Embedded Systems

11. Architecture Synthesis

Lothar Thiele
Contents of Course

1. Embedded Systems Introduction
   2. Software Introduction
   3. Real-Time Models
      4. Periodic/Aperiodic Tasks
      5. Resource Sharing
   6. Real-Time OS
   7. System Components
   8. Communication
   9. Low Power Design
   10. Models
      11. Architecture Synthesis
   12. Model Based Design

Software and Programming

Processing and Communication

Hardware
Contents

- **Models**
  - Scheduling without resource constraints
    - ASAP
    - ALAP
    - Timing Constraints
  - Scheduling with resource constraints
    - List Scheduling
    - Integer Linear Programming
  - Iterative Algorithms
  - Dynamic Voltage Scaling
Architecture Synthesis

Determine a hardware architecture that efficiently executes a given algorithm.

- **Major tasks** of architecture synthesis:
  - *allocation* (determine the necessary hardware resources)
  - *scheduling* (determine the timing of individual operations)
  - *binding* (determine relation between individual operations of the algorithm and hardware resources)

- **Classification** of synthesis algorithms:
  - *heuristics* or *exact methods*

- Synthesis methods can often be applied *independently of granularity* of algorithms, e.g. whether operation is a whole complex task or a single operation.
Models

- **Sequence graph** \( G_S = (V_S, E_S) \) where \( V_S \) denotes the operations of the algorithm and \( E_S \) the dependence relations.

- **Resource graph** \( G_R = (V_R, E_R) \), \( V_R = V_S \cup V_T \) where \( V_T \) denote the resource types of the architecture and \( G_R \) is a bipartite graph. An edge \( (v_s, v_t) \in E_R \) represents the availability of a resource type \( v_t \) for an operation \( v_s \).

- **Cost function** \( c : V_T \rightarrow \mathbb{Z} \)

- **Execution times** \( w : E_R \rightarrow \mathbb{Z}^{\geq 0} \) are assigned to each edge \( (v_s, v_t) \in E_R \) and denote the execution time of operation \( v_s \in V_S \) on resource type \( v_t \in V_T \).
Models

**Example** sequence graph:

- Given *algorithm* (differential equation):

```c
int diffeq(int x, int y, int u, int dx, int a) {
    int x1, u1, y1;
    while ( x < a ) {
        x1 = x + dx;
        u1 = u - (3 * x * u * dx) - (3 * y * dx);
        y1 = y + u * dx;
        x = x1;
        u = u1;
        y = y1;
    }
    return y;
}
```
Models

- **Sequence graph:**

![Sequence graph diagram](image-url)
Models

*Resource graph:*

- Multiplier: $\alpha(r_1) = 1$
  - $c(r_1) = 8$

- ALU: $\alpha(r_2) = 1$
  - $c(r_2) = 3$
Allocation and Binding

An allocation is a function $\alpha : V_T \rightarrow \mathbb{Z}^{\geq 0}$ that assigns to each resource type $v_t \in V_T$ the number $\alpha(v_t)$ of available instances.

A binding is defined by functions $\beta : V_S \rightarrow V_T$ and $\gamma : V_S \rightarrow \mathbb{Z}^{>0}$. Here, $\beta(v_s) = v_t$ and $\gamma(v_s) = r$ denote that operation $v_s \in V_S$ is implemented on the $r$th instance of resource type $v_t \in V_T$. 
A schedule is a function \( \tau : V_S \rightarrow \mathbb{Z}^+ \) that determines the starting times of operations. A schedule is feasible if the conditions

\[
\tau(v_j) - \tau(v_i) \geq w(v_i) \quad \forall (v_i, v_j) \in E_S
\]

are satisfied. \( w(v_i) = w(v_i, \beta(v_i)) \) denotes the execution time of operation \( v_i \).

