Introduction to Code-Level Timing Analysis

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

`bjorn.lisper@mdh.se`

2020-11-18

Mälardalen University (MDH)

- Regional university, 100 km west of Stockholm
- Twin campus Västerås/Eskilstuna
- 16000 students, 900 employees, 70+ professors

Uppsala
Norrtälje
Västerås
Enköping
Arlanda
Mälaren
Eskilstuna
Strängnäs
Stockholm
NYKVARN
Södertälje
Flen
Gnesta
Katrineholm
Trosa
Nynäshamn
Nyköping
Oxelösund
Norrköping

Embedded Systems @ MDH

- The most prominent research environment at MDH
- Research adressing different aspects of embedded systems software and real-time
- 19 research groups
- $>$ 20 full professors, $>$ 50 senior researchers, $\sim$ 100 PhD students
- More info at `www.es.mdh.se`

The Programming Languages Group

A research group within Embedded Systems, headed by me

Research mainly in **static program analysis** for **embedded software**

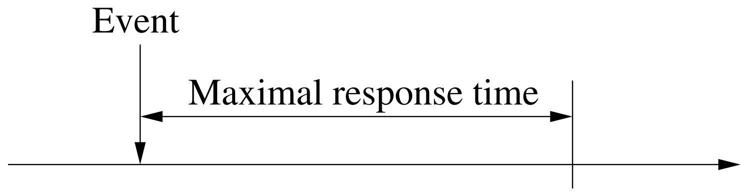
Especially for real-time properties: *Worst Case Execution Time (WCET) Analysis*, but also for other purposes

See

`www.es.mdh.se/research-groups/30-`

Real-Time Systems

Many embedded systems are *real-time systems*. These have *timing constraints*:



They appear in many domains, like for instance:

- Vehicular (automotive, avionics, trains, . . .)
- Telecom, home entertainment
- Medical devices

Interesting to verify that the systems really meet their timing requirements

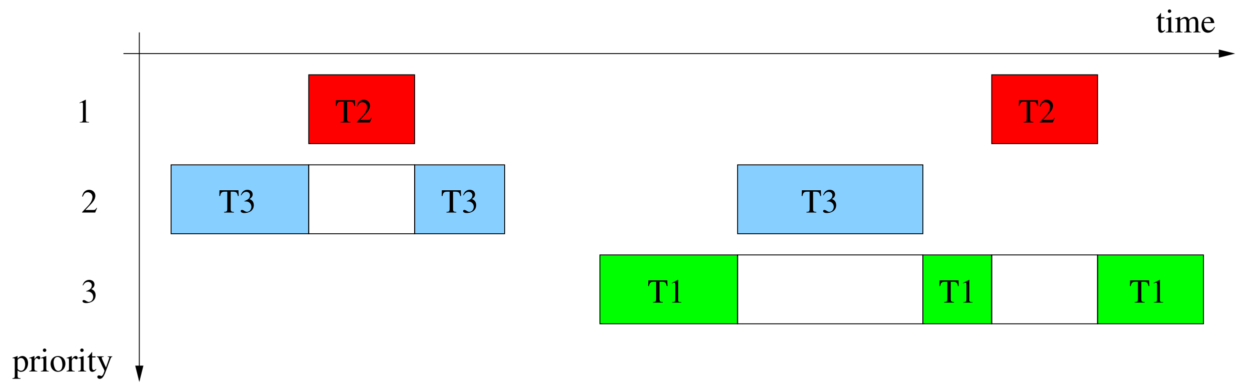
Hard and Soft Real-Time Systems

Two classes:

- **Hard** real-time systems: timing constraints *must* be met (typically safety-critical systems)
  - (vehicles, life-supporting equipment, . . .)
- **Soft** real-time systems: *desirable* that timing constraints are met (but not absolutely necessary)
  - (telecom, home entertainment, . . .)

