5.

Hardware-Software Partitioning

The partitioning algorithm is described in this chapter. The main objective is to implement in hardware the minimum that allows the design to meet its constraints. By using an approach which tries to maximise the use of software components, hardware processors area is decreased at the expense of software host memory, which is considered acceptable. Real-time constraints are the primary concern here. The final design must be able to guarantee the schedulability of all tasks in the system. Hence, the partitioning method is developed to work closely with, in fact involve, the scheduling algorithm.

5.1 Algorithm Overview

Two definitions are key to the proper understanding of the partitioning approach proposed (this work is partially presented in [24]).

1. *Process Feasibility*: A process implementation is considered feasible if, in the absence of resource constraints, its related time constraints are met.

2. *System Feasibility*: A whole system is considered feasible if all its processes are feasible and there exists a scheduling such as the time constraints are met.

The approach used deals separately with each feasibility level described above. The main advantage of this method is the fact that, as it will be seen later, process feasibility can be guaranteed without the need of actually analysing the whole scheduling. A task-level partitioning method is being used which is divided in two main steps (see Figure 5.1):

- Pre-partitioning:
  Performed just once; it is devised to perform a process-feasibility based partitioning.
- System Partitioning:
  Performed interspersed with process scheduling; it takes into account system feasibility, as defined above.

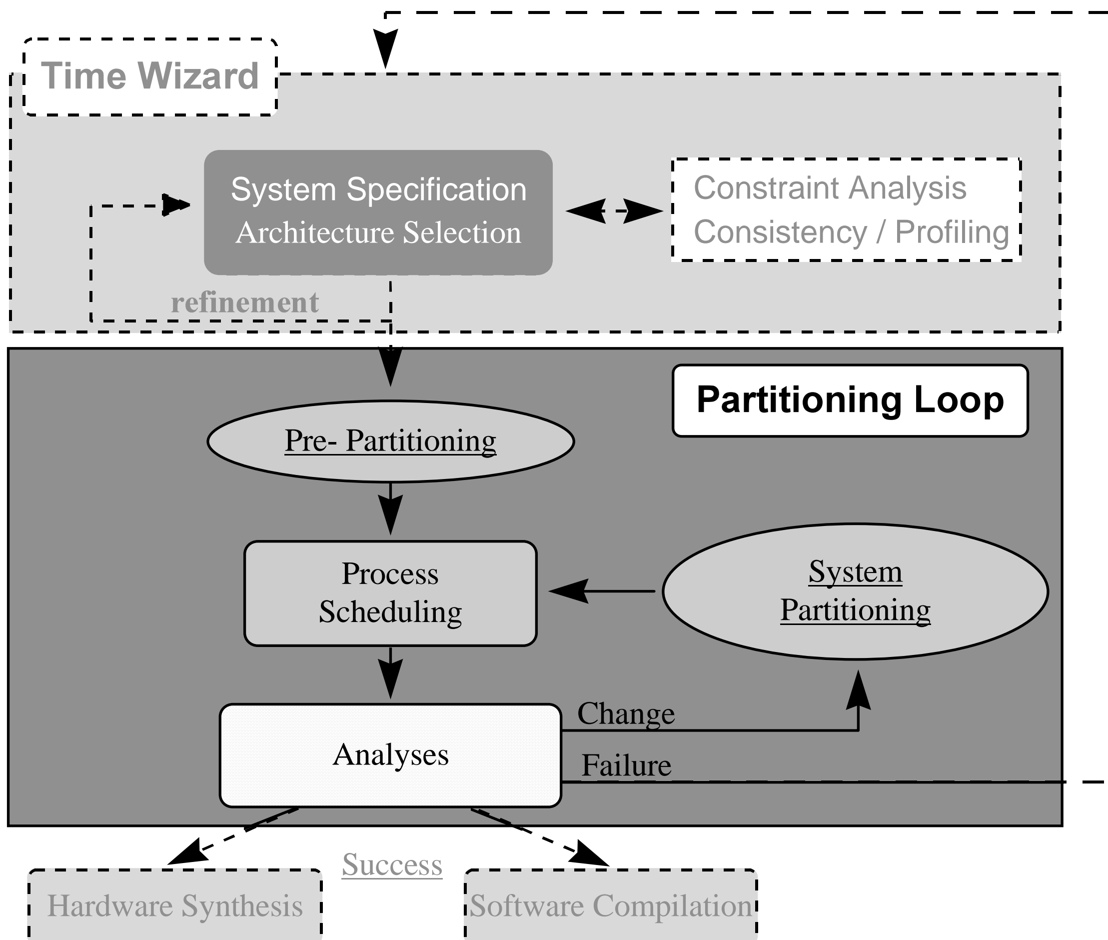


Figure 5.1. Partitioning process (highlighted)

5.2 Pre-Partitioning

In this phase, segment timing constraints are analysed first individually and then inside process nets. Initially all relations and constraints are reset to express the original requirements (see Figure 5.2). Then, a test similar to the consistency check (see section 4.4.2) is performed in each task and whenever possible inconsistent tasks are moved to hardware. This means that if a task i has $r'_i + SwWCET_i > d'_i$, and its implementation is not fixed in software (see Section 3.3.2), then i is moved to hardware in order to decrease its WCET, so that $r'_i + HwWCET_i \leq d'_i$,. However, if the implementation is fixed in software or if $r'_i + HwWCET_i > d'_i$, then the design has failed since either the task cannot be moved to hardware or even if moved it does not meet its deadline, and so there is no possible solution for this set of constraints and temporal parameters.

