations used for each process in the binning string of the node. For example if the binning string of node X is (A, B) and if there are 3 types of mapping on process A and 4 types of mapping on process B, then there will be a

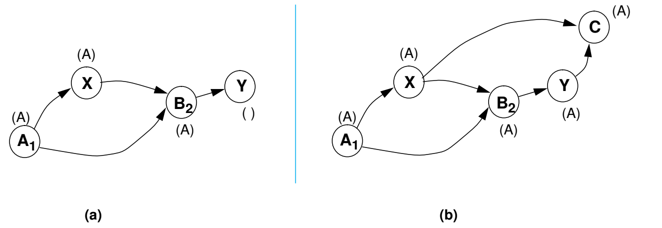

Fig. 11. Two examples to show how the binning strings (shown in parantheses) are calculated

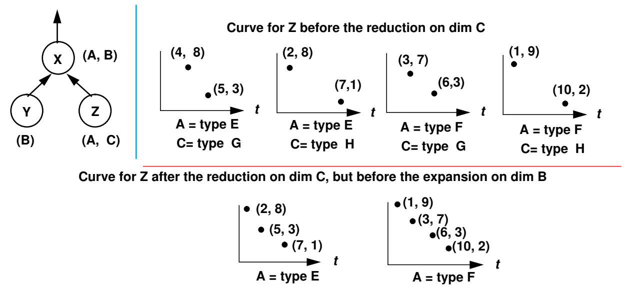

Fig. 12. Example to show the reduction of dimension of the bins

total 12 bins for node X. The first bin will be tagged as (A = type 1, B = type 1), and the second bin will be tagged as (A = type 1, B = type 2), and so on.

C.2 Post-order traversal

Suppose we are processing node X with two children Y and Z (which have already been processed during the post-order traversal). We check the binning strings of Y and Z against that of X. If the binning strings of any child and its parent are different, we have to normalize the dimension of the bins of the child to the same dimension as that of the parent. For a child node with binning string shorter than that of its parent node, we expand the dimension of the bins of that child node by duplicating the corresponding curve to the bins which are added in order to match the binning string of its parent. For example, if child node Y has binning string (B) and its parent X has binning string (A, B), and assuming that there are two types of mappings for both A and B, say types E and F. Originally, Y has two bins tagged with $(B = E)$ and $(B = F)$, respectively. After the expansion, Y will have 4 curves each tagged with a different combination of types of A and B used. In other words, we will duplicate the original curve of node Y for $(B = E)$ tag and create two identical curves for tags $(A = E, B = E)$ and $(A = F, B = E)$. Similar duplication step is applied to the curve of node Y for the $(B = F)$ tag. For a child node with longer binning string than that of its parent node, we reduce the dimension of the bins of that child node by merging the curves which belong to bins that differ only in the ID missing from the binning string of the parent. For example, if child node Z has binning string (A, C) and its parent X has binning string (A, B), we need to reduce on the dimension C and expand on the dimension B for the bins of child node Z. To reduce on the dimension C, we will do a superimpose followed by lower bound operation on the curves of bins corresponding to tags $(A = E, C = E)$ and $(A = E, C = F)$ to obtain the unified curve for the new tag $(A = E)$. Similar operation is needed for the curve in bin tagged $(A = F)$ (cf. Fig. 12).

After we normalize the dimension of each child node, the curve representing the accumulated cost vs. delay on the parent can be constructed by adding the curves of each child and including the contribution of the module alternative (which is consistent with the tag of the bin) matched at that parent. This must be done for every bin, one at a time.

Adding must occur in the common region among all curves to ensure that the resulting merged function reflects feasible matches at the children of n. The curve for successive matchings at the same node n are then merged by applying a *lower-bound merge* operation on the corresponding curves.

