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

Jui-Ming Chang and Massoud Pedram

CENG 98-04

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-4458
February 1998

Abstract

In this paper, 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 lifetimes. The proposed algorithm has been implemented. Experimental results are reported and discussed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Previous work on system level synthesis has mainly focused on fine-grain hardware/software partitioning. Examples include VULCAN II [GM92] and COSYMA [EHB93]. 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. Furthermore, 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 on.

In contrast, coarse-grain partitioning does not decompose the initial specification into basic blocks and does not 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 designers' expertise more easily and can achieve a desired partitioning which satisfies some macroscopic choices more readily [MMS95]. Finally, the resulting solution has more logical coherence which facilitates the top-down design process and allows for debugging of the hardware/software.

Many of the coarse-grain partitioning algorithms 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 [Wol95] [Ben96] [DLJ97]) are directed acyclic graphs ($DAGs$) which 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, in general, the coarse-grain processes may communicate with each other at times other than the end or beginning of their lifetimes. We refer to this kind of communication as *intermediate* communication and to the task graph with intermediate communication as a *generalized task graph*. The problem we are trying to solve can then be stated as follows.

Problem 1.1 *Given a generalized task graph consisting of processes which communicate with each other by arbitrary 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 HW/SW resources so as to minimize the total area cost while satisfying a given deadline.*

Notice that the cost of mapping a process to a library unit (implementation) cannot be exactly determined because of the possibility of sharing the same unit between different processes. In such a case, we can use time-division multiplexing (TDM) to share the unit. The cost should account for this possibility and include the area and delay overhead associated with the *context switching*. We assume in this report 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.

A task graph with intermediate 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 be periodic. In such a case, we can handle the case that the period is greater than or equal to the deadline by performing the same schedule on every period.

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 of non-programmable computational units are ASICs and custom ICs. Examples of programmable communication units are FIFOs with controllers, bidirectional handshake controllers, DMA controllers, bus arbiters, or shared memory access and 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.*

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 only happen if the process lifetimes do not overlap or the processes are mutually exclusive.

Our algorithm consists of three major steps. First, processes are decomposed into subprocesses which perform parts of the required computation. The correct precedence relationships implied by the specified communication mechanism is 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). 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.), we therefore do not want the subprocesses decomposed from the same original coarse grain process to be mapped to different hardware units in the final solution. The scheduling is done 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 that will ensure the de-

composed subprocesses will be mapped not only to the same hardware type, but also to the same hardware instance. The allocation and binding will determine the sharing of hardware among all coarse-grain processes in the system.

The report is organized as follows. In Chapter 2, we summarize related work for coarse-grain HW/SW partitioning. In Chapter 3, we introduce our transformation rules for process decomposition. In Chapter 4, we present the MILP formulation for the Problem 1.1. In Chapter 5, we present our dynamic programming algorithm for solving Problem 1.1. In Chapter 6, we describe the allocation and binding algorithm to be used after the scheduling step. Experimental results and conclusion are provided in Sections 7 and 8, respectively.

Chapter 2

Related work

There are two published works [JEO+94] [KM96] on fine-grain hardware/software partitioning, which use dynamic programming. In these reports, 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 problem, and the dynamic programming formulation is used only to avoid repeated computations in their iterative procedure. Their formulation is therefore completely different from our dynamic programming formulation which traverses the generalized task graph globally.

There are other works which consider communicating processes with end/begin type communication. This kind of task graph is commonly used in real-time and distributed systems [CSL91] [PS89]. There have been several research publications on the coarse-grain HW/SW partitioning which handle task graphs with only end/begin type communication [Wol95] [DH94] [Ben96] [NG94]. 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. The above works use greedy heuristic [Wol95], branch and bound [DH94], or MILP [PP92] [Ben96] as their optimization techniques to explore the solution space.

For the task graphs consisting of only end/begin type of communication, the problem can be easily solved by a dynamic programming algorithm similar to the one used in [CP96]. For task graphs with intermediate 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 as will be explained later in this report.

The work reported in [DIJ95] for coarse-grain system synthesis, separates the synthesis of computational and communication processes into two distinct stages. In this case, it is very difficult to apply a timing constraint (deadline) on the system because part of time in the critical path is used to do the computation whereas another part is used to do the communication. In [Wol95], the gradient search method on the solution space 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 and relocate a message (communication process) from one bus to another and 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. Furthermore, our algorithm is based on dynamic programming and hence produces the optimal solution.

In summary, to our knowledge, the work reported in this report is the first work on coarse-grain process mapping that deals with complex communication among processes (i.e. blocking send/blocking receive, nonblocking send/blocking receive, blocking send/nonblocking receive or nonblocking send/nonblocking receive) using a novel dynamic programming approach.

Chapter 3

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 added. 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. 3.1(a), (b), (c) and (d), respectively. In Fig. 3.1, process S represents the actual process which sends the data from the sending process and R represents the process which sends the *reply* or *acknowledgment* from the receiving process. The arcs with single tail denote the precedence relationships between the nodes connected by that arc. 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 thus they should be finally mapped to the same hardware/software instance.

For a task graph with complex communications among processes, we follow the transformation rules shown in Fig. 3.1 to create to the decomposed task graph.

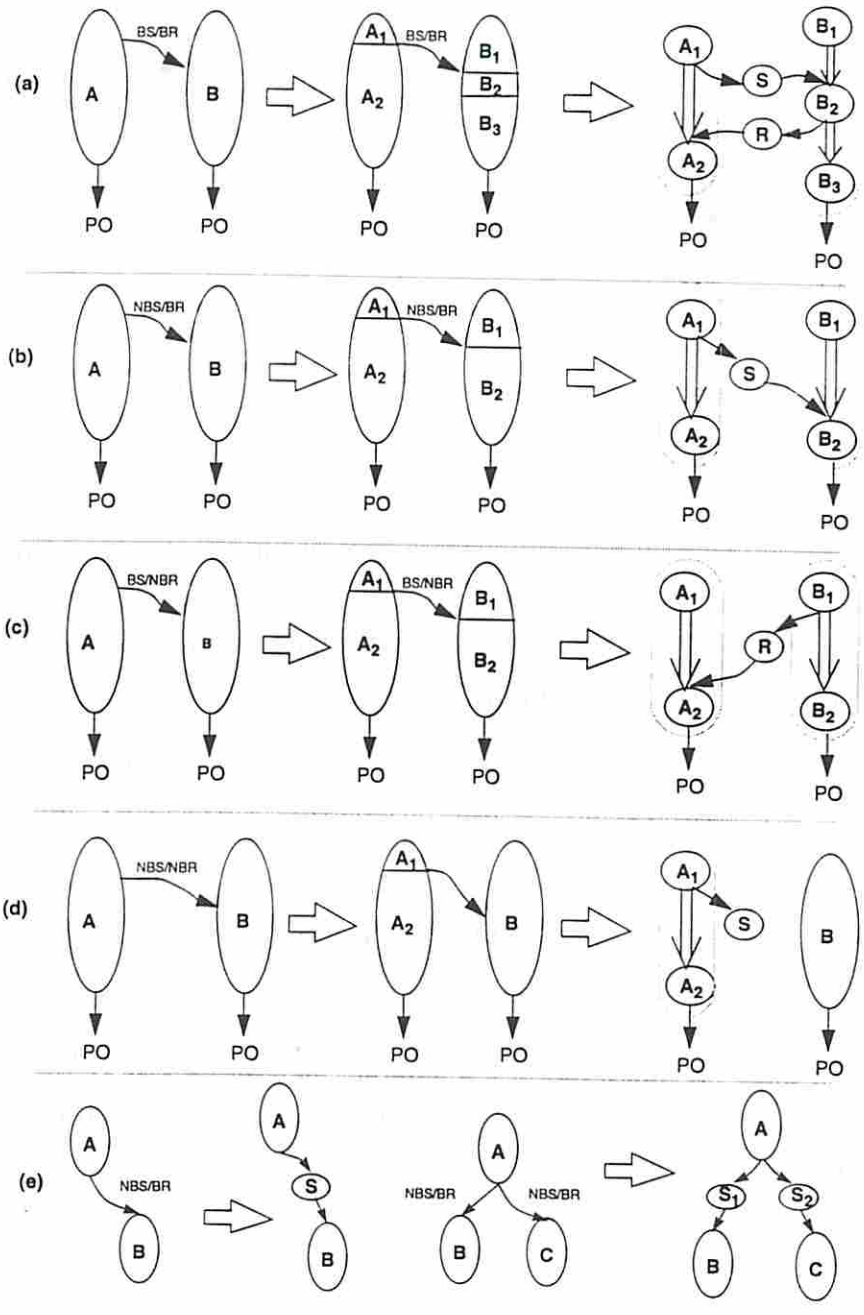

Figure 3.1: Decomposition of communicating processes with different type of intermediate communication

