Allocation, Assignment and Scheduling

Kris Kuchcinski
Krzysztof.Kuchcinski@cs.lth.se

Allocation of System Components

- Defines an architecture by selecting hardware resources which are necessary to implement a given system.
- The components can be, for example, microprocessors, micro-controllers, DSP's, ASIP's, ASIC's, FPGA's, memories, buses or point-to-point links.
- Usually made manually with a support of estimation tools.
- In simple cases can be performed automatically using optimization strategy.

Assignment of System Components

- After allocation the partitioning of system functionality to selected components can be done.
- The partitioning defines the assignment of tasks to particular components.
- If there is number of tasks assigned to the same component, which does not support parallel execution, the execution order need to be decided — task scheduling.

Scheduling

- Depending on the computation model scheduling can be done off-line or during run-time.
- Static vs. dynamic scheduling.
- RTOS support for dynamic scheduling.
- Scheduling can address advanced execution techniques, such as software pipelining.
- Can be applied to tasks allocated to hardware, software as well as hardware operations and software instructions.

Scheduling Approaches

- Static scheduling
  - static cycling scheduling
- Dynamic scheduling
  - fixed priorities — e.g., rate monotonic
  - dynamic priorities — e.g., earliest deadline first
Synthesis of the following code (inner loop of differential equation integrator)

while c do
  begin
    $x_1 := x + dx$;
    $u_1 := u - (3 * x * u * dx) - (3 * y * dx)$;
    $y_1 := y + (u * dx)$;
    $c = x < a$;
  x := x1; u := u1; y := y1;
end.

High-Level Synthesis Scheduling

Behavioral specification
  Language selection
  Parallelism
  Synchronization

procedure example;
  var a, b, c, d, e, f, g : integer;
  begin
    read(a, b, c, d, e);
    $f := e * (a + b)$;
    $g := (a + b) * (c + d)$;
  end;

High-Level Synthesis Scheduling (cont’d)

Design Representation
  parsing techniques
  data-flow analysis
  parallelism extraction
  program transformations
    elimination of high-level constructs
    loop unrolling
    subexpression detection

HLS typical process

Operation Scheduling
  Parallelism/cost trade-off
  Performance measure
  Clocking strategy

Data Path Allocation
  Operation selection
  Register/Memory allocation
  Interconnection Generation
  Hardware Minimization
Data Path Allocation

HLS typical process (cont’d)

Control Allocation
- Selection of control style (PLA, random logic, etc.)
- Clock implementation
- Signal/condition design

Control Allocation

Module Binding

Control description (FSM)

Control ROM
0000: 11000000 0001
0001: 00100000 0010
0010: 00011000 0011
0011: 01000000 0100
...

ASAP Scheduling

ALAP Scheduling
List Scheduling

- Constructive scheduling algorithm which selects operation to be assigned to control steps based on a priority function.
- Priority function can be different in different versions of list scheduling algorithms:
  - higher priority to operations with low mobility, or
  - higher priority to operations with more immediate successors,
  - length of the path from the operation to the end of the block,
  - ...

Force-Directed Scheduling

- The basic strategy is to place similar operations in different control steps to balance the concurrency of the operations without increasing the total execution time.
- By balancing the concurrency of operations, it is ensured that each functional unit has a high utilization and therefore the total number of units required is decreased.
- Three main steps:
  - determine the time frame of each operation,
  - create a distribution graph, and
  - calculate the force associated with each assignment.

Distribution Graph

<table>
<thead>
<tr>
<th>Control step</th>
<th>Addition/Subtraction DG</th>
<th>Multiplication DG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1/3</td>
<td>1/3</td>
</tr>
<tr>
<td>2</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>3</td>
<td>1/3</td>
<td>1/3</td>
</tr>
<tr>
<td>4</td>
<td>1/2</td>
<td>1/2</td>
</tr>
</tbody>
</table>

DG_{mult}(1) = 1/2 + 1/3 = 0.833
DG_{mult}(2) = 1/2 + 1/3 = 0.833
DG_{mult}(3) = 1/2 + 1/2 + 1/3 = 1.333
DG_{mult}(4) = 1/2 + 1/2 = 1

Distribution Graph (cont'd)
**Force Calculation**

- The force associated with the tentative assignment of an operation to c-step $j$ is equal to the difference between the distribution value in that c-step and the average of the distribution values for the c-steps bounded by the operation’s time frame.

