Outline

• Allocation of system components
• Estimation
  – Metrics and cost functions
  – How good is the estimation
• Partitioning
  – Basic algorithms
  – HW partitioning algorithms
  – HW/SW partitioning algorithms
  – System partitioning algorithms
Hardware/Software Codesign

- HW-components
- specification
- standard software (RTOS, ...)

• Decision based on hardware/software partitioning, a special case of hardware/software codesign.

Functionality to be implemented in software or in hardware?

- Decision based on hardware/software partitioning, a special case of hardware/software codesign.
Exploration

- Allocation,
- Partitioning,
- Transformation
- Estimation

solve these problems not by the given order, but iterate many times before we are satisfied with our system-level design.

The Partitioning Problem

**Definition**: The partitioning problem is
- to assign $n$ objects $O = \{o_1, ..., o_n\}$
- to $m$ blocks (also called partitions) $P = \{p_1, ..., p_m\}$, such that
  - $p_1 \cup p_2 \cup ... \cup p_m = O$
  - $p_i \cap p_j = \{\} \quad \forall \quad i, j: i \neq j$ and
  - cost $c(P)$ are minimized.
- In **system synthesis**:
  - objects = problem graph nodes
  - blocks = architecture graph nodes
Quality Metrics

- **HW Cost metrics:**
  - design area (# transistors, gates, registers, etc.)
  - Packaging cost (#pins)
- **SW Cost metrics**
  - Program memory size
  - Data memory size
- **Performance metrics**
- **Other metrics**

Cost Functions

**Measure quality** of a design point
- may include
  - \( C \) ... system cost in [$]
  - \( L \) ... latency in [sec]
  - \( P \) ... power consumption in [W]
- requires **estimation** to find \( C, L, P \)

**Example:** linear cost function with penalty
\[
f(C, L, P) = k_1 \cdot h_C(C; C_{\text{max}}) + k_2 \cdot h_L(L; L_{\text{max}}) + k_3 \cdot h_P(P; P_{\text{max}})
\]
- \( h_C, h_L, h_P \) ... denote how strong \( C, L, P \) violate the design constraints \( C_{\text{max}}, L_{\text{max}}, P_{\text{max}} \)
- \( k_1, k_2, k_3 \) ... weighting and normalization
Cost functions

\[ \text{Costfct} = \]
\[ = k_1 \cdot F (\text{component}_1; \text{size}, \text{component}_1; \text{size constr}) \]
\[ + k_2 \cdot F (\text{component}_2; \text{size}, \text{component}_2; \text{size constr}) \]
\[ + k_3 \cdot F (\text{component}_3; \text{IO}, \text{component}_3; \text{IO constr}) \ldots \]

- \( k \)'s are user provided constants indicating the relative importance of each metric, and
- \( F \) indicates the desirability of a metric's value. A common form of \( F \) returns the degree of constraint violation, normalized such that 0 = no violation, and 1 = very large violation. This form of \( F \) causes the cost function to return zero when a partition meets all constraints, making the goal of partitioning to obtain a cost of zero.

Behavior Closeness Metrics

- **Connectivity** - based on the number of wires shared between the sets of behaviors. Grouping behaviors that share wires should result in fewer pins.
- **Communication** is based on the number of bits of data transferred between the sets of behaviors, independent of the number of wires used to transfer the data. Grouping heavily communicating behaviors should result in better performance, due to decreased communication time.
- **Hardware sharing** is based on the estimated percentage of hardware that can be shared between two sets of behaviors. Grouping behaviors that can share hardware should result in a smaller overall hardware size.
Behavior Closeness Metrics (cont.)

- **Common accessors** is based on the number of behaviors that access both sets of behaviors. Grouping such behaviors should result in fewer overall wires.
- **Sequential execution** is based on the ability to execute behaviors sequentially without loss in performance.
- **Constrained communication** is based on the amount of communication between the sets of behaviors that contributes to each performance constraint. Grouping such behaviors should help ensure that performance constraints are met.
- **Balanced size** is based on the size of the sets of behaviors. Grouping smaller behaviors should eventually lead to groups of balanced size.

