Hardware-Software Codesign

4. System Partitioning

Lothar Thiele
System Design

specification

system synthesis

SW-compilation

instruction set

HW-synthesis

estimation

machine code

net lists

intellectual prop. code

intellectual prop. block
Mapping transforms behavior into structure and execution:

- **allocation**: select components
- **binding**: assign functions to components
- **scheduling**: determine execution order
- ... finally, synthesis results into **implementation**
Levels of Abstractions

Mapping can be done

- at low level: register transfer level (RTL) or netlist level
  - e.g., split a digital circuit and map it to several devices (FPGAs, ASICs)
  - system parameters (e.g., area, delay) relatively easy to determine

- at high level: system level
  - comparison of design alternatives for optimality (design space exploration)
  - system parameters are unknown and difficult to determine
    → to be estimated via analysis, simulation, (rapid) prototyping
Model-Based Synthesis – Example

- considered performance
  - cost $C$: cost of allocated components, e.g., sum
  - latency $L$: due to scheduling (resource sharing)

- conflicting design goals
  - feasible schedule $L \leq L_{\text{max}}$
  - feasible allocation $C \leq C_{\text{max}}$

**optimal $C$:** N:1 mapping

**optimal $L$:** 1:1 mapping
Example – Alternatives

**optimal C:** N:1 mapping

**optimal L:** 1:1 mapping
Cost Functions

*Quantitatively measure performance* of a design point

- system cost $C \, [\$]$
- latency $L \, [\text{sec}]$
- power consumption $P \, [\text{W}]$
- ...

*Estimation* is required to find $C, L, P$ values, for each design point

- *example*: linear cost (preference) function with penalty

\[
f(C, L, P) = k_1 \cdot h_C(C, C_{\max}) + k_2 \cdot h_L(L, L_{\max}) + k_3 \cdot h_P(P, P_{\max})
\]

- $h_C, h_L, h_P$ ... denote how strong $C, L, P$ violate design constraints $C_{\max}, L_{\max}, P_{\max}$
- $k_1, k_2, k_3$ ... weighting and normalization
The Formal Partitioning Problem

assign \(n\) objects \(O=\{o_1, \ldots, o_n\}\) to \(m\) blocks (also called partitions) \(P=\{p_1, \ldots, p_m\}\), such that

- \(p_1 \cup p_2 \cup \ldots \cup p_m = O\) (all objects are assigned – mapped)
- \(p_i \cap p_j = \emptyset\) \(\forall i,j: i \neq j\) (an object is not assigned or “mapped” twice)
- and costs \(c(P)\) are minimized

**Note:** in *system synthesis* (simple model)
- objects = process network graph nodes
- blocks = architecture graph nodes
- cost = measured/estimated with dedicated cost functions (e.g., latency, power, hardware cost)
Partitioning Methods

**Exact methods**
- enumeration
- integer linear programs (ILP) (→ see next slides)

**Heuristic methods**
- constructive methods
  - random mapping
  - hierarchical clustering
- iterative methods
  - Kernighan-Lin algorithm
  - simulated annealing
  - evolutionary algorithms
# Integer Programming Model

**Ingredients:**
- objective function (cost)
- constraints


\[
\text{objective } \quad C = \sum_{x_i \in X} a_i x_i \text{ with } a_i \in R, x_i \in \mathbb{N} \quad (1)
\]

\[
\text{constraints } \quad \forall j \in J : \sum_{x_i \in X} b_{i,j} x_i \geq c_j \text{ with } b_{i,j}, c_j \in \mathbb{R} \quad (2)
\]

**Integer programming (IP) problem:**

minimize objective function (1) subject to constraints (2)

*note:* if all \( x_i \) are constrained to be either 0 or 1, the IP problem is said to be a 0/1 integer programming problem.
Small Example of 0/1 IP

minimize: \[ C = 5x_1 + 6x_2 + 4x_3 \]

subject to: \[ x_1 + x_2 + x_3 \geq 2 \]
\[ x_1, x_2, x_3 \in \{0,1\} \]

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( C )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
</tbody>
</table>

optimal (minimal)
Integer Linear Program for Partitioning

- Binary variables $x_{i,k}$
  - $x_{i,k} = 1$: object $o_i$ in block $p_k$
  - $x_{i,k} = 0$: object $o_i$ not in block $p_k$