The function PrecedenceAdjustOrPartition() attempts to adjust precedence relations in the same way performed by function AdjustRelations() in Section 4.4.1, i.e., if $i \mapsto j$ then $r'_j = \max(r'_i + c_i, r'_j)$, and $d'_i = \min(d'_j - c_j, d'_i)$. However, if i causes an adjustment in j (or vice-versa) which leads to an inconsistent timing constraint, such as $r'_j + c_j > d'_j$, then PrecedenceAdjustOrPartition() creates a *partitioning list* (PL) consisting of i and j, which become candidates to be moved to hardware. Each PL contains a set of segments which are candidates to be moved to partitioning, since their inter-dependencies have induced timing constraint inconsistencies among them. Therefore, at least one of those segments should be moved to hardware in order to speed up its execution time and make it possible for all segments to meet their deadlines. Both tasks i and j are put in a PL because if i causes j to become inconsistent, then j causes i to become inconsistent too. This can be shown by noticing that if $i \mapsto j$ then $r'_i + c_i + c_j$ must be smaller or equal to d'_j, otherwise j is inconsistent. Considering that i causes j to be inconsistent, then $r'_i + c_i + c_j > d'_j \Rightarrow r'_i + c_i > d'_j - c_j \Rightarrow i$ cannot end before j starts. But from the initial condition $i \mapsto j$ and so i should finish before j starts, which implies that i is also inconsistent.

In fact, since precedence relations are transitive, if $i \mapsto j$ and $j \mapsto k$, and i's timing constraints lead to inconsistent constraints in j, what in turn lead to inconsistencies in k, then a PL is created containing i, j, and k, and so on for any other segment in the same conditions.

Similar to the function PrecedenceAdjustOrPartition(), the function Exclusion-AdjustOrPartition() attempts to adjust exclusion relations in the same way performed by function AdjustRelations() in Section 4.4.1, i.e., if $i \otimes j \wedge r'_j + c_j + c_i > d'_i$ then the relation $i \mapsto j$ can be used if $r'_i + c_i + c_j \leq d'_j$. However, if $r'_i + c_i + c_j > d'_j$, then neither i can precede j nor vice-versa, and so i and j are put in a PL so that one of them can be moved to hardware in order to execute faster and its execution time becomes smaller enough to make i and j consistent, otherwise no feasible solution exists and the design has failed.

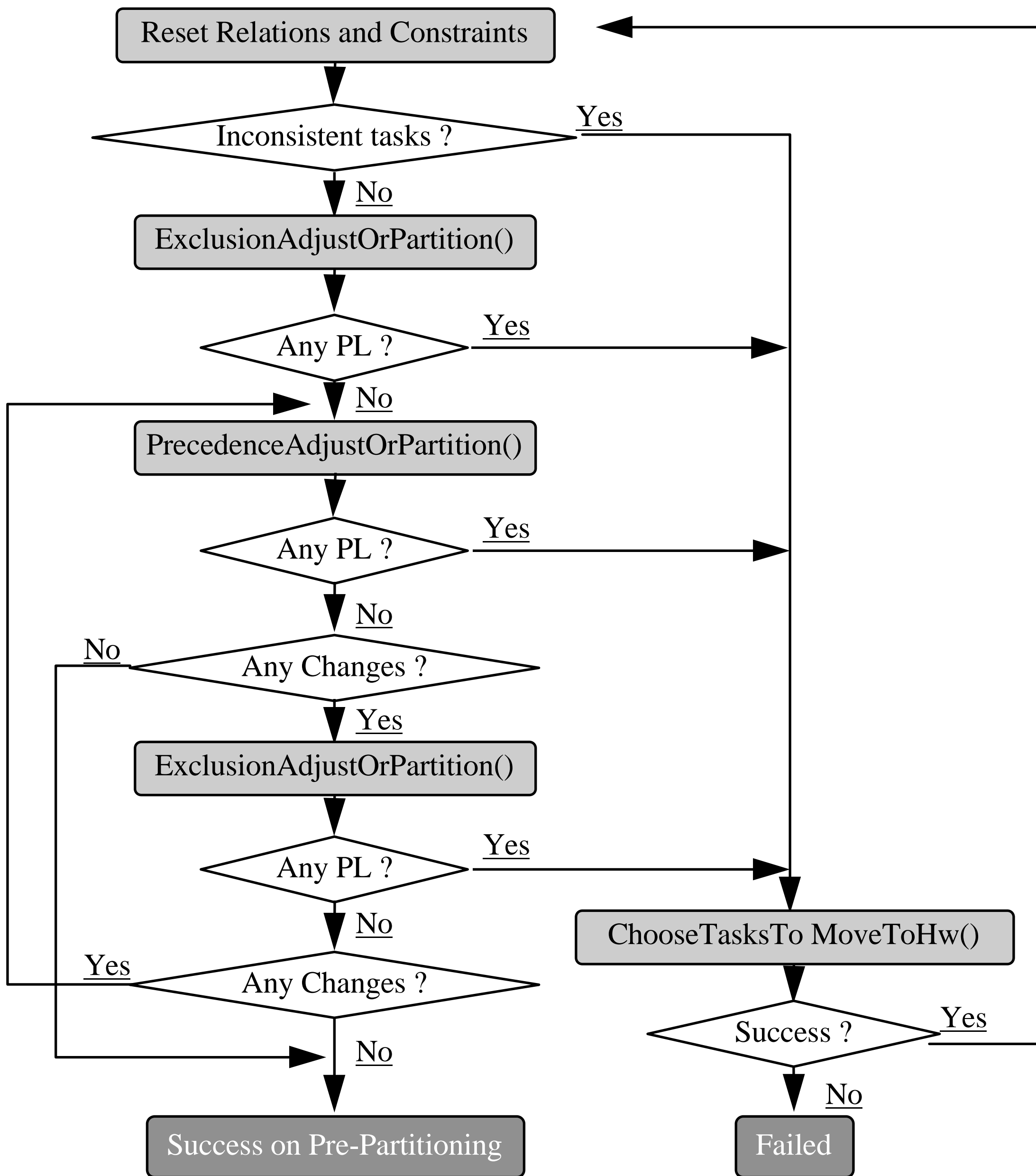


Figure 5.2. Pre-partitioning algorithm

PLs contain sets of segments whose inter-dependencies induce timing constraint inconsistencies among them. Therefore, from each PL at least one segment needs to be