Verifying Timing Constraints

Real-time systems are often structured into *tasks*, which are run according to some *scheduling policy*. E.g., *fixed-priority*:



Timing verification divided into two levels:

- *System-level*, assuming knowledge of execution times of tasks
- *Code-level*, finding out about execution times of tasks

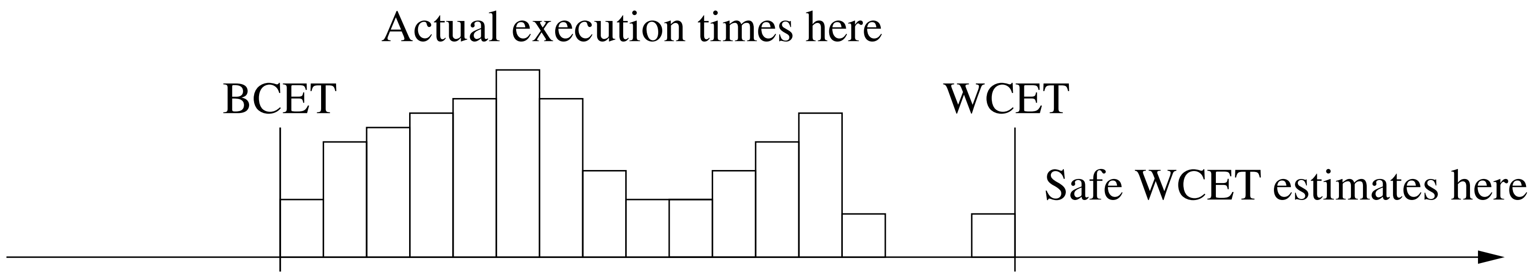
Example: verify that, for sure, a deadline always is met (system level)

Code-Level Timing Analysis

Find out the timing properties of a piece of code (task) running on some given hardware

System-level analysis uses results from code-level analysis

In particular: estimates of the *Worst-Case Execution Time* (WCET):



WCET = longest running time of a sequential program running uninterrupted on a certain hardware

Safe WCET Estimates

WCET estimates that do not underestimate the real WCET are *safe*

If a system-level analysis using safe WCET estimates says that a deadline surely is met, then this is *surely true*

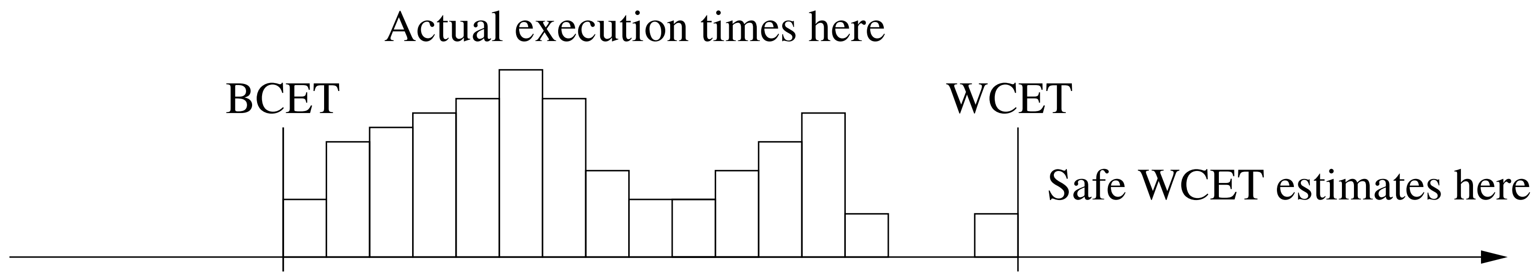
Very important for hard real-time systems – provides a *mathematical proof* that the timing requirements are met

Much higher confidence than for testing only

WCET Analysis

WCET Analysis = estimating the WCET

A **safe** WCET analysis will never underestimate the real WCET



Different Approaches

Based on testing/measurements:

- Pros: easy to automate and apply
- Cons: safety not guaranteed (may underestimate WCET)
- Cons: requires a running system

Based on formal models (static program analysis):

- Pros: can give provably safe WCET estimates
- Pros: code does not have to run, sufficient to have formal models
- Cons: hard to automate completely (Turing's Halting Problem), can be hard to get tight WCET estimates, may be laborious to apply

