4.

Task Scheduling

This chapter presents the scheduling algorithm developed to deal with special needs imposed for the co-design methodology proposed in this dissertation. It is a pre-runtime scheduler in the sense that all scheduling decisions are made off-line before the actual system execution takes place (see Chapter 2).

4.1 Problem Complexity

In this work, the algorithm is tailored to heterogeneous multi-processor applications having pre-emptable periodic processes with release times, maximum execution times, deadlines, and inter-process precedence and exclusion constraints. The objective is to find a feasible schedule for all processes in the system, where all timing and dependence constraints are satisfied.

The need to deal with heterogeneous multi-processor applications comes directly from the co-design target architecture composed by hardware processors and several, possibly different, software hosts. The implication of this in the scheduler is made clear in the following example. Suppose that the target architecture is composed only of identical processors. If this is the case, there is no change in the maximum execution times of processes, when disregarding communication issues, if they are moved from one processor to another. In fact, this characteristic is used by some authors [118] in order to perform allocation and scheduling simultaneously. However, in a co-design environment possibly composed of ASICs, FPGAs, and several different software hosts, such as transputers, DSPs, and general purpose processors, the processes execution times depend on the processor to which they are allocated.

Garey has shown that the problem of finding a feasible schedule for a set of non-pre-emptable processes, with release times and deadlines in a uni-processor or multi-processor architecture is NP-complete in the strong sense [45]. Any algorithm targeted to scheduling uni-processor systems with mutual exclusion relations among tasks also has to consider the special case where all tasks mutually exclude each other. This is identical to the NP-complete problem studied by Garey. Thus, any such algorithm has to be able to deal with the complexity involved in solving that special case (for an introduction to NP-completeness see [45]).

The problem becomes more complex when multi-processor architectures are considered. Process allocation and scheduling, even when the only goal is the minimisation of the overall execution time, i.e. no independent release times and deadlines are considered, with or without communication costs, is a NP-complete problem [109]. Instead of dealing with the high complexity which would incur for tackling allocation and scheduling of a set of tasks with release times, deadlines, precedence and mutual exclusion constraints, in a heterogeneous multi-processor architecture, the algorithm described in this Chapter is aimed to find a feasible scheduling to those tasks and leaves the allocation to be done by the designer.

4.2 Algorithm Overview

The algorithm presented here is based on a pre-run-time scheduling method for single processor architectures developed by Xu and Parnas [119]. Although they have proposed another algorithm for multi-processor applications with non-pre-emptable processes [118], this is not suitable for our purposes since it is applicable to architectures comprised of a set of identical processors which is not the case in co-design.

A best-first branch-and-bound technique is being employed where each node in the search tree uses an earliest-deadline-first method to produce what is called a *basic schedule*. If such a schedule is not feasible, the task with maximum lateness in that node is searched for and *branch sets* consisting of pairs of tasks are created which, by changing their scheduling priorities, may produce a better solution than the current one. Scheduling priorities are changed by introducing new inter-task dependencies, such as precedence relations. For each tuple a new node is created containing the corresponding new dependency as well as all its parent dependencies.

In order to optimise the search for a feasible schedule, a prune method is applied in which a Lateness Lower Bound (LLB) is calculated using the current node schedule and the branch set information. It is a estimation of the minimum lateness that can be achieved from successors of the actual parent node. If a leaf node's LLB is equal to or greater than the least lateness among all tree nodes, it is clear that none of its successor nodes can improve the solution, and so this leaf node is pruned. The technique is called best-first since it is considered that the non-pruned leaf node holding the minimum LLB is the one with the best chance of having a feasible successor and that node is chosen to proceed the search [119].

4.3 Problem Transformation

The original problem is to find a feasible schedule of a set of periodic processes. So, let $\mathbb{P}$ be the process set to be scheduled with each process[3] p characterised by a tuple (R_p, C_p, D_p, T_p, M_p), where R_p is its release time, C_p the computation time, D_p the deadline, T_p the period, and M_p the assigned processor, $M_p \in \mathbb{M}$ (the processor set). All values are integers and R_p and D_p are relative values that refer to the beginning of the period.

Instead of dealing directly with periodic tasks and considering an infinite scheduling time, the algorithm deals with sets of segments over a finite time length equal to the least common multiple (LCM) of all task periods in the process set, which is called *Schedule Period* (*Ts*). Any feasible schedule found over *Ts* is also feasible at any time greater than *Ts*, since all constraints can be considered by analysing the task interactions over that period of time, and so the schedule can be repeated infinitely.

Each execution of a process p over the schedule period is called a *time instance* or *segment* of p, and $\mathbb{PS}$ is the set of segments of all $p \in \mathbb{P}$. Each segment $a \in \mathbb{PS}$ is defined

[3] Over this chapter the terms *processes* and *tasks* are used interchangeably.

by a tuple (r_a, c_a, d_a, μ_a) where r_a is its release time, c_a the computation time, d_a the deadline, and μ_a the assigned processor, $\mu_a \in \mathbb{M}$, r_a and d_a are absolute values and refer to the beginning of *Ts*. Considering that $\forall p \in \mathbb{P}$, $D_p \leq T_p$, the number of *time instances* of *p* is (Ts / T_p).

By extending the relations defined above to segments, the original set $\mathbb{P}$ can be used to create a set $\mathbb{PS}$ and the problem becomes to find a feasible schedule of $\mathbb{PS}$ over the time defined by *Ts*. The symbols $\mapsto$ and $\otimes$ are used to model precedence and mutual exclusion, respectively, between process pairs. It is considered that ($i \otimes j$) $\Leftrightarrow$ ($j \otimes i$). To generate $\mathbb{PS}$ from $\mathbb{P}$ the following rules are used, which preserve inter-dependencies:

1. $\forall p \in \mathbb{P}$, a series of segments $p_k \in \mathbb{PS}$, $1 \leq k \leq Ts / T_p$, is created so that $\mu_{pk} = M_p$, $c_{pk} = C_p$, $r_{pk} = (R_p \,.\, k)$, $r_{pk} = (D_p \,.\, \mathrm{k})$
2. $\forall i, j \in \mathbb{P}$, $i \mapsto j \Rightarrow i_k \mapsto j_h$, $1 \leq k \leq Ts / T_i \wedge 1 \leq h \leq Ts / T_j$.
3. $\forall i, j \in \mathbb{P}$, $i \otimes j \Rightarrow i_k \otimes j_h$, $1 \leq k \leq Ts/T_i \wedge 1 \leq h \leq Ts/T_j$.

Therefore, the above rules can be used to produce data for the search-tree root node. The procedure which generates the root node using those rules is called *CreateRootNode*.

4.4 Segment Basic Schedule

Bearing in mind the transformations above, all considerations are now focused on the problem of scheduling a set of segments $\mathbb{PS}$ while regarding their inter-dependencies, time, and allocation constraints. The earliest-deadline-first (EDF) strategy has been shown to produce optimal or near optimal results in several specific scheduling problems [57] and so it was chosen to serve as basis of the scheduling method proposed here.

Given that all values are integers, a segment *i* can be split into a sequence of indivisible time units (i, 0), (i, 1), ..., (i, c_i-1), where (i, 0) is the first and (i, c_i-1) is the last unit in the sequence, each of them requiring exactly one processor time unit to execute.

Let $\mathbb{U}$ be the set of all time units of all segments in $\mathbb{PS}$, and $\mathbb{R}$ be the set of tuples that denote all time units in all processors in $\mathbb{M}$, so that:

$$\mathbb{U} = \{ (i, k) \mid i \in \mathbb{PS} \wedge 0 \leq k < c_i\text{-}1 \}$$

$$\mathbb{R} = \{ (\mu, t) \mid \mu \in \mathbb{M} \wedge t \in [0, \infty) \}$$

A schedule is a total function $\pi : \mathbb{U} \rightarrow \mathbb{R}$ where:

1) $\forall(i_1, k_1), (i_2, k_2) \in \mathbb{U}, \forall\ (\mu, t) \in \mathbb{R}$:

$\pi(i_1, k_1) = (\mu, t) \wedge (i_1 \neq i_2 \vee k_1 \neq k_2) \Rightarrow \pi(i_2, k_2) \neq (\mu, t)$

2) $\forall(i, k_1), (i, k_2) \in \mathbb{U}, \forall(\mu_1, t_1), (\mu_2, t_2) \in \mathbb{R}$:

$k_1 < k_2 \ \wedge\ \pi(i, k_1) = (\mu_1, t_1)\ \wedge\ \pi(i, k_2) = (\mu_2, t_2) \ \Rightarrow\ t_1 < t_2$

Statement 1) assures that no more than one segment can be executing on the same processor at the same time. Assertion 2) states that the ordering of segment time units must be conserved.

