Embedded Software: Modeling, Synthesis and Validation

Franco Fummi

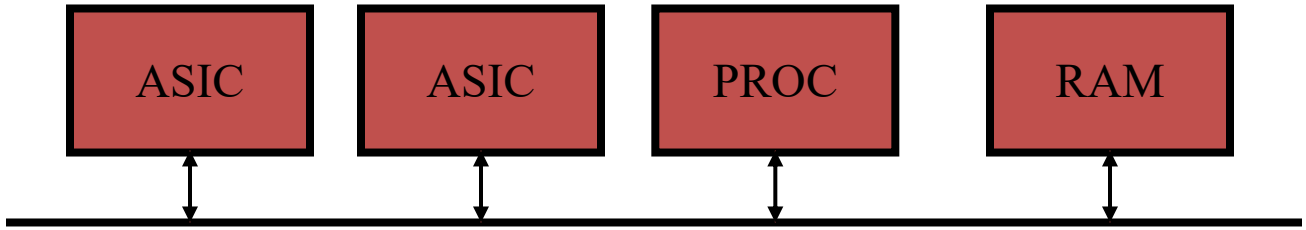
UNIVERSITÀ
di VERONA
Dipartimento
di INFORMATICA

Version 1.1

Contents

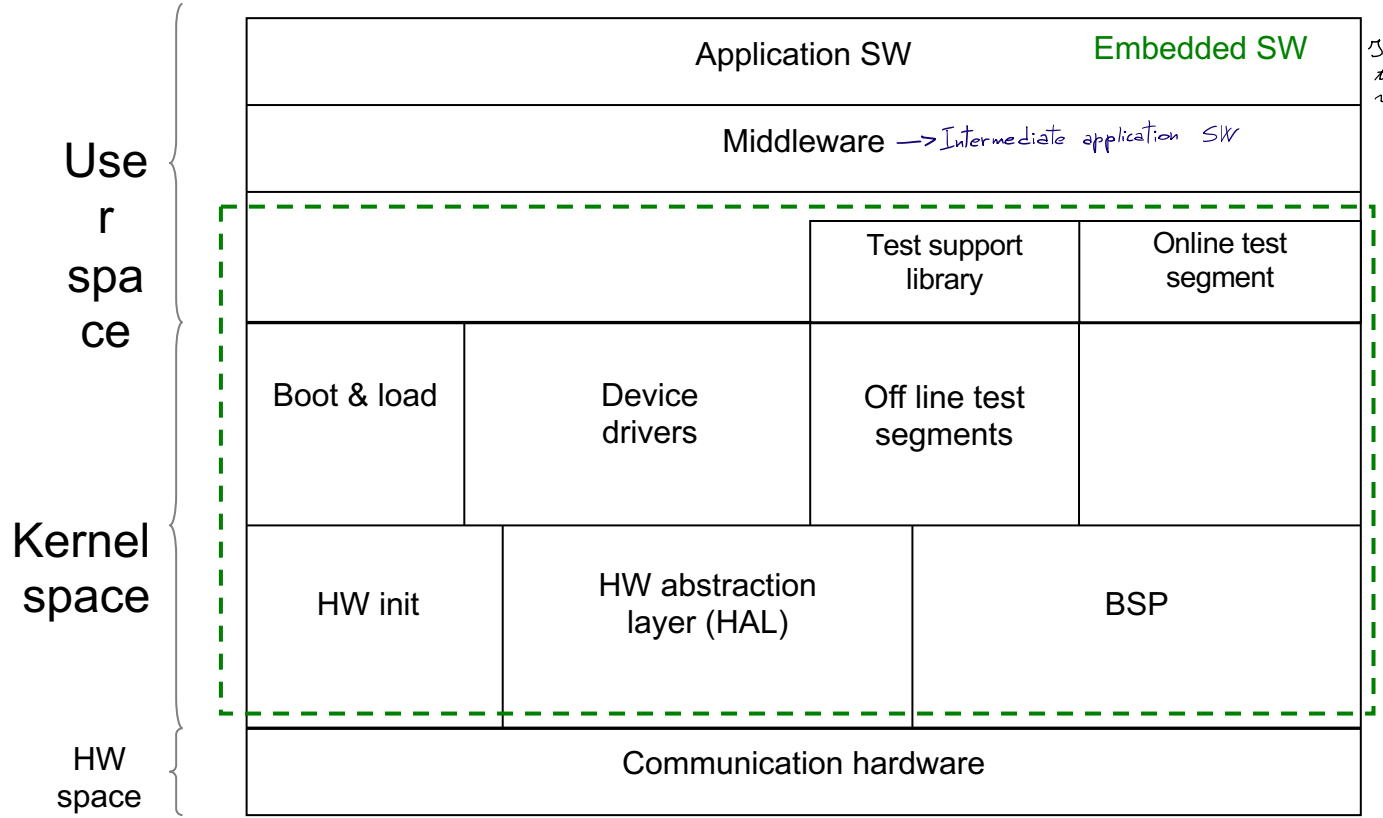
- Embedded Software Synthesis
  - hardware-dependent software
  - software synthesis
- Embedded Software performance evaluation
  - evaluation strategy
  - evaluation correctness
- Embedded Software testing
  - which problem is this?
  - possible solutions

