Scheduling

Felice Balarin
CBL

What's the problem?

- Have some work to do  
  - know subtasks
- Have limited resources
- Have some constraints to meet
- Want to optimize performance

Solution space

- roadmap
- priorities
- divide time
- simulation
- static analysis
- SW synthesis
- RTOS synthesis
- RTOS configuration

Design: policy selection

- Roadmap  
  have a plan in advance  
  - list of tasks
- Prioritize  
  always execute highest priority task  
  - static or dynamic priorities
- Divide time  
  give a fixed time slice to each task

Complexity  
Task inter-dependence  
Task knowledge
Implementation

- **SW synthesis**
  - generate code that combines functionality, communication, coordination
- **RTOS synthesis**
  - import functionality code
  - generate communication and coordination code
- **RTOS configuration**
  - generate configuration files for an existing RTOS

Design-Implementation space

Our work

Outline

- **Overview**
  - shop scheduling
  - data-flow scheduling
  - real-time scheduling
  - OS scheduling
- **Data-flow scheduling**
  - pure
  - Petri nets
- **Real-time scheduling**
- **RTOS generation**
Shop scheduling

Single job, one time
- finite and known amount of work
- multiple resources of different kind
- often minimize lateness
  - could add release, precedence, deadlines, ...

SOLUTION: compute the schedule
APPLICATION: manufacturing

Data-flow scheduling

Single-job, repeatedly
- known amount of work
  - simple subtasks
- multi-processor
- max. throughput, min. latency

SOLUTION: code generation
APPLICATION: signal processing

Data-flow scheduling variants

- Work
  - data dependent (BDF, FCPN)
- Resources
  - many different execution units (HLS)
- Goal
  - min. code, min. buffers, min. resources

Real-time scheduling

Fixed number of repeating jobs
- each job has fixed work
  - job is a sub-task
- processor(s)
- meet individual deadlines

SOLUTION: choose policy, let RTOS implement it
APPLICATION: real-time control
RT scheduling variants

- Work
  - sporadic or event-driven tasks,
  - variable (data dependent) work
  - coordination between tasks:
    - mutual exclusion, precedence, ...
- Goal
  - event loss, input or output correlation, freshness, soft deadlines, ...

OS scheduling

Variable number of random tasks
- know nothing about sub-tasks
- processor + other computer resources
- progress of all tasks, average service time

SOLUTION: OS implements time-slicing

APPLICATION: computer systems

Outline

- Overview
  - shop scheduling
  - data-flow scheduling
  - real-time scheduling
  - OS scheduling
- Data-flow scheduling
  - pure
  - Petri nets
- Real-time scheduling
- RTOS generation
  - scheduling
  - communication

Data-flow scheduling

- Functionality usually represented with a data-flow graph
- Kahn’s conditions allow scheduling freedom
  - if a computation is specified with actors (operators) and data dependency, and
  - every actor waits for data on all inputs before firing, and
  - no data is lost
  - then the firing order doesn’t matter
Data-flow graphs

- Schedule: a firing order that respects data-flow constraints and returns the graph to initial state

A schedule:

Data-flow graphs

- Schedule: a firing order that respects data-flow constraints and returns the graph to initial state

A schedule:

Data-flow graphs

- Schedule: a firing order that respects data-flow constraints and returns the graph to initial state

A schedule:

Data-flow graphs

- Schedule: a firing order that respects data-flow constraints and returns the graph to initial state

A schedule:
Data-flow graphs

A schedule: A

Schedule: a firing order that respects data-flow constraints and returns the graph to initial state

Data-flow graphs

A schedule: A D

Schedule: a firing order that respects data-flow constraints and returns the graph to initial state

Data-flow graphs

A schedule: A D

Schedule: a firing order that respects data-flow constraints and returns the graph to initial state

Data-flow graphs

A schedule: A D B

Schedule: a firing order that respects data-flow constraints and returns the graph to initial state
Data-flow graphs

• Schedule: a firing order that respects data-flow constraints and returns the graph to initial state

Schedule implementation

• Static scheduling (cyclic executive, round robin)
  • A, B, C, D are processes
  • RTOS schedules them repeatedly in order A D B C

Data-flow graphs

• Schedule: a firing order that respects data-flow constraints and returns the graph to initial state

Schedule implementation

• Code synthesis (OS generation)
  • A, B, C, D are subroutines
  • generate: `forever{ call A; call D; call B; call C; }`
  • less robust, better overhead
Schedule implementation