Based on the schedule function above several assertions may be derived. A segment i *executes* at time t on processor μ iff $\exists(i, k) \in \mathbb{U}, (\mu, t) \in \mathbb{R} \mid \pi(i, k) = (\mu, t)$. If no segment executes on that processor at time t then μ is *idle*. A segment i executes from t_1 to t_2 on processor μ iff it uses all processor time units in the interval $[t_1, t_2)$, i.e., $\exists(i, k+t_1), (i, k+t_2\text{-}1) \in \mathbb{U} \wedge \forall t, t_1 \leq t < t_2: \pi(i, k+t) = (\mu, t)$. A *gap* exists from t_1 to t_2 on processor μ iff μ is idle in the interval $[t_1, t_2)$, i.e., $\nexists\ (i, k) \in \mathbb{U} \wedge \forall t, t_1 \leq t < t_2: \pi(i, k+t) = (\mu, t)$. In a schedule, the execution time of the first and the last time unit of a segment i is called *start time* s_i, and *completion time* e_i, respectively, so that $\pi(i, 0) = (\mu_i, s_i,)$ and $\pi(i, c_i) = (\mu_i, e_i\text{-}1)$, $\mu_i \in \mathbb{M}$. The lateness of a segment i is defined as *lateness*$(i) = e_i - d_i$. The *schedule lateness* is the maximum lateness among all segments in that schedule.

The symbol ∇ is used to represent another relation between segments called *pre-emption*, so that $i \nabla j$ implies that if j is executing and i is able to start executing then j suspends its execution and i starts. Pre-emption cannot be used among segments in different processors and it is not a relation that can be specified by the user but instead is used by the branch-and-bound algorithm to improve partial results, as it will be seen later.

4.4.1 Timing Relation Adjustments

Timing relations are adjusted according to the existent inter-dependencies. These adjustments improve the basic scheduling algorithm efficiency in the search for *eligible* segments and also helps in the node pruning process.

Precedence relation adjustments are such that if $i \mapsto j$ then $r'_j = \max(r'_i + c_i, r'_j)$, since j cannot start executing before i finishes and before it own release time. Conversely, i

must stop executing before j starts executing and before its own deadline, so $d'_i = \min(d'_j - c_j, d'_i)$.

Mutual exclusion relation adjustments are used to make the relations tighter by introducing new precedence relations. For example, if $i \otimes j \wedge r'_j + c_j + c_i > d'_i$ then j cannot execute before i and the relation $i \mapsto j$ can be used instead[4]. Adjustments made in a segment i may affect other segments which are related to i. So, precedence and mutual exclusion adjustments are performed interspersely until no adjustments exist to be done.

AdjustRelations(n) is the name given to the function which performs the above transformations to adjust all segments relations in a node n in the search tree. The example below shows the application of this function. The original data is shown in Figure 4.1 (a) and (b). Initially, precedence relations are verified. The earliest time A stops executing is $r'_A + c_A = 20$. Since $A \mapsto C$, task C cannot start before A stops executing and before its own release time, which is 15. Then C's release time can be adjusted to be the maximum between 20 and 15, which results in the new $r'_C = 20$. The same applies to D, since $A \mapsto D$, but given that $r'_D = 55$ which is already greater than 20, D's release time is not changed.

Conversely, since $A \mapsto C$ and $A \mapsto D$, task A need to finish before C and D start executing and before its own deadline, which is 100. The latest time C can start executing and still meet its deadline is $d'_C - c_C = 80$. The same applies to D and $d'_D - c_D = 85$. Then the deadline of A can be adjusted to be the minimum between 100, 80, and 85, which results in the new $d'_A = 80$. The changes due to precedence relations are highlighted in Figure 4.1.c.

Now consider the values in Figure 4.1.c and the mutual exclusion relations shown in Figure 4.1.a. Since $A \otimes B$, either A executes before B or vice-versa, i.e., they cannot be executing at the same time. However, if A executes before B, then $r'_A + c_A + c_B$ must be smaller or equal to d'_B, otherwise B will stop executing after its deadline, which is not allowed. Since $r'_A + c_A + c_B = 50$ is greater than $d'_B = 40$, A cannot be allowed to execute before B and the only option left is B to execute before A, which is possible since $r'_B + c_B + c_A \leq d'_A$. So the relation $B \mapsto A$ can be added to the design in order to better define the partial execution order between A and B. The result of this step is in Figure 4.1.d.

[4] If $r'_i + c_i + c_j > d'_j$ then i cannot precede j either. In this case, i and j would candidates for partitioning, as it will be explained in the next chapter.

Since the application of precedence and mutual exclusion adjustments above have changed the system specification, it is necessary to reapply them again in order to verify if these changes affect other timing constraints and inter-dependencies. So, precedence adjustments are applied again which results in the changes highlighted in Figure 4.1.e. Mutual exclusion adjustments lead to the changes shown in Figure 4.1.f and another application of precedence adjustments cause the changes emphasised in Figure 4.1.g. No further changes are possible and no further attempts are made to adjust relations or temporal constraints.

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

(a)

Task	A	B	C	D
r’	0	10	15	55
c	20	30	20	30
d’	100	40	105	110

(b)

Precedence adjustments:

$r'_C = \max(20, 15) = 20$

$d'_A = \min(85, 80, 100) = 80$

Task	A	B	C	D
r’	0	10	**20**	55
c	20	30	20	30
d’	**80**	40	105	110

(c)

$\Rightarrow$

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

(d)

$\Rightarrow$

Task	A	B	C	D
r’	**40**	10	**60**	**60**
c	20	30	20	30
d’	80	40	105	110

(e)

$\Rightarrow$

$A \mapsto C$
$A \mapsto D$
$B \mapsto A$
$C \mapsto D$

(f)

$\Rightarrow$

Task	A	B	C	D
r’	40	10	60	**80**
c	20	30	20	30
d’	**60**	40	**80**	110

(g)

Figure 4.1. Timing constraints adjustments

4.4.2 Consistency Check

Timing constraints can be inconsistent. For example, each segment must have $r'_i + c_i \leq d'_i$, otherwise no feasible solution exists. Also, some of the original and derived constraints may be redundant, e.g., $\forall i, j \in \mathbf{PS}$, if $d'_i < r'_j$ and the relations $i \mapsto j$ or $i \otimes j$ exist, they can be removed to improve the algorithm implementation. Another example is $(i \mapsto j \wedge i \otimes j)$ which can be reduced to $i \mapsto j$. This consistency analysis is used for node pruning and for reducing the algorithm complexity.

The marked relations among processes or segments shown below are also considered inconsistent, and so are avoided by the algorithm.

	$\mu_i \neq \mu_j$	$i \otimes j$	$j \mapsto i$	$j \nabla i$
$i \otimes j$				✗
$i \mapsto j$			✗	✗
$i \nabla j$	✗	✗	✗	✗

Table 4.1. Relations inconsistencies

The function *consistent*(*n*) is defined as *true* if a node *n* in the search tree is consistent when all considerations above are taken into account, and *false* otherwise.