The latency \( L \) of a schedule is the time difference between start node \( v_0 \) and end node \( v_n \):

\[
L = \tau(v_n) - \tau(v_0)
\]
**Scheduling**

- **Example:**

  \[ L = \tau(v_{12}) - \tau(v_0) = 7 \]

  \[
  \begin{align*}
  \tau(v_0) &= 1 \\
  \tau(v_1) &= \tau(v_{10}) = 1 \\
  \tau(v_2) &= \tau(v_{11}) = 2 \\
  \tau(v_3) &= 3 \\
  \tau(v_6) &= \tau(v_4) = 4 \\
  \tau(v_7) &= 5 \\
  \tau(v_8) &= \tau(v_5) = 6 \\
  \tau(v_9) &= 7 \\
  \tau(v_{12}) &= 8 
  \end{align*}
  \]
Binding

Example ($\alpha(r_1) = 4$, $\alpha(r_2) = 2$):

$\beta(v_1) = r_1, \gamma(v_1) = 1$,
$\beta(v_2) = r_1, \gamma(v_2) = 2$,
$\beta(v_3) = r_1, \gamma(v_3) = 2$,
$\beta(v_4) = r_2, \gamma(v_4) = 1$,
$\beta(v_5) = r_2, \gamma(v_5) = 1$,
$\beta(v_6) = r_1, \gamma(v_6) = 3$,
$\beta(v_7) = r_1, \gamma(v_7) = 3$,
$\beta(v_8) = r_1, \gamma(v_8) = 4$,
$\beta(v_9) = r_2, \gamma(v_9) = 1$,
$\beta(v_{10}) = r_2, \gamma(v_{10}) = 2$,
$\beta(v_{11}) = r_2, \gamma(v_{11}) = 2$
Multiobjective Optimization

- Architecture Synthesis is an optimization problem with more than one objective:
  - Latency of the algorithm that is implemented
  - Hardware cost (memory, communication, computing units, control)
  - Power and energy consumption

- Optimization problems with several objectives are called “multiobjective optimization problems”.
## Multiobjective Optimization

Let us suppose, we would like to select a typewriting device. Criteria are

- mobility (related to weight)
- comfort (related to keyboard size and performance)

<table>
<thead>
<tr>
<th>Icon</th>
<th>Device</th>
<th>weight (kg)</th>
<th>comfort rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>🖥️</td>
<td>PC of 2009</td>
<td>20.00</td>
<td>10</td>
</tr>
<tr>
<td>🖥️</td>
<td>PC of 1984</td>
<td>7.50</td>
<td>7</td>
</tr>
<tr>
<td>🖥️</td>
<td>Laptop</td>
<td>3.00</td>
<td>9</td>
</tr>
<tr>
<td>🖥️</td>
<td>Typewriter</td>
<td>9.00</td>
<td>5</td>
</tr>
<tr>
<td>🖥️</td>
<td>Touchscreen Smartphone</td>
<td>0.11</td>
<td>2</td>
</tr>
<tr>
<td>🖥️</td>
<td>PDA with large keyboard</td>
<td>0.09</td>
<td>3</td>
</tr>
<tr>
<td>🖥️</td>
<td>PDA with small keyboard</td>
<td>0.11</td>
<td>4</td>
</tr>
<tr>
<td>🖥️</td>
<td>Organizer with tiny keyboard</td>
<td>0.08</td>
<td>1</td>
</tr>
</tbody>
</table>
Multiobjective Optimization
Pareto-Dominance

**Definition**: A solution $a \in \mathcal{X}$ weakly Pareto-dominates a solution $b \in \mathcal{X}$, denoted as $a \preceq b$, if it is as least as good in all objectives, i.e., $f_i(a) \leq f_i(b)$ for all $1 \leq i \leq n$. Solution $a$ is better than $b$, denoted as $a < b$, iff $(a \preceq b) \land (b \not\preceq a)$.
Pareto-optimal Set

A solution is named **Pareto-optimal**, if it is not Pareto-dominated by any other solution in X.

The set of all Pareto-optimal solutions is denoted as the Pareto-optimal set and its image in objective space as the **Pareto-optimal front**.
Contents

- Models
- Scheduling without resource constraints
  - ASAP
  - ALAP
  - Timing Constraints
- Scheduling with resource constraints
  - List Scheduling
  - Integer Linear Programming
- Iterative Algorithms
- Dynamic Voltage Scaling
Scheduling Algorithms

Classification

- **unlimited resources**: no constraints in terms of the available resources are defined.
- **limited resources**: constrains are given in terms of the number a types of available resources.