Simple Target Platform



- A set of dedicated HW units
- A programmable core or co-processor
- Memory
  - Including SW program storage
- Interfaces and interconnections

Hardware-dependent Software (HdS)



If you are selling an embedded o.s., I have to trust you if you don't give me an open-source version (note: not FREE! just open-source)

Middleware → Intermediate application SW

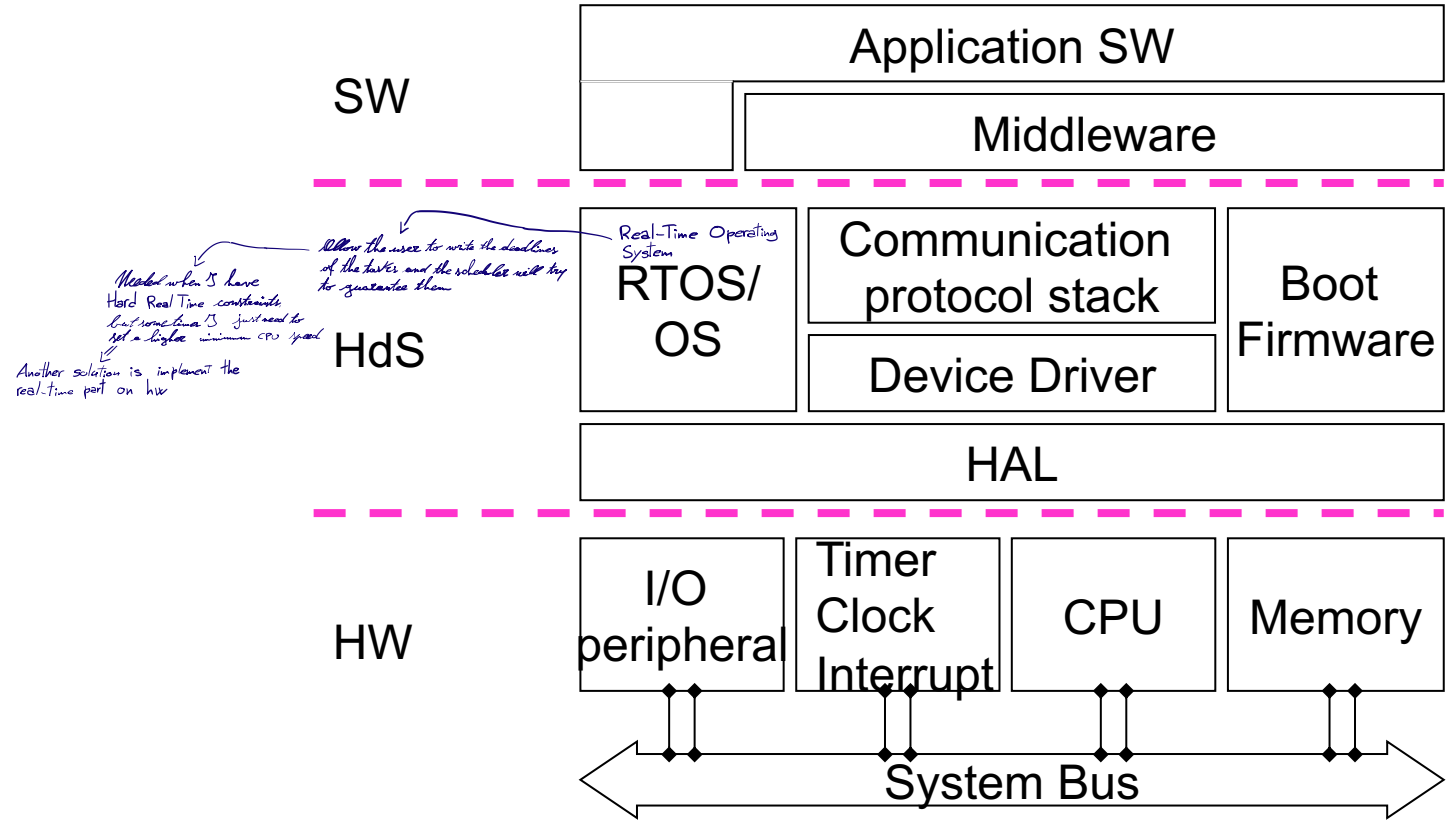
Minimum set of components of an o.s. that we need