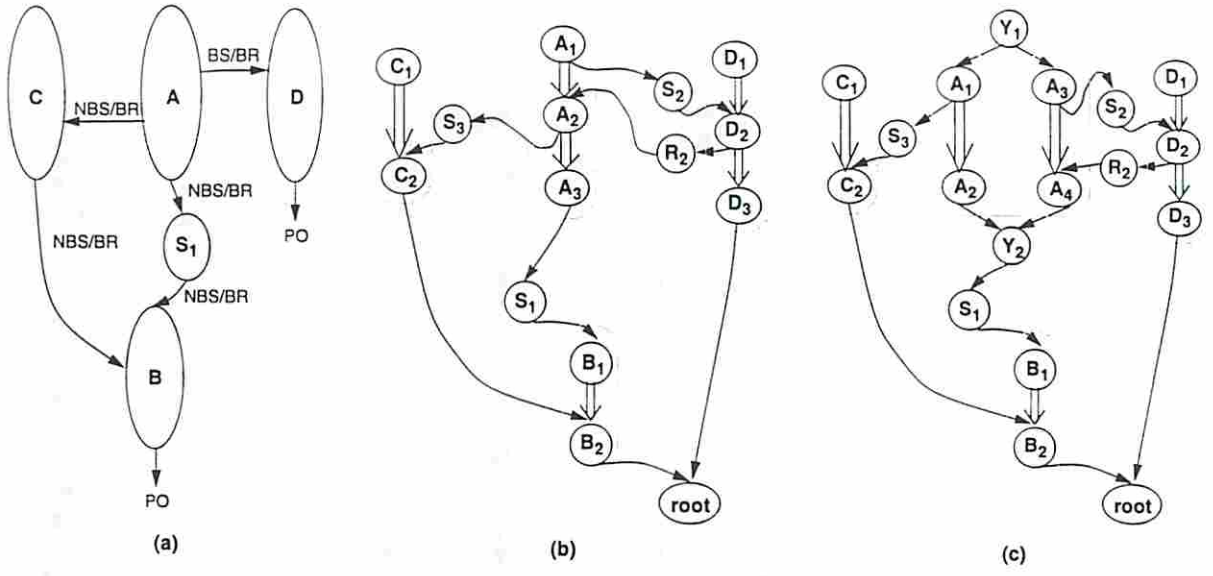

Figure 3.2: Example to illustrate decomposition of single and multi-threaded processes

When there are more than one intermediate communications for a given process (cf. process A in Fig. 3.2(a)), the decomposition of this process depends on whether it is single threaded or multi-threaded. For a single threaded process, the intermediate communication is referenced to the same time line as that of the thread. In this case, the appropriate transformation rules are applied to all intermediate communications at different points of the time line (cf. Fig. 3.2(b)). For a process with multi-threads, the intermediate 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. After that, the appropriate transformation rule is applied on each thread that serves the time line for the corresponding intermediate communication (cf. Fig. 3.2(c)).

Chapter 4

MILP Formulation for the Scheduling

After the decomposition of the original task graph, some (computational) process P_i are decomposed into a number 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 mapping to the same programmable processor type with different processor utilization factors as two 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 the 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 needed for the MILP formulations:

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

$$S_{i,j} \in Z^+ \cup \{0\} \text{ is the start time for (sub)process } P_{i,j}.$$

The following are the constants used. $d_{i,j,k}$ 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:

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

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

Precedence Constraint:

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

Type Consistency Constraint: $x_{i,j,k} = x_{i,1,k}, \ \forall k, \ \forall i, \ \forall 1 < j \leq U_i$

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

Chapter 5

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.

5.1 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. 5.2. The corner points on those curves are non-inferior points. A point is inferior to another point if both its cost and delay are 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 runnung on that type of hardware. In case the processor is shared among multiple processes, the delay cost of each process must account for the overhead of context switching.

In this report, 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 should include 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.

5.2 Simple task graphs

For a task graph without re-convergent fanout and with only end/begin type communications, the algorithm used in [CP96] can be directly used without going through the process decomposition phase. This algorithm would then produce the optimal hardware/software mapping for a tree-like 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. 3.1(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.

5.3 Complex task graphs

Handling task graphs with processes that have re-convergent fanout and use intermediate 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 5.3.1 *The scheduling of a generalized task graph which includes intermediate communication with blocking communication mechanism (i.e. Problem 1.1) is* NP-complete.

Proof by restriction [GJ79].

The intermediate communication among coarse-grain processes is possible and occurs frequently. Using the original dynamic programming in [CP96], we may get some point on the curve of a node with re-converging inputs. This point may results 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. 5.4). In addition, using the original algorithm in [CP96], during the post-order graph traversal, we may drop some points that actually lead to

the optimal solution (cf. Example 5.3.0.0.1).

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 5.3.0.0.1 *Here we use an example to show why binning is necessary to obtain the correct solution to our problem.*

Suppose in Fig. 5.2(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. 5.2(a). Each subprocess A_1, A_2 and process C can be mapped to either implementation E or F.

*Using dynamic programming (*post-order *traversal) without binning, we get a final accumulated curve at PO with each point annotated with the implementation type of A_1 and A_2 as shown in 5.2(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 (*post-order *traversal) with binning, we get two curves at PO as shown in Fig. 5.2(c), each curve is tagged with the implementation used for both A_1 and A_2. There is no type inconsistent solution here. Furthermore, point (9,46) in curve tagged by $A = E$ in Fig. 5.2(c) is missing from the solution obtained by using the dynamic programming without binning. The reason is that this point were inferior to point (9, 45) and hence got dropped. Furthermore, applying dynamic programming (*pre-order, *recursively) with timing constraint = 13 on PO, we get a solution $A_2 = F$, and the timing constraint passed to A_1 will be $13 - 6 = 7$. This will pick point $A_1 = E$ in the curve of A_1 , which is again a type inconsistent solution. Under the same timing constraint but using dynamic programming with binning, we get a solution both $A_1 = F$ and $A_2 = F$.* □

Creating the binning strings for each node in the task graph

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

Definition 5.3.0.0.1 *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 5.3.0.0.2 *[MR88] 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. 5.1, 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 report, 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 [CP96], the post-order and pre-order traversal can be performed on the individual PO's sequentially for the min-cost solution under timing constraint.

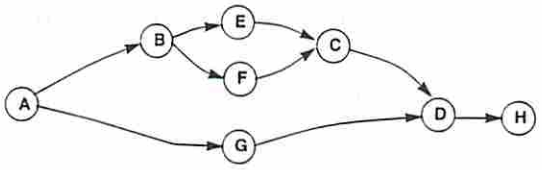

Figure 5.1: Example to show the definition of re-convergence

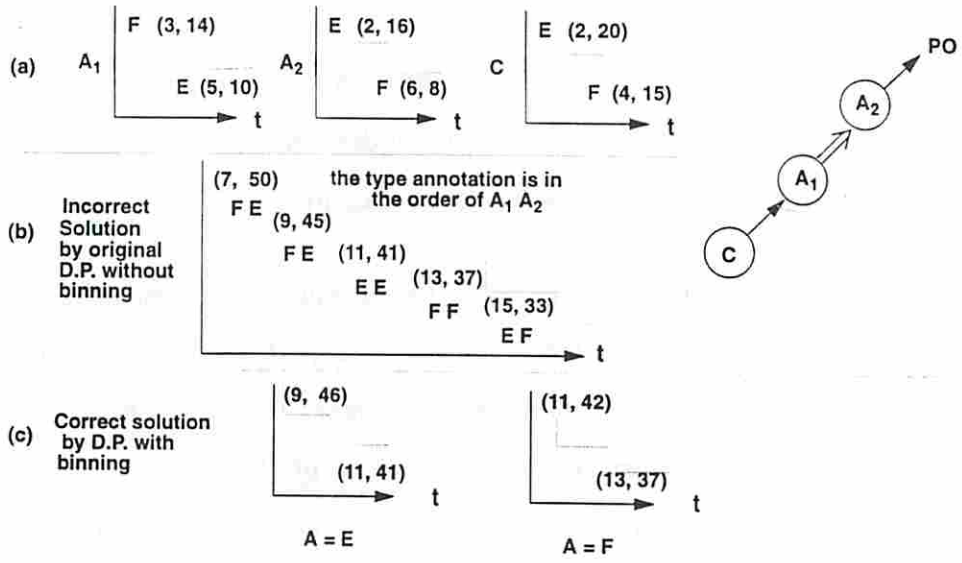

Figure 5.2: Example to show why binning is required during D.P.