Because our decomposed task graph is a DAG instead of a tree, we face the problem of how to pass up the cost of a multiple fanout node to its parents during the post-order traversal. We use the a heuristic whereby the cost value of a multiple fanout node is divided by its fanout count when propagated upward in the DAG. This heuristic produces the **exact** total cost at the root. This is true as long as multiple primary outputs are merged into a single root. The proof is straight forward (similar to flow conservation in network flow problem).

The curve addition and merging are performed recursively until the root of the graph is reached. The resulting curve is saved in the corresponding bin of the graph at its corresponding node. The set of (t, c) pairs corresponding to the composite curve for the tag at the root node gives the set of all possible arrival time-cost trade-offs for the user to choose from.

C.3 Pre-order traversal

Pre-order traversal begins at the root of the decomposed task graph and proceeds toward the leaves. Consider a node X of the graph. The (output) arrival time and the type constraint for the node are know. Our task is to determine the arrival times and the type constraints for each of its child nodes.

Consider a child Z of node X. We are assured that at least one of the tagged curves of X is consistent with the type constraint passed down to X. If there is exactly one such curve stored at X, we pick the minimum-cost point of the curve that which satisfies the arrival time constraint of X. Otherwise, there are more than one tagged curves that are consistent with the type constraint passed down to node X. In this case, we find the corresponding best cost point on each curve (which satisfies the timing constraint) and among them pick the solution which has the overall minimum-cost. Next, we update the type constraint for node Z as the Union of type constraint passed down to node X and the constraint implied by the tag of the chosen point on the tagged curve (or bin) and set the timing constraint of Z as the timing constraint at X minus the delay of the match at X.

A node with multiple fanouts will be visited multiple times during the pre-order traversal. During each visit, the arrival time and possibly type constraint of the node may change in order to guarantee that arrival time and type consistency constraints for all paths emanating from that node toward the root of the graph are satisfied. Note that because of our introduction of a single root for the graph binning string assignment procedure, we are guaranteed not to see conflicting type consistency constraints from different fanout branches of the multiple fanout node. This is illustrated in the next example.

Example V.3: We use a simple example to show the results of traversal on an example task graph with and without merging the multiple PO's into a single root. The task graph for the example is shown in Fig. 13(b) with unmerged PO's. In this graph nodes corresponding to communication processes ($S's$ and $R's$) are deleted (the delay and cost for them are set to zero) for the sake of clarity. The module curves for all node are shown in the bottom of Fig. 13. The binning string for each node is shown in its right hand side within parentheses and the process ID's are separated by a comma. If we specify timing constraint = 18 at PO_1 during pre-order, we obtain a solution that makes B_1, B_2 and B_3 use type F processor. However, the same timing constraint imposed at PO_3 results in a solution where A uses type E, B_1 and B_2 use type E, and C uses type F. We can convert B_3 from type F to type E to create a type consistent solution for $B's$. Unfortunately, this increases the arrival time at PO_1 to 21, which violates the timing constraint. In conclusion, the sequential post/pre-order traversal on multiple PO's may either create a type inconsistent (unacceptable) solution or a solution that violates the given timing constraint.

Consider the same system but with the multiple PO's merged into a single root as shown in Fig. 13(a). Applying the post-order followed by the pre-order traversal on the root with timing constraint =18 produces a solution where all $A's$ use type E, all $B's$ use type F, and all $C's$ use type E processors. The solution is type consistent and satisfies the given timing constraint. In fact, after post-order, we can get the optimal solutions under any given timing constraints very quickly because node Y has binning string (A, B, C) and whenever we get to node Y during the pre-order we already get the solution for the type of the processors that will be used by $A's$, $B's$ and $C's$. The further traversal of nodes under node Y is needed only if there are some processes that do not have their ID's in the binning string of Y.

Theorem V.1.3: The dynamic programming with binning (with the proposed binning string construction algorithm) solves Problem I.1 optimally and satisfies all type consistency constraints.