HdS = Hardware-dependent Software

Minimum set of sw that allows us to develop sw on a board/platform

Now it's most likely to find a Linux kernel (not the standard one for PCs)

HdS v.s. Software



Hardware and software synthesis

- Implementation of hardware and software components after partitioning
- Constraints and optimization criteria similar to those for partitioning
- Area and code size traded-off against performance (dominant for real-time systems)
- Cost considerations ⇨ off-the-shelf processor
  - separation of software and hardware synthesis, relying on a pre-designed customizable interface
- Exception: synthesis of ASIP and microcode

System-level co-synthesis flow

- Consistent system-level modeling
- Partitioning into dedicated and programmable units
- HW synthesis of dedicated units
  - Based on research or commercial standard synthesis tools
- SW synthesis for programmable units (processors)
  - Based on specialized compiling techniques
- Interface synthesis
  - Definition of HW/SW interface and synchronization
  - Drivers of peripheral devices

Sw synthesis problems

- Target architecture is an ASIP
  - Develop a specific compiler
- Non-executable system-level specification of computer-aided partitioning
  - Synthesize high-level or assembly code
- Interface to HW with given protocol
  - Synthesize interfacing routines

Re-targetable compilers

- Compiler technology suitable for different architectural back-ends
  - ASIPs have specific instruction sets, memory and interconnection resources
- Code quality (i.e. execution speed) is important whereas compilation time is less critical
- Assembly code programming is still common practice

Re-targetable compilers

- Portable compilers
  - Compiler needs significant re-write for porting
- Compiler compilers
  - Generates compiler from architectural templates
- Machine-independent compilers
  - Applicable to different architectures

Re-targetable compilers

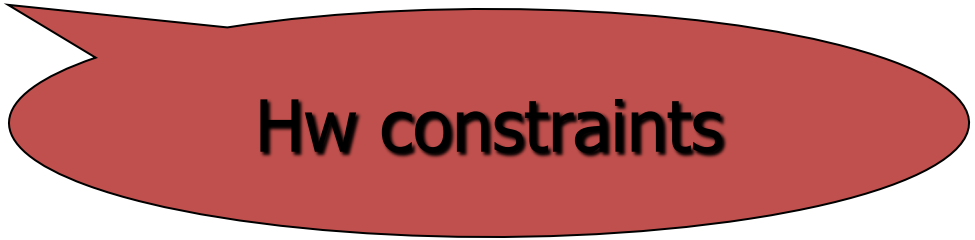
- Compile code into intermediate form and optimize
  - Standard optimizing compiler algorithms
- Instruction selection
  - Pattern matching techniques
- Instruction scheduling
  - Satisfaction of real-time constraints
- Register allocation
- Micro-code compaction