Performance

- Behavior's **execution time** is calculated as the sum of the behavior's internal computation time (ict) and communication time (commtime).
  - The *ict* is the execution time on a particular component, assuming all accessed behaviors and variables take zero time.
  - The communication time (*commtime*) includes
    - time to transfer data to/from accessed behaviors and variables, and
    - time for such accessed behaviors to execute (e.g., the time for a called procedure to execute and return).
- This model leads to some inaccuracy, since some computation and communication could occur in parallel, but the model provides reasonable accuracy while enabling rapid estimations.
**Performance Metrics**

**Clock cycle effect on Execution Time** and resources required

\[
b.\text{exec_time} = b.ict \times p + b.\text{comm_time} \\
\]

\[
b.\text{comm_time} = \sum_{c_k \in b.\text{outchannels}} c_k.\text{accfreq} \times (c_k.\text{ttime}_{\text{bus}} + (c_k.\text{dst}).\text{exec_time}) \\
\]

\[
c_k.\text{ttime}_{\text{bus}} = \text{bus.time} \times \left( \frac{c_k.\text{bits}}{\text{bus.width}} \right) \\
\text{bus.time} = \text{bus.timesame if } (c_k.\text{dst}).p = p, \\
= \text{bus.timediff otherwise}. \\
\]

- **Pre-estimation** -- A behavior's \textit{ict} based on profiling = determines the execution count of each basic block (a sequence of statements not containing a branch)
- **Online estimation** -- Given a partition of every functional object to a component, the actual \textit{ict}, bus values, and bus times become known; execution time can be evaluated.

---

**Execution Time**

\[
b.\text{exec_time} = b.ict \times p + b.\text{comm_time} \\
\]

\[
b.\text{comm_time} = \sum_{c_k \in b.\text{outchannels}} c_k.\text{accfreq} \times (c_k.\text{ttime}_{\text{bus}} + (c_k.\text{dst}).\text{exec_time}) \\
\]

\[
c_k.\text{ttime}_{\text{bus}} = \text{bus.time} \times \left( \frac{c_k.\text{bits}}{\text{bus.width}} \right) \\
\text{bus.time} = \text{bus.timesame if } (c_k.\text{dst}).p = p, \\
= \text{bus.timediff otherwise}. \\
\]

- **Pre-estimation** -- A behavior's \textit{ict} based on profiling = determines the execution count of each basic block (a sequence of statements not containing a branch)
- **Online estimation** -- Given a partition of every functional object to a component, the actual \textit{ict}, bus values, and bus times become known; execution time can be evaluated.
HW Estimation Model

\[ \text{clk} > \text{delay}(\text{SR}) + \text{delay}(\text{CL}) + \text{delay}(\text{RF}) + \text{delay}(\text{MUX}) + \text{delay}(\text{FU}) + \text{delay}(\text{NS}) + \text{delay}(\text{SR}) + \text{setup}(\text{SR}) + \sum \text{delay}(n) \]

CLOCK CYCLE ESTIMATION

Maximum-operator-delay method

\[ \text{clk}(\text{MOD}) > \text{Max} \left[ \left( \text{delay}(t) \right) \right] \]

Control Step Estimation

Operator-use Method

Estimate the number of control steps required to execute a behavior, given the resources:

<table>
<thead>
<tr>
<th>t</th>
<th>num(t)</th>
<th>clocks(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mult</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>sub</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- The method partitions all statements into a set of nodes such that all statements in a node could be executed concurrently.

\[ u_1 := u \times dx \]
\[ u_2 := 5 \times w \]
\[ u_3 := 3 \times y \]
\[ y_1 := i \times dx \]
\[ w := w + dx \]
\[ u_4 := u_1 \times u_2 \]
\[ u_5 := dx \times u_3 \]
\[ y := y + y_1 \]
\[ u_6 := u - u_4 \]
\[ u := u_6 - u_5 \]
\[ u_3 := 3 \times y \]
\[ y_1 := i \times dx \]
\[ w := w + dx \]
\[ u_4 := u_1 \times u_2 \]
\[ u_5 := dx \times u_3 \]
\[ y := y + y_1 \]
\[ u_6 := u - u_4 \]
\[ u := u_6 - u_5 \]

\[ \text{maximum (with control loop)} \]
\[ \text{max}(7, j) = 9 \]
\[ \text{max}(5, j) = 4 \]
Clock Cycle Estimation