- **iterative algorithms**: an initial solution to the architecture synthesis is improved step by step.
- **constructive algorithms**: the synthesis problem is solved in one step.
- **transformative algorithms**: the initial problem formulation is converted into a (classical) optimization problem.
Scheduling Without Resource Constraints

The scheduling method can be used
- as a **preparatory step** for the general synthesis problem
- to **determine bounds** on feasible schedules in the general case
- if there is a **dedicated resource** for each operation.

Given is a sequence graph $G_S = (V_S, E_S)$ and a resource graph $G_R = (V_R, E_R)$. Then the latency minimization without resource constraints is defined as

$$L = \min \{ \tau(v_n) : \tau(v_j) - \tau(v_i) \geq w(v_i) \ \forall (v_i, v_j) \in E_S \}$$
Contents

- Models
- Scheduling without resource constraints
  - ASAP
  - ALAP
  - Timing Constraints
- Scheduling with resource constraints
  - List Scheduling
  - Integer Linear Programming
- Iterative Algorithms
- Dynamic Voltage Scaling
The ASAP Algorithm

**ASAP = As Soon As Possible**

\[
\text{ASAP}(G_S(V_S, E_S), w) \{ \\
\quad \tau(v_0) = 1; \\
\quad \text{REPEAT} \{ \\
\quad \quad \text{Determine } v_i \text{ whose predec. are planned; } \\
\quad \quad \tau(v_i) = \max\{\tau(v_j) + w(v_j) \ \forall (v_j, v_i) \in E_S\} \\
\quad \} \text{ UNTIL (} v_n \text{ is planned); } \\
\quad \text{RETURN } (\tau); \\
\}
\]
The ASAP Algorithm

**Example:**

\[ w(v_i) = 1 \]
Contents

- Models
- Scheduling without resource constraints
  - ASAP
  - \textit{ALAP}
  - Timing Constraints
- Scheduling with resource constraints
  - List Scheduling
  - Integer Linear Programming
- Iterative Algorithms
- Dynamic Voltage Scaling
The ALAP Algorithm

ALAP = As Late As Possible

ALAP(G_S(V_S, E_S), w, L_{max}) {
    \tau(v_n) = L_{max} + 1;
    \text{REPEAT} \{
        \text{Determine } v_i \text{ whose succ. are planned;}
        \tau(v_i) = \min\{\tau(v_j) \forall (v_i, v_j) \in E_S\} - w(v_i)
    \} \text{ UNTIL } (v_0 \text{ is planned});
    \text{RETURN } (\tau);
}
The ALAP Algorithm

Example:

\( L_{\text{max}} = 7 \)
\( w(v_i) = 1 \)
Contents

- Models
- Scheduling without resource constraints
  - ASAP
  - ALAP
  - *Timing Constraints*
- Scheduling with resource constraints
  - List Scheduling
  - Integer Linear Programming
- Iterative Algorithms
- Dynamic Voltage Scaling
Scheduling with Timing Constraints

- Different classes of timing constraints:
  - **deadline** (latest finishing times of operations), for example
    \[ \tau(v_2) + w(v_2) \leq 5 \]
  - **release times** (earliest starting times of operations), for example
    \[ \tau(v_3) \geq 4 \]
  - **relative constraints** (differences between starting times of a pair of operations), for example
    \[ \tau(v_6) - \tau(v_7) \geq 4 \]
    \[ \tau(v_4) - \tau(v_1) \leq 2 \]
Scheduling with Timing Constraints

We will model all timing constraints using \textit{relative constraints}. Deadlines and release times are defined relative to the start node \( v_0 \).

