Real Time Programming Languages

1. General

1.1. Real Time System
System or device, that has to satisfy certain timing constraints while functioning.
Hard timing constraints for safety critical systems.

1.2. Programming Models

1.2.1. Synchronous Programming Model
- model based on programs reacting to events in zero time
- output based on input is computed immediately
- implementation: program approximates synchrony by computing a reactor to an event before the next event occurs (ensured by the compiler)
- no scheduler is necessary

1.2.2. Scheduled Programming Model
- program consits of processes, threads, taks
- scheduler determines which part runs at which time
- compiler only implements functionality
- runtime system (scheduler) implements the scheduling

1.2.3. Time Triggered Model
- computation takes a fixed non zero amount of time
- event safety $\Rightarrow$ time safety, but the other way does not hold

1.3. Layers of reactive program
- interface: tasks care of input reception and output production
- reactive kernel: implements logic ot the system
- data handling layer: perform computaion requested by kernel

2. Esterel

Synchronous Language

reacting to events in zero time („instantaneously“), i.e, based on the **synchrony hypothesis:**
- reaction takes no time w.r.t. environment
- subprocesses take no time w.r.t. other subprocesses
- interprocess communication takes no time
- statements take time if they say so: await. . ., every. . .

2.1. Esterel in General

Advantage	Disadvantage
model of time	limited flexibility (no dynamic memory)
deterministic	limited tool support
easy to analyze	semantically not easy for programmer
translateable to HW description	implementation is not efficient
much easier to verify formally	paradigm doesn't fit for all problems

2.1.1. Signals
Signal coherence rules:
- by default: absent
- if emitted: present
- can be both: no
- multiple values per cycle: no
- propagation of changes: instantaneous

Internal Signals:
- have a limited scope which lies within a module, i.e., is not externally visible
- can be seen as „local“ inputs and outputs
- same semantics as interface signals
- useful for letting parallel threads interact („||“)

2.1.2. Loops
loops are always bounded, means the body of a loop must take time. otherwise: instantaneous loop $\Rightarrow$ invalid

2.1.3. ?-operator
- applies to valued signals and sensors
- their value of the current tick can be obtained with the ?-operator, e.g.: `O(?I)` assigns the value of `I` to `O`

2.1.4. pre-operator
- „timeshift“ in Esterel
- `pre()` returns the value of valued signals from when they were previously present (i.e., not from the previous tick!), e.g.: `O(pre(?I))` assings the previous value of `I` to `O`

2.1.5. Variables
- the only thing that do not fall under the signal coherence rules (i.e., they can take multiple values per tick and the order of statements does matter)
- cannot be used as signal expressions (i.a.: `abort, present, ...`), and cannot be shared between concurrent threads
- scope is defined explicitely with the keywords `in` (opens scope) and `end` (closes scope)

2.1.6. Modules/Renaming
comparable to functions in C, but modules have no parameters, instead use renaming $\Rightarrow$ `run` can replace signals, variable or assign constants

2.1.7. Important Shorthands

Derived Statement	Kernel Statements
`halt`	`loop pause end`
`sustain s`	`loop emit s; pause end`
`present s then p`	`present s then p else nothing end`
`await s`	`trap T in loop pause; present s then exit T end end loop end`
`await immediate s`	`trap T in loop present s then exit T end; pause end loop end`
`suspend p when immediate s`	`suspend present s then pause end; p when s`
`abort p when (immediate) s`	`trap T in suspend p when (immediate) s; exit T || await (immediate) s; exit T; end`
`weak abort p when (immediate) s`	`trap T in p; exit T || await (immediate) s; exit T; end`
`loop p each s`	`loop abort p; halt when s end loop`
`every (immediate) s do p end`	`abort (immediate) s; loop p each s`

2.2. Logical Correctness
A signal is logically correct, if there exists exactly one status (present or absent) for it that respects the signal coherence law.
There are two properties which a signal can exhibit:
1. **deterministic**: it has ≤ 1 coherent state
2. **reactive**: it has ≥ 1 coherent state

Consequently, a logically correct signal is both determinstic and reactive.