Static WCET Analysis: The Standard Setup

Must in principle explore *all paths* in the program

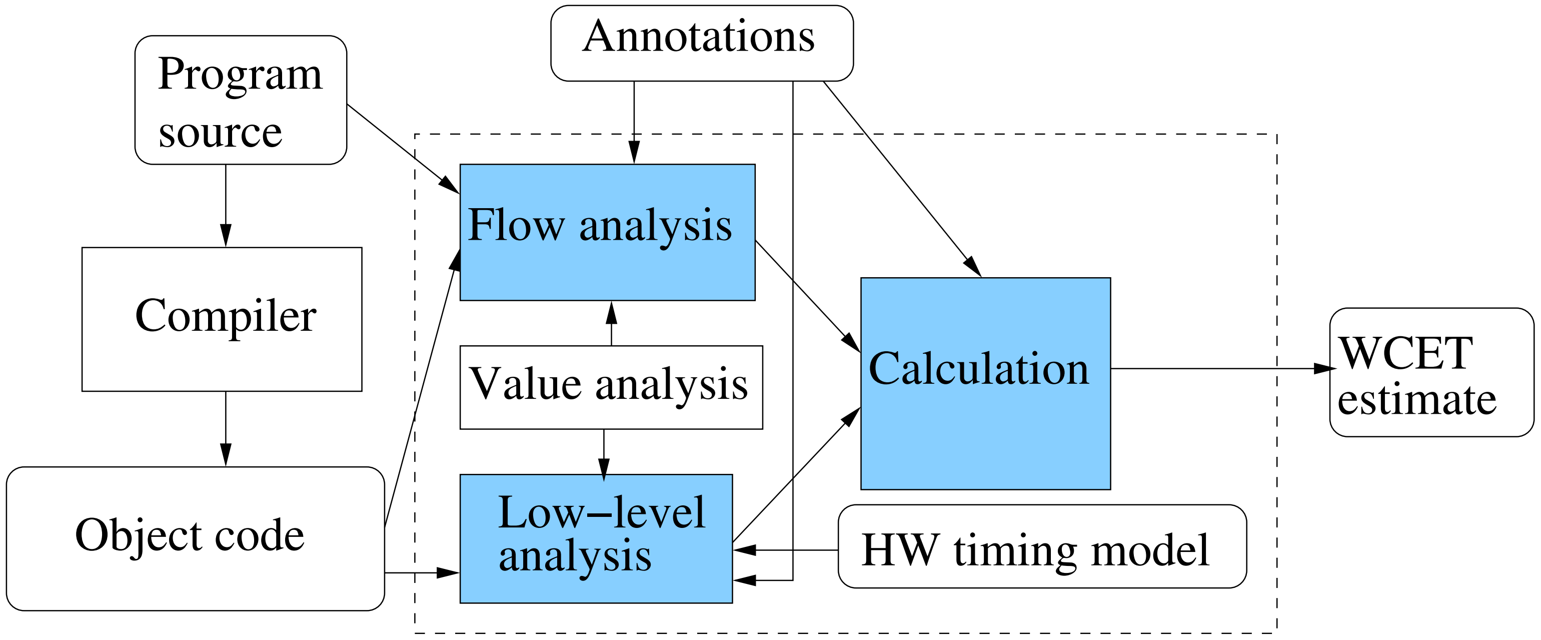
There can be *very many paths*

Must thus make *controlled approximations* to obtain an estimate in reasonable time

Thus, WCET analysis is typically broken down into the following steps:

- Constrain possible program flows (*flow analysis*)
- Estimate hardware impact to bound WCET for program fragments (*low-level analysis*)
- Use information to produce a safe WCET estimate (calculation)

Structure of WCET analysis



Flow Analysis

Purpose: to automatically discover constraints on program flow

1. Bounds on # of loop iterations:

```
i = 1;
while i < 100 do
  ....
  i = i+2
```

All loops must be bounded in order to find a finite WCET estimate

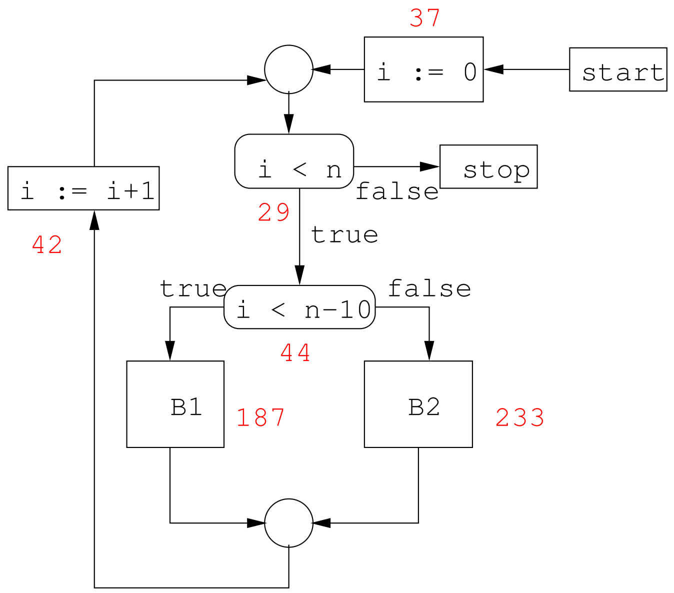
2. Infeasible path constraints can yield sharper WCET estimates:

```
if i < 5 then ...
  ....
if i > 7 then ...
```