4.4.3 Segment Eligibility

A segment *i* is said to be *eligible* to execute at time *t* if the following properties are observed:

1. a) $r'_i \leq t \wedge \neg(e_i \leq t)$

   b) $\nexists j \mid j \otimes i \wedge s_j < t \wedge \neg(e_j \leq t)$

   c) $\nexists j \mid j \mapsto i \wedge \neg(e_j \leq t)$

2. $\nexists j \mid j$ holds the properties described in 1), above, and

   a) $\mu_i = \mu_j \wedge \begin{cases} j \nabla i \\ \vee \\ \neg (j \nabla i) \wedge \begin{cases} d'_j < d'_i \\ \vee \\ d'_j = d'_i \wedge c_j > c_i \end{cases} \end{cases}$

   $\vee$

   b) $\mu_i \neq \mu_j \wedge j \otimes i \wedge \begin{cases} d'_j < d'_i \\ \vee \\ d'_j = d'_i \wedge c_j > c_i \end{cases}$

Conditions 1.a, 1.b, and 1.c deal with time constraints and segment inter-dependencies, respectively. So, a segment *i* is only eligible to execute if *t* is greater than the release time of *i* and *i* has not completed yet. Also, if there is a segment *j* which has not finished yet and *j* must precede *i*, or *j* has already started and excludes *i*, then *i* must wait until *j* completes.

Condition 2 merely deals with priorities, i.e., if two segments satisfy Condition 1 then Condition 2 is used to determine which of them is the eligible one.

Pre-emption relations are added by the branch-and-bound algorithm to change the scheduling order of the segments, in an attempt to find a feasible solution. So, if two segments i and j in the same processor have a true Condition 1, and there exists a relation $i \nabla j$, then segment i will execute even if $d'_j < d'_i$.

4.4.4 Basic Schedule

A *basic schedule* of a set of segments $\mathbb{PS}$ is one where segments start after their start times, and which obeys all precedence, exclusion, and processor allocation requirements, and the allocation obey the *eligible* rules. A *feasible schedule* is one that satisfies the properties of a basic schedule and $\forall i \in \mathbb{PS}$, $e_i \leq d'_i$. An *optimal schedule* is one that has the minimum lateness among all schedules. The list below summarises the rules of a basic schedule.

$\forall\ i, j \in \mathbb{PS}$:

1. $s_i \geq r'_i$
2. $i \mapsto j \Rightarrow e_i \leq s_j$
3. $i \otimes j \Rightarrow e_i \leq s_j \vee e_j \leq s_i$
4. i is allocated to $\mu_i \Rightarrow \forall k,\ 0 \leq k < c_i - 1,\ \nexists\ (\mu, t) \in \mathbb{R}: \mu \neq \mu_i \wedge \pi(i, k) = (\mu, t)$
5. i executes at time t iff i is *eligible* at time t.

The algorithm shown next computes a basic scheduling in a node n based on the above rules, and considers a problem containing *numSeg* segments and *numP* processors to schedule. Comments start with the symbol "//" and continue until the end of line. The following variables are used:

- t : current time value
- lastT : time value used on the previous iteration
- s and e are size *numSeg* arrays so that s[i] and e[i] represents s_i and e_i, respectively.
- started : size *numSeg* array; started[i] is set to *true* if $s_i \geq t$, and *false* otherwise
- completed : size *numSeg* array; completed[i] is *true* if $e_i \leq t$, and *false* otherwise.
- compTimeLeft : size *numSeg* array; compTimeLeft[i] holds the amount of remaining execution time for segment i at time t.

- seg : size *numP* array; seg[*m*] holds the identity of the segment currently executing on processor *m* at time *t*.
- idle : size *numP* array; idle[*m*] is *true* if processor *m* is idle at *t*, and *false* otherwise.

```
ProduceBasicSchedule(n)
    {
    lastT = -1;
    $\forall i \in \mathbb{PS}$:
        {
        started[i] = false;
        completed[i] = false;
        compTimeLeft[i] = $c_i$;
        }
    $\forall m \in \mathbb{M}$:
        idle[m] = true;
    While ($\exists$i $\in \mathbb{PS}$ | $\neg$completed[i])
        {
        if $\exists r'_i >$ lastT, $t_1$ = min{$r'_i$ | $r'_i >$ lastT }, otherwise $t_1 = \infty$;
        if $\exists m \in \mathbb{M}$ | $\neg$idle[m], $t_2$ = min{compTimeLeft[seg[m]] + lastT | $\neg$idle[m]}, else $t_2 = \infty$;
        t = min($t_1$, $t_2$);
        $\forall m \in \mathbb{M}$ | $\neg$idle[m]
            {
            // seg[m] is executing from lastT to t, so decrease its compTimeLeft by
            // the amount (t - lastT):
            compTimeLeft[seg[m]] = compTimeLeft[seg[m]] - (t - lastT);
            if (compTimeLeft[seg[m]] = 0)            // if seg[m] completed on time t
                {
                e[seg[m]] = t;
                completed[i] = true;
                idle[m] = true;
                }
            }
        $\forall i \in \mathbb{PS}$ | i is eligible at time t
            {
            seg[$\mu_i$] = i;
            if $\neg$started[i]
                {
                started[i] = true;
                s[i] = t;
                }
            }
        lastT = t;
        }
    }
```

4.5 Basic Schedule Improvement

Consider that the basic schedule is not feasible, i.e., there is at least a segment *i* with *lateness*(*i*) = $e_i - d'_i > 0$. In order to improve any segment lateness it is necessary to re-schedule that segment to finish earlier. Hence, to improve the lateness of the latest process *l*, and so improve the schedule lateness, *l* has to be scheduled earlier. If there are more than one latest segment, *l* is considered to be the one, among those, that completed last.

Although it is possible to find a feasible solution, if one exists, by exhaustively changing all segment pairs relative execution order, this is a very expensive method. Not every change will lead to a lateness improvement and others may be redundant. Let $\mathbb{Z}(l)$ be the set of segments whose completion time precedes and includes *l*'s completion time in a period of continuous execution, i.e., within that period there does not exist any instant *t* when all processors in $\mathbb{M}$ are idle. $\mathbb{Z}(l)$ is a first attempt to reduce the number of relevant segments for scheduling improvement.

$\mathbb{Z}(l)$ can be defined as (notice below that *k* may be equal to *l*):

1. $l \in \mathbb{Z}(l)$

2. $\forall i$, if $\exists\ k \in \mathbb{Z}(l) \mid r'_k < e_i \wedge e_i < e_l$

   $\Rightarrow\ i \in \mathbb{Z}(l)$

The following theorems are used to limit the number of number of changes that need to be tried in order to improve the a schedule.

Theorem 4.1: $\mathbb{Z}(l)$ is a set of segments whose completion time precede and include *l*'s completion time and which execute within a period of time that stretches from $min\{\ r'_i \mid i \in \mathbb{Z}(l)\ \}$ to e_l where there does not exist any instant *t* when all processors in $\mathbb{M}$ are idle.

Proof: See Appendix C.

Theorem 4.2: If the schedule is not feasible then a feasible schedule can only be found if it is possible to change the relative execution order of segments in $\mathbb{Z}(l)$ such that *l* is scheduled earlier.

Proof: See Appendix C.

If the schedule is not feasible, the latest segment l has to be scheduled earlier. In order to do this, let's define two subsets of $\mathbb{Z}(l)$, such that $\mathbb{ZI}(l) = \{i \mid i \in \mathbb{Z}(l) \wedge \mu_i = \mu_l\}$and $\mathbb{ZE}(l) = \{i \mid i \in \mathbb{Z}(l) \wedge \mu_i \neq \mu_l\}$.

4.5.1 Uni-processor Schedule Improvement

First consider the case of a uni-processor architecture, so that $\mathbb{ZI}(l) = \mathbb{Z}(l)$ and $\mathbb{ZE}(l) = \emptyset$. There exist two properties of a basic schedule which are important here [119]:

1. e_l is the earliest possible completion time for the entire set of segments in $\mathbb{Z}(l)$.
2. Any non-optimal schedule can be improved on only by scheduling some segment $k \in \mathbb{Z}(l) \mid d'_l < d'_k$.