Proof: The proof follows easily from the correctness of our binning construction algorithm and the fact that the principle of optimality for dynamic programming holds. ■

Note that the above statement about the optimality

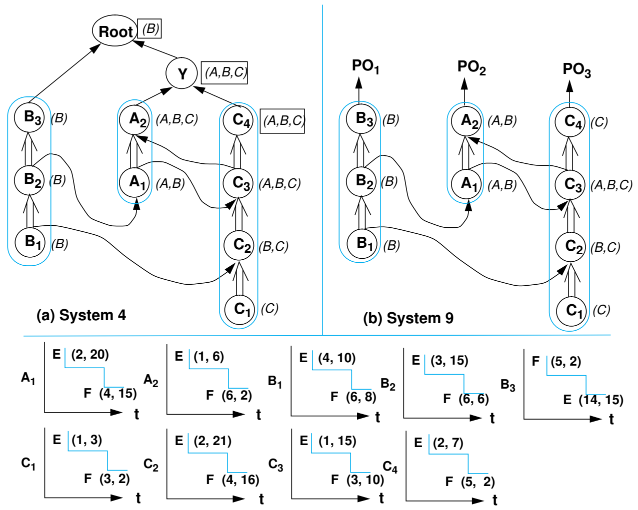

Fig. 13. Figure to illustrate the need to merge PO's into a single root

of our proposed solution only apply to the problem of the scheduling under timing and type consistency constraints. The complete problem that include the simultaneous scheduling and sharing is not solved in an optimal fashion.

D. Complexity Analysis

For simple task graphs defined in Section V-B, the task graphs remain the same after the initial process decomposition step. The same algorithm used in [14] can be used. The time complexity using dynamic programming algorithm for solving problems involved with this class of task graphs is pseudo-polynomial.

Let us scale delay values for all nodes (subprocesses) under different process mapping to become integers. Furthermore, we denote the maximum computation time for a tree-like decomposed task graph (using the worst-case integer delay values on any path) by T_{max} and assume that T_{max} is bounded from above by an integer Q. Let $|\mathcal{I}| = n$ where n is the total number of nodes (*decomposed* computation and communication (sub)processes) in the decomposed task graph.

Suppose that the maximum number of possible process mappings for each subprocess (node) is K and the maximum length among all binning strings is m. r is equal to the total number of *original coarse-grain (un-decomposed) computational processes* plus the total number of communication processes in the decomposed task graph that are themselves multiple fanout processes or have their own decomposed subprocesses with fanout count greater than one or themselves or their decomposed subprocesses are in the transitive fan-in cone of some multiple fanout nodes in the decomposed task graph. Using our process decomposition method, all communication (sub)processes have fanout count ≤ 1. In a decomposed task graph with n nodes, $0 \leq m \leq r < n$. Then there will be at most K^m bins in each node in the decomposed task graph. Then the maximum possible number of points in each node of the decomposed task graph will be $Q \cdot K^m$.

The number of area-delay points on each node in the decomposed task graph is bounded from above by $Q \cdot K^m$. The algorithm thus has a time complexity of $n \cdot Q \cdot K^m$. Delay function merging and adding can be done in polynomial time in the number of points on the curves involved in the operations. Therefore, our algorithm for solving the coarse-grain process mapping with complex communication problem runs in $n \cdot Q \cdot K^m$ time.

Note that, the value of m is in general dependent on the structure of the decomposed task graph. In the worst case, m can be as large as n. In practice, m is however, much smaller than n. For example, for the frequently encountered simple task graphs defined in Section V-B, m is zero. In this case, the algorithm has pseudo-polynomial time complexity on the decomposed task graph.