In-lined code synthesis
- A, B, C, D are code fragments
- generate: \texttt{for}\{A; D; B; C; \}
- even less robust, even better overhead

Data-flow scheduling

Resources
- fixed or arbitrary number of processors

Goal:
- max. throughput given a fixed number of processors
- min. processors to achieve required throughput

Data-flow scheduling goals

Max. throughput given a fixed number of processors
- it is NP-hard to determine max. achievable throughput

Min. processors to achieve required throughput
- if there are loops than there is a fundamental upper bound
- easy to compute

Throughput bound

\[ \frac{1}{\text{max}_{\text{loops}}(\text{Time/Delay})} \]

N+2'nd output of A can be computed at least 7 time units after the Nth
Scheduling heuristics

Non-overlapped scheduling
- Look at one iteration
- Use list scheduling algorithm (developed for shop scheduling)

Overlapped scheduling
- less developed

Inter-iteration constraints

- Remove delayed edges
- List scheduling:
  - maintain list of tasks that could be scheduled
  - schedule one with longest path

List scheduling

- Assume two processors
List scheduling

![Diagram](image)

List scheduling

![Diagram](image)

Inter-iteration constraints

- Unfold k iteration (e.g. k=2)
- Do list scheduling

![Diagram](image)

List scheduling

- Rate optimal (not true in general)

![Diagram](image)
Static data flow

- Loop scheduling
- Code size
- Buffer size

Loop scheduling

ABCBCCC
A (2 B (2 C))
A (2 B) (4 C)
A (2 B C) (2 C)

Loop scheduling and code size

A (2 B (2 C))
A;
for i = 1 ... 2 {
    B;
    for i = 1 ... 2 {
        C;
    }
}

- single appearance schedules minimize in-lined code size

Buffer size

ABCBCCC
A (2 B (2 C)) 20 20
A (2 B) (4 C) 20 40
A (2 B C) (2 C) 20 30
Data-flow scheduling

- Perfect design-time information
  - Fixed amount of repeating work
    - data-independent
  - Input streams from the environment always available
  - Simple global constraints

  data dependency => Petri nets
  timing constraints => real-time scheduling

Outline

- Overview: four classes of scheduling
- Data-flow scheduling
  - pure
  - Petri nets
- Real-time scheduling
- RTOS generation
  - scheduling
  - communication

Real-time scheduling

- System is a set of tasks
  - tasks have known execution times
- 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)

Real-time scheduling problems

- Analysis:
  - Are timing requirements met for a given scheduling policy?

- Synthesis:
  - Find a scheduling policy that meets timing requirements.
Scheduling policies

● off-line (pre-run-time, static)
  – round-robin
  – static cyclic

● on-line (run-time, dynamic)
  – static priority
  – dynamic priority
  – preemptive or not

Off-line scheduling

Round-robin:
● pick an order of tasks, e.g. A B C D
● execute them forever in that order

```c
forever {
    if ( enabled(A ) ) execute A ;
    if ( enabled(B ) ) execute B ;
    if ( enabled(C ) ) execute C ;
    if ( enabled(D ) ) execute D ;
}
```
● much like basic data-flow

Off-line scheduling

Static cyclic:
● pick a sequences of tasks, e.g. A B C B D
● execute that sequence forever

```c
forever {
    if ( enabled(A ) ) execute A ;
    if ( enabled(B ) ) execute B ;
    if ( enabled(C ) ) execute C ;
    if ( enabled(B ) ) execute B ;
    if ( enabled(D ) ) execute D ;
}
```
● much like static data-flow

On-line scheduling

Priority based:
● if several tasks are enabled, execute one with the highest priority

Static priority
  – priorities are assigned off-line
Dynamic priority
  – priorities may change at run-time
On-line scheduling

Preemptive
- at any time, execute the highest-priority enabled task (even if it means suspending active task)

Non-preemptive
- once a task is chosen to be executed, it is run to completion even if some higher priority task becomes enabled in the meantime

Off-line vs. on-line scheduling

- Plus side:
  - simple to implement
  - low overhead (no preemption or priority calculation)
  - easy to analyze and synthesize
- Minus side
  - bad service to urgent tasks
  - independent of actual requests
  - lots of wasted checks

Agenda

On-line scheduling
- static priority
  - Liu-Layland model
  - reactive model
- dynamic priority
  - earliest-deadline first
  - priority inversion
    - priority ceiling protocol

Preemptive static priority scheduling

- Liu-Layland [73] consider systems consisting of tasks:
  - enabled periodically
  - with fixed run time
  - that should be executed before enabled again
  - scheduled preemptively with statically assigned priorities
Liu-Layland results

Critical instant occurs when a task is enabled at the same time as all higher priority tasks.