- Cost $c_{i,k}$, if object $o_i$ is in block $p_k$

- Integer linear program:

  \[
  \begin{align*}
  &\text{minimize} & \sum_{k=1}^{m} \sum_{i=1}^{n} x_{i,k} \cdot c_{i,k} & \quad 1 \leq k \leq m, 1 \leq i \leq n \\
  &\text{subject to} & \sum_{k=1}^{m} x_{i,k} = 1 & \quad 1 \leq i \leq n \\
  & & x_{i,k} \in \{0,1\} & \quad 1 \leq i \leq n, 1 \leq k \leq m
  \end{align*}
  \]
Example – Partitioning

e.g., optimized for a load balanced system

<table>
<thead>
<tr>
<th>task</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PE1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>exe. time</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE0</td>
<td>5</td>
<td>15</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>PE1</td>
<td>10</td>
<td>20</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

load balancing:
\[ \text{load}_{\text{PE0}} = 5 + 15 \]
\[ \text{load}_{\text{PE1}} = 10 + 10 \]
Variations in ILP

Additional constraints:

- e.g., maximum $h_k$ objects in block $k$

$$
\sum_{i=1}^{n} x_{i,k} \leq h_k \quad 1 \leq k \leq m
$$

Maximizing the cost function:

- can be done by setting $C' = -C$ in a minimization problem
**ILP for synthesis**

* Solving the synthesis problem with ILP is very popular:
  - If not solving to optimality, runtimes are acceptable and a solution with guaranteed quality can be determined.
  - Scheduling can be integrated.
  - Various additional constraints can be added.

- However, finding the right equations to model the constraints is an art.
Remarks on Integer Programming

**Integer programming is NP-complete**

- In practice, runtimes can increase exponentially with the size of the problem.

- But problems of some thousands of variables can still be solved with commercial solvers (depending on the size/structure of the problem) or approximation algorithms (heuristics).

- IP models can be a good starting point for designing heuristic optimization methods.
Partitioning Methods

- **exact methods**
  - enumeration
  - integer linear programs (ILP)

- **heuristic methods**
  - *constructive methods (see next slides)*
    - random mapping
    - hierarchical clustering
  - iterative methods
    - Kernighan-Lin algorithm
    - simulated annealing
    - evolutionary algorithms
Constructive Methods

Examples

- random mapping
  - each object is assigned to a block randomly

- hierarchical clustering
  - stepwise grouping of (e.g., two) objects
  - and evaluate closeness function (how desirable it is to group objects)

Constructive methods are often used to generate a starting partition for iterative methods
Hierarchical Clustering Example (1)

Closeness function: arithmetic mean of weights

$\mathbf{v}_5 = \mathbf{v}_1 \cup \mathbf{v}_3$
Hierarchical Clustering Example (2)

\[ v_6 = v_2 \cup v_5 \]
Hierarchical Clustering Example (3)

\[ V_7 = V_6 \cup V_4 \]
Hierarchical Clustering – Summary

step 0: \{v_1, v_2, v_3, v_4\}

step 1: \{v_2, v_4, v_5\}

step 2: \{v_4, v_6\}

step 3: \{v_7\}

\(v_7 = v_6 \cup v_4\)

\(v_6 = v_2 \cup v_5\)

\(v_5 = v_1 \cup v_3\)
Partitioning Methods

- **exact methods**
  - enumeration
  - integer linear programs (ILP)

- **heuristic methods**
  - constructive methods
    - random mapping
    - hierarchical clustering
  - iterative methods (**see next slides**)
    - Kernighan-Lin algorithm
    - simulated annealing
    - evolutionary algorithms
Iterative Methods (1)

*Often used principle for iterative methods:*

- start with some initial configuration (partitioning)
- repeatedly search *neighborhood* (similar partitions) and *select a neighbor* as candidate
- evaluate *fitness (cost) function of candidate*
  - accept candidate using acceptance rule
  - if not, select another neighbor
- stop if quality is sufficiently high, if no improvement can be found, or after some fixed time

*Ingredients:*

- initial configuration, function to find a *neighbor* as next candidate, cost function, acceptance rule, stop criterion
Iterative Methods (2)

Simple iterative improvement or “hill climbing”:
- candidate is always and only accepted if cost is lower (or fitness is higher) than current configuration
- stop when no neighbor with lower cost (higher fitness) can be found

Disadvantages:
- local optimum as best result
- local optimum depends on initial configuration
- generally no upper bound on iteration length
Iterative Methods – Illustration

Fitness

A
X

B

Hillclimb

C
How to Cope with Disadvantages?