moved to hardware in order to improve the chances of satisfying the constraints, which is done by the function ChooseTasksToMoveToHw(). Every time tasks are moved to hardware the algorithm re-starts from the beginning in order to re-consider the initial timing constraints without adjustments plus the temporal characteristics of the tasks moved to hardware. Notice that when a task is moved to hardware, all its temporal instances (i.e., segments) are moved as well. So, suppose that there exist two PLs $\{a_1, b, c\}$ and $\{a_2, e\}$, with a_1 and a_2 being temporal instances of the same task A. If A is moved to hardware, then the condition that at least one segment needs to be moved from each PL is satisfied, since a_1 and a_2 are both moved.

The functions PrecedenceAdjustOrPartition() and ExclusionAdjustOrPartition() are executed in a loop until no more adjustments are made or the partitioning has failed. The algorithm stops either if at any iteration no adjustments are made in any relation or constraint (again, similar to the function AdjustRelations()), or if the function Choose-TasksToMoveToHw() (see Section 5.4) fails to move tasks to hardware. Each function is explained in detail in the sub-sections to follow.

5.2.1 Consistency-Based Partitioning

This step analyses each task (not segments) considering their original timing constraints, i.e., before performing relation adjustments. The algorithm can be seen below (the notation used here is defined in previous chapters).

$\forall i \in \mathbf{PS}$:

1) if μ_i is a software host $\wedge\ r_i + SwWCET_i > d_i$:

   if $\neg\ FixedImp_i \Rightarrow$ move i to hardware.

   otherwise no feasible solution exists.

2) if μ_i is a hardware processor $\wedge\ r_i + HwWCET_i > d_i \Rightarrow$ no feasible solution exists.

5.2.2 Partitioning Lists Creation

As discussed in section 4.4.1, adjustment relations can be performed in two cases:

1. $\forall\ i, j \in \mathbf{PS} \mid i \otimes j \wedge r'_i + c_i + c_j > d'_j \Rightarrow j$ cannot precede i and so the relation $i \mapsto j$ can be used instead of $i \otimes j$.
2. $\forall\ i, j \in \mathbf{PS} \mid i \mapsto j \Rightarrow r'_j = \max(r'_i + c_i, r'_j)$, $d'_i = \min(d'_j - c_j, d'_i)$.

However, the new relations and constraints existing after the adjustments may present timing inconsistencies. Suppose that two segments A and B have $r_A = r_B = 0$, $c_A = c_B = 60$, $d_A = d_B = 100$. Although A and B are individually consistent, if A $\mapsto$ B, the adjustments using (2) would lead to $r'_B = 60$, and $d'_A = 40$. Also, if A $\otimes$ B and the adjustment in (1) is used the same problem would result.

The functions ExclusionAdjustOrPartition() and PrecedenceAdjustOrPartition() attempt to perform the above mentioned adjustments. Whenever an adjustment leads to inconsistent segments, PLs are created as detailed in the next sections.

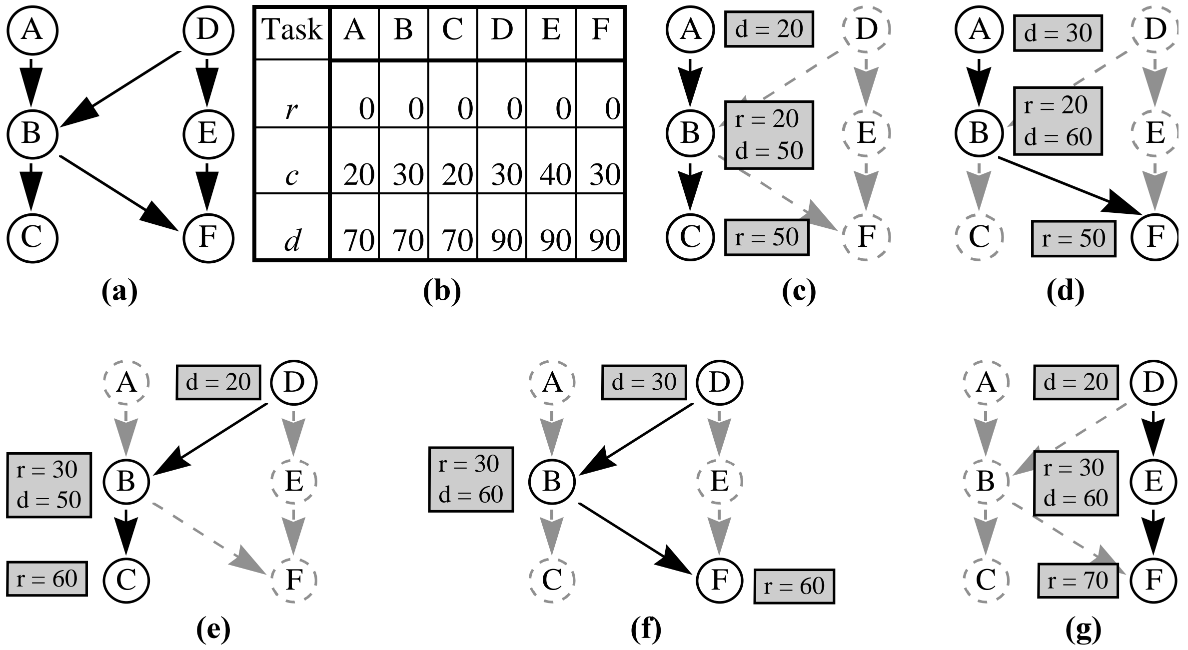
5.2.2.1 Precedence-Based Partitioning Lists

This step takes into account the influence of precedence relations on time constraints and is implemented by PrecedenceAdjustOrPartition().