Clock Slack

- Minimize idle time of functional units
- **Clock Slack** = portion of clock cycle for which the FU is idle

\[
\text{delay}(+) = 49\text{ns} \quad \text{delay}(-) = 56\text{ns} \quad \text{delay}(x) = 163\text{ns}
\]

\[
\text{slack}(\text{clk}, t_i) = \left( \left[ \text{delay}(t_i) + \text{clk} \right] \times \text{clk} \right) - \text{delay}(t_i)
\]

\[
\text{slack}(163,x) = \left( \left[ 163 + 163 \right] \times 163 \right) - 163 = 0\text{ns}
\]

\[
\text{slack}(163,-) = \left( \left[ 56 + 163 \right] \times 163 \right) - 56 = 107\text{ns}
\]

\[
\text{slack}(163,+) = \left( \left[ 49 + 163 \right] \times 163 \right) - 49 = 114\text{ns}
\]

Slack-minimization method

\[
\text{ave}_{-}\text{slack}(\text{clk}) = \frac{\sum_{i=1}^{T} (\text{occur}(t_i) \times \text{slack}(\text{clk}, t_i))}{\sum_{i=1}^{T} \text{occur}(t_i)}
\]

\[
\text{utilization}(\text{clk}) = 1 - \frac{\text{ave}_{-}\text{slack}(\text{clk})}{\text{clk}}
\]

\[
\text{slack}(65,x) = (3 \times 65) - 163 = 32\text{ns}
\]

\[
\text{slack}(65,-) = (1 \times 65) - 56 = 9\text{ns}
\]

\[
\text{slack}(65,+) = (1 \times 65) - 48 = 17\text{ns}
\]

\[
\text{utilization}(65\text{ns}) = 1 - 24.4/65 = 0.62 = 62\%
\]

The clock utilization is repeated for all clock values from 14 ns to 163 and the maximum 92% was achieved at a clock of 56 ns.
Control Steps

- Control Unit sequences operations through a series of control steps
- 1 control step corresponds to a single state
- The number of control steps affects the complexity of the control logic in the implementation

```
behavior B
begin
  A:=A+1
  ...
end B
```

Execution time

Communication
Message generated by 1 behavior (producer) received by other behavior (consumer)
- avgrate(C)
- peakrate(C)

Inter-event timing

Functionality Partitioning

- **Hardware partitioning** techniques aim to partition functionality among hardware modules (ASICs or blocks on an ASIC)
- Most such techniques partition at the granularity of arithmetic operations
- Partitioning functionality among a **hardware/software** architecture at the level of:
  - statement
  - statement sequence
  - subroutine/task levels.
Manual partitioning

- Provide the ability to manually relocate objects,
- Allows user control of the relative weights of various metrics in the cost function
- Automatically provide hints of what changes might yield improvements to the current partition.
- Closeness hints provide a list of object pairs, sorted by the closeness of the objects in each pair. Closeness is based on a weighted function of various closeness metrics.

Hardware/software partitioning

- No need to consider special purpose hardware in the long run?
- Correct for fixed functionality, but wrong in general, since “By the time MPEG-n can be implemented in software, MPEG-n+1 has been invented” [de Man]

Functionality to be implemented in software or in hardware?
General Partitioning Methods

- Exact methods:
  - enumeration
  - Integer Linear Programs (ILP)
- Heuristic methods:
  - constructive methods
    - random mapping
    - hierarchical clustering
  - iterative methods
    - Simulated Annealing
    - Evolutionary Algorithms (EA)

Example of HW/SW partitioning

Inputs
1. Target technology
2. Design constraints
3. Required behavior
HW/SW codesign: approach

Steps of a partitioning algorithm (1)