Minimum, maximum and equality constraints can be converted into each other:

- **Minimum constraint:**
  \[
  \tau(v_j) \geq \tau(v_i) + l_{ij} \quad \rightarrow \quad \tau(v_j) - \tau(v_i) \geq l_{ij}
  \]

- **Maximum constraint:**
  \[
  \tau(v_j) \leq \tau(v_i) + l_{ij} \quad \rightarrow \quad \tau(v_i) - \tau(v_j) \geq -l_{ij}
  \]

- **Equality constraint:**
  \[
  \tau(v_j) = \tau(v_i) + l_{ij} \quad \rightarrow \quad \tau(v_j) - \tau(v_i) \leq l_{ij} \land \tau(v_j) - \tau(v_i) \geq l_{ij}
  \]
Weighted Constraint Graph

Timing constraints can be represented in form of a weighted constraint graph:

A weighted constraint graph $G_C = (V_C, E_C, d)$ related to a sequence graph $G_S = (V_S, E_S)$ contains nodes $V_C = V_S$ and a weighted edge for each timing constraint. An edge $(v_i, v_j) \in E_C$ with weight $d(v_i, v_j)$ denotes the constraint $\tau(v_j) - \tau(v_i) \geq d(v_i, v_j)$. 
Weighted Constraint Graph

In order to represent a feasible schedule, we have one edge corresponding to each precedence constraint with

\[ d(v_i, v_j) = w(v_i) \]

where \( w(v_i) \) denotes the execution time of \( v_i \).

A consistent assignment of starting times \( \tau(v_i) \) to all operations can be done by solving a single source longest path problem.

A possible algorithm (**Bellman-Ford**) has complexity \( O(|V_C| |E_C|) \):

Iteratively set \( \tau(v_j) := \max\{\tau(v_j), \tau(v_i) + d(v_i, v_j) : (v_i, v_j) \in E_C\} \) for all \( v_j \in V_C \) starting from \( \tau(v_i) = -\infty \) for \( v_i \in V_C \setminus \{v_0\} \) and \( \tau(v_0) = 1 \).
Weighted Constraint Graph

**Example:** \( w(v_1) = w(v_3) = 2 \quad w(v_2) = w(v_4) = 1 \)
\[
\tau(v_0) = \tau(v_1) = \tau(v_3) = 1, \quad \tau(v_2) = 3,
\]
\[
\tau(v_4) = 5, \quad \tau(v_n) = 6, \quad L = \tau(v_n) - \tau(v_0) = 5
\]
Contents

- Models
- Scheduling without resource constraints
  - ASAP
  - ALAP
  - Timing Constraints
- Scheduling with resource constraints
  - List Scheduling
  - Integer Linear Programming
- Iterative Algorithms
- Dynamic Voltage Scaling
Scheduling With Resource Constraints

Given is a sequence graph $G_S = (V_S, E_S)$, a resource graph $G_R = (V_R, E_R)$ and an associated allocation $\alpha$ and binding $\beta$.

Then the minimal latency is defined as

\[
L = \min \{ \tau(v_n) : \\
(\tau(v_j) - \tau(v_i) \geq w(v_i, \beta(v_i)) \forall (v_i, v_j) \in E_S) \land \\
(|\{v_s : \beta(v_s) = v_t \land \tau(v_s) \leq t < \tau(v_s) + w(v_s, v_t)\}| \leq \alpha(v_t) \forall v_t \in V_T, \forall 1 \leq t \leq L_{max})\}
\]

where $L_{max}$ denotes an upper bound on the latency.
Contents

- Models
- Scheduling without resource constraints
  - ASAP
  - ALAP
  - Timing Constraints
- Scheduling with resource constraints
  - List Scheduling
  - Integer Linear Programming
- Iterative Algorithms
- Dynamic Voltage Scaling
List Scheduling

\[
\text{LIST}(G_S(V_S, E_S), G_R(V_R, E_R), \alpha, \beta, \text{priorities}) \{ \\
 t = 1; \\
 \text{REPEAT} \{ \\
 \quad \text{FORALL } v_k \in V_T \{ \\
 \quad \quad \text{determine candidates to be scheduled } U_k; \\
 \quad \quad \text{determine running operations } T_k; \\
 \quad \quad \text{choose } S_k \subseteq U_k \text{ with maximal priority} \\
 \quad \quad \quad \text{and } |S_k| + |T_k| \leq \alpha(v_k); \\
 \quad \quad \tau(v_i) = t \ \forall v_i \in S_k; \} \\
 t = t + 1; \\
 \} \text{ UNTIL } (v_n \text{ planned}) \\
\text{RETURN } (\tau); \}
\]
List Scheduling

One of the most *widely used* algorithms for scheduling under resource constraints.