A *precedence graph* is one where vertices represent segments and edges represent the precedence relations among segments. A traversal algorithm is used to adjust constraints and test for inconsistencies on each possible path in the precedence graphs existing on the design (a specification may have more than one precedence graph). The complete algorithm is shown below.

1. Identify non-preceded segments, i.e., those which have no predecessors.
2. For each path starting on a non-preceded segment:
   - 2.1. Walk through the path adjusting the constraints.
   - 2.2. Test for inconsistencies: create a PL containing all inconsistent segments on that path.
   - 2.3. Return constraints to previous values (this avoids interference from a path search to another).
3. If there are no PLs adjust all constraints based on precedence.

The example of Figure 5.3 is used to illustrate the algorithm execution. The precedence graph to be searched can be seen in Figure 5.3.a and the initial timing constraints in Figure 5.3.b. A and D are non-preceded segments. Figure 5.3 (c) to (g) highlight the paths searched, and the constraint adjustments associated with each path. In Figure 5.3 (e) and (g) all segments in the highlighted paths become inconsistent, which results in the creation of two PLs: {B, C, D}, and {D, E, F}.



Task	A	B	C	D	E	F
r	0	0	0	0	0	0
c	20	30	20	30	40	30
d	70	70	70	90	90	90

Figure 5.3. Application of the function PrecedenceAdjustOrPartition()

5.2.2.2 Exclusion-Based Partitioning Lists

Two different approaches are used here: one employed only first time the function ExclusionAdjustOrPartition() is called, and another used when called from inside the pre-partitioning loop (see Figure 5.2). In the first call, mutual exclusion relations are tested so that $\forall\ i, j \mid i \otimes j$, if $r_i + c_i + c_j > d_j \wedge r_j + c_i + c_j > d_i \Rightarrow$ PL = $\{i, j\}$.

When called from the pre-partitioning loop the algorithm proceeds in the following manner:

$\forall\ i, j \mid i \otimes j \wedge r'_i + c_i + c_j > d'_j \wedge r'_j + c_i + c_j > d'_i$:

1. Add relation $i \mapsto j$, use the algorithm to create PLs based on precedence relations, remove relation $i \mapsto j$.
2. Add relation $j \mapsto i$, use the algorithm to create PLs based on precedence relations, remove relation $j \mapsto i$.

At any iteration, if no PLs are added and $\exists\ i \mid r'_i + c_i + c_j > d'_j$ but $r'_j + c_i + c_j \leq d'_i$, then the relation $j \mapsto i$ is added and the pre-partitioning loop proceeds. This corresponds to the relation adjustment shown in Chapter 3.

The reasoning behind the approach used in the first call is simple. If two segments i and j mutually exclude each other and $r'_i + c_i + c_j > d'_j \wedge r'_j + c_i + c_j > d'_i$, then even if one

of them was allocated to another identical processor (with same WCET) the inconsistency would persist. So, the only solution is to reduce the WCET of *i* or *j*, which may be achieved by implementing it in hardware.

Nevertheless, this approach cannot be used during the remaining iterations since it could lead to wrong results. This occurs due to the fact that it does not take into account any changes made in previous iterations. For example, suppose a set of segments {A, B, C} with the characteristics described in Figure 5.4 (a) and (b). Initially all segments are allocated to the same software host *sw1* and the only segment able to move to hardware is C. No new precedence relations are added to the design by applying ExclusionAdjustOr-Partition().

$A \otimes B$
$A \mapsto C$
$B \mapsto C$

Task	A	B	C
µ	sw1	sw1	sw1
FixedImp	✓	✓	
r'	0	0	30
SwWCET	25	35	50
HwWCET	n/a	n/a	10
d'	90	90	90

⇒

Task	A	B	C
µ	sw1	sw1	sw1
FixedImp	✓	✓	
r'	0	0	45
SwWCET	25	35	50
HwWCET	n/a	n/a	10
d'	40	40	90

(a) **(b)** **(c)**

Figure 5.4. Application of the function ExclusionAdjustOrPartition()

The next step is to apply PrecedenceAdjustOrPartition(), which performs the adjustments shown in Figure 5.4.c, but generates no PLs. Since there were changes, ExclusionAdjustOrPartition() is applied again. If the same algorithm used in the first call is used again, a PL containing A and B is created. Given that A and B have fixed implementations and so cannot be moved to hardware, it would seem that there is no possible solution for the partitioning. However, it can be observed that if C is moved to hardware the system becomes feasible.

Consider again the example of Figure 5.4. Using the approach proposed for calls from inside the pre-partitioning loop, the relation $A \mapsto B$ is added and the function PrecedenceAdjustOrPartition() is called. The PL {A, B, C} is created. Then the relation $A \mapsto B$ is removed and the same procedure is used with the addition of $B \mapsto A$, which

results in the creation of the PL (A, B, C} again. Since A and B have fixed implementation, C is chosen to be moved to hardware and the feasible solution shown in Figure 5.5 is achieved.