- Translation of the behavior into an internal graph model
- Translation of the behavior of each node from VHDL into C
- Compilation
  - All C programs compiled for the target processor,
  - Computation of the resulting program size,
  - Estimation of the resulting execution time (simulation input data might be required)
- Synthesis of hardware components:
  \[ \forall \text{ leaf node, application-specific hardware is synthesized.} \]
  High-level synthesis sufficiently fast.
Steps of a partitioning algorithm (2)

- **Flattening of the hierarchy:**
  Granularity used by the designer is maintained. Cost and performance information added to the nodes. Precise information required for partitioning is pre-computed
- **Generating and solving a mathematical model of the optimization problem:**
  Integer programming IP model for optimization. Optimal with respect to the cost function (approximates communication time)

Steps of a partitioning algorithm (3)

- **Iterative improvements:**
  Adjacent nodes mapped to the same hardware component are now merged.
Steps of a partitioning algorithm (4)

Interface synthesis:
After partitioning, the glue logic required for interfacing processors, application-specific hardware and memories is created.

Integer programming models

Ingredients:
- Cost function
- Constraints

Involving linear expressions of integer variables from a set \( X = \{ x_i \} \)

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

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

**Def.**: The problem of minimizing (1) subject to the constraints (2) is called an integer programming (IP) problem.

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.
Example

\[ C = 5x_1 + 6x_2 + 4x_3 \]
\[ x_1 + x_2 + x_3 \geq 2 \]
\[ x_1, x_2, x_3 \in \{0,1\} \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></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

Remarks on integer programming

- Maximizing the cost function can be done by setting \( C' = -C \)
- Integer programming is NP-complete :(
- In practice, running times 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 and structure of the problem.
- IP models can be a good starting point for modelling, even if in the end heuristics have to be used to solve them.
An IP model for HW/SW Partitioning

Notation:

- Index set $I$ denotes task graph nodes.
  - Each $i \in I$ corresponds to a task graph node.
- Index set $L$ denotes task graph node types.
  - Each $\ell \in L$ corresponds to a task graph node type, e.g. square root, DCT (Discrete Cosine Transform) or FFT.
- Index set $KH$ denotes hardware component types.
  - e.g. there is one index value $k_1 \in KH$ for the DCT hardware component type and another one $k_2 \in KH$ for the FFT hardware component type.
  - For each of the hardware component there may be multiple copies or "instances"; each instance is identified by an index $j \in J$.
- Index set $KP$ denotes processors. All processors are assumed to be of the same type.

An IP model for HW/SW Partitioning

$X_{i,k} = 1$ if node $v_i$ is mapped to hardware component type $k \in KH$ and 0 otherwise.

$Y_{i,k} = 1$ if node $v_i$ is mapped to processor $k \in KP$ and 0 otherwise.

$NY_{\ell,k} = 1$ if at least one node of type $\ell$ is mapped to processor $k \in KP$ and 0 otherwise.

$T$ is a mapping from task graph nodes to their types:

$T : I \rightarrow L$

The cost function accumulates the cost of hardware units:

$C = \text{cost(processors)} + \text{cost(memories)} + \text{cost(application specific hardware)}$
Operation assignment constraints (1)

\[ \forall i \in I : \sum_{k \in KH} X_{i,k} + \sum_{k \in KP} Y_{i,k} = 1 \]

All task graph nodes have to be mapped either in software (KP) or in hardware (KH).

All decision variables (\(X_{i,k}\) – for HW and \(Y_{i,k}\) – for SW) are assumed to be positive integers.

Additional constraints to guarantee they are either 0 or 1:

\[ \forall i \in I : \forall k \in KH : X_{i,k} \leq 1 \]
\[ \forall i \in I : \forall k \in KP : Y_{i,k} \leq 1 \]

Operation assignment constraints (2)

\[ \forall \ell \in L, \forall i : T(v_i) = c_\ell, \forall k \in KP : NY_{\ell,k} \geq Y_{i,k} \]

• For all types \(\ell\) of operations and for all nodes \(i\) of this type: if \(i\) is mapped to some processor \(k\) (i.e., \(Y_{i,k} = 1\)), then that processor must implement the functionality of \(\ell\) – i.e., a copy of the SW that implements that functionality must be in the processor’s memory.