Flow analysis is hard to automate completely (halting problem)

Low-Level Analysis

Finds upper bounds to local WCETs for *basic blocks* (short instruction sequences):



Uses hardware timing model for instructions

A Difficulty

Modern hardware can have *complex timing models*:

- Pipelined instruction execution
- Caches
- Out-of order instruction execution (superscalar)
- Branch prediction
- etc. . .

Typically makes basic block execution times dependent on *hardware state*

A safe analysis must use the worst execution time, which may vary widely: can yield large pessimism. Much research how to mitigate this

Calculation

Combining flow information and local WCET bounds

Three different approaches:

- **Tree-based**. Based on high-level program syntax tree, presumes well-structured code. Fast but somewhat imprecise.
- **Path-based**. Local search over all paths in loop bodies to find local execution time maxima. More precise than tree-based.
- **Implicit Path Enumeration**. Constraint-based approach, computes WCET estimate with *Integer Linear Programming* (ILP). General, and precise.

State of the Practice

Tools exist (commercial, and research prototypes), like:

- aiT (absInt GmbH, commercial)
- RapiTime (Rapita Systems Ltd, commercial)
- Bound-T (Tidorum Oy, was commercial: now open source)
- SWEET (MDH, academic)
- TuBound (TU Vienna, academic)
- Chronos (NU Singapore, academic)
- Heptane (Rennes, academic)
- Otawa (Toulouse, academic)

Used mainly in automotive and avionics

Fairly complex processors can be handled, but analysis times can be long

No multicore yet

Level of automation is still a problem

Future Trends

A major challenge: the shift to **parallel hardware and software**

Opens several cans of worms:

- Shared hardware resources can yield very unpredictable execution times
- Parallel programs can suffer from race conditions, deadlocks
- Not possible anymore to structure the analysis into three distinct phases

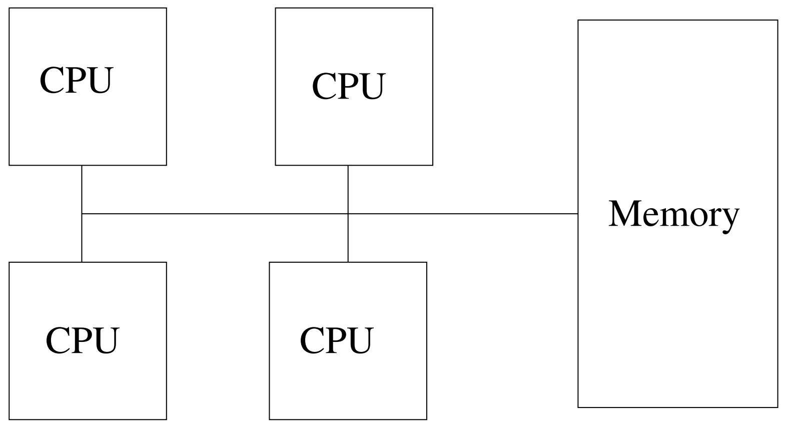
Hot research area, but results are still restricted

Current parallel hardware designs and programming paradigms are just not timing-predictable

Safety-critical parallel systems require timing-predictable hardware and software architectures!