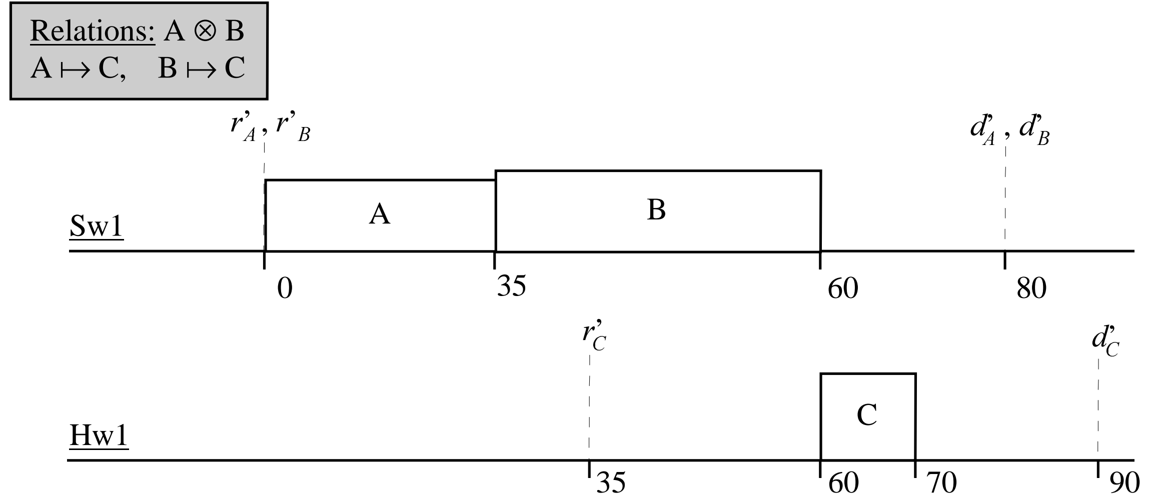


Figure 5.5. Feasible partitioning

5.3 System Partitioning

A similar approach to the creation of set **Z**(l) is used here. When searching for a feasible schedule using the branch-and-bound algorithm, the *last late segment* (*lls*) is used instead of the latest segment, where if **LS** is the set of segments with lateness > 0, then $lls = \{ i \mid e_i = \max\{ e_j \mid j \in \mathbf{LS}\}\}$. If there is a late segment *lls*(n) at a node n, a set called *partitioning set* - **SPar**(n, *lls*(n)) - is created which, similar to **Z**(l), contains all segments which influences this lateness. The argument here is the same used for lateness improvement discussed in section 4.5.

If a successor node n has a schedule which is not feasible but no segment in **SPar**(*pn, lls*(*pn*)) is late, where *pn* means parent node, the following may happen:

1. If *lls*(n) was not late in the root node, then during the branch-and-bound search some segment in **SPar**(*pn, lls*(*pn*)) caused *lls*(n) to become late. So, **SPar**(n, *lls*(n)) is created and combined to **SPar**(*pn, lls*(*pn*)) to continue the search, i.e., **SPar**(n, *lls*(n)) = **SPar**(n, *lls*(n)) ∪ **SPar**(*pn, lls*(*pn*)).
2. Otherwise, **SPar**(n, *lls*(n)) is created but not combined to **SPar**(*pn, lls*(*pn*)).

Similar to pre-partitioning, this approach also enables several partitioning sets to be created during any branch-and-bound scheduling attempt, since each node may have a new **SPar**(n, $lls(n)$).

<u>CreateSPar(*n*, *lls*(*n*))</u>

{

1. $lls(n) \in$ **SPar**(n, $lls(n)$)
2. $\forall\ i$:

   $\exists\ k \in$ **SPar**(n, $lls(n)$) |

   $e_i = s_k \wedge r'_k < e_i \wedge$

   $$\begin{cases} \mu_i = \mu_k \wedge (\ d'_i \le d'_k \vee i \mapsto k \vee i \otimes k \vee i\ \nabla\ k\ ) \\ \vee \\ \mu_i \ne \mu_k \wedge (\ i \mapsto k \vee i \otimes k\ ) \end{cases}$$

   $\Rightarrow i \in$ **SPar**(n, $lls(n)$)
3. If lateness($lls(n)$) < 0 in the root node

   $\Rightarrow$ **SPar**(n,$lls(n)$) = **SPar**(n,$lls(n)$) $\cup$ **SPar**(pn, $lls(pn)$).

}

At the end of a non-successful branch-and-bound scheduling iteration, several partitioning sets may have been created throughout the search tree nodes. The partitioning set which $lls(n)$ appears earlier in the root node schedule is chosen for partitioning, since this means that a feasible schedule was found for segments in partitioning sets of predecessor nodes (which appear later in the schedule). In case of ties, the node with minimum lateness is chosen.

The total time units (*TTU*) that need to be freed on $lls(n)$ software host is calculated and used to define a lower bound on the number of tasks that need to be moved to hardware in order to make feasible the schedule of the segments in **SPar**(n, $lls(n)$). TTU is calculated as follows:

TTU = $Rsp + Csp - Dsp$, where: $Rsp = \min\{\ r'_i \mid i \in$ **SPar**$(n,\ lls(n))\ \}$

$Dsp = \min\{\ d'_i \mid i \in$ **SPar**$(n,\ lls(n))\ \}$

$Csp = \sum c_i \mid i \in$ **SPar**$(n,\ lls(n))$

If TTU > 0, tasks are moved to hardware so that $\forall i \in$ **SPar**(n, $lls(n)$) and i is moved to hardware, $\sum SwWCET_i \ge$ TTU. If TTU ≤ 0, only one task is moved to hardware.

5.4 Cost Function

Time-based parameters are used as terms of the cost function, since the main objective is to find a feasible solution in regards to time constraints. Other authors have covered other aspects of partitioning costs, such as [8, 40].