Proof. Let task m with period occur at $t_1$. Let some higher priority task i occur at $t_2+k\times T_i$, $k=0,1,2$

Either

```
   t1  t2  t2+Tm
   i done   m done
```

or

```
   t1  t2  t2+Tm
   i done   m done
```

In either case, interference of i to m can only increase if we slide $t_1$ to $t_2$.

Rate-monotonic scheduling

Assigning higher priority to tasks with shorter period is optimal

Proof: Assume system is feasible. Consider the initial busy period (i.e. the worst case), and two tasks violating the RMS rule. Exchanging their priorities maintains feasibility.

Utilization bounds

utilization = $\sum_{tasks} (\text{execution time}) / (\text{period})$

Any set of $n$ tasks with utilization of less than $n(2^{1/n} - 1)$ is schedulable

- for $n=2,3,...$, $n(2^{1/n} - 1) = 0.83, 0.78, ... \ln(2)=0.69$
- many sets of tasks reach higher utilization

Audsley’s Algorithm

bound $p$-busy interval $(B_p)$ where only tasks with priority $p$ or higher execute

- maximum $B_p$ is the worst case response time
- if $B_p < T$, then task $i$ never misses its event
  1. Need only to know for which tasks $P_k > P_i$
  2. Lowering $P_i$ cannot help

\[ E_i \quad T_i \quad B_{pi} \quad T_i \quad \text{time} \]
Static Priority Schedule Validation

Audsley [91]:
- for a task in Liu-Layland’s model find its worst case response time

Audsley’s algorithm
- let $E_i$’s be run times $T_i$’s periods
- how much can $i$ be delayed by a higher priority task $k$:
  - each execution delays it by $E_k$
  - while $i$ is executing $k$ will be executed $\lceil B_i / T_k \rceil$
- $B_i = E_i + \sum_{k>i} \lceil B_i / T_k \rceil * E_k$

Solving implicit equation
- iteration
  - $B_{i,0} = E_i$
  - $B_{i,n+1} = E_i + \sum_{k>i} \lceil B_{i,n} / T_k \rceil * E_k$
  - will converge if processor utilization if less than 1

Static Priority Scheduling

ANALYSIS: Schedule validation
- Are timing constraints satisfied for a given priority assignment?

SYNTHESIS: Priority assignment
- Find a priority assignment which meets all timing constraints.
**Audsley’s Modification**

- \( T_1, O_1 \rightarrow P_1, E_1 \)
- \( T_2, O_2 \rightarrow P_2, E_2 \)
- \( T_n, O_n \rightarrow P_n, E_n \)

\( T_i \) - period or minimum time between occurrences (MTBO)
\( P_i \) - priority
\( E_i \) - execution time
\( O_i \) - offset

- independent tasks enabled periodically or sporadically starting at \( O_i \)
- correct if there are no missed events

**SYNTHESIS for Audsley’s Modification**

- Rate-monotonic is not optimal
- Assume there exists a test such that:
  - to check whether a task satisfies its timing constraints need to know only who has higher priority (don’t need complete assignment)
  - if a task fails a test with priority \( k \), then it also fails with any priority \( p < k \)
- Busy period analysis is such a test

```plaintext
let no tasks have priority assignment
for each priority \( k \) from lowest to the highest
if \( B_k < T_i \) for some task \( i \) without priority assignment
    assign priority \( k \) to task \( i \)
else
    quit
end if
end for
```

- It is optimal, if the test is exact

**What’s wrong with LL model?**

- Liu-Layland model yields strong results but does not model reactivity well
- Our model:
  - models reactivity directly
  - abstracts functionality
  - allows efficient conservative schedule validation
Reactive Model

- tasks are enabled by internal and external events
- external events occur sporadically
- internal events are task executions
- correct if there are no missed events

ANALYSIS of Reactive Model

- Internal events:
  - not missed if priority assignment meets some constraints
- External events
  - checked by busy period analysis