The initial process decomposition step is necessary for any method (including MILP or exhaustive search) which handles a task graph with mid-way communication. Both MILP and exhaustive search will have exponential complexity on n. We have also included an example (cf. Fig. 16) from the communication field with complex communications among computational processes, the MILP solution or exhaustive search on the decomposed task graph (with tonal n=39 nodes) runs forever due to the exponential behavior on the large number of variables in MILP (195 variables, 123 equations, 51 inequalities) or the number of nodes for exhaustive search (with the decomposed subprocess regrouped, there will be total 22 nodes; if each node has 4 possible implementations, there will be 4^{22} combination needed to explore using exhaustive search). However, using our new method on this same example, m is 9 and the time complexity is 4^9 if $K = 4$. It only takes 6.329 seconds on a Pentium PC 233 Mhz to solve the scheduling problem using our new method.

VI. Allocation and Binding

As a result of the scheduling phase, the computational subprocesses decomposed from the same original coarse-grain process are mapped to the same type of processor implementation or custom ICs. They have not however been mapped to the same instance of the processor or custom IC.

Our first step is to *regroup* these subprocesses back into their whole coarse-grain process and assign them to the same processor instance. From this point on, the allocation and binding will treat the regrouped subprocesses as a single process as if they have not been decomposed. The lifetime of that process is the time span from the beginning of the first subprocess to the end of the last subprocess.

Processes are generally separated into different classes if they are mapped to different types of hardware units. Within each class, the allocation and binding (sharing) is then performed.

Processes which are mapped to programmable units such as CPUs, DSPs, DMAs, or other controllers for communication, can share the same instance of the unit through TDM even if their lifetimes overlap. In addition to the programmable communication units, part of the buses or the shared memory and/or local buffers needed for communication may be shared in a TDM fashion by the the corresponding communication processes. The requirements for sharing one programmable unit instance are that the processes are mapped to the same type of unit, and the sum of the utilization factors of those processes is less than 100%. We perform the allocation and binding by using a *modified bin packing algorithm* which ensures that every regrouped coarse-grain process is bound to the same hardware instance throughout its lifetime.

Processes which are mapped to the same hardware type but *do not necessarily have the same* utilization factors (even if their lifetimes overlap), it may still share the same unit if the sum of their utilization factors does not exceed 100%.

For non-programmable units such as custom ICs or other communication units, sharing is possible only if either the process lifetimes do not overlap or the processes are mutually exclusive.

A. TDM scheduling

As a result of the allocation and binding phase, we may assign processes with overlapping lifetimes to the same instance of a programmable processor. The programmable processor is shared by these processes in a TDM manner. The processes are granted time slices proportional to their utilization factor by a simple operating system. This step, called the TDM scheduling, is detailed next.

Consider some instance of a programmable processor and the set of coarse-grain processes assigned to it. We divide the time line for this processor into a set of intervals which are delineated by the begin or end points of each coarse-grain process whose life span overlaps with that interval, is served by the processor instance. The service time for each such process is proportional to the ratio of the utilization factor of that process to the sum of the utilization factors of all processes whose life spans overlap with the interval (Note that this sum is less than or equal to 1). Note that the processor instance may be idle for some periods of time during each interval. This is to ensure that each process is completed in the time decided by the dynamic programming based scheduling of previous section and not any shorter time (which may lead to a timing violation). Furthermore, note that the same process may receive different service times (time slices) during different intervals due to the service requirement/demands of other processes in those intervals.

The modified bin packing algorithm ensures that the bandwidth requirement for communication units which are shared by several communication processes are met by using a TDM scheduler in a similar way as the TDM scheduling is done for programmable processors.

Theorem VI.1: The TDM scheduling with the modified bin packing preservs the planned global timing.

Proof: Each process which is assigned to a programmable processor instance is granted the proper amount of service time in each time interval so as to preserve the total computation time for each process as planned by the DP-based scheduling. Thus the global timing will not be violated. ■

B. Extension to allow subprocesses with different processor utilization

Previously, we restrict that all subprocesses decomposed from the same original process be mapped to the same type (and the same instance) of hardware with the same utilization factor. We can relax this and allow the subprocesses to mapped the same type but different utilization.