Although segment information is used to define the candidates to be moved, the cost-function comparison is done at the task level. This is so because, as explained later, when a task is moved to hardware all its temporal instances are moved as well.

Let $nSeg_A$ be the number of segments derived from a task A that belong to PLs or to the partitioning set, depending on the current phase. For each task A which has a segment belonging to any PL, and A is not already in hardware and does not have a fixed implementation in software, calculate the partitioning parameters $Par1$ to $Par5$ as described below. They are used to evaluate the cost in moving a task to hardware, as it will be seen shortly.

$Par1_A$ is proportional to the improvement in moving task A to hardware, when considering all time freed by A in its former software host, i.e.:

$$Par1_A = SwWCET_A \,.\, nSeg_A$$

$Par2_A$ is proportional to the improvement in moving A to hardware, when considering only the time gain by decreasing $WCET_A$ (from $SwWCET_A$ to $HwWCET_A$). This is important when considering that, even though A has been moved to another processor, not all time freed in the former software host may be available for use due to mutual exclusion and precedence relations. So, $Par2_A$ is calculated as:

$$Par2_A = (SwWCET_A - HwWCET_A) \,.\, nSeg_A$$

$Par3_A$ is a measure of the potential parallelism among the segments derived from task A and segments derived from other tasks. The definitions below are necessary to understand how to calculate $Par3_A$:

- $D(i) = c_i \,/\, (d'_i - r'_i)$, is the probability of a segment i execute at any time unit within the period $[r'_i, d'_i)$.
- $OP(i, j)$ is the execution overlapping period between two segments i and j, and it is equal to $\min(d'_i, d'_j) - \max(r'_i, r'_j)$, if $r'_i < d'_j \wedge r'_j < d'_i$, otherwise it is zero.

- $PE(i,j) = D(i).D(j).OP(i, j)$, is the average number of time units during which i and j may be able to execute in parallel if either is moved to hardware.
- PET(i) is the average number of time units of i during which another segment j is executing in parallel. If this value is not bigger than $SwWCET_i$, then PET(i) = $\sum$ PE(i, j), otherwise it is equal to $SwWCET_i$. If $i \mapsto j \vee j \mapsto i \vee i \otimes j$, the result is further multiplied by $(SwWCET_i - HwWCET_i) / SwWCET_i$ since these segments cannot operate in parallel, and so j can only use any part of the execution time of i that become available.

All values are calculated using data from the root node, since data from other nodes are altered during the branch-and-bound scheduling and may not represent all possibilities. Finally, $Par3_A$ is calculated as:

$$Par3_A = \sum PET(i), \text{ where } i \text{ is a temporal instance of task } A.$$

$Par4_A$ takes into account the fact that in an unfeasible system scheduled by the earliest deadline algorithm, all segments satisfy release-time constraints, although some segments do not satisfy their deadline constraints (see eligibility rules on section 4.4.3). So, segments that are scheduled earlier have a greater effect when moved to hardware than other that come later. For example, suppose that segments i, j and k precede each other, and that all three are late. If k is moved to hardware, it may become consistent. However, this will have no effect on the lateness of i or j. On the other hand, if i were moved to hardware and $r'_j < e_i \wedge r'_j < e_j$, all three tasks would benefit from the change. PET(i) is used since it is necessary to consider if there exist another segment which may benefit from moving i to hardware.

During pre-partitioning, $Par4_A$ cannot be calculated by the above equation since no scheduling information exists at that stage. Therefore, during pre-partitioning release times (r'_i) are used instead of start times (s_i). $Par4_A$ is calculated as follows:

$$Par4_A = \sum s_i \,.\, \text{PET}(i), \text{ where } i \text{ is a temporal instance of task } A.$$

$Par5_A$ corresponds to the number of bytes transferred between task A and tasks in other processors. The reasoning for this parameter is that communication between tasks in the same software host is faster and less expensive in terms of design time and hardware cost than communication between processors.

$$Par5_A = \sum \text{nBytes}(A, i) \,.\, nSeg_A, \text{ with } \mu_i = \mu_A$$

The final cost of moving a task *A* to hardware is calculated as follows (where W1-5 are weights defined by the user):

$$CT_A = - \text{Par1}_A.\text{W1} - \text{Par2}_A.\text{W2} - \text{Par3}_A.\text{W3} - \text{Par4}_A.\text{W4} + \text{Par5}_A.\text{W5}$$

5.5 Moving tasks to hardware

There are two methods of moving tasks to hardware: automatic and manual. In automatic mode, tasks are selected based on the cost of moving them to hardware and the task with minimum value is chosen to move. In the pre-partitioning phase, one segment from each PL needs to be moved to hardware. So, one task at a time is moved, and all PLs are verified. If there are PLs which did not have any segment moved, other task is chosen. This continues until all PLs have at least one segment moved to hardware.

During system partitioning, tasks from the partitioning set are moved to hardware in turn until $\sum SwWCET_A \geq$ TTU, where *A* is a moved task.

In manual mode, the designer is presented with all tasks belonging to a PL or a **SPar**(*n, lls*(*n*)), depending on the current design phase. Each task is shown with its respective cost parameters and final value, and the task which has the minimum cost is displayed with an identification tag. The designer may accept the partitioner choice or refuse and choose another task to be moved.

There are two reasons for design failure when moving tasks to hardware. If during pre-partitioning there is a PL whose segments cannot be moved to hardware, or if during system partitioning it is impossible to make $\sum SwWCET_A \geq$ TTU, the design is not feasible. This may happen if all candidate tasks are already in hardware or if their implementation is fixed. Such a failure would mean that it is impossible to provide a feasible system with the initial requirements, and so the design must be changed.