**Principles:**

- To each operation there is a *priority* assigned which denotes the urgency of being scheduled. This *priority is static*, i.e. determined before the List Scheduling.
- The algorithm schedules *one time step after the other*.
- $U_k$ denotes the set of operations that (a) are mapped onto resource $v_k$ and whose predecessors finished.
- $T_k$ denotes the currently running operations mapped to resource $v_k$. 
List Scheduling

**Example:**

![Diagram of a list scheduling example with nodes labeled 1 to 5 and two graphs GS and GR with nodes r1 and r2 and labels α(r1) = 1 and α(r2) = 1.]
List Scheduling

Solution via *list scheduling*:
- In the example, the solution is independent of priority.
- Because of the *greedy* principle, all resources are directly occupied.
- List scheduling is a *heuristic algorithm*.

In this example, it does not yield the minimal latency!
List Scheduling

Solution via an *optimal method*:

- *Latency is smaller* than with list scheduling.
- An example of an optimal algorithm is the transformation into an *integer linear program*.
Contents

- Models
- Scheduling without resource constraints
  - ASAP
  - ALAP
  - Timing Constraints
- Scheduling with resource constraints
  - List Scheduling
  - *Integer Linear Programming*
- Iterative Algorithms
- Dynamic Voltage Scaling
**Integer Linear Programming**

**Principle:**

Synthesis Problem → transformation into ILP → Integer Linear Program (ILP) → optimization of ILP → Solution of ILP → back interpretation → Solution of Synthesis Problem
Integer Linear Program

- Yields **optimal solution** to synthesis problems as it is based on an exact mathematical description of the problem.
- Solves **scheduling, binding and allocation** simultaneously.
- **Standard optimization** approaches (and software) are available to solve integer linear programs:
  - in addition to linear programs (linear constraints, linear objective function) some variables are forced to be integers.
  - much more complex than solving linear program
  - efficient methods are based on (a) branch and bound methods and (b) determining additional hyperplanes (cuts).
Many **variants** exist, depending on available information, constraints and objectives, e.g. minimize latency, minimize resources, minimize memory. Just an example is given here!!

For the following example, we use the **assumptions**:  
- The **binding is determined** already, i.e. every operation $v_i$ has a unique execution time $w(v_i)$.  
- We have determined the **earliest and latest starting times** of operations $v_i$ as $l_i$ and $h_i$, respectively. To this end, we can use the ASAP and ALAP algorithms that have been introduced earlier. The maximal latency $L_{\text{max}}$ is chosen such that a feasible solution to the problem exists.
Integer Linear Program

minimize: \( \tau(v_n) - \tau(v_0) \)

subject to \( x_{i,t} \in \{0, 1\} \quad \forall v_i \in V_S \quad \forall t : l_i \leq t \leq h_i \) (1)

\[
\sum_{t=l_i}^{h_i} x_{i,t} = 1 \quad \forall v_i \in V_S
\] (2)

\[
\sum_{t=l_i}^{h_i} t \cdot x_{i,t} = \tau(v_i) \quad \forall v_i \in V_S
\] (3)

\( \tau(v_j) - \tau(v_i) \geq w(v_i) \quad \forall (v_i, v_j) \in E_S \) (4)

\[
\sum_{\forall i : (v_i, v_k) \in E_R} \sum_{p' = \max\{0, t-h_i\}} \min\{w(v_i)-1, t-l_i\} \]

\( x_{i,t-p'} \leq \alpha(v_k) \)

\( \forall v_k \in V_T \quad \forall t : 1 \leq t \leq \max\{h_i : v_i \in V_S\} \) (5)
Integer Linear Program

**Explanations:**

- (1) declares variables $x$ to be binary.
- (2) makes sure that exactly one variable $x_{i,t}$ for all $t$ has the value 1, all others are 0.
- (3) determines the relation between variables $x$ and starting times of operations $\tau$. In particular, if $x_{i,t} = 1$ then the operation $v_i$ starts at time $t$, i.e. $\tau(v_i) = t$.
- (4) guarantees, that all precedence constraints are satisfied.
- (5) makes sure, that the resource constraints are not violated. For all resource types $v_k \in V_T$ and for all time instances $t$ it is guaranteed that the number of active operations does not increase the number of available resource instances.
**Explanations:**