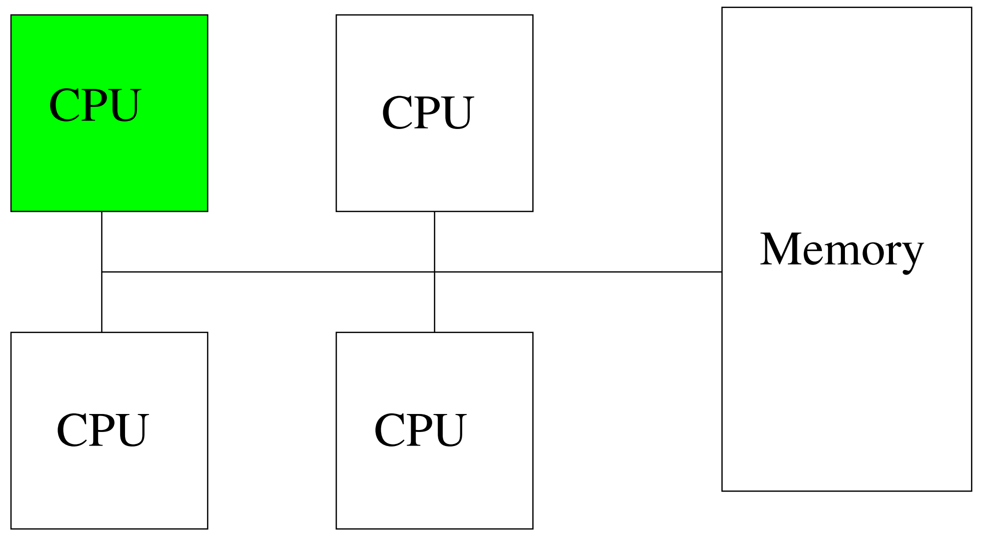
An Example: Multi-Core Processors

A vanilla multicore processor:

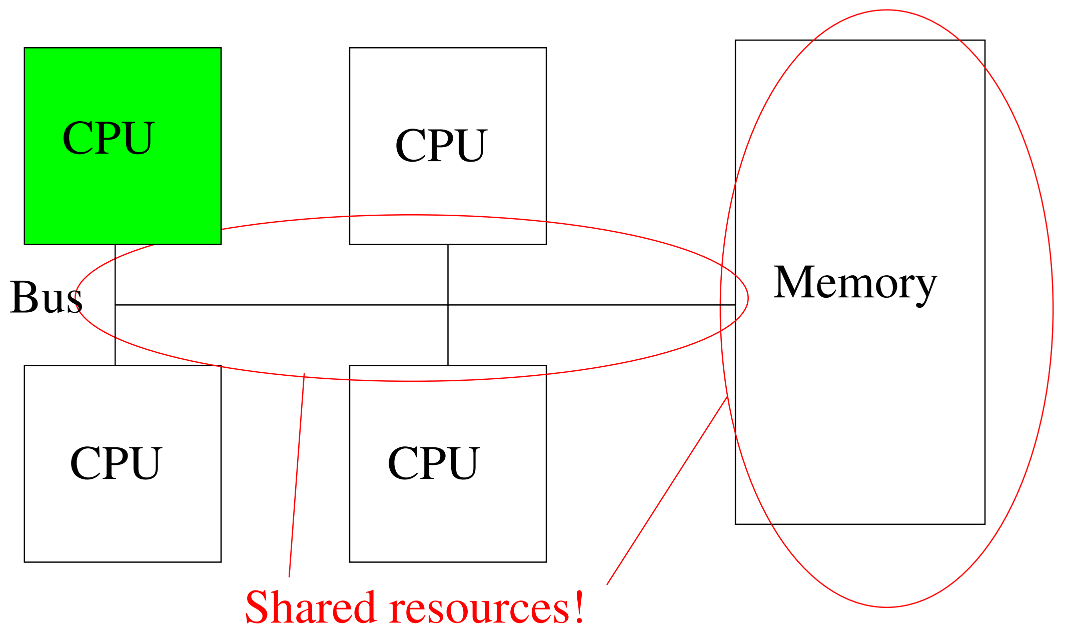


A number of processor cores, and a memory connected by a bus

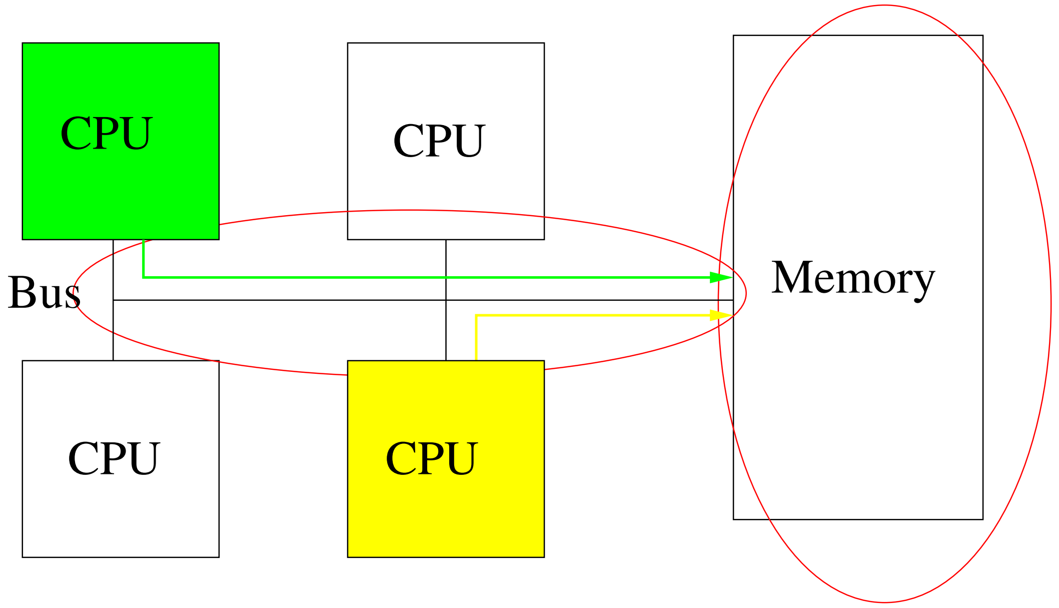
Why Timing Analysis Becomes Problematic



Consider a task running on a core in a multicore system



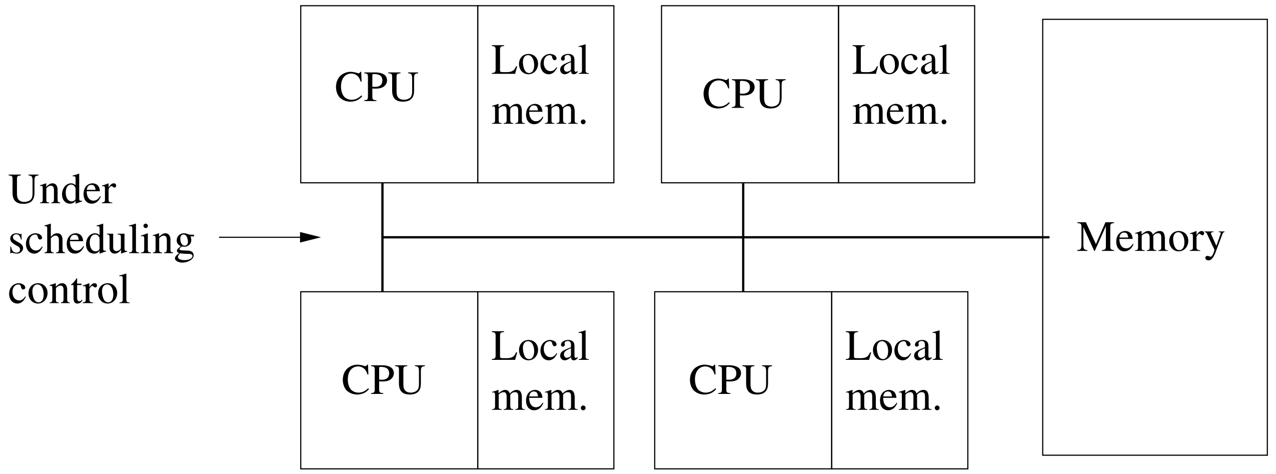
The core will now *share resources* with other cores



When making a memory access, it will have to *compete* with the other cores for the shared resources needed

Memory access times thus become dependent on what is running on the other cores! A safe worst case access time can be very pessimistic. Can yield *very pessimistic WCET bounds*

A Possible Solution



Add local memories (scratchpads). Store local data there. Only shared data in the shared memory

Let each core have a reserved time slot for accessing bus $+$ memory

Problem: multi-cores are not built like this

No good solution exists for currently available multi-cores

Conclusions

Code-level timing analysis is an important tool for verifying timing properties in safety-critical systems

WCET analysis is the most important analysis for this purpose

Can handle systems with current embedded sequential processors, and “reasonable” software

Some effort may be required to obtain tight WCET estimates

No good solution for current multi-cores

Hot research topic how to handle parallel hardware and software