This extension will not complicate the scheduling phase, the only thing that may be changed is that in the bins that are tagged with some mapping type in the leave nodes that is decomposed from certain original process will have more more than one points to start with. For example a process A_1 which is decomposed from an original process A with binning string (A) has 3 bins that are tagged as A = Intel Pentium, A = AMD K6 and A = TI 320C25. In each bin, there might be multiple points corresponding to different processor utilization factors.

The allocation and Binding method is still similar, but the interval is now defined at the beginning or ending of the subprocesses (now those subprocesses may have different processor utilization) in stead of the beginning or ending of the coarse-grain processes. Within each interval, the total utilization factor is still a constant. But in this extension, there will be much more intervals to process.

C. Handling other cost functions, e.g. energy

Our optimization method based on dynamic programming with binning can also be used on different cost functions without modification. Previously, we used the area cost to illustrate the problem and the design space. We can easily replace the area cost function with a composite cost function, such as the cost function related to area and energy consumption. Although the composite cost function may be a non-linear function, it is quite common to use a linear combination of area and energy as the cost function. That is, cost = $\alpha \cdot A$ + (1- α) $\cdot E$, $0 \leq \alpha \leq 1$.

We assume that all of the nonprogrammable hardware units will be turned off when they are not performing any task and will be turned on only when they are active. For programmable computational and communication units, we assume they consume a very small amount of energy when no process is running on them. Note that the total energy used for mapping a process to a programmable processor with different processor utilization factors will remain the same. This is because mapping with lower CPU utilization will reduce the power consumption and may also reduce the area cost due to higher potential for resource sharing; It will however increase the time to complete the process on the mapped processor and thus its energy cost will remain nearly the same.

For the case that the cost function is the total energy used, then a time constrained minimal energy solution for a given tast graph will assign a 100% processor utilization factor for every process mapped to a programmable processor. The resulting design will consist of a mapping that uses a lot of chip area, because the processes mapped to programmable processors will no longer be able to share the processor instance. This result is due to the fact that all other processor utilization factors (which allow for processor sharing) will result in the same egergy, but will have larger delay values. Hence they will be inferior to the solution with 100% utilization factor and hence are dropped during dynamic programming.

For the case that the composite cost function is a linear combination of area and energy, some of the points in the cost vs. delay curve for mapping a process into the same type of processor with different utilization factors become non-inferior points due to the area component in the cost function and will thus be kept in the curve. This is a desirable cost function because we not only want to reduce the total energy used, but also keep the chip area small.

D. Discussion

Scheduling by dynamic programming presented in Section V estimated the area cost of a process mapped to a programmable processor as the area cost of that processor times the utilization factor and the area cost of a process mapped to non-programmable hardware component, as the area cost of the hardware component. Before the DP-based scheduling, the actual begin and end times of the processes cannot be completely determined. As a result, it is not possible to accurately account for the possibility of sharing during the scheduling phase. During the allocation and binding phase, we perform the modified bin packing algorithm to maximize the possibility of sharing of the hardware resources. There may of course be some difference between the estimated accumulated area cost of a node during the scheduling and the minimum area cost after one performs the modified bin packing algorithm on all of the nodes in the transitive fanin cone of the node. We do not elaborate on this point here, however it can be shown that if sharing is accounted for during the scheduling, the principle of the optimality of DP will be violated. Separation of scheduling and allocation phases however makes the whole problem more manageable; Furthermore, the difference in cost estimates during the two phases naturally arises in any algorithm based on multiple phase optimizations. This is not uncommon in VLSI CAD where conventional flows separate the whole design process into several sequential phases which causes inconsistency between the estimated cost of some cost function in the current phase and that in the next phase. An example is the placement and routing in layout synthesis. During the placement phase, one cost function to be minimized is total wiring length. However, before the detailed routing is completed, the estimation of the wiring lengths is not accurate.