A consequence of Theorem 5-1 when applied to a uni-processor architecture is that there is no gap in the interval $(min\{r'_i \mid i \in \mathbb{Z}(l)\}, e_l)$, since there is only one processor and it cannot be idle within that period. So, taking into account property 1 above and Theorem 5-2, it is clear that the only way of improving l's schedule is by making it use some processor time units that was being used by another segment in $\mathbb{Z}(l)$. However, there is no advantage in changing the relative execution order between l and another segment $i \in \mathbb{Z}(l)$ if $d'_i \leq e_l$, since from property 1 this would make i more or as late as l. Also, it is impossible to change the relative execution order of two segments if a precedence relation exist between them.

Thus, for a segment $i \in \mathbb{Z}(l)$ with $d'_i > e_l$, if $i \otimes l$ and it is possible to change (i, l) relative execution order (i.e., $r'_l + c_l + c_i \leq d'_i$) then create a child node with the added relation $l \mapsto i$, if this addition does not cause any inconsistency with any relation that may already exist (see Section 4.4.2), and try scheduling again. On the other hand, if $\neg(i \otimes l)$ and all other considerations hold, and $\exists k \mid k$ executes between $(s_i, e_l) \wedge (i \mapsto k \vee i \nabla k)$, then create a child node with the added relation $l \nabla i$. In this later case, $\forall k$ that executes between s_i and e_l, if $k \otimes i$ then add the relation $k \mapsto i$ otherwise add $k \nabla i$, and try scheduling again. These relations need to be added so that the execution of i starts just before the execution of l, and the pre-emption may occur. This is why i cannot precede or pre-empt k, otherwise no relation could be added to i and k, and the occurrence of pre-emption would not be guaranteed.

A summary of the above described method is found below. To this summary the sets $\mathbb{G}\mathbf{1}$ and $\mathbb{G}\mathbf{2}$ of segment pairs are defined, such that $\mathbb{G}\mathbf{1}$ holds the pairs (i, j) whose relative execution order changes by adding the relation $j \mapsto i$, and $\mathbb{G}\mathbf{2}$ holds the pairs to which the relation $j \nabla i$ is added. These sets are also called *branch sets*.

1. $\forall i \in \mathbb{ZI}(l) \mid d'_i > e_l \wedge \neg(i \mapsto l) \wedge \neg(i \nabla l) \wedge$

   a) $i \otimes l \wedge r'_l + c_l + c_i \leq d'_i$

   $\Rightarrow (i, l) \in \mathbb{G}\mathbf{1}$.

   $\vee$

   b) $\neg(i \otimes l) \wedge \neg(l \nabla i) \wedge r'_i + c_i + c_l \leq d'_i \wedge$

   $\nexists j \in \mathbb{ZI}(l) \mid j$ executes in the interval $(s_i, e_l) \wedge e_i \leq s_j \wedge e_j \leq e_l \wedge (i \mapsto j \vee i \nabla j)$

   $\Rightarrow (i, l) \in \mathbb{G}\mathbf{2}$.

2. $\forall (i, j) \in \mathbb{G}\mathbf{1}$, try a new schedule with all previous relations plus $(j \mapsto i)$.

3. $\forall (i, j) \in \mathbb{G}\mathbf{2}$, try a new schedule with all previous relations plus $(j \nabla i)$.

   $\forall k \in \mathbb{ZI}(l) \mid k$ executes in the interval (s_i, e_j)

   $$\wedge \begin{cases} i \otimes k \Rightarrow \text{add relation } (k \mapsto i) \\ \vee \\ \neg(i \otimes k) \Rightarrow \text{add relation } (k \nabla i) \end{cases}$$

4.5.2 Multi-processor Schedule Improvement

In a multi-processor architecture, the scheduling effects that segments in $\mathbb{ZE}(l)$ cause over segments in $\mathbb{ZI}(l)$ need to be taken into account. From the eligibility rules (see Section 4.4.3) it is possible to verify that only those segments $i \in \mathbb{ZE}(l)$ that precede or exclude segments $j \in \mathbb{ZI}(l)$ are of importance. Also only those segments i and j such that i executes before j (i.e. $e_i \leq s_j$) and $r'_j < e_i$ are directly affected (from eligibility Conditions 1.a and 1.b). For example, Figure 4.2 shows some of those effects by showing a gap created by segment $h \in \mathbb{ZE}(l)$. Segment A does not exert any direct influence on segment B, but it does on segment C. So, a possible improvement may exist if A and C change their relative scheduling order. In this example, this is exactly what happens as it is shown in Figure 4.3.

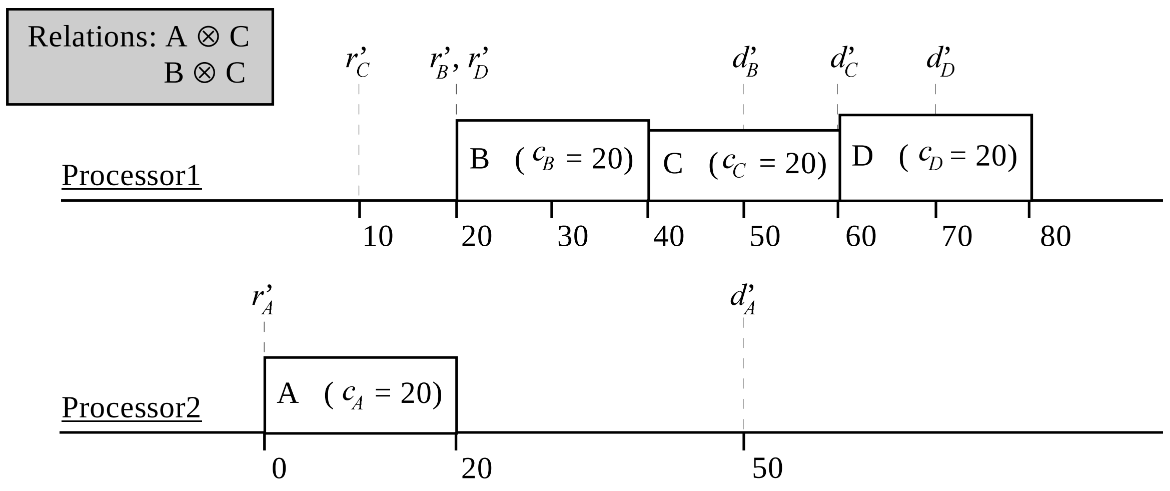


Figure 4.2. Scheduling gap.

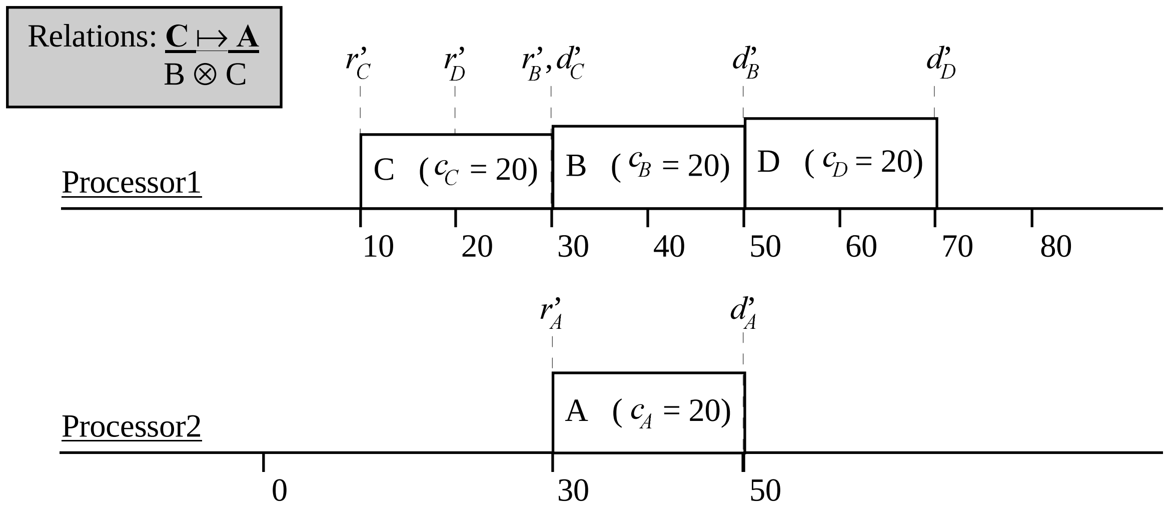


Figure 4.3. Narrowing the gap.