However, in this report, 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 5.3.0.0.1 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. 5.3(a). 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. If we visited a node for the 2nd 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 [CLR90] 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. 5.3(b).

Theorem 5.3.0.0.1 *Suppose a process B is decomposed into subprocesses B_1, B_2, B_3, ..., 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, ..., C_m with the precedence relationships $C_1 \prec C_2 \prec C_3 \prec \ldots, \prec C_m$. Let D_1, D_2, ..., D_m denote the re-convergent nodes of C_1, C_2, ..., C_m, respectively. Note that D_i's are in general sets of nodes. We also denote the set of all nodes which lie on the re-convergent fanout paths from C_i to D_i as T_i. Then we have $T_1 \supseteq T_2 \supseteq T_3 \supseteq ,\ldots, \supseteq T_m$.*[1]

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 5.3.0.0.2 *Suppose a process A is decomposed into A_1, A_2, ... ,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.*

Example 5.3.0.0.3 *Suppose in Fig. 5.5, A_1 is a subprocess decomposed from process A. In Fig. 5.5(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. 5.5(b), Y lies some path from A_1 to C, therefore, Y needs the ID of A in its binning string.*

(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)

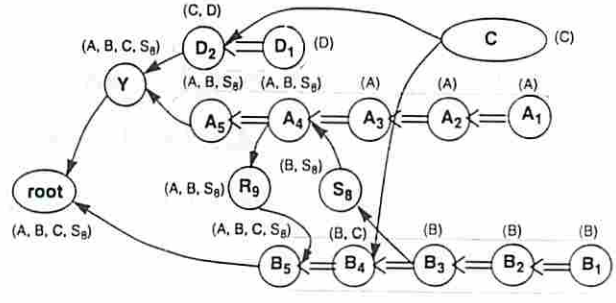

Figure 5.3: Pseudo code for creating binning strings

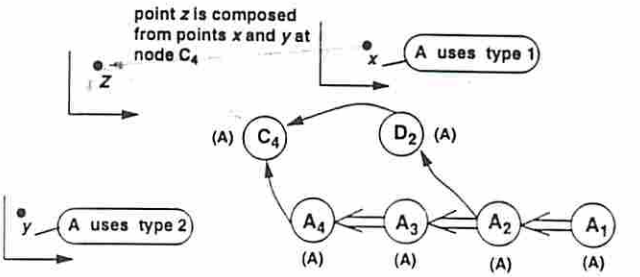

Figure 5.4: Fig. to be used in Theorem 5.3.0.0.2

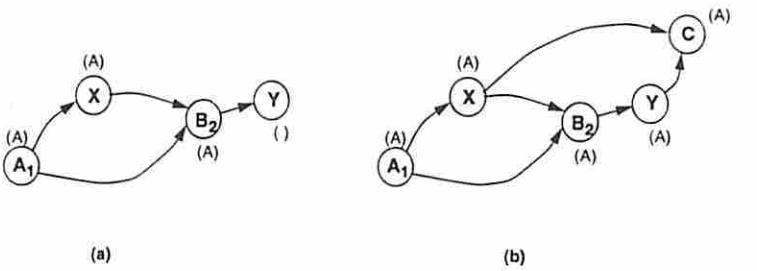

Figure 5.5: Two similar situations on node Y which need different binning strings (shown within the parentheses)

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

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 will expand the dimension of the bins of that child node by duplicating the same 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 will 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, B, C) and its parent X has binning string (A, B), then we will do a

[1] All proofs are omitted to save space

superimpose followed by lower bound operation on the curves of bins corresponding to tags $(A = E, B = E, C = E)$ and $(A = E, B = E, C = F)$ to obtain the unified curve for the new tag $(A = E, B = E)$. Similar operation is needed for all other combinations of A and B implementations.

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 in order 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 root 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.

Pre-order traversal

Pre-order traversal begins at the root of the decomposed task graph and proceeds toward the leaves. Consider a node, say 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 node X with a child Z. 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 5.3.0.0.1 *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. 5.6(b) with un-merged 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. 5.6. 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 will 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. 5.6(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 5.3.0.0.1 *The dynamic programming with binning (with our binning strings construction) solves Problem 1.1 optimally and satisfies all type consistency constraints.*

5.4 Complexity Analysis

For simple task graphs defined in Chapter 5.2, the task graphs remain the same after the initial process decomposition step. The same algorithm used in [CP96] can be used. The time complexity using dynamic programming algorithm for solving problems involved with this class of task graphs is pseudo-polynomial.

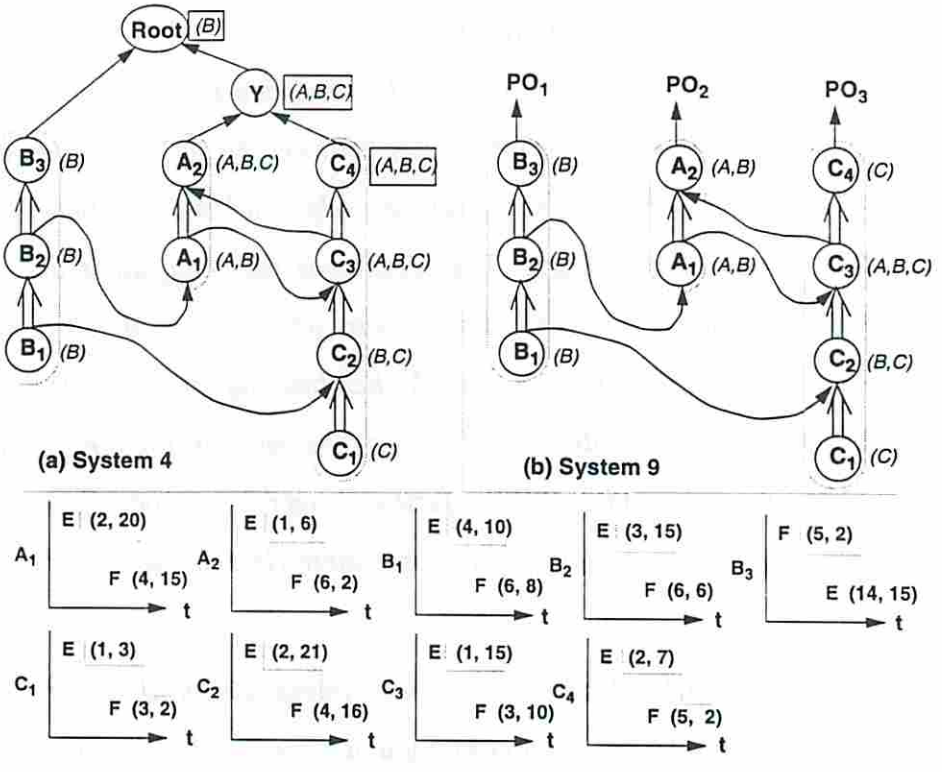

Figure 5.6: Figure to illustrate the need to merge $PO's$ into a single root

Let's scale delay values for all nodes (subprocesses) under different process mapping to become integers. Furthermore, let's denotes 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 $\mid \mathcal{I} \mid = n$ 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 energy-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 Chapter 5.2, 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 methods (including MILP or exhaustive search) to handle a task graph with intermediate communication. Both MILP and exhaustive search will have exponential complexity on n. We have also included an example (cf. Fig. 7.3) 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.

Chapter 6

Allocation and Binding

After the scheduling phase, the computational subprocesses decomposed from the same original coarse-grain process will be mapped to the same type of processor implementation or custom ICs. They have however not 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.

In general, processes are 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.

For the processes mapped to programmable units such as CPUs, DSPs, DMAs, or other controllers for communication, it is possible for them to share the same instance of the unit by TDM manner 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. Details are omitted to save space.

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.