Assignment Constraints

- Strictly increasing:
  \[ \delta(x,0) = E_a + E_c + E_d + E_e + E_b + E_d \]
  \[ \delta(x,P_1) = E_b + E_c + E_d \]
  \[ \delta(b,P_a) = E_c + E_d \]
- How about strictly decreasing?
- Strictly decreasing:
  - minimizes processor load
  - optimal if there is one external event

Decreasing Assignment

- Decreasing assignments will miss events if an external event occurs while some shared task is enabled

in the sequence:
\[ x a y a ... \]
event (a,b) is missed
**Modified Decreasing Assignment**

- If \( i \) enables \( k \), then \( P_i > P_k \) unless \( i \) is *merge point* (where threads from two external events meet)
- Merge points must have lower priority than any direct or indirect successors

\[
\delta(y,1) = E_4 + E_2 + \delta(2,1) + E_1 + \delta(1,1)
\]

\[
\delta(2,1) = E_4 + E_3
\]

\[
\delta(1,1) = E_4 + E_1
\]

**Private Events**

- Some private events are more critical:
  - \((d,b)\) is not missed if \( B_{P_b} < T_y \)
  - if \((d,b)\) is not missed neither is \((y,d)\)
  - after \( b \) is assigned a priority, we can assign any remaining (higher) priority to \( d \)
- Task is *safe* unless it is:
  - merge point
  - private and has no successors

\[
\delta(x, i) = \sum_k E_k + \sum_j \delta(j, i)
\]

where
- \( k,j \): merge-free successors of \( a \) with priority of at least \( i \)
- \( j \): merge point
- Need only to know for which tasks \( P_k > P_i \).
Priority Assignment Algorithm

let no tasks have priority assignment
let partial order G represent modified decreasing assignment
for each priority k from to lowest to the highest
  if there is some safe minimal task i in G
    assign priority k to task i and remove i from G
  else if B_k < T_x for all external fan-ins x of some minimal task i in G
    assign priority k to task i and remove i from G
  else
    quit
end if
end for

● If any modified decreasing assignment can be validated by busy analysis, the algorithm will find it

Priority Assignment Algorithm: Example

STARS

A methodology for worst-case analysis of discrete systems that can be used to find a conservative bound on response time.

● and power
● and bus utilization

Example: VM pager

CONTROL:
if (present (message)) {
  frames = message; last = size_of(message);
}
if ((present (play)) || (present (request)) && last > 0) {
  emit frame (frames[last--]);
}

BUFFER:
if (present (frame)) {
  samples = frame; last = 50;
}
if (present (tick) && last > 0) {
  emit sample (samples[last--]);
  if (last == 20) emit request();
}
Signatures

- Abstractions of signals
- Event counts: a good choice
- Must satisfy two conditions:
  - signatures can be compared
  - want to know which one is worse

VM Pager Signatures

- A vector with components:
  - ms # of message events
  - pl # of play events
  - tk # of request events
  - fr # of frame events
  - rq # of tick events
  - sm # of sample events
  - ...

Signature abstractions

- Abstract behaviors
- Maps times and signatures to signatures
- Must be conservative:

VM Pager Signature Abstractions

- # of message events $F_{ms}(s, T) = T/625$
- # of play events $F_{pl}(s, T) = T/10,000$
- # of request events $F_{rk}(s, T)$
- # of frame events $F_{fr}(s, T)$
- # of tick events $F_{rk}(s, T)$
- # of sample events $F_{sm}(s, T)$
- ...
Example: VM pager

BUFFER:
if (present (frame) {
samples = frame; last = 50;}
if (present (tick) && last > 0) {
emit sample (samples[last --]);
if (last == 20) emit request();}

F_{rq}(s, T) = \min(f, tk/30)
F_{sm}(s, T) = \min(tk, 50*f_r)

Workload Function: VM pager

BUFFER:
if (present (frame) {
samples = frame; last = 50;}
if (present (tick) && last > 0) {
emit sample (samples[last --]);
if (last == 20) emit request();}

W(s) = 20*(fr+tk) + 20*fr + 20*sm + 10*rq + (... control part...)