Software synthesis

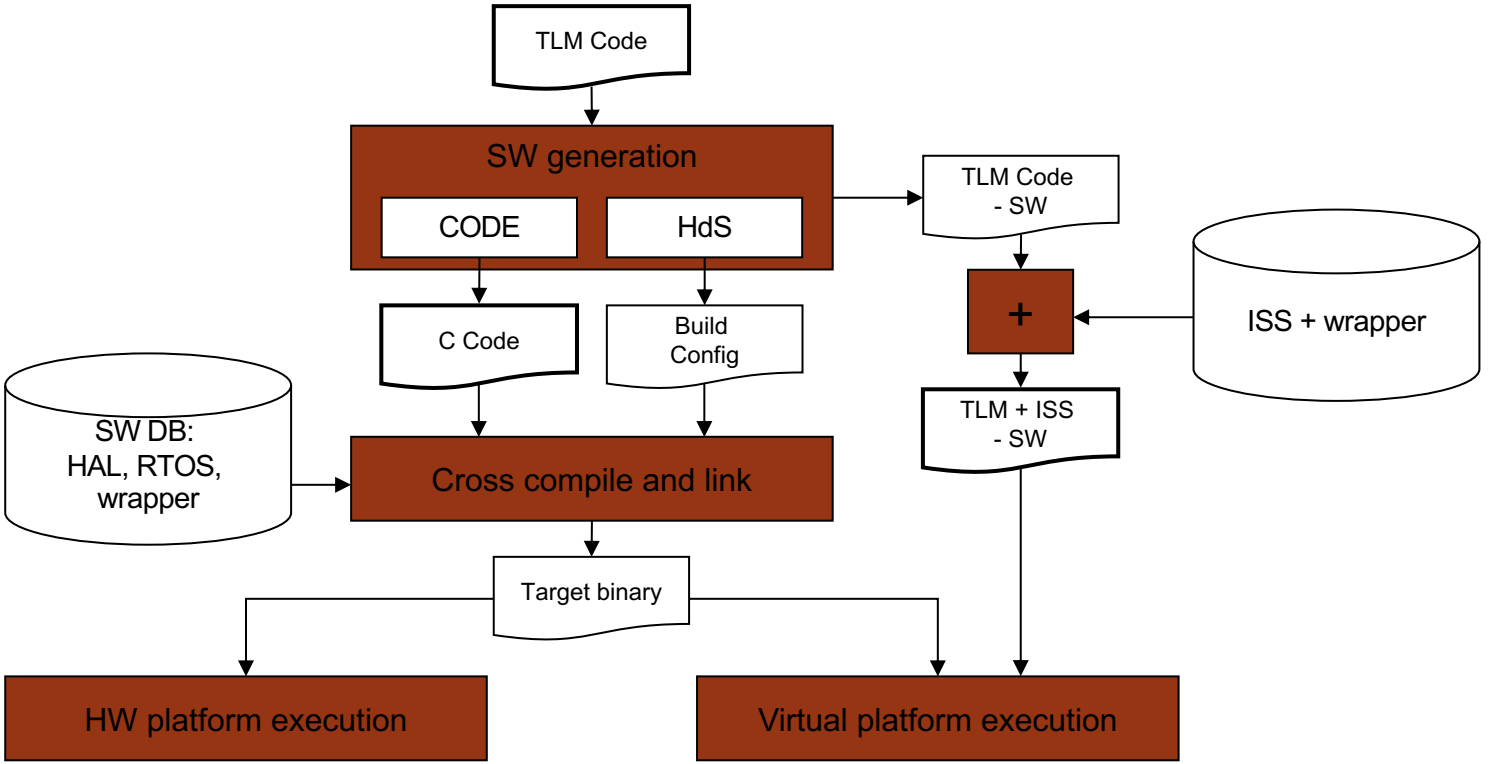
- Constrained problem for embedded software:
  - no virtual memory, reduced dynamic memory allocation and dynamic task creation
- SW functions identified by program threads
- A single processor requires thread serialization or interleaving of originally concurrent tasks
- Scheduling of threads and instructions
  - Satisfying performance constraints
- System-level run-time scheduler to synchronize SW and HW functions

Sw synthesis from SystemC (I)

- Sw processes are isolated from the global system description
- Hw/Sw interfaces are generated
- SystemC keywords remapped on C++
- Software performance measurement

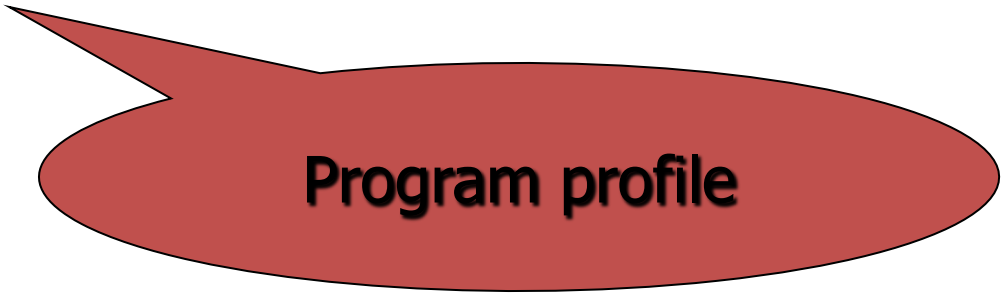


Sw synthesis from SystemC (II)



Sw performance measurement

- Use of profiling tools
- C library linked to executable code (e.g. GNU gprof) → evaluate the critical parts of global performance of a SW
- Program execution (problems?)