Logical Correctness Check
1. Assume Siganl s is present and check hypothesis
2. Assume Siganl s is absent and check hypothesis
3. Repeat step 1 and 2 for all signals (input and output)
4. Logical Correct if number of correct hypothesis $= 1$

A program, that is logical correct has exactly one globally coherent state and is much simpler to debug and analyze.

```
1 -- not reactive
2 module M:
3     output O;
4     present O then
5         nothing;
6     else
7         emit O;
8     end present;
9 end module;
```

```
1 -- not deterministic
2 module M:
3     output O;
4     present O then
5         emit O;
6     else
7         nothing;
8     end present;
9 end module;
```

2.3. Model Checking/Functional Verification
Formal proof of functional correctness of a program.
- Two main properties to check:
  - **safety:** „something bad never happens“
  - **liveness:** „something good eventually happens“
- Specifying Properties:
  - *state formula*: property is element of AP (propositional logic), e.g., Airbus must never go to parking mode unless weight on wheels (WoW)
  - *path formula:* defines a sequence of states, e.g., when the pilot hits brakes, it has to start braking within n cycles
- Checking the Specification

Module Checking in Esterel
1. write an observer module in Esterel, that runs in parallel to your program under analysis (PUA)
2. compile Esterel to hardware description and run xeve
3. xeve answers „possibly emitted“ or „possibly not emitted“ for every checked output from the observer

3. Processor Architecture

3.1. Single/Multi-Cycle Processors

3.1.1. Single-Cycle
- clock cyle is determined by the longest path in the machine
- Advantage: design is easier

3.1.2. Multi-Cycle
- implementation: break up instructions into steps, each such step executes in 1 clock cycle
- different instructions require different number of clock cycles
- more efficient and less hardware

3.2. Pipelines
- relevant for WCET, because hazards will cause timing anomalies.
- Used in all modern processors, to keep every portion of the processor busy with some instruction.
- hazards and resulting stalls will not let us reach ideal speedup

3.3. Hazards

3.3.1. Data Hazards
- Read-after-Write (RaW)
- Write-after-Write (WaW)
- Write-after-Read (WaR)
- Read-after-Read (RaR)

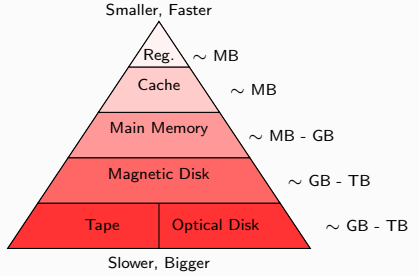
3.3.2. Structural Hazard
More than one instruction needs the same resource at the same time.

3.3.3. Control Hazard
A conditional jump (e.g., `if`) took place, which was not expected. Now we have the wrong instruction in the pipeline.

3.3.4. Avoiding Hazards
- **Data Hazards:** The compiler
  - must take care, that there is „enough time“ between two instructions
  - could also re-arrange the instructions, such that an instruction is computed much earlier than needed.
- **Structural Hazards:** depend on the ISA and processor architecture. MIPS does not have them.
- **Control hazards:** can be reduced, but not avoided. The processor tries to predict what branches („if“s) are taken

3.3.5. Living with the remaining hazards
- **stalling:** For data and structural hazards. Entire pipeline is suspended, until the critical instruction is completed
- **flushing:** For control hazards. Entire pipeline is cleared. It is loaded thereafter with the correct instructions.
- **bubbling:** For data and structural hazards. Processor inserts bubbles (`nop` , „“no operation““) into the pipeline. Bubbles do nothing but delay the upcoming instructions.

3.4. Memory Hierachy
memory closer to CPU is faster but smaller



3.5. Caches
- Relevant for WCET, because hits and misses have different timing.
- Every level of the memory hierachy can be regarded as a „cache“.
  - **cache hit:** You find the data that you are looking for being in the cache
  - **cache miss:** Data you are looking for is not in this cache, you have to get it from the next lower level. This takes a lot of time!

3.5.1. Direct-mapped caches
- if you know the address of your data (which you always do), then you know exactly in which place of the cache it could be
- this means, you do not have to search the cache for your data (it's fast!)
- but it wastes the precious cache space (think of modulo. . .)

3.5.2. Associative Cache
- if you know the address of your data, then still there is a (limited) number of places in the cache, where it could be
- you have to look in all those places (it's slower!)
- but you use your cache much better

3.5.3. Locality
if an item is referenced:
- temporal locality: it will tend to be referenced soon
- spacial locality: nearby will tend to be referenced soon

3.5.4. Cache Calculations

Blocksize	BS
address space	AS
way set-associative cache	w
Number of blocks	$NB = size/BS$
number of sets	$NS = NB/w$
number of bits for offset field	$WO = \mathrm{ld}(BS)$
number of bits for index field	$WI = \mathrm{ld}(NS)$
number of bits for tag field	$WT = AS - WO - WI$

$AS - WI - WO$	WI	WO
TAG field	INDEX field	OFFSET field

4. WCET

4.1. WCET Analysis

Determine how long your code takes in the worst case to complete a certain procedure, to verify the synchrony hypothesis and to perform a schedulability analysis (when multiple programs run in parallel: does each of them finish in time?)