Chapter 7

Experimental Results

Our dynamic programming with binning, named Codex-dp (for Co-design of Communicating Systems Using Dynamic Programming), and is implemented in C and tested on a number of circuits. Table. 7.1 shows the information about the examples used. Prakash1 and Prakash2 are taken from the two examples in [PP92], Yen is taken from [Wol95], and Bender is an example taken from [Ben96]. 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 sorrounding 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 HW units as well by running the process on the HW 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. 7.1 with deadline = 80.0(ms) taken from the CPM system [Ste95]. The task graph for example 2 is shown in Fig. 7.2 with deadline = 100.0(ms). The decomposed task graph for example 3 is shown in Fig. 5.6(a) with deadline = 18.0(ms) using our library (Curves shown in Fig. 5.6 do not correspond to our library modules). The task graph for example 4 is shown in Fig. 3.2(a), and its decomposed task graph is shown in Fig. 3.2(c) with deadline = 50.0(ms). The task graph for example 5 is shown in Fig. 7.3, its decomposed task graph is too large to be included in this report with deadline = 200.0(ms). This task graph is the sub-block performing the **voice activity detection** used in GSM (Group Special Mobile) [Ste95]. 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. 7.1, columns 2 and 3 show the values of m and n seen by Codex-dp. Columns 4 and 5 give the total number of computational and communication units needed after the allocation and binding using the modified bin packing algorithm. In column 6, we show the number of mixed analog and digital units used. Columns 7 and 8 give the CPU time used and the area cost (cm^2) used to implement the examples by Codex-dp. In column 9, we show the number of variables, inequalities and equations if the scheduling is done by the MILP formulation described in Section 4 assuming that each process has four possible implementations. In column 10, 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.

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 area cost and total computation time for example 5. As can be seen, decreasing the deadline constraint, increases the area cost of the optimal solution. Similar trend exists for all other examples.

Example	m	n	# of CPUs	# of C.U.	# of Mixed A/D	CPU time (s) of Codex-dp	total cost of Codex-dp	# of var. + ineq. + eq. for MILP	complex Exhausti
Prakash1	2	11	1	1	0	0.036	63.7	55+13+11	O(4
Prakash2	6	22	2	1	0	0.507	122.5	110+26+22	O(4
Yen	1	12	1	1	0	0.043	63.7	60+13+12	O(4
Bender	7	13	2	2	0	1.143	137.2	60+17+12	O(4
ex1	0	13	1	1	1	0.086	79.7	65+12+13	O(4
ex2	1	14	2	1	2	0.114	148.5	70+15+0	O(4
ex3	3	11	2	0	0	0.064	98.0	49+14+6	O(4
ex4	4	18	3	1	0	0.107	171.5	63+21+7	O(4
ex5-1	9	39	3	3	0	6.329	200.9	195+51+123	O(4
ex5-2	9	39	2	3	0	6.412	151.9	195+51+123	O(4
ex5-3	9	39	2	2	0	6.356	127.4	195+51+123	O(4

Table 7.1: Experimental results

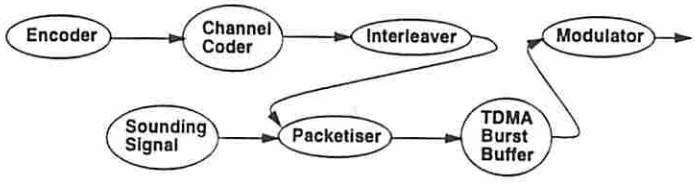

Figure 7.1: A very simple task graph with only end/begin communication

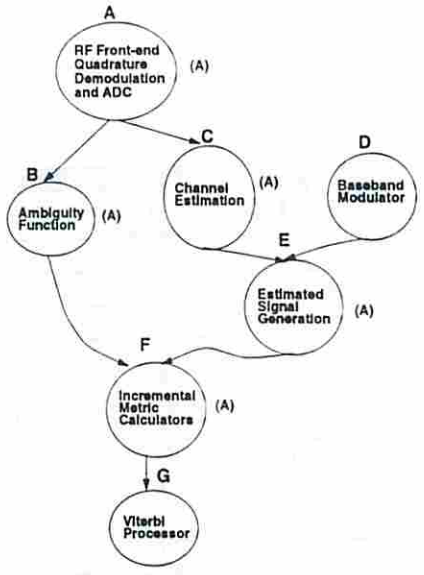

Figure 7.2: Task graph with only end/begin communication but with re-convergent fanout

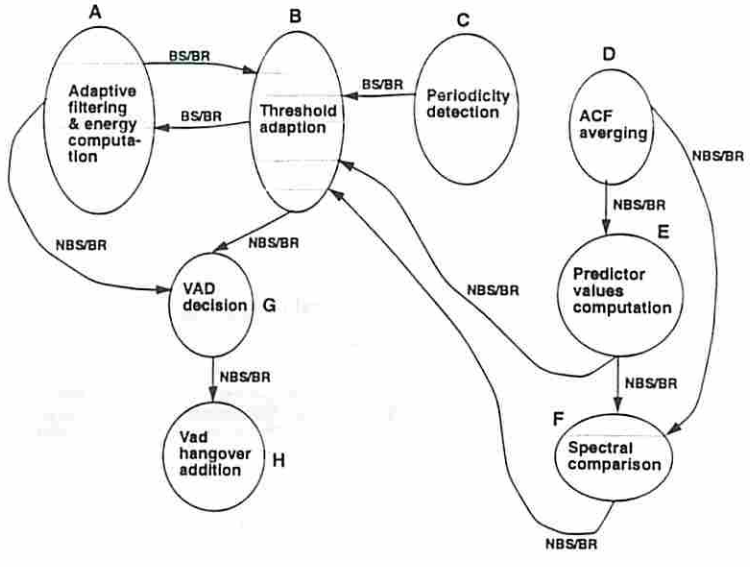

Figure 7.3: Task graph of Voice Activity Detection (VAD) used in the GSM system

Chapter 8

Conclusion

In this report, we presentd 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 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.

Bibliography

[Ben96] A. Bender. MILP Based Task Mapping for Heterogenous Multiprocessor Systems. In *Proceedings European Design Automation Conference*, 1996.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, McGraw-Hill, 1990.

[CP96] J.-M. Chang and M. Pedram. Energy Minimization Using Multiple Supply Voltages. In *Proceedings International Symposium for Low Power Electronic and Design*, August 1996.

[CSL91] W. Chu, C. Sit, and K. Leung. Task Response Time for Real-Time Distributed Systems with Resources Contentions. *IEEE Transactions on Software Engineering*, 10(17), October 1991.

[DH94] J. D'Ambrosio and X. Hu. Configuration-Level Hardware/Software Partitioning for Real-Time Embeded Systems. In *Proceedings IEEE International Workshop on Hardware/Software Codesign*, 1994.

[DIJ95] 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.

[DLJ97] B. P. Dave, G. Lakshminarayana, and N. Jha. Cosyn: Hardware-Software Co-Synthesis of Embeded Systems. In *Proceedings IEEE-ACM Design Automation Conference*, 1997.

[EHB93] R. Ernst, J. Henkel, and Th. Benner. Hardware/Software Co-Synthesis for Microcontrollers. *IEEE Design and Test Magazine*, 10(4), December 1993.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman and Company, 1979.

[GM92] R. Gupta and G. D. Micheli. System-level Synthesis using Re-programmable Components. In *Proceedings European Design Automation Conference*, 1992.

[JEO+94] 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.

[KM96] P. Knudsen and J. Madsen. PACE: A Dynamic Programming Algorithm for Hardware/Software Partitioning. In *Proceedings IEEE International Workshop on Hardware/Software Codesign*, 1996.

[MMS95] G. De Micheli and editor M. Sami. *Hardware/Software Co-Design, pages 22, 84, 85, 96, 217.* Kluwer Academic Publishers, 1995.

[MR88] 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.

[NG94] S. Narayan and D.D Gajski. Synthesis of System-Level Bus Interface. In *Proceedings European Design Automation Conference*, 1994.

[PP92] S. Prakash and A. Parker. SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 16, December 1992.

[PS89] D.-T. Peng and K. Shin. Static Allocation of Periodic Tasks with Precedence Constraints. In *Proceedings International Conference on Distributed Computing Systems*, 1989.

[Ste95] R. Steele. *Mobile Radio Communications.* Pentech Press, London, 1995.

[Wol95] T.-Y. Yen W. Wolf. Communication Synthesis for Distributed Embedded Systems. In *Proceedings IEEE International Conference on Computer-Aided Design*, 1995.