• Decision variables must also be 0/1 variables:
\[ \forall \ell \in L, \forall k \in KP : NY_{\ell,k} \leq 1. \]
Resource & design constraints

- $\forall k \in KH$, the cost (area) used for components of that type is calculated as the sum of the costs of the components of that type. This cost should not exceed its maximum.
- $\forall k \in KP$, the cost for associated data storage area should not exceed its maximum.
- $\forall k \in KP$ the cost for storing instructions should not exceed its maximum.
- The total cost ($\Sigma_{k \in KH}$) of HW components should not exceed its maximum
- The total cost of data memories ($\Sigma_{k \in KP}$) should not exceed its maximum
- The total cost instruction memories ($\Sigma_{k \in KP}$) should not exceed its maximum

Scheduling / precedence constraints

- For all nodes $v_{i1}$ and $v_{i2}$ that are potentially mapped to the same processor or hardware component instance, introduce a binary decision variable $b_{i1,i2}$ with $b_{i1,i2}=1$ if $v_{i1}$ is executed before $v_{i2}$ and $= 0$ otherwise.
  Define constraints of the type (end-time of $v_{i1}$) $\leq$ (start time of $v_{i2}$) if $b_{i1,i2}=1$ and (end-time of $v_{i2}$) $\leq$ (start time of $v_{i1}$) if $b_{i1,i2}=0$
- Ensure that the schedule for executing operations is consistent with the precedence constraints in the task graph.
Other constraints

**Timing constraints**
These constraints can be used to guarantee that certain time constraints are met.

• Some less important constraints omitted.

Example

- HW types H1, H2, and H3 with costs of 20, 25, and 30.
- Processors of type P.
- Tasks T1 to T5.
- Execution times:

<table>
<thead>
<tr>
<th>T</th>
<th>H1</th>
<th>H2</th>
<th>H3</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Operation assignment constraints (1)

<table>
<thead>
<tr>
<th>T</th>
<th>H1</th>
<th>H2</th>
<th>H3</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>20</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>20</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>20</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

∀i ∈ I : \(\sum_{k \in KH} X_{i,k} + \sum_{k \in KP} Y_{i,k} = 1\)

A maximum of 1 component (KH or KP) is used to run a task:

- \(X_{1,1} + Y_{1,1} = 1\) (task 1 either mapped to H1 or to P1)
- \(X_{2,1} + Y_{2,1} = 1\) (task 2 either mapped to H2 or to P1)
- \(X_{3,1} + Y_{3,1} = 1\) (task 3 either mapped to H3 or to P1)
- \(X_{4,1} + Y_{4,1} = 1\) (task 4 either mapped to H3 or to P1)
- \(X_{5,1} + Y_{5,1} = 1\) (task 5 either mapped to H1 or to P1)

Task indices: \(I = \{1, 2, 3, 4, 5\}\)

HW components indices: \(KH = \{1, 2, 3\}\)

SW component index: \(KP = \{1\}\)

Operation assignment constraints (2)

Apply slide 36:

• Assume that the types of tasks \(T_1\) to \(T_5\) are \(l = 1, 2, 3, 3,\) and 1, respectively; then:

\[\forall l \in L, \forall i : T(v_i) = c_l, \forall k \in KP : NY_{l,k} \geq Y_{i,k}\]

If node 1 (\(T_1\)) is mapped to the processor \(P_1\), then the function \(l=1\) must be implemented on that processor.

The same function \(l=1\) must also be implemented on that processor if task \(T_5\) is mapped to the processor \(P_1\).
Other equations

- HW types H1, H2 and H3 with costs of 20, 25, and 30.
- Processors of type P.

\[(\ldots)\] represents the number of instances of HW components

<table>
<thead>
<tr>
<th>T</th>
<th>H1</th>
<th>H2</th>
<th>H3</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Apply slide 30:

Cost function:

\[
C = 20 \#(H1) + 25 \#(H2) + 30 \#(H3) + \text{cost}(\text{processor}) + \text{cost}(\text{memory})
\]

Apply slide 39:

- Time constraints leading to: Application specific hardware required for time constraints under 100 time units.