5.6 Possible Results

Due to the precedence and exclusion relations which are the driving force during pre-partitioning, tasks that are moved to hardware during that phase imply the need for

reducing their execution times, i.e., moving those tasks to other processors is not important if their execution times are not reduced.

For example, if $i \mapsto j$ and $r_i = r_j = 0$, $c_i = c_j = 30$, $d_i = d_j = 50$, after constraint adjustments these values would become $r'_i = 0$, $d'_i = 20$, $r'_j = 30$, $d'_j = 50$, and both segments would not be feasible. If i is moved to another processor with the same execution time, the problem would persist. The only solution is to reduce i or j'sexecution time by moving one of them to a faster processor, which would mean a move to hardware. A faster software host could be used instead, but in this case the move would have to be done by the user, since the approach used here do not allow automatic moves across software hosts.

At the same time, system partitioning is concerned with problems derived from excessive software host utilisation. At this phase, the speed-up gain when moving a task to hardware is interesting but not strictly required. The parallelism acquired and the decrease of processor utilisation that occurs when moving tasks to other hardware processors is enough to solve the problems dealt with by the system partitioning algorithm. This could be used to help rapid prototyping of systems. After pre-partitioning, the specification can be changed so that all remaining tasks allocated to software hosts have their *HwWCET* made equal to their *SwWCET*. At the end of the partitioning, all tasks moved to hardware during system partitioning could be allocated to processors identical to their previous software hosts, and no special hardware would be necessary for those tasks.

Therefore, pre-partitioning can be related with execution speed-up whereas system partitioning could be solved by parallelism. Actually, this is not exactly so. In order to consider all segment inter-dependencies during pre-partitioning, mutual exclusion relations among more than two segments need to be considered. To do so, techniques such as clique partitioning [84] would have to be used to group tasks that mutually exclude each other, and permutation would be necessary to verify all possible scheduling ordering. However, this is left as future work since it involves only known algorithms.

The partial results obtained during each partitioning phase may help the designer in order to alter the specification and tailor it to specific purposes, such as rapid prototyping. These solutions are summarised in the table below. In the *Pre-partitioning* and *System Partitioning* columns, the possible results of each phase are shown. It is assumed that the initial architecture is composed of only one software host in order to assist with the comprehension of this table.

Pre-Partitioning	System Partitioning	Possible Solutions
All in software	No changes	All software / Uni-processor
All in software	Changes required	All software / Multi-processor or Sw-Hw / Uni-processor
Hardware required	No changes	Sw-Hw / Uni-processor
Hardware required	Changes required	Sw-Hw / Uni-processor or Sw-Hw / Multi-processor

Table 5.1. Possible Solutions

5.7 Example

A simple example is presented here in which the system specified in Figure 5.6 is partitioned. The original timing information is summarised on Table 5.2. The changes made in temporal requirements after executing the function ExclusionAdjustOrPartition() are shown in Figure 5.7 and Table 5.3 (all highlighted cells in the tables below refer to changes in constraints when compared to the original specification). Tasks {T11, T12, T21, T22, T23} are grouped in a PL and so are candidates to be moved to hardware. T21 is chosen and moved to hardware (here represented by processor Hw1, see Table 5.4), where its computation time is 10 time units. However, the scheduler fails to find a feasible solution with the current solution and task T0 becomes 20 time units late (see Figure 5.8). The system partitioning then occurs (Table 5.5) and T23 is moved to hardware (processor Hw2). The scheduler is executed again, producing the feasible result shown in Figure 5.9. Some intermediate steps were omitted from the explanation. A full description of the partitioning process in this example can be seen in the automatically generated report in Appendix D.

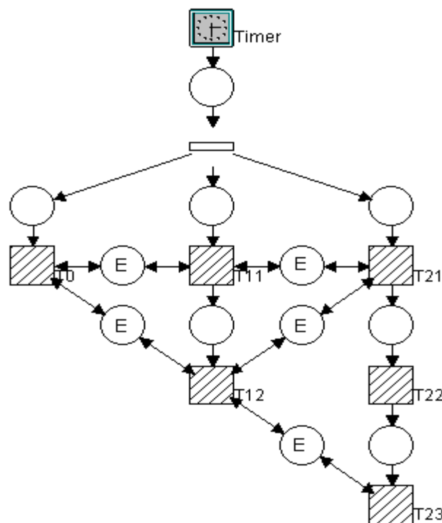


Figure 5.6. System specification

Task	T0	T11	T12	T21	T22	T23
µ	Sw	Sw	Sw	Sw	Sw	Sw
FixedImp	✓					
r	0	1	40	60	0	90
c(sw/hw)	50	40/20	10/10	50/10	20/10	50/30
d	151	51	91	140	140	140

Table 5.2. Tasks initial timing information

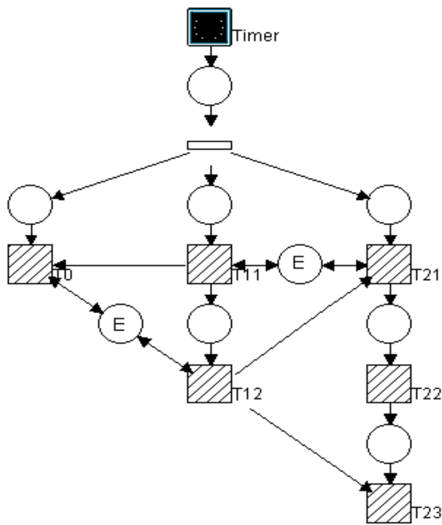


Figure 5.7. Adjusted Specification

Task	T0	T11	T12	T21	T22	T23
μ	Sw	Sw	Sw	Sw	Sw	Sw
r'	41	1	41	60	110	130
c	50	40	10	50	20	50
d'	151	10	20	70	90	140