Any segment i that executes after e_h (i.e. $s_i \geq e_h$) and have $r'_i < e_h$ may be able to use some processor time units in the interval (r'_i, e_h) if:

Case a. The relative execution order between h and some segment $i \in \mathbf{ZI}(l)$ which it excludes is able to change, e.g. if $i \otimes h \wedge e_h \leq s_i$, then a change is possible if $r'_i + c_i + c_h \leq d'_h$, or

Case b. If h is scheduled earlier, since h would finish earlier than the current e_h, leaving more processor time units to be used by segments with $s_i \geq e_h \wedge r'_i < e_h$.

Suppose that an attempt to provide an earlier schedule to h (Case b) would also consider changing the relative scheduling order of any process that h excludes and that executes before h. If this is the case, a solution to case a) does not need to consider any segment $i \in \mathbb{ZI}(l)$ such that $i \otimes h \wedge e_i \leq s_h$. Considering the above and the previous section explanations, the case described in a) can be taken into account by the following method:

$$\forall i \in \mathbb{ZE}(l), \text{ if } \exists j \in \mathbb{ZI}(l) \mid i \otimes j \wedge e_i \leq s_j \wedge r'_j < e_i \wedge \neg(i \mapsto j) \wedge r'_j + c_j + c_i \leq d'_i$$
$$\Rightarrow (i, j) \in \mathbb{G}\mathbf{1}$$

Case (b) can be considered by noticing that it is required from h exactly the same that is desired to the latest segment l in order to improve an unfeasible schedule, which is to have it scheduled earlier. Therefore, a recursive call using h as parameter to the procedures developed to l solves the problem, and also validates the assumption made to solve Case a).

Since segments not allocated to μ_l can influence the scheduling of l only if they precede or exclude some segment in μ_l or yet if they exert influence on the scheduling of some segment h as described in case b above, set $\mathbb{ZE}(l)$ definition can be made more restrictive to consider only those segments with the aforementioned characteristics. Thus, the general improvement method can then be described as:

```
CreateBranchSets( l )
  {
  Create ZI(l), and ZE(l):
   1. $l \in \mathbb{ZI}(l)$
   2. $\forall i$, if $\exists k \in \mathbb{ZI}(l) \mid r'_k < e_i \wedge e_i < e_l \wedge \mu_i = \mu_l \quad \Rightarrow i \in \mathbb{ZI}(l)$
   3. $\forall i$, if $\exists k \in \mathbb{ZI}(l) \mid r'_k < e_i \wedge e_i < e_k \wedge (i \otimes k \vee i \mapsto k) \wedge \mu_i \neq \mu_l \quad \Rightarrow i \in \mathbb{ZE}(l)$
  Create G1 and G2:
   1. $\forall i \in \mathbb{ZI}(l) \mid d'_i > e_l \wedge \neg(i \mapsto l) \wedge \neg(i \nabla l) \wedge$
      a) $i \otimes l \wedge r'_l + c_l + c_i \leq d'_i$
         $\Rightarrow (i, l) \in \mathbb{G}\mathbf{1}$.
      b) $\neg(i \otimes l) \wedge \neg(l \nabla i) \wedge r'_i + c_i + c_l \leq d'_i \wedge$
         $\nexists j \in \mathbb{ZI}(l) \mid j$ executes in the interval $(s_i, e_l) \wedge e_i \leq s_j \wedge e_j \leq e_l \wedge (i \mapsto j \vee i \nabla j)$
         $\Rightarrow (i, l) \in \mathbb{G}\mathbf{2}$.
   2. $\forall i \in \mathbb{ZE}(l)$: if $\exists j \in \mathbb{ZI}(l) \mid i \otimes j \wedge e_i \leq s_j \wedge r'_j < e_i \wedge \neg(i \mapsto j) \wedge r'_j + c_j + c_i \leq d'_i$
         $\Rightarrow (i, j) \in \mathbb{G}\mathbf{1}$.

   3. $\forall i \in \mathbb{ZE}(l)$: if $\exists j \in \mathbb{ZI}(l) \mid (i \otimes j \vee i \mapsto j) \wedge e_i \leq s_j \wedge r'_j < e_i$
         $\Rightarrow$ CreateBranchSets( $i$ ).
  }
```

```
CreateChildNodes( l )
    {
    CreateBranchSets( l );

    $\forall(i, j) \in \mathbb{G}\mathbf{1}$, create a child node with all relations in $l$ plus $(j \mapsto i)$;

    $\forall(i, j) \in \mathbb{G}\mathbf{2}$, create a child node with all relations in $l$ plus $(j \nabla i)$ and

        $\forall k \in \mathbb{ZI}(l) \mid k$ executes in the interval $(s_i, e_j)$

        $\wedge \begin{cases} i \otimes k \Rightarrow \text{add relation } (k \mapsto i) \\ \vee \\ \neg (i \otimes k) \Rightarrow \text{add relation } (k \nabla i) \end{cases}$
    }
```

4.6 Branch-And-Bound Algorithm

A search tree is used where in each node an attempt is made to produce a feasible schedule. If a node does not hold a feasible solution, child nodes are created where each one holds new relations based on the pairs (i, j) belonging to the branch sets $\mathbb{G}\mathbf{1}$ and $\mathbb{G}\mathbf{2}$ associated to the current node schedule, in order to improve the previous result. Therefore, each node $n \in \mathbb{ST}$ (the search tree node set) has a segment set $\mathbb{S}(n)$ containing its parent node information plus new information added by the branch-and-bound algorithm. To improve the algorithm efficiency, the order in which child nodes schedules are produced is related to the value of d'_i from the greatest to the smallest. The reasoning behind this comes from the fact that in the child node schedule the segments i and j change their execution order, i.e. i will finish after the time j was previously finishing, and so there is a greater chance of not causing i to become unfeasible for larger values of d'_i.

The *Lateness*(n) of a node n is defined as the lateness of the schedule in n, i.e., *Lateness*(n) = max{lateness(i) | $i \in \mathbb{S}(n)$}. A node is considered *feasible* if its schedule is feasible.

4.6.1 Node Pruning

There are ways of making the algorithm more efficient by identifying nodes which do not lead to feasible solutions. Since new relations are added to each node, timing relation adjustments need to be carried out (see Section 4.4.1). However, the consistency check (see

Section 4.4.2) can reveal any problems caused by the introduction of those relations and prune any inconsistent node.

Another way of increasing the algorithm efficiency is by predicting how good the solutions found by any successor node can be based on the current node schedule, and pruning nodes which lead to bad results. Let *LLB*(*n*) be a lower bound on the lateness of schedules derived from node *n*, i.e., no successor node of *n* can provide a solution with a lateness smaller than LLB(*n*). If a node *n* is unfeasible (i.e., lateness(*n*) > 0) and lateness(*n*) ≤ LLB(*n*) or LLB(*n*) > 0 then *n* is pruned, since none of its successor nodes will be able to produce either a schedule better than *n* or a feasible one. Even stronger, if the minimum lateness among all nodes created so far (named *minLateness*) is less than or equal to LLB(*n*), then *n* can be pruned since none of its successor nodes can produce a better schedule than the one with *minLateness*. Also, let $\mathbb{LN}$ be the set of all non-pruned leaf nodes. If during the search the minimum lateness among all nodes (named *minLateness*) is less than or equal to the minimum LLB of all nodes in $\mathbb{LN}$ (named *minLLB*), the node holding *minLateness* has an optimal schedule.