- (5) The first sum selects all operations that are mapped onto resource type $v_k$. The second sum considers all time instances where operation $v_i$ is occupying resource type $v_k$:

$$w(v_i) - 1 \sum_{p' = 0}^{x_{i,t-p'}} = \begin{cases} 
1 & : \forall t : \tau(v_i) \leq t \leq \tau(v_i) + w(v_i) - 1 \\
0 & : \text{sonst}
\end{cases}$$
Contents

- Models
- Scheduling without resource constraints
  - ASAP
  - ALAP
  - Timing Constraints
- Scheduling with resource constraints
  - List Scheduling
  - Integer Linear Programming
- Iterative Algorithms
- Dynamic Voltage Scaling
Iterative Algorithms

**Iterative algorithms** consist of a set of indexed equations that are evaluated for all values of an index variable $l$:

$$x_i[l] = F_i[\ldots, x_j[l - d_{ji}], \ldots] \quad \forall l \quad \forall i \in I$$

Here, $x_i$ denote a set of indexed variables, $F_i$ denote arbitrary functions and $d_{ji}$ are constant index displacements.

Examples of well known representations are **signal flow graphs** (as used in signal and image processing and automatic control), **marked graphs** and special forms of **loops**.
Iterative Algorithms

Several representations of the same iterative algorithm:

- One indexed equation with constant index dependencies:

\[ y[l] = au[l] + by[l - 1] + cy[l - 2] + dy[l - 3] \quad \forall l \]

- Equivalent set of indexed equations:

\[ x_1[l] = au[l] \quad \forall l \]
\[ x_2[l] = x_1[l] + dy[l - 3] \quad \forall l \]
\[ x_3[l] = x_2[l] + cy[l - 2] \quad \forall l \]
\[ y[l] = x_3[l] + by[l - 1] \quad \forall l \]
Iterative Algorithms

- **Extended sequence graph** $G_S = (V_S, E_S, d)$: To each edge $(v_i, v_j) \in E_S$ there is associated the *index displacement* $d_{ij}$. An edge $(v_i, v_j) \in E_S$ denotes that the variable corresponding to $v_j$ depends on variable corresponding to $v_i$ with displacement $d_{ij}$.

- Equivalent *marked graph*:
Iterative Algorithms

- Equivalent *signal flow graph*:

```
  u  a  d  c  b  y
```

- Equivalent *loop program*:

```java
while(true) {
  t1 = read(u);
  t5 = a*t1 + d*t2 + c*t3 + b*t4;
  t2 = t3;
  t3 = t4;
  t4 = t5;
  write(y, t5);
}
```
Iterative Algorithms

- An *iteration* is the set of all operations necessary to compute all variables $x_i[l]$ for a fixed index $l$.
- The *iteration interval* $P$ is the time distance between two successive iterations of an iterative algorithm. $1/P$ denotes the *throughput* of the implementation.
- The *latency* $L$ is the maximal time distance between the starting and the finishing times of operations belonging to one iteration.
- In a pipelined implementation (*functional pipelining*), there exist time instances where the operations of different iterations $l$ are executed simultaneously.
- In case of *loop folding*, starting and finishing times of an operation are in different physical iterations.
**Iterative Algorithms**

- **Implementation principles**
  - A *simple possibility*, the edges with $d_{ij} > 0$ are removed from the extended sequence graph. The resulting simple sequence graph is implemented using *standard methods*.

**Example** with unlimited resources:

![Diagram of a sequence graph with labeled nodes and execution times.](image)

- Execution times $w(v_i)$
- One iteration
- One physical iteration
- $L = 7$
- $P = 7$
- No pipelining
Iterative Algorithms

- **Implementation principles**
  - Using *functional pipelining*: Successive iterations overlap and a higher throughput (1/P) is obtained.