- Repeat algorithm many times with different initial configurations
- Use information gathered in previous runs
- Use a more complex “acceptance rule” to jump out of local optimum
- Use a more complex strategy that accepts sometimes randomly generated solutions
Iterative Methods – Simple Greedy Heuristic

Iterate until no improvement in cost:
re-group the object pairs that leads to the largest cost gain

Example: cost = number of edges crossing the partitions
before re-group: 5; after re-group: 4; gain = 1
Iterative Methods – Kernighan-Lin

*Improved algorithm: Kernighan-Lin:*

- as long as a better partition is found
  - from all possible pairs of objects
    - *virtually* re-group the “best” (lowest cost of resulting partition)
  - from the remaining (not yet touched) objects
    - *virtually* re-group the “best” pair
  - continue until all objects have been re-grouped
  - from these $n/2$ partitions, take the one with smallest cost and *actually* perform the corresponding re-group operations
Illustration of KL Algorithm (1)

Example: partitioning of digital circuit

communication cost from node x to node y

cost matrix c(x,y)

\[
\begin{array}{cccccccc}
  & a & b & c & d & e & f & g & h \\
 a & 0 & 0 & .5 & 0 & .5 & 0 & 0 & 0 \\
b & 0 & 0 & .5 & .5 & 0 & 0 & 0 & 0 \\
c & .5 & .5 & 0 & .5 & 1 & .5 & 0 & 0 \\
d & 0 & .5 & .5 & 0 & 0 & 1 & 0 & 0 \\
e & .5 & 0 & 1 & 0 & 0 & .5 & 1 & 0 \\
f & 0 & 0 & .5 & 1 & .5 & 0 & .5 & .5 \\
g & 0 & 0 & 0 & 0 & 1 & .5 & 0 & .5 \\
h & 0 & 0 & 0 & 0 & 0 & .5 & .5 & 0 \\
\end{array}
\]
**Illustration of KL Algorithm (2)**

**first re-group**

<table>
<thead>
<tr>
<th>pair</th>
<th>( E_x - I_x )</th>
<th>( E_y - I_y )</th>
<th>( c(x, y) )</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a, c)</td>
<td>0.5 – 0.5</td>
<td>2.5 – 0.5</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>(a, f)</td>
<td>0.5 – 0.5</td>
<td>1.5 – 1.5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(a, g)</td>
<td>0.5 – 0.5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(a, h)</td>
<td>0.5 – 0.5</td>
<td>0 – 1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>(b, c)</td>
<td>0.5 – 0.5</td>
<td>2.5 – 0.5</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>(b, f)</td>
<td>0.5 – 0.5</td>
<td>1.5 – 1.5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(b, g)</td>
<td>0.5 – 0.5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(b, h)</td>
<td>0.5 – 0.5</td>
<td>0 – 1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>(d, c)</td>
<td>1.5 – 0.5</td>
<td>2.5 – 0.5</td>
<td>0.5</td>
<td>2</td>
</tr>
<tr>
<td>(d, f)</td>
<td>1.5 – 0.5</td>
<td>1.5 – 1.5</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>(d, g)</td>
<td>1.5 – 0.5</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(d, h)</td>
<td>1.5 – 0.5</td>
<td>0 – 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(e, c)</td>
<td>2.5 – 0.5</td>
<td>2.5 – 0.5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(e, f)</td>
<td>2.5 – 0.5</td>
<td>1.5 – 1.5</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>(e, g)</td>
<td>2.5 – 0.5</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(e, h)</td>
<td>2.5 – 0.5</td>
<td>0 – 1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**some definitions**

- \( E_i \) = external costs of vertex \( i \)
- \( I_i \) = internal costs of vertex \( i \)
- \( D_i = E_i - I_i \) = desirability to move a vertex (\( x \) or \( y \))
- \( \text{gain} = D_x + D_y - 2* c(x, y) \) = gain due to change in cut costs
Illustration of KL Algorithm (3)

second re-group

<table>
<thead>
<tr>
<th>pair</th>
<th>$E_x - I_x$</th>
<th>$E_y - I_y$</th>
<th>$c(x, y)$</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(a, f)$</td>
<td>0 – 1</td>
<td>1 – 2</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>$(a, g)$</td>
<td>0 – 1</td>
<td>1 – 1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>$(a, h)$</td>
<td>0 – 1</td>
<td>0 – 1</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>$(b, f)$</td>
<td>0.5 – 0.5</td>
<td>1 – 2</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>$(b, g)$</td>
<td>0.5 – 0.5</td>
<td>1 – 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$(b, h)$</td>
<td>0.5 – 0.5</td>
<td>0 – 1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>$(e, f)$</td>
<td>1.5 – 1.5</td>
<td>1 – 2</td>
<td>0.5</td>
<td>-2</td>
</tr>
<tr>
<td>$(e, g)$</td>
<td>1.5 – 1.5</td>
<td>1 – 1</td>
<td>1</td>
<td>-2</td>
</tr>
<tr>
<td>$(e, h)$</td>
<td>1.5 – 1.5</td>
<td>0 – 1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