The above descriptions can be summarised as follows:

- $minLateness = \min\{\text{lateness}(n) \mid n \in \mathbb{ST}\}$
- $minLLB = \min\{\text{LLB}(n) \mid n \in \mathbb{LN}\}$
- $\forall n \in \mathbb{ST}$, if $\neg(\text{consistent}(n) \vee \text{minLateness} \leq \text{LLB}(n) \vee \text{LLB}(n) > 0 \Rightarrow n \notin \mathbb{LN}$
- At any time during the execution of the branch-and-bound algorithm, if $minLateness \leq minLLB \Rightarrow$ the global optimal schedule has already been found (although it may not be feasible).

4.6.2 Calculating *LLB(n)*

For a node *n*, if $\mathbb{G}\mathbf{1} = \emptyset \wedge \mathbb{G}\mathbf{2} = \emptyset$ then there is no way of improving lateness(*n*) and so LLB(*n*) = lateness(*n*). However, if this is not the case, three methods are used to calculate the values LLB1(*n*), and LLB2(*n*), and the maximum value between them is assigned to LLB(*n*). In the estimation of LLB(*n*) a trade-off between accuracy and speed is considered. The better the accuracy the greater the amount of time to calculate LLB(*n*). In fact, an exact estimation would be as hard and computation intensive as generating all possible solutions [107].

4.6.2.1 LLB1(n)

LLB1(n) is a trivial lateness lower bound which takes into account each segment individual time constraints. It is calculated by noticing that each segment must execute after its release time and before its deadline. So, LLB1(n) can be calculated as:

$$\text{LLB1}(n) = \max\{r'_i + c_i - d'_i \mid i \in \mathbb{PS}(n)\}.$$

4.6.2.2 LLB2(n)

LLB2(n) takes into account segment interaction by considering the processor utilisation during an interval of time by several segments, and any possible improvements that can be made to the current schedule. Firstly, suppose that a set of segments $\mathbb{S}$ allocated to a same processor μ_S execute within the interval $[t_1, t_2)$. Instead of considering each segment timing constraints separately, consider that each $i \in \mathbb{S}$ has a modified release time r^*_i and deadline d^*_i, so that $r^*_i = \min\{r'_j \mid j \in \mathbb{S}\}$, and $d^*_i = \max\{d'_j \mid j \in \mathbb{S}\}$. The total computation time C_S of all segments in $\mathbb{S}$ is $C_\text{S} = \sum_{j \in \mathbb{S}} c_j$. Therefore, if no other constraints are considered, the lateness of the last segment $l \in \mathbb{S}$ can be calculated as lateness(l) = $r^*_l + C_\text{S} - d^*_l$, and this value can be used as a lower bound since at least one of the segments in $\mathbb{S}$ will be at least as late as l when all constraints are considered.

The evaluation described above, although useful can be made better by considering gaps in the schedule which cannot be closed. For example, consider the problem shown in Figure 4.4. Processor 1 is idle from time 40 to 60. No matter how the schedule order changes, the minimum size that gap may become is 10, since no segment executing before the gap (i.e. A or B) can be scheduled later than the maximum of their deadlines, which is $d'_A = 45$, and no segment executing after the gap (i.e. C or D) can be scheduled earlier than the minimum of their release times, which is $r'_D = 55$. Thus, the lower bound evaluation proposed in the above paragraph should be performed separately on the set of segments {A, B} and {D, E}, and the greatest of them be considered as LLB2. If $d'_A \geq r'_B$, the gap could be closed completely and the evaluation should consider the set {A, B, D, E}.

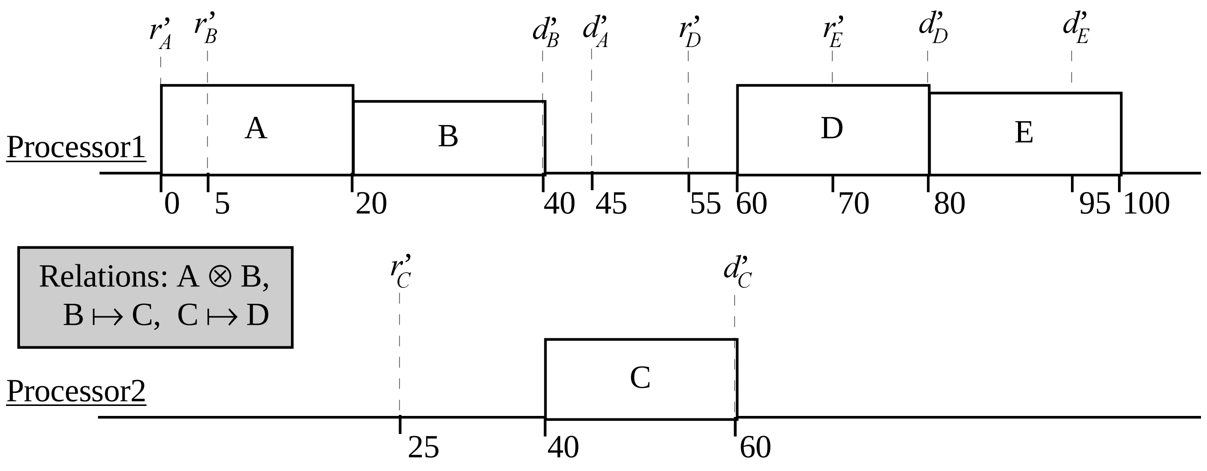


Figure 4.4. Lateness Lower Bound Evaluation

The complete description of LLB2(n) follows. Let (g_1, g_2, ..., g_N) represent the **N** periods of time in processor μ_l, such that μ_l is idle during each g_i, and g_i occurs within the interval of time when segments in $\mathbb{ZI}(l)$ are executing, i.e. between $\min\{s_u \mid u \in \mathbb{ZI}(l)\}$ and e_l, and $\nexists j, k \mid r'_k \le d'_j$, $r'_k = \min\{r'_v \mid s_v \ge eg(g_i) \wedge v \in \mathbb{ZI}(l)\}$ and $d'_j = \max\{d'_v \mid e_v \le sg(g_i) \wedge v \in \mathbb{ZI}(l)\}$, where $sg(g_i)$ and $eg(g_i)$ are the start and the end of g_i, i = (1,...,**N**), respectively. Define:

- $r^*(g_i) = \min\{r'_j \mid e_j \ge eg(g_i) \wedge j \in \mathbb{ZI}(l)\}$, i = (0, 1,...,**N**).
- $d^*(g_i) = \max\{d'_j \mid e_j \le sg(g_i) \wedge j \in \mathbb{ZI}(l)\}$, i = (1,...,**N, N+1**).
- $eg(g_0) = \min\{s_j \mid j \in \mathbb{ZI}(l)\}$. $sg(g_{N+1}) = e_l$ (i.e. $\max\{e_j \mid j \in \mathbb{ZI}(l)\}$)
- $C(g_i) = \sum c_j \mid j \in \mathbb{ZI}(l) \wedge eg(g_{i-1}) \le e_j \le sg(g_i)$
- LLBg(g_i) = $r^*(g_{i-1}) + C(g_i) - d^*(g_i)$

Then LLB2 = max{LLBg(g_i) | i = (1,..., N+1) } and finally,

$$\boxed{\text{LLB}(n) = \max\{\text{LLB1}(n), \text{LLB2}(n)\}}$$

In the example of Figure 4.4, LLB1 = max {-25, -15, -15, -5, -5} = -5, LLB2 = max {0 + 40 - 45, 55 + 40 - 95} = 0, and LLB = max{-5, 0 } = 0.

4.6.3 Branch-and-Bound Implementation

The following procedure takes into account all ideas and procedures previously discussed in this chapter in order to implement the branch-and-bound algorithm. CN represents the child node set of a given node, which initially contains the root node. All variables and functions used have already been discussed or may be easily understood from the context.