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

Jui-Ming Chang and Massoud Pedram

CENG 98-04

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-4458
February 1998

Abstract

In this paper, 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 lifetimes. The proposed algorithm has been implemented. Experimental results are reported and discussed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Previous work on system level synthesis has mainly focused on fine-grain hardware/software partitioning. Examples include VULCAN II [GM92] and COSYMA [EHB93]. 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. Furthermore, 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 on.

In contrast, coarse-grain partitioning does not decompose the initial specification into basic blocks and does not 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 designers' expertise more easily and can achieve a desired partitioning which satisfies some macroscopic choices more readily [MMS95]. Finally, the resulting solution has more logical coherence which facilitates the top-down design process and allows for debugging of the hardware/software.

Many of the coarse-grain partitioning algorithms 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 [Wol95] [Ben96] [DLJ97]) are directed acyclic graphs (*DAGs*) which 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, in general, the coarse-grain processes may communicate with each other at times other than the end or beginning of their lifetimes. We refer to this kind of communication as *intermediate* communication and to the task graph with intermediate communication as a *generalized task graph.* The problem we are trying to solve can then be stated as follows.

Problem 1.1 *Given a generalized task graph consisting of processes which communicate with each other by arbitrary 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 HW/SW resources so as to minimize the total area cost while satisfying a given deadline.*

Notice that the cost of mapping a process to a library unit (implementation) cannot be exactly determined because of the possibility of sharing the same unit between different processes. In such a case, we can use time-division multiplexing (TDM) to share the unit. The cost should account for this possibility and include the area and delay overhead associated with the *context switching.* We assume in this report 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.

A task graph with intermediate 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 be periodic. In such a case, we can handle the case that the period is greater than or equal to the deadline by performing the same schedule on every period.

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 of non-programmable computational units are ASICs and custom ICs. Examples of programmable communication units are FIFOs with controllers, bidirectional handshake controllers, DMA controllers, bus arbiters, or shared memory access and 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.*

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 only happen if the process lifetimes do not overlap or the processes are mutually exclusive.

Our algorithm consists of three major steps. First, processes are decomposed into subprocesses which perform parts of the required computation. The correct precedence relationships implied by the specified communication mechanism is 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). 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.), we therefore do not want the subprocesses decomposed from the same original coarse grain process to be mapped to different hardware units in the final solution. The scheduling is done 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 that will ensure the de-

composed subprocesses will be mapped not only to the same hardware type, but also to the same hardware instance. The allocation and binding will determine the sharing of hardware among all coarse-grain processes in the system.

The report is organized as follows. In Chapter 2, we summarize related work for coarse-grain HW/SW partitioning. In Chapter 3, we introduce our transformation rules for process decomposition. In Chapter 4, we present the MILP formulation for the Problem 1.1. In Chapter 5, we present our dynamic programming algorithm for solving Problem 1.1. In Chapter 6, we describe the allocation and binding algorithm to be used after the scheduling step. Experimental results and conclusion are provided in Sections 7 and 8, respectively.

Chapter 2

Related work

There are two published works [JEO+94] [KM96] on fine-grain hardware/software partitioning, which use dynamic programming. In these reports, 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 problem, and the dynamic programming formulation is used only to avoid repeated computations in their iterative procedure. Their formulation is therefore completely different from our dynamic programming formulation which traverses the generalized task graph globally.

There are other works which consider communicating processes with end/begin type communication. This kind of task graph is commonly used in real-time and distributed systems [CSL91] [PS89]. There have been several research publications on the coarse-grain HW/SW partitioning which handle task graphs with only end/begin type communication [Wol95] [DH94] [Ben96] [NG94]. 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. The above works use greedy heuristic [Wol95], branch and bound [DH94], or MILP [PP92] [Ben96] as their optimization techniques to explore the solution space.

For the task graphs consisting of only end/begin type of communication, the problem can be easily solved by a dynamic programming algorithm similar to the one used in [CP96]. For task graphs with intermediate 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 as will be explained later in this report.

The work reported in [DIJ95] for coarse-grain system synthesis, separates the synthesis of computational and communication processes into two distinct stages. In this case, it is very difficult to apply a timing constraint (deadline) on the system because part of time in the critical path is used to do the computation whereas another part is used to do the communication. In [Wol95], the gradient search method on the solution space 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 and relocate a message (communication process) from one bus to another and 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. Furthermore, our algorithm is based on dynamic programming and hence produces the optimal solution.

In summary, to our knowledge, the work reported in this report is the first work on coarse-grain process mapping that deals with complex communication among processes (i.e. blocking send/blocking receive, nonblocking send/blocking receive, blocking send/nonblocking receive or nonblocking send/nonblocking receive) using a novel dynamic programming approach.

Chapter 3

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 added. 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. 3.1(a), (b), (c) and (d), respectively. In Fig. 3.1, process S represents the actual process which sends the data from the sending process and R represents the process which sends the *reply* or *acknowledgment* from the receiving process. The arcs with single tail denote the precedence relationships between the nodes connected by that arc. 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 thus they should be finally mapped to the same hardware/software instance.

For a task graph with complex communications among processes, we follow the transformation rules shown in Fig. 3.1 to create to the decomposed task graph.

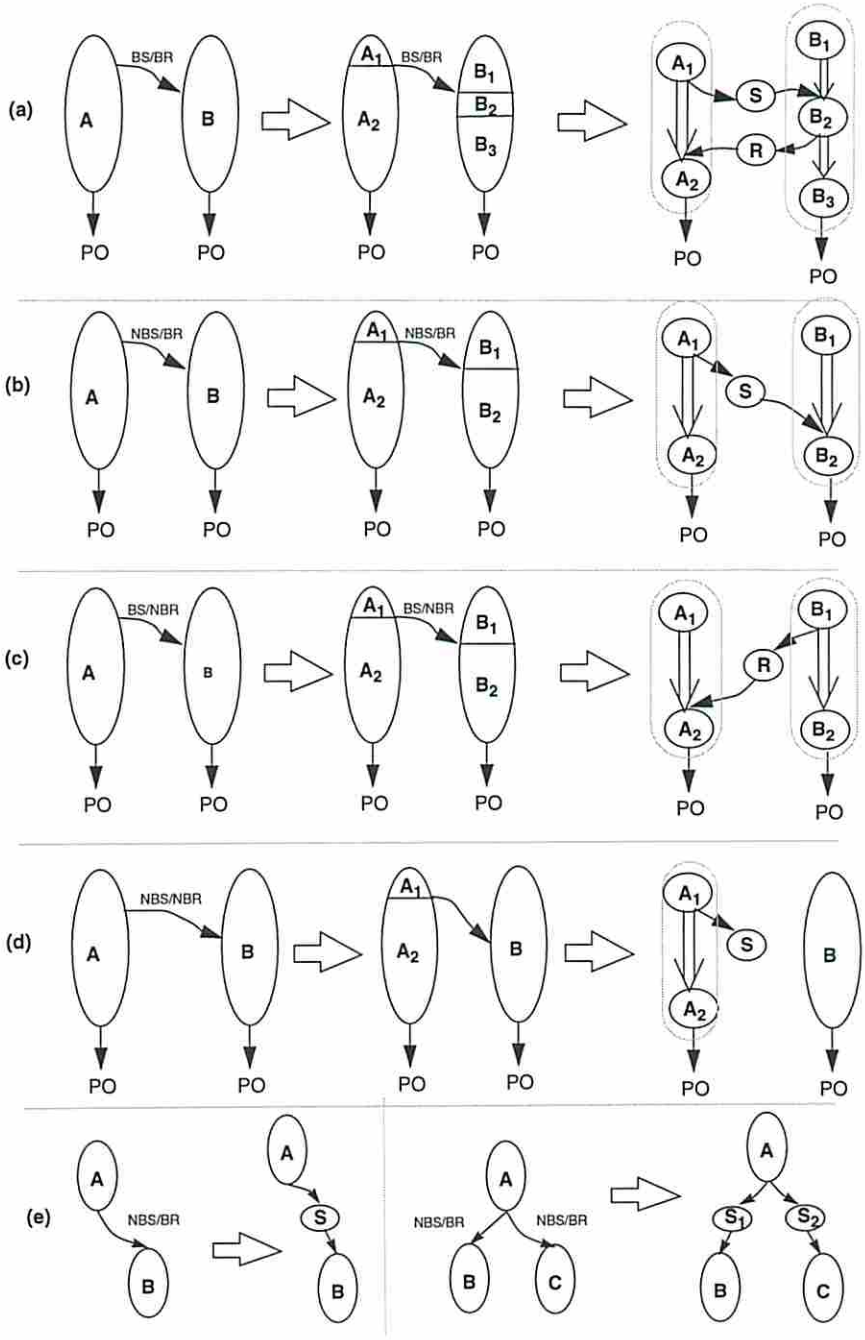

Figure 3.1: Decomposition of communicating processes with different type of intermediate communication