STARS
1 Pick a signature
2 Choose a signature abstraction F and workload function W and verify they are monotone and conservative
3 Solve
   s = F(s,T)
   T = W(s)
4 T is a bound on response time
   the processor cannot be continuously busy for more than T time units

Automatic signature abstraction

If
- signatures are linear constraints over event counts
- components have Boolean transition function
then
- can build the best possible signature abstraction automatically
Automatic abstraction example

```c
CONTROL:
    if (present (message) { frames = message; last = 3
    if ((present (play) || present (request)) && last > 0) { emit frame (frames[last --]); }

- model last with inputs l0 - l3 and outputs n0-n3
- build BDD
  - 28 nodes
- OMEGA input file (145 lines, ~70 variables)
- OMEGA prints out:
  - frames <= 3 message AND frames <= play + request
```

Agenda

On-line scheduling
- static priority
- dynamic priority
  - critical sections
    - priority ceiling protocol
  - earliest-deadline first

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

```plaintext
task: C1 C2
```

Deadlocks

- Two tasks, fixed priority

```
Task H: C2 C1
Task L: C1 C2
H
L
deadlock
```
Priority inversion

- Three tasks, fixed priority

  Task H
  Task M
  Task L

  Priority inversion

Priority ceiling 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

Deadlocks

- Two tasks, fixed priority

  Task H
  Task L

  ceiling(C1) = ceiling(C2) = H

Priority inversion

- Three tasks, fixed priority

  Task H
  Task M
  Task L

  ceiling(C) = H
Priority ceiling protocol

- No deadlocks
- Priority inversion limited to one CS

Agenda

On-line scheduling
- static priority
- dynamic priority
  - critical sections
  - earliest-deadline first

Liu-Layland model

- system consists of tasks:
  - enabled periodically
  - with fixed run time
  - that should be executed before enabled again

Why dynamic priority?

- Static priority may not work
- $E_1=2 \quad T_1=5$
- $E_2=4 \quad T_2=7$
Earliest deadline first

- Give highest priority to tasks with closest deadline
  
  \[ E_1 = 2 \quad T_1 = 5 \]
  
  \[ E_2 = 4 \quad T_2 = 7 \]

EDF can schedule any set of tasks with utilization < 1

Outline

- Overview
- Data-flow scheduling
- Real-time scheduling
- RTOS generation

RTOS functions

- Enable communication between software tasks, hardware and other system resources
- Coordinate software tasks
  - keep track which tasks are ready to execute
  - schedule them

System: Network of CFSMs
Implementations

- CFSMs can be implemented:
  - in hardware: HW-CFSMs
  - in software: SW-CFSMs
  - by built-in peripherals (e.g. timer): MP-CFSMs

Events: SW to SW

- for every event, RTOS maintains
  - global values
  - local flags

Events: atomicity problems

- TASK 1 detects $y$ AND NOT $x$, which is never true
- to avoid, need atomic detects

Events: SW to SW

- for atomicity:
  - always read from frozen
  - others always write to live
  - at the beginning of execution, switch
Events: HW to SW

- event can be **polled** or driving an **interrupt**
- for polled events:
  - allocate I/O port bits for value, occurrence and acknowledge flags
  - generate the polling task that acknowledges and emits all polled events that have occurred

Events: HW to SW

- for events driving an interrupt:
  - allocate I/O port bits for value,
  - allocate an interrupt vector,
  - create an interrupt service routine that emits an event

Events: interrupts

- interrupt service routine:
  
  ```
  { emit x
  }
  ```

- optional interrupt service routine:
  
  ```
  { emit x
  execute SW-CFSM
  }
  ```

Events: SW to HW

- allocate I/O port bits for value and occurrence flag
- use existing ports or memory-mapped ports
- write value to I/O port
- create a pulse on occurrence flag
Events: SW to/from MP

- every peripheral must have a library with
  - `init` function (to be called at initialization time)
  - deliver function for each input (to be called by `emit`)
  - `detect` function for each output (to be called by `poll-taker`)
  - interrupt service routine (containing `emit`)

Coordination

- consider SW-CFSM ready to run whenever it has some not consumed input events
- generate code for
  - round robin
  - non preemptive static priority
  - preemptive static priority

Round robin

- for a given order (e.g. A B C D) generate

```c
void round_robin(void)
{
    if (enabled(A)) execute A;
    if (enabled(B)) execute B;
    if (enabled(C)) execute C;
    if (enabled(D)) execute D;
}
```

Non-preemptive static priority

- if order of priorities is A B C D, generate:

```c
void non_preemptive(void)
{
    if (enabled(A)) execute A;
    if (enabled(B)) execute B;
    if (enabled(C)) execute C;
    if (enabled(D)) execute D;
}
```
Preemptive static priority

- at the end of every emit add

```java
while (!A:enabled() && priority(A) > priority(current)) {
    execute A;
}
```