VII. Experimental Results

Our dynamic programming with binning, named Codex-dp (for Co-design of Communicating Systems Using Dynamic Programming), is implemented in C and tested on a number of circuits. Table I shows the information about the examples used. Prakash1 and Prakash2 are two examples taken from [12], Yen is taken from [4], and Bender is an example taken from [5]. In every example, we do the process decomposition and insert appropriate communication processes (of end/begin type) in the original task graphs. All of the experiments are done for our module library which contains a number of programmable processors and communication units. We allow the sharing of these resource through time-division multiplexing. Our library also contains some non-programmable and mixed analog and digital circuits. More precisely, our library contains CPUs such as the Intel Pentium and Motorola 68030 and DSPs such as TI 302C25 and Communication units such as Intel DMA controller with the surrounding circuitry and 10Mb/s Ethernet controller, and mixed analog and digital units such as Modulator and Mixers used in communication. A pre-processing step determines the area/delay cost of each process when it is mapped to various hardware units in the library. We do this by using the chip areas of the hardware units, as well as by running the process on the hardware unit and measuring the total computation time.

We also report results on 5 more examples from various sources. The task graph for example 1 is shown in Fig. 14 with deadline = 80.0(ms) taken from the CPM system [19]. The task graph for example 2 is shown in Fig. 15 with deadline = 100.0(ms). The decomposed task graph for example 3 is shown in Fig. 13(a) with deadline = 18.0(ms) using our library (Curves shown in Fig. 13 do not correspond to our library modules). The task graph for example 4 is shown in Fig. 2(a), and its decomposed task graph is shown in Fig. 2(c) with deadline = 50.0(ms). The task graph for example 5 is shown in Fig. 16, its decomposed task graph is too large to be included in this paper with deadline = 200.0(ms). This task graph is the sub-block performing the **voice activity detection** used in GSM (Group Special Mobile) [19]. For this example, we used three different deadlines and report the results in row ex5-1, ex5-2, ex5-3. The corresponding deadlines were set to 170, 300, 510 (ms), respectively.

In Table I, column 2 shows the values of m and n seen by Codex-dp (cf. V-D). Column 3 gives the total number of processor and communication units needed after the allocation and binding. Columns 4 and 5 give the CPU time used by Codex-dp (in *seconds* on a 200 MHz Pentium Pro) and the estimated area cost (in cm^2) required to implement each circuit. Column 6 gives the numbers of variables, inequalities and equations if the scheduling is formulated as a mixed integer linear program, MILP (cf. Section IV) assuming that each process has four possible implementations. In column 7, we show the complexity of using exhaustive search after regrouping all of the computational subprocesses back into their original coarse-grain processes and still assuming that each process has four possible implementations. Note that we could not present comparative results with other approaches for codesign of coarse-grain communicating processes, because the other approaches do not support mid-way communication.

As can be seen from the table, Codex-dp produces optimal scheduling results in very short time compared to the expected time for MILP or exhaustive search. Furthermore, the entries for ex5-1, ex5-2, ex5-3 show the trade-off between the area cost and the total computation time for example 5. As can be seen, decreasing the deadline constraint, increases the area cost of the optimal solution. A similar trend exists for all other examples.

Ckt	m, n	pu,cu cnt	Codex-dp c-time	Codex-dp cost	MILP form.	Exh. srch
Prak1	2,11	1,1	0.036	63.7	55,13,11	4^{11}
Prak2	6,22	2,1	0.507	122.5	110,26,22	4^{22}
Yen	1,12	1,1	0.043	63.7	60,13,12	4^{12}
Bend.	7,13	2,2	1.143	137.2	60,17,12	4^{13}
ex1	0,13	1,1	0.086	79.7	65,12,13	4^{13}
ex2	1,14	2,1	0.114	148.5	70,15,0	4^{14}
ex3	3,11	2,0	0.064	98.0	49,14,6	4^{9}
ex4	4,18	3,1	0.107	171.5	63,21,7	4^{8}
ex5-1	9,39	3,3	6.329	200.9	195,51,123	4^{22}
ex5-2	9,39	2,3	6.412	151.9	195,51,123	4^{22}
ex5-3	9,39	2,2	6.356	127.4	195,51,123	4^{22}