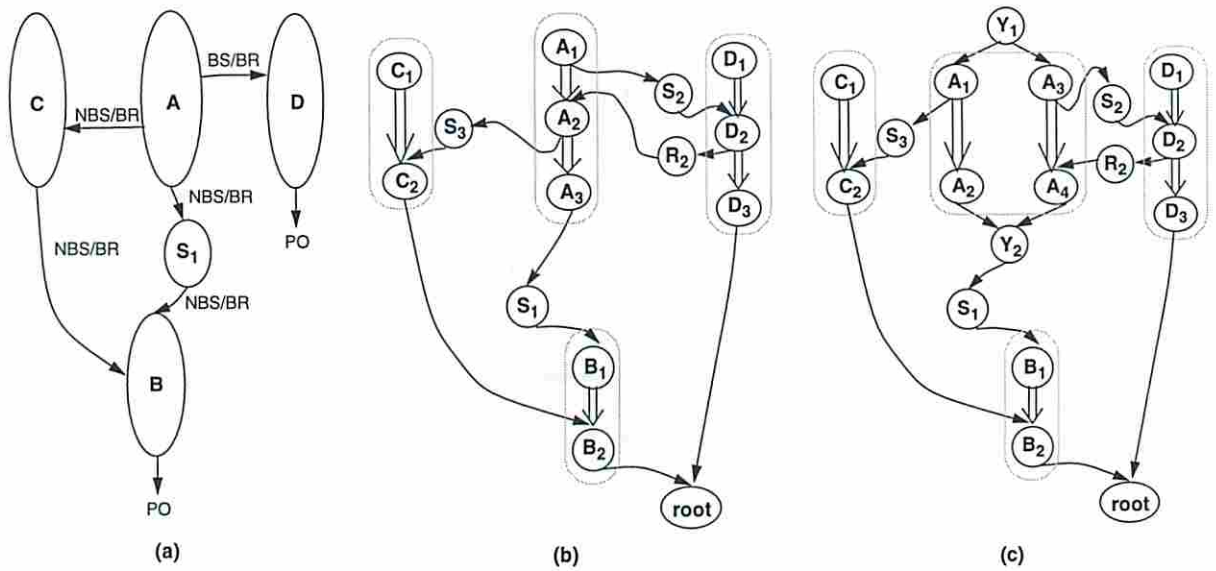

Figure 3.2: Example to illustrate decomposition of single and multi-threaded processes

When there are more than one intermediate communications for a given process (cf. process A in Fig. 3.2(a)), the decomposition of this process depends on whether it is single threaded or multi-threaded. For a single threaded process, the intermediate communication is referenced to the same time line as that of the thread. In this case, the appropriate transformation rules are applied to all intermediate communications at different points of the time line (cf. Fig. 3.2(b)). For a process with multi-threads, the intermediate 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. After that, the appropriate transformation rule is applied on each thread that serves the time line for the corresponding intermediate communication (cf. Fig. 3.2(c)).

Chapter 4

MILP Formulation for the Scheduling

After the decomposition of the original task graph, some (computational) process P_i are decomposed into a number 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 mapping to the same programmable processor type with different processor utilization factors as two 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 the 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 needed for the MILP formulations:

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

$$S_{i,j} \in Z^+ \cup \{0\} \text{ is the start time for (sub)process } P_{i,j}.$$

The following are the constants used. $d_{i,j,k}$ 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:

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

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

Precedence Constraint:

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

Type Consistency Constraint: $x_{i,j,k} = x_{i,1,k}, \ \forall k, \ \forall i, \ \forall 1 < j \leq U_i$

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

Chapter 5

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.

5.1 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. 5.2. The corner points on those curves are non-inferior points. A point is inferior to another point if both its cost and delay are 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 runnung on that type of hardware. In case the processor is shared among multiple processes, the delay cost of each process must account for the overhead of context switching.

In this report, 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 should include 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.

5.2 Simple task graphs

For a task graph without re-convergent fanout and with only end/begin type communications, the algorithm used in [CP96] can be directly used without going through the process decomposition phase. This algorithm would then produce the optimal hardware/software mapping for a tree-like 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. 3.1(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.

5.3 Complex task graphs

Handling task graphs with processes that have re-convergent fanout and use intermediate 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 5.3.1 *The scheduling of a generalized task graph which includes intermediate communication with blocking communication mechanism (i.e. Problem 1.1) is* NP-complete.

Proof by restriction [GJ79].

The intermediate communication among coarse-grain processes is possible and occurs frequently. Using the original dynamic programming in [CP96], we may get some point on the curve of a node with re-converging inputs. This point may results 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. 5.4). In addition, using the original algorithm in [CP96], during the post-order graph traversal, we may drop some points that actually lead to

the optimal solution (cf. Example 5.3.0.0.1).

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 5.3.0.0.1 *Here we use an example to show why binning is necessary to obtain the correct solution to our problem.*

Suppose in Fig. 5.2(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. 5.2(a). Each subprocess A_1, A_2 and process C can be mapped to either implementation E or F.

*Using dynamic programming (*post-order *traversal) without binning, we get a final accumulated curve at PO with each point annotated with the implementation type of A_1 and A_2 as shown in 5.2(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 (*post-order *traversal) with binning, we get two curves at PO as shown in Fig. 5.2(c), each curve is tagged with the implementation used for both A_1 and A_2. There is no type inconsistent solution here. Furthermore, point (9,46) in curve tagged by $A = E$ in Fig. 5.2(c) is missing from the solution obtained by using the dynamic programming without binning. The reason is that this point were inferior to point (9, 45) and hence got dropped. Furthermore, applying dynamic programming (*pre-order, *recursively) with timing constraint = 13 on PO, we get a solution $A_2 = F$, and the timing constraint passed to A_1 will be $13 - 6 = 7$. This will pick point $A_1 = E$ in the curve of A_1 , which is again a type inconsistent solution. Under the same timing constraint but using dynamic programming with binning, we get a solution both $A_1 = F$ and $A_2 = F$.* □

Creating the binning strings for each node in the task graph

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

Definition 5.3.0.0.1 *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 5.3.0.0.2 *[MR88] 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. 5.1, 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 report, 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 [CP96], the post-order and pre-order traversal can be performed on the individual PO's sequentially for the min-cost solution under timing constraint.

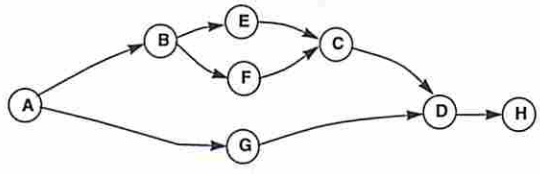

Figure 5.1: Example to show the definition of re-convergence

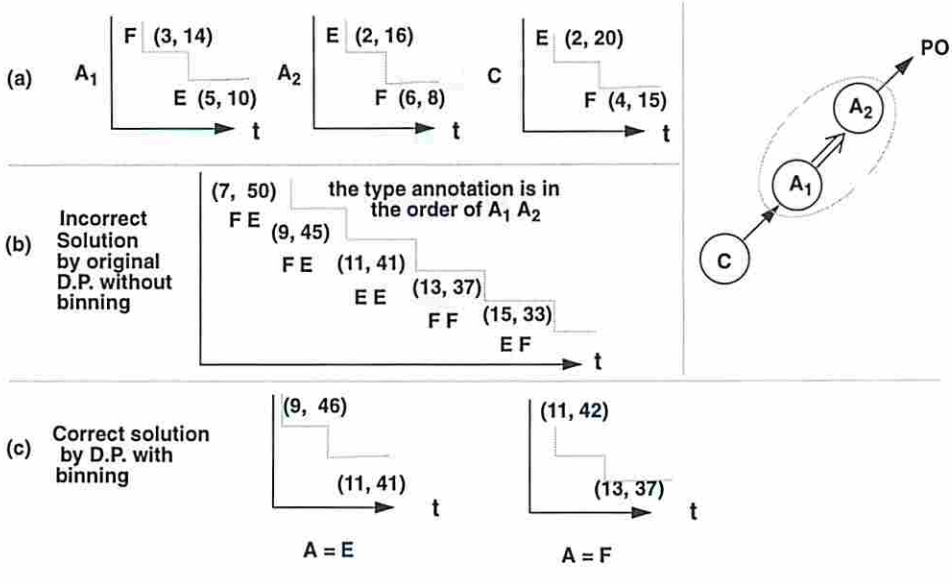

Figure 5.2: Example to show why binning is required during D.P.