WCET Analysis using IPET/ILP

1. **cross-compile:** for a MIPS-based-processor
2. **disassemble:** get the assembler code for analysis
3. **find basic blocks (BB):** group instructions that do not branch/jump (straight line of code with only one enterence and exit point)
4. **generate control flow graph (CFG):** assembly from 2. turned into a directed graph
5. **derive structural constraints:** read-off from CFG → „such-that"-equations for ILP
6. **add user constraints:** e.g., logical constraints. „This loop will not run more than x times."
7. **Feed ILP (Integer Linear Programming) solver:** objective fcn + constraints go in, the block counts come out (x_i), together with the obj. fcn. value, which is WCET

4.1.1. Definition WCET

The WCET is the maximum possible execution time a program or procedure can need for completion, considering all possible inputs, all possible internal states/execution paths and assuming uninterrupted processing time. It depends on the specific program and the specific processor it is running on.

4.2. Compiling Esterel

Esterel can be compiled to

- **finite state machine:** signal not in states; very fast, but long code
- **netlist-based compilation (default):** boolean logic circuits; scales well, but very inefficient
- **control-flow:** scales well, fast, but bad causality, rarely used

4.3. I(ine)-Blocks

- I-block: maximum sequence of code within a basic block such that when the first instruction of the I-block is accessed, either the whole I-block is in the cache, or none of its contents is in the cache
- A basic block B_i is partitioned into I-blocks $B_{i,1}, B_{i,2}, \ldots, B_{i,n}$
- The execution times of an I-block $B_{i,j}$ are given by $c_{i,j}^{\text{hit}}$ and $c_{i,j}^{\text{miss}}$

$$\text{WCET} = \sum_{i=1}^{N} \sum_{j=1}^{n_i} c_{i,j}^{\text{hit}} x_{i,j}^{\text{hit}} + c_{i,j}^{\text{miss}} x_{i,j}^{\text{miss}}$$

4.4. WCET analysis with caches

$$\text{WCET} = \max_{x^{\text{hit}}, x^{\text{miss}}} \sum_{i=1}^{N} c_i^{\text{hit}} x_i^{\text{hit}} + c_i^{\text{miss}} x_i^{\text{miss}}$$

4.5. Safety

- **event-safety:** in each state of a trace at most a single process action is enabled
- **time-safety:** all tasks complete before their output is read
- **space-safety:** scheduler doesn't choose unfinished part to execute, when it's shared variable are not yet written

5. Ada

Scheduled Model Language

Real time is the physical time as observed in the external environment.

5.1. Ada in Gerneral

Ada	C++
`if then ; else ;`	`if ; else ;`
`case when (when others)`	`switch case (default)`
`loop`	`for`
`while loop`	`while`
`Records`	`struct`
`Access`	`Pointer`
`Generics`	`Templates`

5.1.1. Difference to C

- harder to compile, but less bugs after
- once it compiles, it works
- harder to mix data types
- programmer has to think harder → better code

5.2. Tasking

A task runs concurrently to the rest of the Ada program (=thread) (main program is also a task)

5.3. Ravenscar Profile

- Full Determinism, easier schedulability analysis, memory boundness
- task are:
  - *time-triggered* (periodic) (released by a single `delay until`)
  - *event-triggered* (sporadic) (released by a single protected entry e.g., to realize interrupts)
- no task hierachies, no rendvous, static task set
- all entries have capacity of one task

5.3.1. Schedulability Analysis

process to show, whether all tasks can keep their deadlines.
task := (P, e, d) (with P = Period, e = WCET, d = Deadline) schedulable if:

$$u = \sum_{i=1}^{n} \frac{e_i}{P_i} \leq n \left(\sqrt[n]{2} - 1 \right)$$

5.3.2. Scheduling Policy

- Fixed priority preemption scheduling with immediate priority inheritance and static CPU assingment
- tasks with highest priority always runs
- protected objects get ruling priority, which is higher than all other prioritys
- call inherits priority of protected objects