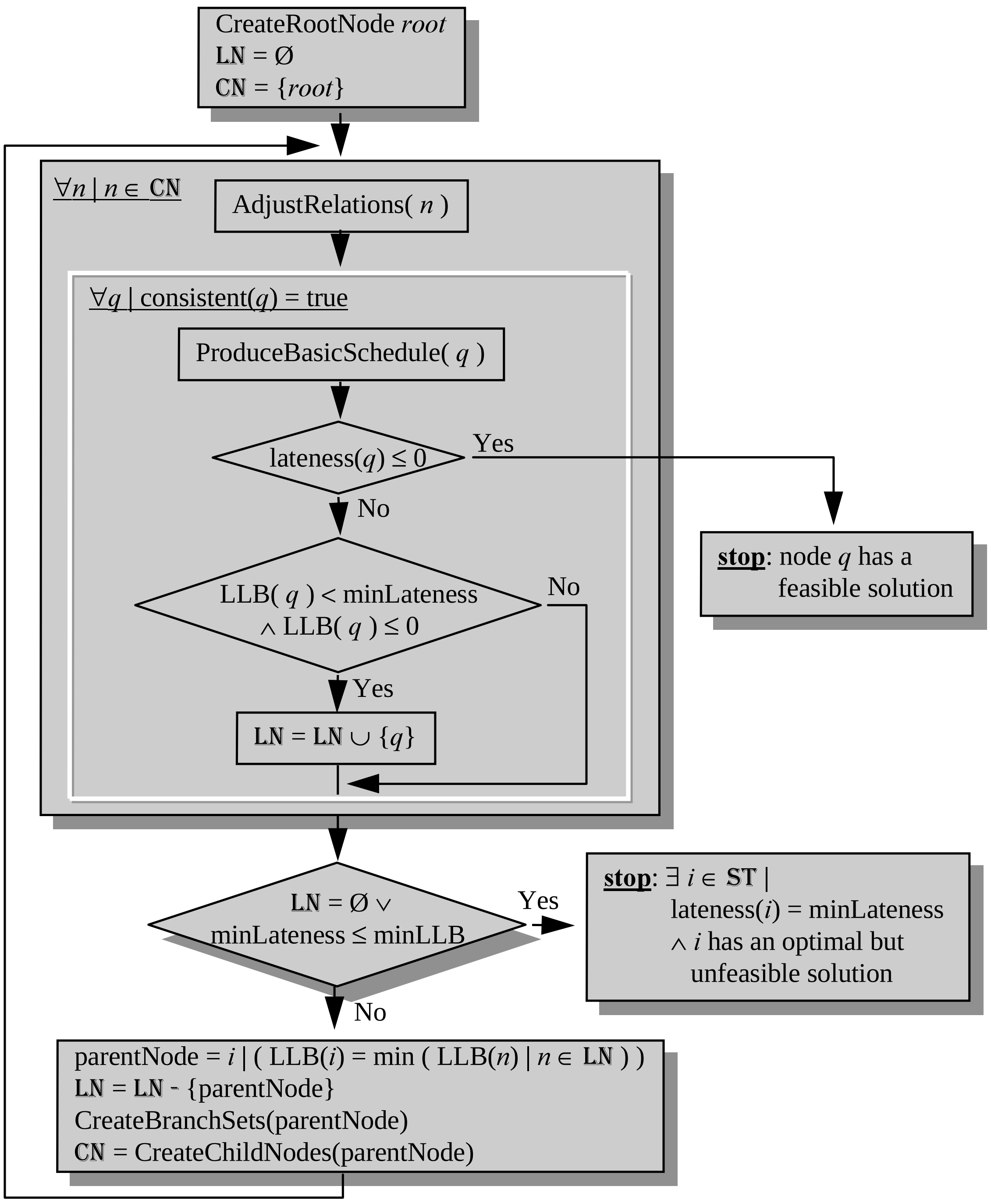

CreateRootNode *root*
LN = Ø
CN = {*root*}
∀*n* | *n* ∈ CN
AdjustRelations(*n*)
∀*q* | consistent(*q*) = true
ProduceBasicSchedule(*q*)
lateness(*q*) ≤ 0
Yes
No
stop: node *q* has a feasible solution
LLB(*q*) < minLateness ∧ LLB(*q*) ≤ 0
No
Yes
LN = LN ∪ {*q*}
LN = Ø ∨ minLateness ≤ minLLB
Yes
stop: ∃ *i* ∈ ST | lateness(*i*) = minLateness ∧ *i* has an optimal but unfeasible solution
No
parentNode = *i* | (LLB(*i*) = min (LLB(*n*) | *n* ∈ LN))
LN = LN - {parentNode}
CreateBranchSets(parentNode)
CN = CreateChildNodes(parentNode)

Figure 4.5. Branch-and-Bound Algorithm

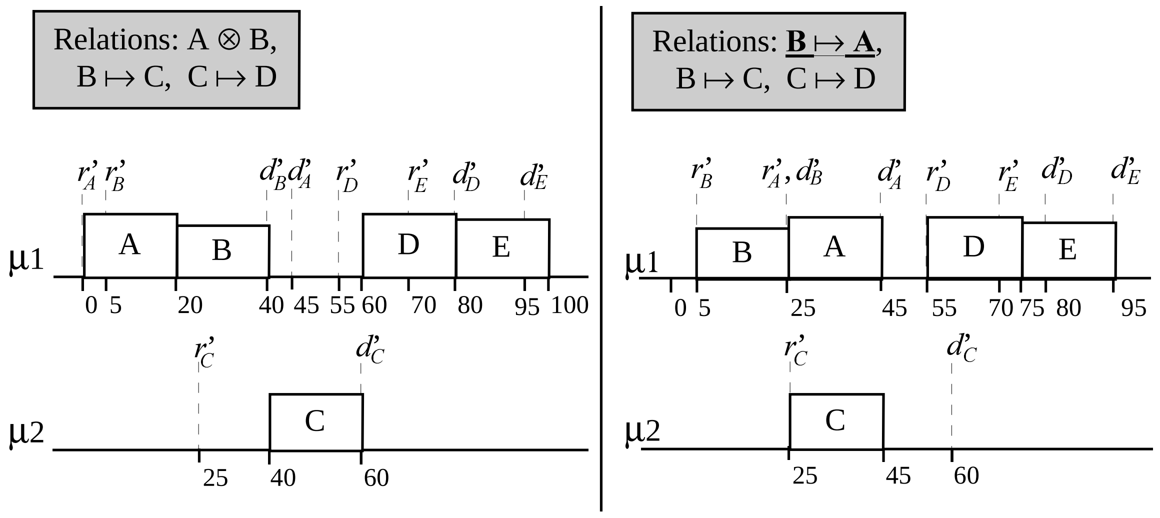
4.6.4 Context Switching Considerations

The only possible time a segment switch may take place is either at the adjusted release time or at the completion time of a segment. Furthermore, each segment can pre-empt any other segment just once [119]. So, it is possible to consider context switching by the addition of the following terms to each segment's computation time (notice that only term 3 is specifically added due to pre-emption):

1. the time required to save the status of a pre-empted segment;
2. the time required to load a new segment;
3. the time required to restart a pre-empted segment.