Pareto Points

(60, 45) is a Pareto point as it dominates the other 2 points
Result (by educated guessing)

- For a time constraint of 100 time units and cost(P) < cost(H3):

<table>
<thead>
<tr>
<th>T</th>
<th>H1</th>
<th>H2</th>
<th>H3</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td></td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td></td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>

Solution
- T1 → H1
- T2 → H2
- T3 → P
- T4 → P
- T5 → H1

Separation of scheduling-partitioning

- Combined scheduling/partitioning very complex;
- Heuristic: Compute estimated schedule
- Perform partitioning for estimated schedule
- Perform final scheduling
- If final schedule does not meet time constraint, go to 1 using a reduced overall timing constraint.
Application example

- Audio lab (mixer, fader, echo, equalizer, balance units); slow SPARC processor
- 1µ ASIC library
- Allowable delay of 22.675 µs (~ 44.1 kHz)

SPARC processor
ASIC (Compass, 1 µ)
External memory

Outdated technology; just a proof of concept.

Running time for COOL optimization

Only simple models can be solved optimally.
Deviation from optimal design

Hardly any loss in design quality.

Running time for heuristic
Design space for audio lab

![Graph showing design space for audio lab]

- Everything in software: 72.9 µs, $0 \lambda^2$
- Everything in hardware: 3.06 µs, $457.9 \times 10^6 \lambda^2$
- Lowest cost for given sample rate: 18.6 µs, $78.4 \times 10^6 \lambda^2$

Final remarks

- **COOL approach:**
  - shows that formal model of hardware/SW codesign is beneficial; IP modeling can lead to useful implementation even if optimal result is available only for small designs.
- **Other approaches for HW/SW partitioning:**
  - starting with everything mapped to hardware; gradually moving to software as long as timing constraint is met.
  - starting with everything mapped to software; gradually moving to hardware until timing constraint is met.
  - Binary search.
Design Quality Estimation

Estimation

More accurate estimates require more time!

1. Pre-estimation: Each functional object (behavior, variable and channel) is annotated with information, (the number of bytes for a behavior when compiled to a particular processor, the average frequency of channel access, or the number of channel bits). Pre-estimation occurs only once at the beginning of exploration, is independent of any particular partition and allocation, and may take seconds… minutes…

2. Online-estimation: Pre-estimated annotations are combined in complex expressions to obtain metric values for a particular partition and allocation. Online-estimation occurs hundreds or thousands of times during manual or automated exploration, so it must take ms.
Typical Estimation Models

<table>
<thead>
<tr>
<th>Design model</th>
<th>Additional tasks</th>
<th>Accuracy</th>
<th>Fidelity</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem</td>
<td>Mem allocation</td>
<td>Low</td>
<td>Low</td>
<td>fast</td>
</tr>
<tr>
<td>Mem+FUs</td>
<td>FU allocation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+ FUs +Reg</td>
<td>Lifetime analys.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+ FUs +Reg+Muxes</td>
<td>FU binding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+ FUs +Reg+Muxes+Wiring</td>
<td>Floor planning</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mem = memories  
FUs = functional units  
Reg = registers  
Muxes = multiplexers

Accuracy VS Speed

- Accuracy ($A$) = measure how close the estimate ($E$) is to the actual value ($M$) of the metric measured after design implementation ($D$)

$$ A = 1 - \frac{|E(D) - M(D)|}{M(D)} $$

- Simplified estimation models yield fast estimators but with less accuracy
Fidelity of Estimation

- percentage of correctly predicted comparisons between design implementations

- Let \( D = \{D_1, D_2, ..., D_n\} \) be a set of implementations of a given specification

- Define

\[
\mu_{ij} = \begin{cases} 
1 & \text{if } E(D_i) > E(D_j) \text{ and } M(D_i) > M(D_j) \text{ or } E(D_i) < E(D_j) \text{ and } M(D_i) < M(D_j) \\
0 & \text{otherwise} 
\end{cases}
\]

- The fidelity \( F \) of an estimation can be defined as a percentage of correct predictions:

\[
F = 100 \times \frac{2}{n(n-1)} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mu_{ij}
\]

Estimation Fidelity

- 100%
- 33%