$$\text{Force}(j) = \text{DG}(j) - \sum_{i=\text{start of operation}}^{\text{end of operation}} \frac{\text{DG}(i)}{\text{duration}}$$

- Assignment of operation 10 to control step 3

<table>
<thead>
<tr>
<th>Force $\text{Force}(3) = \text{DG}<em>{\text{mult}}(3) - \text{average DG}</em>{\text{mult}}$ value over time frame of operation 10 $= 1.333 - (1.333 + 1)/2 = 0.167$, Force$(4) = -0.167$</th>
</tr>
</thead>
</table>

--

**Forced Directed Scheduling Algorithm**

- Once all the forces are calculated, the operation-control step pair with the lowest force is scheduled.
- The distribution graphs and forces are then updated and the above process is repeated until all operations are scheduled.

--

**Advanced Scheduling Topics**

- Control constructs
  - Conditional sharing — sharing of resources for mutually exclusive operations,
  - Speculative execution.
- Chaining and multi-cycling.
- Pipelined components.
- Pipelining.
- Registers and memory allocation.
- Low-power issues.

--

**Conditional Dependency Graph**

--

**Conditional Resource Sharing**

- Adder
- Multiplier

--
**Speculative Execution**

Speculatively executed operation

**Chaining and multicycling**

Chaining

Multicycle

**Pipelined Components**

Multiplier 1

**Fifth Order Elliptic Wave Filter DFG**

**Pipelining — Example**

**Scheduling of H.261 Video Coding Algorithm**

$T_{exec} = 2963, T_{total} = 29630$
Scheduling of H.261 Video Coding Algorithm

- $T_{\text{basic}} = 3373, T_{\text{pipe init}} = 1154, T_{\text{exec}} = 13759$

Real-Time Scheduling

- System is a set of tasks
  - tasks have known execution times (usually WCET)
- Tasks are enabled by repeating events in the environment
  - some known timing constraints
- Task executions must satisfy timing requirements
- Single processor (hard)
  - multiprocessors — much harder, mostly negative results (e.g. NP-hardness)

Scheduling Policies

- **Static** (pre-run-time, off-line)
  - round-robin,
  - static cyclic.
- **Dynamic** (run-time, on-line)
  - static priority,
  - dynamic priority,
  - preemptive or not.

Assumptions and Definitions

- Fixed number of periodic and independent tasks.
- Parameters:
  - execution period — $T$
  - worst-case execution time — $C$
  - execution deadline — $D$
- Usually assumed that $T = D$

Static Scheduling

- **Round-robin**:
  - pick an order of tasks, e.g. A B C D,
  - execute them forever in that order.
- **Static cyclic**:
  - pick a sequences of tasks, e.g. A B C B D,
  - execute that sequence forever.
- Much like static data-flow.
- If there are arbitrary task periods the schedule duration is equal least common multiplier (LCM) of task periods (MPEG 44.1 kHz * 30 Hz requires 132 300 repetitions).
- Problem with sporadic tasks.

Dynamic Scheduling — Static Priorities

- Priorities are assigned to tasks off-line.
- A task with the highest priority is always executed among enabled tasks.
- Single processor execution.
- Preemptive schedule.
- Rate Monotonic Scheduling (RMS) — a task with a shorter period is assigned a higher priority.
- RMS is optimal — no fixed priority scheme does better.
Rate Monotonic Scheduling - An Example

Task 1: 20ms
Task 2: 40ms

Priority(Task 1) < Priority(Task 2)

Rate Monotonic Scheduling

Utilization

\[ Utilization = \sum_{i} \frac{C_i}{T_i} \]

All deadlines met when \( Utilization \leq n(2^{\frac{1}{n}} - 1) \)

For \( n=2 \), \( 2^{\frac{1}{2}} - 1 = 0.83 \),

For \( n \to \infty \), \( n(2^{\frac{1}{n}} - 1) \to \ln(2) = 0.69 \)

An example

<table>
<thead>
<tr>
<th>Task</th>
<th>Period (T)</th>
<th>Rate (1/T)</th>
<th>Exec Time (C)</th>
<th>Utilization (U)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100 ms</td>
<td>10 Hz</td>
<td>20 ms</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>150 ms</td>
<td>6.67 Hz</td>
<td>40 ms</td>
<td>0.267</td>
</tr>
<tr>
<td>3</td>
<td>350 ms</td>
<td>2.86 Hz</td>
<td>100 ms</td>
<td>0.286</td>
</tr>
</tbody>
</table>

For \( n=2 \), \( 2^{\frac{1}{2}} - 1 = 0.779 \), \( \Rightarrow \) tasks are schedulable

Rate Monotonic Scheduling Implementation

- Table of tasks with a task priority and its state (enables, etc.).
- At context switch select the task with the highest priority.
- Linear complexity, \( O(n) \) where \( n \) = number of tasks.