4.7 Examples

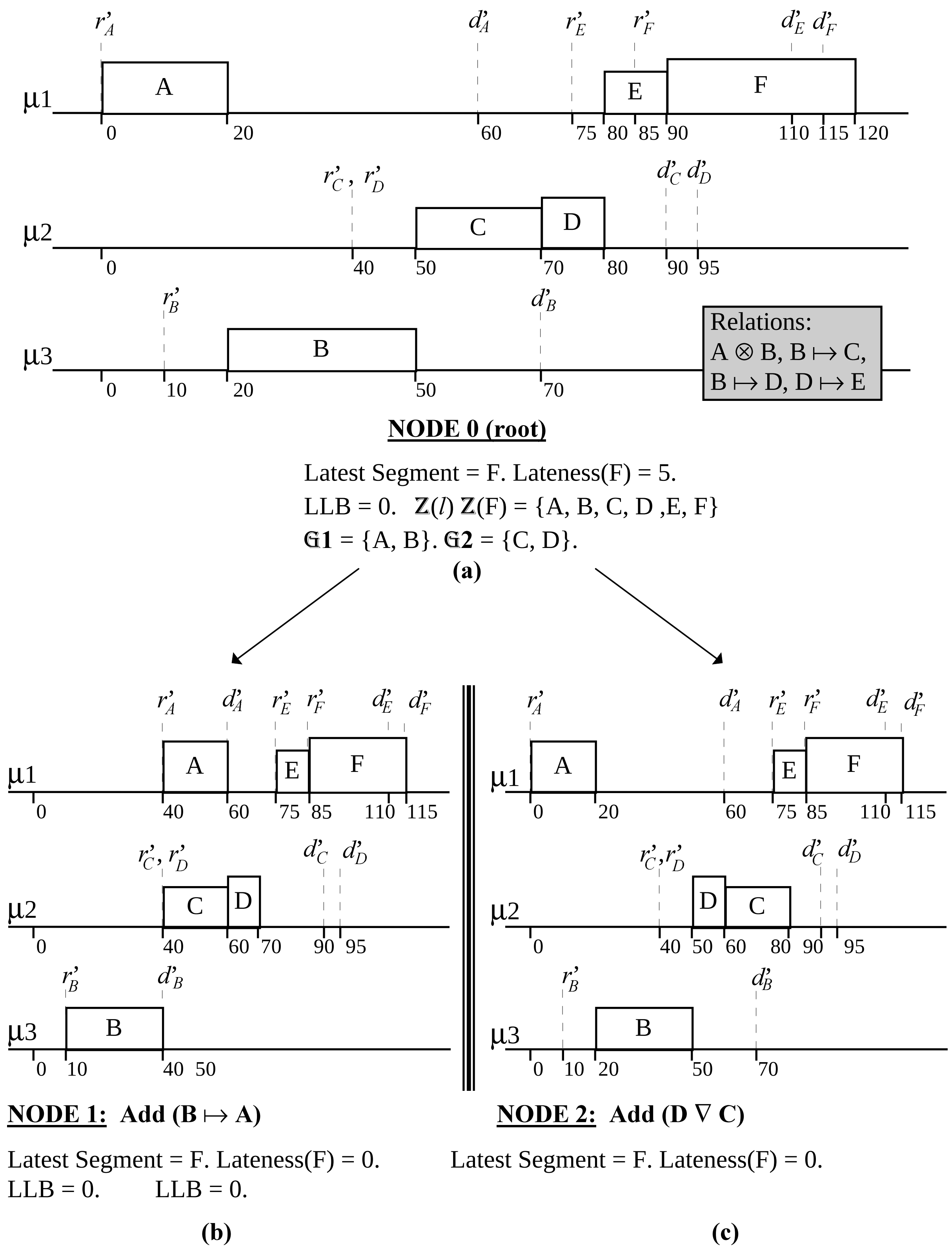
In the examples below, the algorithm was allowed to produce all branches, not stopping when a feasible schedule was found.


Relations: A ⊗ B, B ↦ C, C ↦ D
Relations: **B ↦ A**, B ↦ C, C ↦ D
r'A r'B d'B d'A r'D r'E d'D d'E
μ1
A B D E
0 5 20 40 45 55 60 70 80 95 100
r'C d'C
μ2
C
25 40 60
r'B r'A, d'B d'A r'D r'E d'D d'E
μ1
B A D E
0 5 25 45 55 70 75 80 95
r'C d'C
μ2
C
25 45 60

<u>NODE 0 (root)</u>
Latest Segment = E. Lateness(E) = 5.
LLB = 0. $\mathbb{Z}$(E) = {A, B, C, D ,E}
$\mathbb{G}$1 = {A, B}. **$\mathbb{G}$2** = Ø.

<u>NODE 1</u>
Latest Segment = E. Lateness(E) = 0.
LLB = 0.

Figure 4.6. Scheduling Example 1



NODE 0 (root)

Latest Segment = F. Lateness(F) = 5.
LLB = 0. $\mathbb{Z}(l)$ $\mathbb{Z}$(F) = {A, B, C, D ,E, F}
$\mathbb{G}$**1** = {A, B}. $\mathbb{G}$**2** = {C, D}.

(a)

NODE 1: Add (B ↦ A)

Latest Segment = F. Lateness(F) = 0.
LLB = 0. LLB = 0.

(b)

NODE 2: Add (D ∇ C)

Latest Segment = F. Lateness(F) = 0.

(c)

Figure 4.7. Scheduling Example 2

All solutions for the examples proposed by Xu and Parnas in their article [119] are improved using the algorithm proposed here and solved in the root node due to the use of the *AdjustRelations*() function. For example, even though their fifth example is the one which generates the biggest number of nodes when using their algorithm, it can be seen in Figure 4.8 that here it is solved in the root node due to the adjustments in the initial specifications shown in the tables below.

$\boxed{A \otimes D}$

(a)

Task	A	B	C	D
r'	0	20	30	90
c	60	20	20	20
d'	122	121	120	110

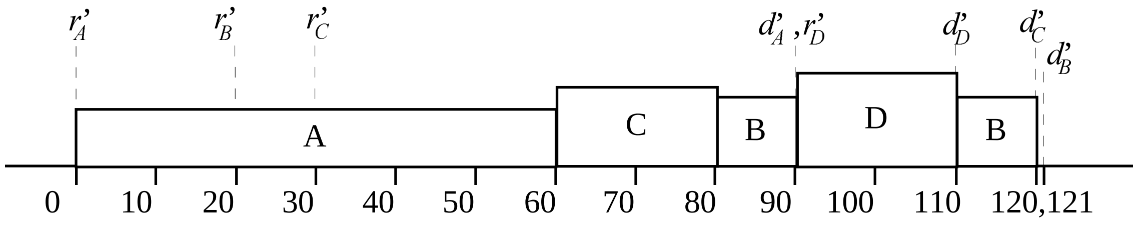
(b)

$\Rightarrow \boxed{A \mapsto D} \Rightarrow$

(c)

Task	A	B	C	D
r'	0	20	30	90
c	60	20	20	20
d'	**90**	121	120	110

(d)



(e) Lateness(root) = 0, LLB(root) = 0.

Figure 4.8. An uni-processor example from [119]

4.8 Summary

The scheduling algorithm shown in this chapter achieves the special requirements for co-design mentioned in the introduction section. To the best of the author's knowledge, there is no other algorithm able to produce a solution whenever one exists for the pre-runtime scheduling of a set of processes with deadlines, release times, precedence, exclusion relations, and which allows process pre-emption, in a heterogeneous multi-processing architecture. A mathematically sound branch-and-bound technique is proposed to solve the problem. Another original point is the use of a pre-runtime scheduling algorithm in a co-design environment.

By using static scheduling, 100% processor utilisation can be obtained, there is no need for complex operating systems (in some cases no operating systems at all), and context switch can be made short since the exact points of their occurrence in the processes are known in advance. These characteristics make this approach very attractive for embedded devices, mainly because of the fact that computational resources are normally at a premium on these systems.

Although for big designs the time to obtain a solution may be quite long, it is important to notice that there is no burden for the actual processors that implement the system since the whole scheduling is done beforehand, possibly by using powerful computers. This characteristic also guarantees that the schedule is sound without doubt before marketing the product. Of course this guarantee is dependent on how good the worst-case estimations are. However, the damage caused by a system crash is proportional to the criticality of that system. For safety critical systems it is reasonable to argue that no event should be unpredicted and that schedulability should be guaranteed before execution [6].

The algorithm described here can also be used for other purposes. For example, introducing simple modifications to avoid pre-emption would allow it to be used for scheduling operations for high-level hardware synthesis, for synchronous or asynchronous approaches.

On the other hand, pre-runtime schedulers are not very flexible when it comes to dealing with sporadic tasks or system upgrading. Although some work has been done to overcome these deficiencies [19], much more is still needed. On-line schedulers may provide some of this flexibility, for example by providing better processor utilisation when sporadic tasks are considered [105, 113]. Nevertheless, more complex operating systems are required in this approach which reduces the available time to run application tasks. Also, complex process interactions are not dealt with properly. For example, high overheads are incurred for precedence relations [106].

The use of co-design partitioning can help solve the problem of low software host utilisation due to sporadic task responsiveness found in pre-runtime scheduling methods, since during partitioning those tasks can be moved to hardware processors, whenever necessary.