**some definitions**

- $E_i$ = external costs of vertex $i$
- $I_i$ = internal costs of vertex $i$
- $D_i = E_i - I_i$ = desirability to move a vertex ($x$ or $y$)
- gain = $D_x + D_y - 2 \times c(x, y)$ = gain due to change in cut costs
Illustration of KL Algorithm (4)

third re-group

<table>
<thead>
<tr>
<th>pair</th>
<th>$E_x - I_x$</th>
<th>$E_y - I_y$</th>
<th>$c(x, y)$</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a, f)</td>
<td>0 − 1</td>
<td>1.5 − 1.5</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>(a, h)</td>
<td>0 − 1</td>
<td>0.5 − 0.5</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>(e, f)</td>
<td>0.5 − 2.5</td>
<td>1.5 − 1.5</td>
<td>0.5</td>
<td>-3</td>
</tr>
<tr>
<td>(e, h)</td>
<td>0.5 − 2.5</td>
<td>0.5 − 0.5</td>
<td>0</td>
<td>-2</td>
</tr>
</tbody>
</table>

some definitions

- $E_i$ = external costs of vertex $i$
- $I_i$ = internal costs of vertex $i$
- $D_i = E_i - I_i$ = desirability to move a vertex ($x$ or $y$)
- $\text{gain} = D_x + D_y - 2c(x, y)$ = gain due to change in cut costs
Illustration of KL Algorithm (5)

... and final re-group
Illustration of KL Algorithm (6)

- Two best solutions found:

<table>
<thead>
<tr>
<th>$i$</th>
<th>pair</th>
<th>$gain(i)$</th>
<th>$\sum gain(i)$</th>
<th>cutsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>$(d, c)$</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>$(b, g)$</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>$(a, f)$</td>
<td>-1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>$(e, h)$</td>
<td>-1</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

- Start from one of these solutions the whole process again … .
Simulated Annealing – Underlying Philosophy

- Inspired from the physical process of annealing (from metallurgy), where a “structured” lattice structure of a solid is achieved by
  1. heating up the solid to its melting point
  2. … and then slowly cooling down until it solidifies to a low-energy state
Simulated Annealing – Underlying Philosophy (2)

- Solids take on a **minimal-energy state** during cooling down *if the temperature is decreased sufficiently slowly*

- There is a non-zero probability that a particle “jumps” to a higher-energy state \((e_{i+1} > e_i)\):

\[
P(e_i, e_{i+1}, T) = e^{\frac{e_i - e_{i+1}}{k_B T}}
\]

- \(k_B = \text{Boltzmann constant}\)
- \(T = \text{temperature}\)
- \(e_i = \text{current energy state}\)
- \(e_{i+1} = \text{next energy state}\)
Simulated Annealing Applications

Application to combinatorial optimization:

- energy = cost of a solution (partition)

- cost decreases with temperature (a global parameter)

- increases in cost are accepted with a certain probability (that depends both on the difference between cost values and also on “temperature”)
Simulated Annealing Algorithm

By analogy with the physical process:

- replace existing solutions by (randomly generated) new feasible solutions from a neighborhood
- improve a solution by always accepting better-cost neighbors (if selected) but allow for a (stochastically) guided acceptance of worse-cost neighbors
- gradual cooling: gradually decrease the probability of accepting worse-cost solutions
  - selecting solutions is almost random when $T$ is large
  - … but increasingly selects the better cost solution as $T$ goes to zero

Advantage

- allowance for “uphill” moves potentially avoids local optima
Simulated Annealing – Possible Coding

temp = temp_start;
cost = c(P);

while (Frozen() == FALSE) {
    while (Equilibrium() == FALSE) {
        P' = RandomMove(P);
cost' = c(P');
deltacost = cost' - cost;
        if (Accept(deltacost, temp) > random[0,1]) {
            P = P';
cost = cost';
        }
    }
    temp = DecreaseTemp(temp);
}

Accept(deltacost, temp) = e \frac{-deltacost}{k\cdot temp}
Simulated Annealing – Possible Coding (contn.)

- **RandomMove**(P)
  - choose a random solution in the neighborhood of P

- **DecreaseTemp()**, **Frozen()**
  - cooling down; there are many different choices, for example:
    - initially: \( \text{temp} := 1.0; \)
    - in any iteration: \( \text{temp} := \alpha \times \text{temp} \) (typ.: \( 0.8 \leq \alpha \leq 0.99 \))
  - frozen after a certain time or if there is no further improvement

- **Equilibrium()**
  - usually after a defined number of iterations

- **Complexity**
  - from exponential to constant, depending on the choice of the functions **Equilibrium()**, **DecreaseTemp()**, and **Frozen()**
  - the longer the runtime, the better the quality of results