Measurement example

% Time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.34	0.02	0.02	7208	0.00	0.00	open
16.67	0.03	0.01	244	0.04	0.12	offtime
16.67	0.04	0.01	8	1.25	1.25	memccpy
16.67	0.05	0.01	7	1.43	1.43	write
16.67	0.06	0.01				*mcount*
0.00	0.06	0.00	236	0.00	0.00	tzset
0.00	0.06	0.00	192	0.00	0.00	tolower
0.00	0.06	0.00	47	0.00	0.00	strlen
0.00	0.06	0.00	45	0.00	0.00	strchr
0.00	0.06	0.00	1	0.00	50.00	main
0.00	0.06	0.00	1	0.00	0.00	memcpy
0.00	0.06	0.00	1	0.00	10.11	print
0.00	0.06	0.00	1	0.00	0.00	*profil*
0.00	0.06	0.00	1	0.00	50.00	report

Measurement legenda (1)

- % time
  - This is the percentage of the total execution time your program spent in this function. These should all add up to 100%.
- cumulative seconds
  - This is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.
- self seconds
  - This is the number of seconds accounted for by this function alone.

Measurement legenda (2)

- calls
  - This is the total number of times the function was called.
- self ms/call
  - This represents the average number of milliseconds spent in this function per call, if this function is profiled.
- total ms/call
  - This represents the average number of milliseconds spent in this function and its descendants per call.

Measurement correctness

Some sw blocks may not be called in the test! An IF-ELSE could be only half- executed if the test isn't well structured

- Sampling period: =>Execute the test many times and changing frequency
  - time interval between two measurements (e.g., 0.01 sec. $\Rightarrow$ 100hz)
- Total execution time:
  - time window used to make the measurement (e.g., 0.06 sec.)
    - number of samples
    - sampling error

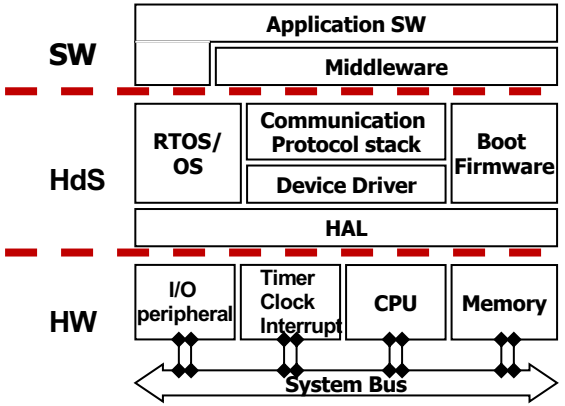
E.g. *Write* function

Definition of testbench

- Does testbench affect the measurement?
  - Yes, definitely!
- How to select testbenches:
  - algorithm information
  - coverage metrics
    - branch coverage
    - statement coverage
    - condition coverage
    - path coverage ...
  - statistical analysis

Embedded Software Testing Difficulties

- Execution platform (hw) is often designed in parallel to embedded sw:
  - a model of the hw is required
  - no easy sw execution
    - no easy testing
- Embedded sw is often hw dependent sw (HdS):
  - correctness is timing dependent
  - sw behavior depends on asynchronous hw events
  - an actual platform or a complex hw model is required
    - no easy testing



A Feasible Approximated Solution

- Functional test pattern based testing
  - it is common use
  - it is not exhaustive
    - some bugs can remain undetected
- To improve efficiency:
  - functional ATPG
    - an hw/sw fault model is necessary
  - functional test qualification
    - an hw/sw fault model is necessary
  - mutation coverage
    - could be the common hw/sw fault model

A Possible Exact Solution

- Property based testing
  - it is of difficult use
  - it is exhaustive if the set of properties is complete
    - definition of a set of good properties
- To improve efficiency:
  - automatic properties generation
    - an hw/sw specification model is required
  - model checking
    - hw/sw model checking is still a research area
  - properties qualification
    - mutation coverage could be useful

The Dreaming Solution

- Correct-by-construction embedded software generation
  - sw is refined from a hw/sw model
    - an hw/sw specification model is required
  - hw dependent sw is automatically generated
    - the hw model drives sw generation
  - hw/sw specification model primitives are automatically mapped on embedded operating system primitives
    - abstract models of operating systems
  - NO embedded sw testing WOULD BE required
    - tools are not on the market