TABLE I
EXPERIMENTAL RESULTS

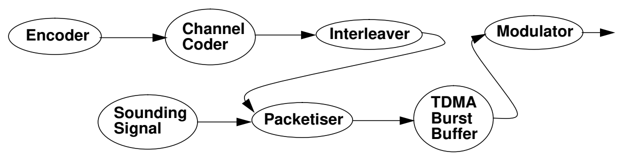

Fig. 14. A very simple task graph with only end/begin communication

VIII. Conclusion

We presented an algorithm based on dynamic programming with binning to solve a min-cost, time-constrained simultaneous scheduling and mapping problem for a set of computational processes which communicated by means of

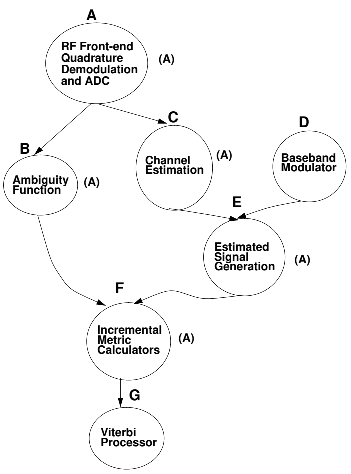

Fig. 15. Task graph with only end/begin communication but with re-convergent fanout

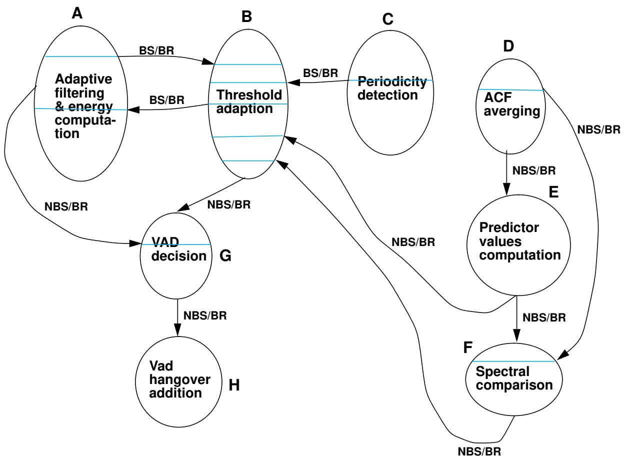

Fig. 16. Task graph of Voice Activity Detection (VAD) used in the GSM system

blocking/nonblocking communication mechanism at times other than the beginning or end of their lifetimes. The proposed algorithm produces optimal results, and is much faster to solve than the MILP formulation. A final resource allocation and sharing step will follow the dynamic programming step and produce the actual instantiation of the processor types to hardware instances. This last step is done using a modified bin packing heuristics.

References

[1] R. Gupta and G. D. Micheli, "System-level Synthesis using Reprogrammable Components," in *Proceedings European Design Automation Conference*, 1992.

[2] R. Ernst, J. Henkel, and Th. Benner, "Hardware/Software Co-Synthesis for Microcontrollers," *IEEE Design and Test Magazine*, vol. 10, no. 4, Dec. 1993.

[3] G. De Micheli and editor M. Sami, *Hardware/Software Co-Design, pages 22, 84, 85, 96, 217*, Kluwer Academic Publishers, 1995.

[4] T.-Y. Yen W. Wolf, "Communication Synthesis for Distributed Embedded Systems," in *Proceedings IEEE International Conference on Computer-Aided Design*, 1995.

[5] A. Bender, "MILP Based Task Mapping for Heterogenous Multiprocessor Systems," in *Proceedings European Design Automation Conference*, 1996.