Critical sections

- Access to a shared resource should be mutually exclusive to access a resource
  - lock the resource → critical section starts
    - may fail and block the task
  - process the resource
  - unlock the resource critical section ends

Rate Monotonic Scheduling — Problems

- Static cyclic scheduling is better if possible.
- Critical sections in tasks and communication create problems
  - Deadlock
  - Priority inversion
- Methods for solving priority inversion problem
  - priority inheritance
  - priority ceiling protocol.

Deadlock
Priority Inversion

Priority inheritance

When a job $J_i$ tries to enter a critical section and it is already locked by a lower priority task $J_k$, then $J_i$ waits and $J_k$ inherits the priority of $J_i$.

The queue of jobs waiting for a resource is ordered by decreasing priority.

Priority inheritance is transitive.

At any time, the priority at which a critical section is executed is always equal to the highest priority of the jobs that are currently blocked on it.

When a job exits critical section it usually resumes the priority it had when it entered critical section.

When released, a resource is granted to the highest priority job if any waiting for it.

Priority Inheritance Protocol

Priority ceiling protocol

Assumptions

- A task cannot voluntarily suspend itself.
- Semaphores cannot be held between invocations.
- Semaphores must be locked in a nested manner.

Protocol

- Every CS has a ceiling: priority of a highest task that may enter it.
- A task is allowed into a CS only if its priority is higher than ceilings of all active CS’s.
- If task A is blocking some higher priority task B, then A gets the priority of B while in CS.

Dynamic Scheduling — Dynamic Priorities

Dynamic priorities needed since sometime static priorities might not meet deadlines.

An Example

$C_1 = 2, T_1 = 5$
$C_2 = 4, T_2 = 7$

Task 1
Task 2
Earliest Deadline First

- Minimizes number of missed deadlines.
- Tasks with earliest deadline has priority.

An Example

\[
\begin{align*}
&\text{Task 1:} & \text{Task 2:} \\
&2 & 5 & 7 & 10 & 14 \\
\end{align*}
\]

\[
C_1 = 2, \ T_1 = 5 \\
C_2 = 4, \ T_2 = 7
\]

- Any sequence is optimal that puts the jobs in order of non-decreasing deadlines

Earliest Deadline First

- EDF can achieve 100% utilization until overload occurs.
- Cannot guarantee meeting deadlines for arbitrary data arrival times.

An example

<table>
<thead>
<tr>
<th>Task</th>
<th>Period (T)</th>
<th>Rate (1/T)</th>
<th>Exec Time (C)</th>
<th>Utilization (U)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>20 ms</td>
<td>50 Hz</td>
<td>10 ms</td>
<td>0.5</td>
</tr>
<tr>
<td>B</td>
<td>50 ms</td>
<td>20 Hz</td>
<td>25 ms</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Earliest Deadline First

- Additional assumptions
  - arbitrary release times and deadlines, and
  - arbitrary and unknown (to the scheduler) execution times.
- The EDF algorithm is optimal in that if there exist any algorithm that can build a valid (feasible) schedule on a single processor, then the EDF algorithm also builds a valid (feasible) schedule.

Earliest Deadline First Implementation

- At each preemption, sort tasks by time-to-deadline.
- Need for efficient sorting algorithm — \( O(n \log n) \)
- Choose ready task closest to the deadline.

What Is Missing

- Data dependencies between tasks
- Context switching time (jitter)
- Multiprocessor scheduling
- Memory considerations

Data dependencies

- Data dependencies allow us to improve utilization.
  - Restrict combination of processes that can run simultaneously.
  - P1 and P2 can’t run simultaneously.
Context-switching time

- Non-zero context switch time can push limits of a tight schedule.
- Hard to calculate effects -- depends on order of context switches.
- In practice, OS context switch overhead is small.

Literature