**Example** with unlimited resources (note data dependencies across iterations!)

\[
\begin{array}{cccccc}
 & & & & & \\
& u & 0 & x_1 & 0 & x_2 & 0 & x_3 & 0 & 1 \\
0 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
\end{array}
\]

- 4 resources
- functional pipelining
- loop folding

**Diagram**

- One physical iteration
- One iteration
- \( P = 2 \)
- \( L = 7 \)
Iterative Algorithms

- Solving the synthesis problem using *integer linear programming*:
  - Starting point is the ILP formulation given for simple sequence graphs.
  - Now, we use the *extended sequence graph* (including displacements $d_{ij}$).
  - *ASAP* and *ALAP* scheduling for upper and lower bounds $h_i$ and $l_i$ use only edges with $d_{ij} = 0$ (remove dependencies across iterations).
  - We suppose, that a suitable *iteration interval* $P$ is chosen beforehand. If it is too small, no feasible solution to the ILP exists and $P$ needs to be increased.
Iterative Algorithms

- Eqn.(4) is replaced by:

\[
\tau(v_j) - \tau(v_i) \geq w(v_i) - d_{ij} \cdot P \quad \forall (v_i, v_j) \in E_S
\]

Proof of correctness:

\[
\tau(v_i) + w(v_i) \leq \tau(v_j) + d_{ij} \cdot P
\]
Iterative Algorithms

- Eqn. (5) is replaced by

\[ \sum_{\forall i: (v_i, v_k) \in E_R} \sum_{p' = 0}^{w(v_i) - 1} \sum_{\forall p: l_i \leq t - p' + p \cdot P \leq h_i} x_{i, t - p' + p \cdot P} \leq \alpha(v_k) \]

\[ \forall 1 \leq t \leq P, \forall v_k \in V_T \]

Sketch of **Proof**: An operation \( v_i \) starting at \( \tau(v_i) \) uses the corresponding resource at time steps \( t \) with

\[ t = \tau(v_i) + p' - p \cdot P \]

\[ \forall p', p : 0 \leq p' < w(v_i) \land l_i \leq t - p' + p \cdot P \leq h_i \]

Therefore, we obtain

\[ \sum_{p' = 0}^{w(v_i) - 1} \sum_{\forall p: l_i \leq t - p' + p \cdot P \leq h_i} x_{i, t - p' + p \cdot P} \]
Contents

- Models
- Scheduling without resource constraints
  - ASAP
  - ALAP
  - Timing Constraints
- Scheduling with resource constraints
  - List Scheduling
  - Integer Linear Programming
- Iterative Algorithms
- *Dynamic Voltage Scaling*
Dynamic Voltage Scaling

- If we transform the DVS problem into an integer linear program optimization: we can *optimize the energy* in case of *dynamic voltage scaling*.

- As an *example*, let us model a set of tasks with dependency constraints.
  - We suppose that a task $v_i \in V_S$ can use one of the execution times $w_k(v_i) \forall k \in K$ and corresponding energy $e_k(v_i)$. There are $|K|$ different voltage levels.
  - We suppose that there are *deadlines* $d(v_i)$ for each operation $v_i$.
  - We suppose that there are no resource constraints, i.e. all tasks can be executed in parallel.
Dynamic Voltage Scaling

minimize: \[ \sum_{k \in K} \sum_{v_i \in V_S} y_{ik} \cdot e_k(v_i) \]
subject to:
\[ y_{ik} \in \{0, 1\} \quad \forall v_i \in V_S, k \in K \quad (1) \]
\[ \sum_{k \in K} y_{ik} = 1 \quad \forall v_i \in V_S \quad (2) \]
\[ \tau(v_j) - \tau(v_i) \geq \sum_{k \in K} y_{ik} \cdot w_k(v_i) \quad \forall (v_i, v_j) \in E_S \quad (3) \]
\[ \tau(v_i) + \sum_{k \in K} y_{ik} \cdot w_k(v_i) \leq d(v_i) \quad \forall v_i \in V_S \quad (4) \]
Dynamic Voltage Scaling

**Explanations:**

- The objective functions just sums up all individual energies of operations.
- Eqn. (1) makes decision variables $y_{ik}$ binary.
- Eqn. (2) guarantees that exactly one implementation (voltage) $k \in K$ is chosen for each operation $v_i$.
- Eqn. (3) implements the precedence constraints, where the actual execution time is selected from the set of all available ones.
- Eqn. (4) guarantees deadlines.