However, in this report, 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 5.3.0.0.1 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. 5.3(a). 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. If we visited a node for the 2nd 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 [CLR90] 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. 5.3(b).

Theorem 5.3.0.0.1 *Suppose a process B is decomposed into subprocesses B_1, B_2, B_3, ..., 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, ..., C_m with the precedence relationships $C_1 \prec C_2 \prec C_3 \prec \ldots, \prec C_m$. Let D_1, D_2, ..., D_m denote the re-convergent nodes of C_1, C_2, ..., C_m, respectively. Note that D_i's are in general sets of nodes. We also denote the set of all nodes which lie on the re-convergent fanout paths from C_i to D_i as T_i. Then we have $T_1 \supseteq T_2 \supseteq T_3 \supseteq ,\ldots, \supseteq T_m$.*[1]

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 5.3.0.0.2 *Suppose a process A is decomposed into A_1, A_2, ... ,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.*

Example 5.3.0.0.3 *Suppose in Fig. 5.5, A_1 is a subprocess decomposed from process A. In Fig. 5.5(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. 5.5(b), Y lies some path from A_1 to C, therefore, Y needs the ID of A in its binning string.*

(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)

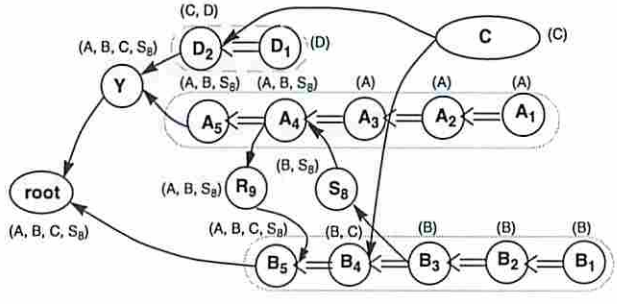

Figure 5.3: Pseudo code for creating binning strings

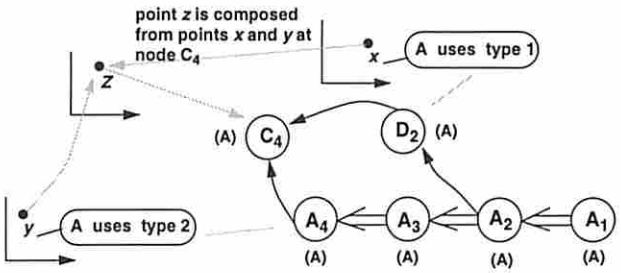

Figure 5.4: Fig. to be used in Theorem 5.3.0.0.2

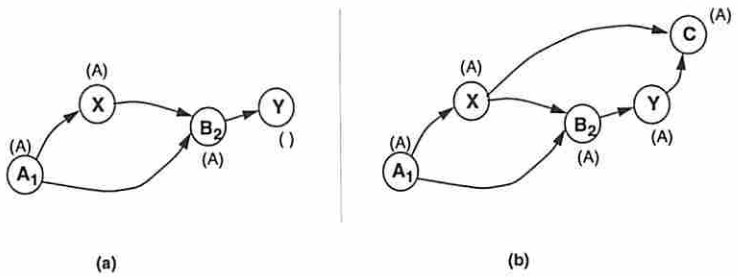

Figure 5.5: Two similar situations on node Y which need different binning strings (shown within the parentheses)

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

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 will expand the dimension of the bins of that child node by duplicating the same 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 will 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, B, C) and its parent X has binning string (A, B), then we will do a

[1] All proofs are omitted to save space

superimpose followed by lower bound operation on the curves of bins corresponding to tags $(A = E, B = E, C = E)$ and $(A = E, B = E, C = F)$ to obtain the unified curve for the new tag $(A = E, B = E)$. Similar operation is needed for all other combinations of A and B implementations.

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 in order 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 root 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.

Pre-order traversal

Pre-order traversal begins at the root of the decomposed task graph and proceeds toward the leaves. Consider a node, say 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 node X with a child Z. 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 5.3.0.0.1 *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. 5.6(b) with un-merged 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. 5.6. 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 will 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. 5.6(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 5.3.0.0.1 *The dynamic programming with binning (with our binning strings construction) solves Problem 1.1 optimally and satisfies all type consistency constraints.*

5.4 Complexity Analysis

For simple task graphs defined in Chapter 5.2, the task graphs remain the same after the initial process decomposition step. The same algorithm used in [CP96] can be used. The time complexity using dynamic programming algorithm for solving problems involved with this class of task graphs is pseudo-polynomial.

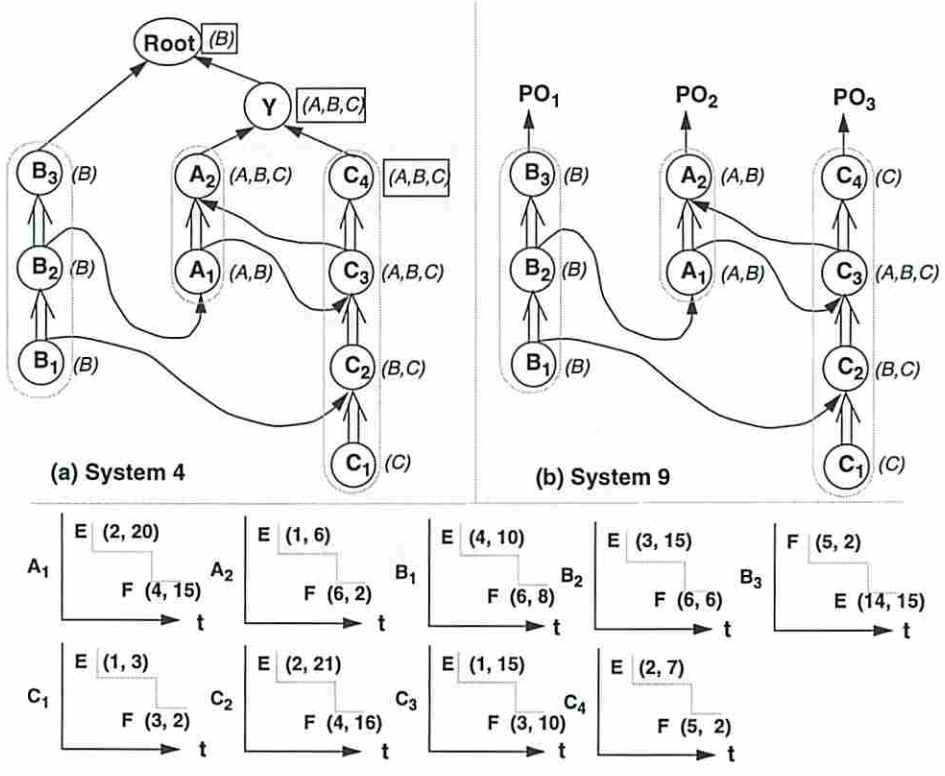

Figure 5.6: Figure to illustrate the need to merge $PO's$ into a single root

Let's scale delay values for all nodes (subprocesses) under different process mapping to become integers. Furthermore, let's denotes 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 $\mid \mathcal{I} \mid = n$ 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 energy-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 Chapter 5.2, 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 methods (including MILP or exhaustive search) to handle a task graph with intermediate communication. Both MILP and exhaustive search will have exponential complexity on n. We have also included an example (cf. Fig. 7.3) 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.

Chapter 6

Allocation and Binding

After the scheduling phase, the computational subprocesses decomposed from the same original coarse-grain process will be mapped to the same type of processor implementation or custom ICs. They have however not 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.

In general, processes are 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.

For the processes mapped to programmable units such as CPUs, DSPs, DMAs, or other controllers for communication, it is possible for them to share the same instance of the unit by TDM manner 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. Details are omitted to save space.

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.