Table 5.3. Release times and deadlines after adjustments

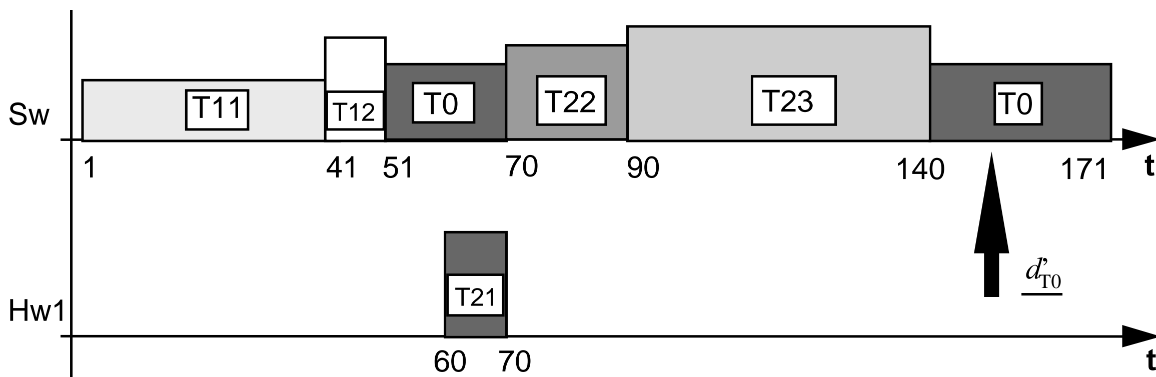


Figure 5.8. Schedule after pre-partitioning

Task	T0	T11	T12	**T21**	T22	T23
μ	Sw	Sw	Sw	**Hw1**	Sw	Sw
r'	51	1	41	60	70	90
c	50	40	10	10	20	50
d'	151	51	90	70	90	140

Table 5.4. Pre-partitioning. Values after adjustment

Task	T0	T11	T12	T21	T22	**T23**
μ	Sw	Sw	Sw	Hw1	Sw	**Hw2**
r'	51	1	41	60	70	90
c	50	40	10	10	10	30
d'	151	51	91	90	110	140

Table 5.5. System partitioning. Values after adjustment

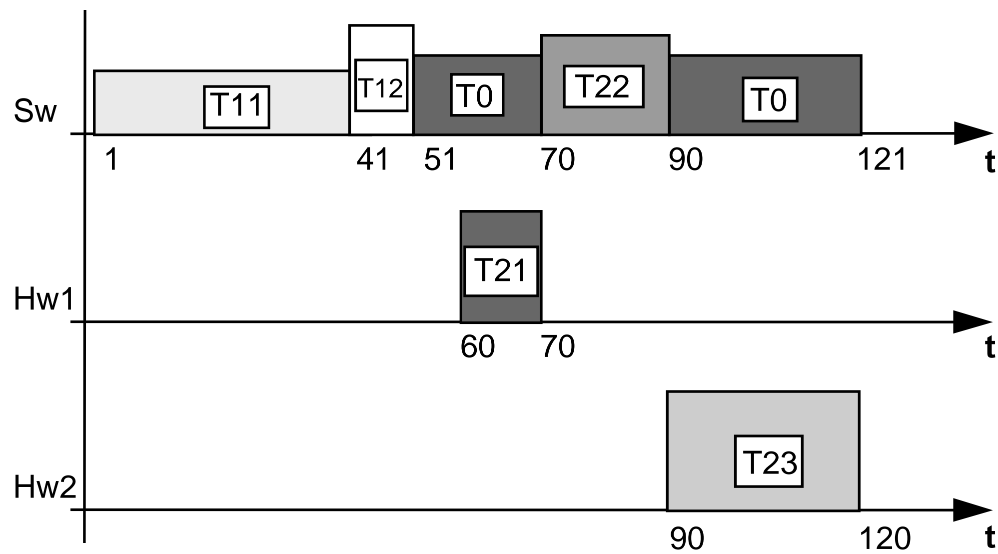


Figure 5.9. Final scheduling results

5.8 Summary

The techniques developed in this dissertation to provide automatic and manual partitioning when considering real-time issues were shown in this chapter. Partitioning is divided in two phases: pre-partitioning and system partitioning. Pre-partitioning deals with task's inter-dependencies, and is more concerned with attaining execution time speed-up. System partitioning is directed towards processor utilisation and achievement of parallelism, and works closely with the scheduling algorithm shown in Chapter 4.

The cost function parameters used were developed to represent time-related features such as speed-up, parallelism, and processor utilisation. Other parameters targeted to deal with area, power, etc., may be incorporated to the cost function but are out of the scope of the current work.

The design of systems based on pre-runtime scheduling techniques particularly benefits from the use of co-design since the problems related to providing resources to highly responsive sporadic tasks may be solved by moving those tasks to hardware. In general those tasks can be transformed so that the part that requires high responsiveness is implemented separately from the rest of the task. Since this part is generally small, the onus in moving this task to hardware would not be great.

The ability to work with uni- and multi-software-host architectures opens the possibilities of using the system described in the development of embedded systems. It is

useful from the specification phase, through prototyping, to the final product implementation.

On the down side, the complexity of the scheduling algorithm makes each system partitioning iteration slow and memory hungry. Nevertheless, the techniques shown here are able to move several tasks at each iteration, which decreases the number of iterations necessary to find a solution. Also, the system partitioning algorithm works very closely with the scheduler and so is dependent on the methods used by it. This would make it difficult to change the system partitioning algorithm to work with rate-monotonic based schedulers.