[6] B. P. Dave, G. Lakshminarayana, and N. Jha, "Cosyn: Hardware-Software Co-Synthesis of Embedded Systems," in *Proceedings IEEE-ACM Design Automation Conference*, 1997.

[7] A. Jantsch, P. Ellervee, J. Oberg, A. Hemani, and H. Tenhunen, "Hardware/Software Partitioning and Minimizing Memory Interface Traffic," in *Proceedings European Design Automation Conference*, 1994.

[8] P. Knudsen and J. Madsen, "PACE: A Dynamic Programming Algorithm for Hardware/Software Partitioning," in *Proceedings IEEE International Workshop on Hardware/Software Codesign*, 1996.

[9] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman and Company, 1979.

[10] J. D'Ambrosio and X. Hu, "Configuration-Level Hardware/Software Partitioning for Real-Time Embedded Systems," in *Proceedings IEEE International Workshop on Hardware/Software Codesign*, 1994.

[11] S. Narayan and D.D Gajski, "Synthesis of System-Level Bus Interface," in *Proceedings European Design Automation Conference*, 1994.

[12] S. Prakash and A. Parker, "SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems," *Journal of Parallel and Distributed Computing*, vol. 16, Dec. 1992.

[13] T. Benner, R. Ernst, and A. Ősterling, "Scalable Performance Scheduling for Hardware-software Co-synthesis," in *Proceedings European Design Automation Conference*, 1995.

[14] J.-M. Chang and M. Pedram, "Energy Minimization Using Multiple Supply Voltages," in *Proceedings International Symposium for Low Power Electronic and Design*, Aug. 1996.

[15] J.-M. Davaeu, T. Ismail, and A.A. Jerraya, "Synthesis of System-Level Communication by Allocation-Based Approach," in *Proceedings IEEE International Symposium on System Synthesis*, 1995.

[16] W.-N. Li, A. Lim, P. Agrawal, and S. Sahni, "On The Circuit Implementation Problem," in *Proceedings IEEE-ACM Design Automation Conference*, 1992.

[17] F. Maamari and J. Rajski, "A Reconvergent Fanout Analysis for Efficient Exact Fault Simulation of Combinational Circuits," in *Proceedings IEEE International Symposium on Fault-Tolerant Computing*, 1988.

[18] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, The MIT Press, McGraw-Hill, 1990.

[19] R. Steele, *Mobile Radio Communications*, Pentech Press, London, 1995.

Jui-Ming Chang is a research scientist in Hewlett-Packard Laboratories, Palo Alto. He received B.S. degree in electrical engineering from National Taiwan University in 1989 and his M.S. and Ph.D. degrees in computer engineering from the University of Southern California, majoring in VLSI CAD in 1993 and 1998, respectively. His research interests include CAD of VLSI circuits and systems, specializing in behavioral-level (high-level) and system-level (system-on-chip) synthesis and hardware/software co-design.

Massoud Pedram is an associate professor of Electrical Engineering - Systems at the University of Southern California. He received his B.S. degree in Electrical Engineering from the California Institute of Technology in 1986 and Ph.D. degree in Electrical Engineering and Computer Sciences from the University of California, Berkeley in 1991. He is a recipient of the NSF's Young Investigator Award (1994) and the Presidential Faculty Fellows Award (a.k.a. PECASE Award) (1996). His current work focuses on developing computer aided design methodologies and techniques for low power design and coupling physical design to logic synthesis.

Dr. Pedram has served on the technical program committee of a number of conferences and workshops, including ASP-DAC, DATE and ICCAD. He also served as the Technical Co-chair and General Co-chair of the International Symposium on Low Power Electronics and Design in 1996 and 1997, respectively. He is a member of IEEE-CAS and ACM-SIGDA and an associate editor of ACM TODAES and IEEE TCAD.