Chapter 7

Experimental Results

Our dynamic programming with binning, named Codex-dp (for Co-design of Communicating Systems Using Dynamic Programming), and is implemented in C and tested on a number of circuits. Table. 7.1 shows the information about the examples used. Prakash1 and Prakash2 are taken from the two examples in [PP92], Yen is taken from [Wol95], and Bender is an example taken from [Ben96]. 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 sorrounding 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 HW units as well by running the process on the HW 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. 7.1 with deadline = 80.0(ms) taken from the CPM system [Ste95]. The task graph for example 2 is shown in Fig. 7.2 with deadline = 100.0(ms). The decomposed task graph for example 3 is shown in Fig. 5.6(a) with deadline = 18.0(ms) using our library (Curves shown in Fig. 5.6 do not correspond to our library modules). The task graph for example 4 is shown in Fig. 3.2(a), and its decomposed task graph is shown in Fig. 3.2(c) with deadline = 50.0(ms). The task graph for example 5 is shown in Fig. 7.3, its decomposed task graph is too large to be included in this report with deadline = 200.0(ms). This task graph is the sub-block performing the **voice activity detection** used in GSM (Group Special Mobile) [Ste95]. 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. 7.1, columns 2 and 3 show the values of m and n seen by Codex-dp. Columns 4 and 5 give the total number of computational and communication units needed after the allocation and binding using the modified bin packing algorithm. In column 6, we show the number of mixed analog and digital units used. Columns 7 and 8 give the CPU time used and the area cost (cm^2) used to implement the examples by Codex-dp. In column 9, we show the number of variables, inequalities and equations if the scheduling is done by the MILP formulation described in Section 4 assuming that each process has four possible implementations. In column 10, 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.

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 area cost and total computation time for example 5. As can be seen, decreasing the deadline constraint, increases the area cost of the optimal solution. Similar trend exists for all other examples.

Example	m	n	# of CPUs	# of C.U.	# of Mixed A/D	CPU time (s) of Codex-dp	total cost of Codex-dp	# of var. + ineq. + eq. for MILP	complexity usi Exhaustive Sea
Prakash1	2	11	1	1	0	0.036	63.7	55+13+11	$O(4^{11})$
Prakash2	6	22	2	1	0	0.507	122.5	110+26+22	$O(4^{22})$
Yen	1	12	1	1	0	0.043	63.7	60+13+12	$O(4^{12})$
Bender	7	13	2	2	0	1.143	137.2	60+17+12	$O(4^{13})$
ex1	0	13	1	1	1	0.086	79.7	65+12+13	$O(4^{13})$
ex2	1	14	2	1	2	0.114	148.5	70+15+0	$O(4^{14})$
ex3	3	11	2	0	0	0.064	98.0	49+14+6	$O(4^{9})$
ex4	4	18	3	1	0	0.107	171.5	63+21+7	$O(4^{8})$
ex5-1	9	39	3	3	0	6.329	200.9	195+51+123	$O(4^{22})$
ex5-2	9	39	2	3	0	6.412	151.9	195+51+123	$O(4^{22})$
ex5-3	9	39	2	2	0	6.356	127.4	195+51+123	$O(4^{22})$

Table 7.1: Experimental results

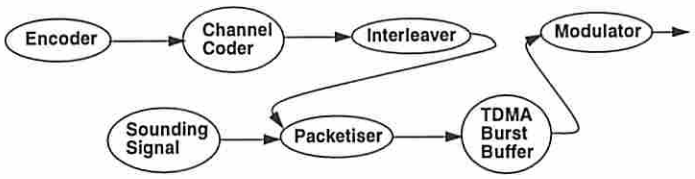

Figure 7.1: A very simple task graph with only end/begin communication

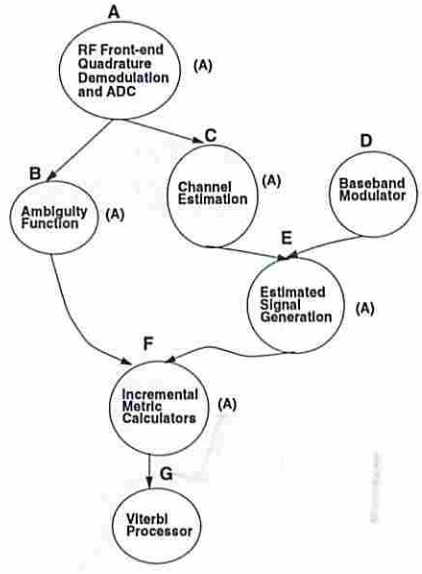

Figure 7.2: Task graph with only end/begin communication but with re-convergent fanout

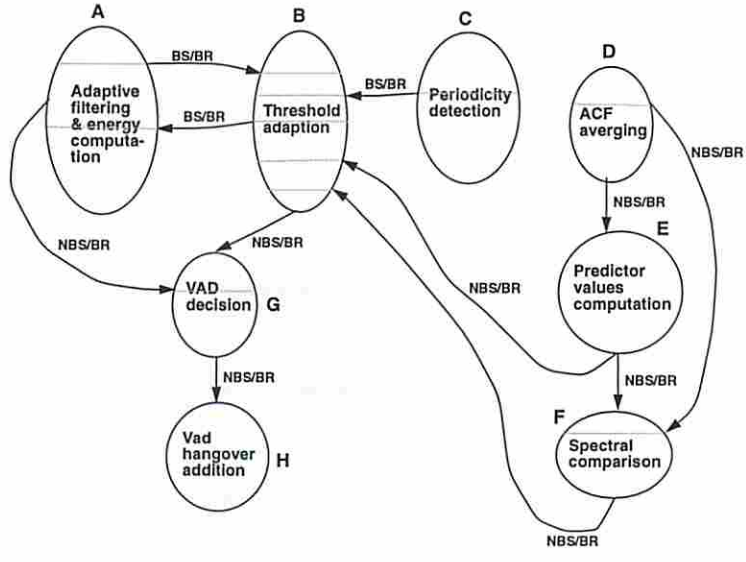

Figure 7.3: Task graph of Voice Activity Detection (VAD) used in the GSM system

Bibliography

[Ben96] A. Bender. MILP Based Task Mapping for Heterogenous Multiprocessor Systems. In *Proceedings European Design Automation Conference*, 1996.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, McGraw-Hill, 1990.

[CP96] J.-M. Chang and M. Pedram. Energy Minimization Using Multiple Supply Voltages. In *Proceedings International Symposium for Low Power Electronic and Design*, August 1996.

[CSL91] W. Chu, C. Sit, and K. Leung. Task Response Time for Real-Time Distributed Systems with Resources Contentions. *IEEE Transactions on Software Engineering*, 10(17), October 1991.

[DH94] J. D'Ambrosio and X. Hu. Configuration-Level Hardware/Software Partitioning for Real-Time Embeded Systems. In *Proceedings IEEE International Workshop on Hardware/Software Codesign*, 1994.

[DIJ95] 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.

Chapter 8

Conclusion

In this report, we presentd 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 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.

[PP92] S. Prakash and A. Parker. SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 16, December 1992.

[PS89] D.-T. Peng and K. Shin. Static Allocation of Periodic Tasks with Precedence Constraints. In *Proceedings International Conference on Distributed Computing Systems*, 1989.

[Ste95] R. Steele. *Mobile Radio Communications.* Pentech Press, London, 1995.

[Wol95] T.-Y. Yen W. Wolf. Communication Synthesis for Distributed Embedded Systems. In *Proceedings IEEE International Conference on Computer-Aided Design*, 1995.

[DLJ97] B. P. Dave, G. Lakshminarayana, and N. Jha. Cosyn: Hardware-Software Co-Synthesis of Embeded Systems. In *Proceedings IEEE-ACM Design Automation Conference*, 1997.

[EHB93] R. Ernst, J. Henkel, and Th. Benner. Hardware/Software Co-Synthesis for Microcontrollers. *IEEE Design and Test Magazine*, 10(4), December 1993.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman and Company, 1979.

[GM92] R. Gupta and G. D. Micheli. System-level Synthesis using Re-programmable Components. In *Proceedings European Design Automation Conference*, 1992.

[JEO+94] 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.

[KM96] P. Knudsen and J. Madsen. PACE: A Dynamic Programming Algorithm for Hardware/Software Partitioning. In *Proceedings IEEE International Workshop on Hardware/Software Codesign*, 1996.

[MMS95] G. De Micheli and editor M. Sami. *Hardware/Software Co-Design, pages 22, 84, 85, 96, 217.* Kluwer Academic Publishers, 1995.

[MR88] 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.

[NG94] S. Narayan and D.D Gajski. Synthesis of System-Level Bus Interface. In *Proceedings European Design Automation Conference*, 1994.