



A Hybrid Approach for Performance Evaluation of Distributed Embedded Software



Wolfgang Rosenstiel

University of Tübingen and

FZI/SiM

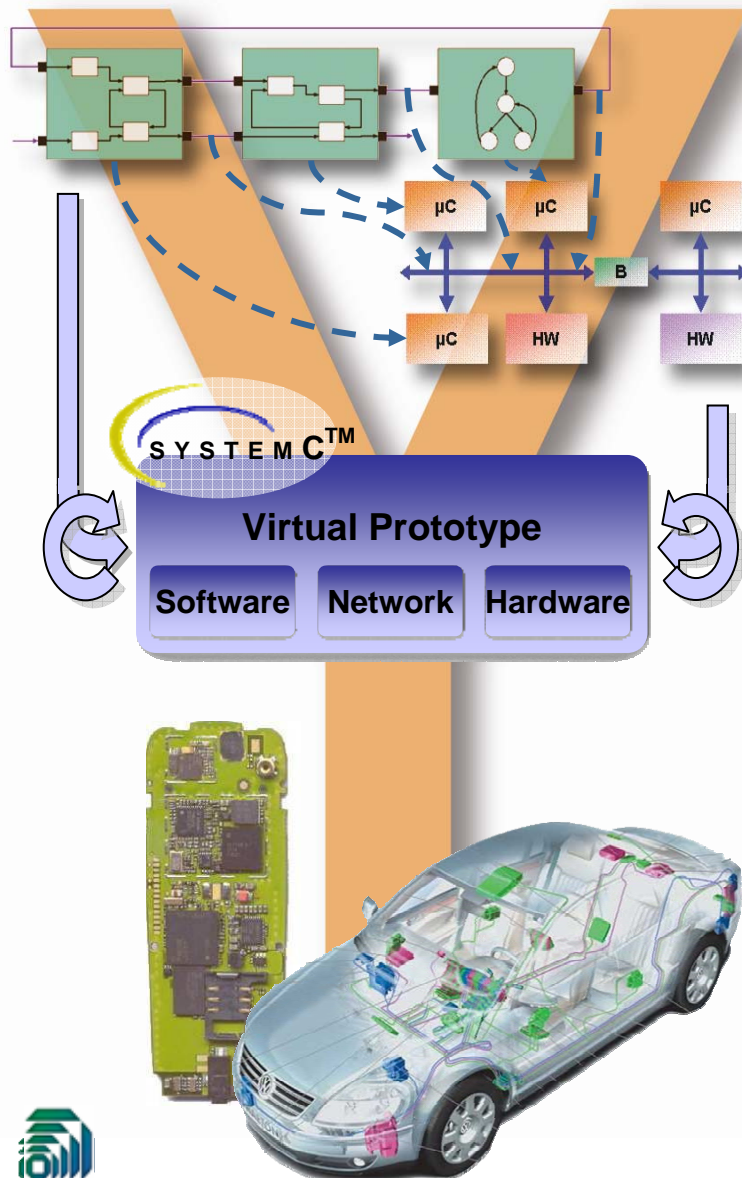


SIM

Outline

- **Embedded Systems Design Flow**
- **SystemC for Modeling and Simulation**
- **Performance Analysis of Hardware-Dependent Software**
 - Analytic approaches
 - Simulation approaches
 - **Hybrid approaches**

DESIGN OF DISTRIBUTED EMBEDDED Systems



- **Comprehensive Modeling of Distributed Systems**

- platform dependent development of the application software (UML, Matlab/Simulink, C++)
- early consideration of the planned target platform
- mapping of function blocks on architecture components
- use of virtual prototypes for the abstract modeling of the target platform

- **Early Evaluation of the Target Platform**

- evaluation of the application software under consideration of the target platform
- optimization of the target platform under consideration of the application software
- performance, power dissipation and reliability

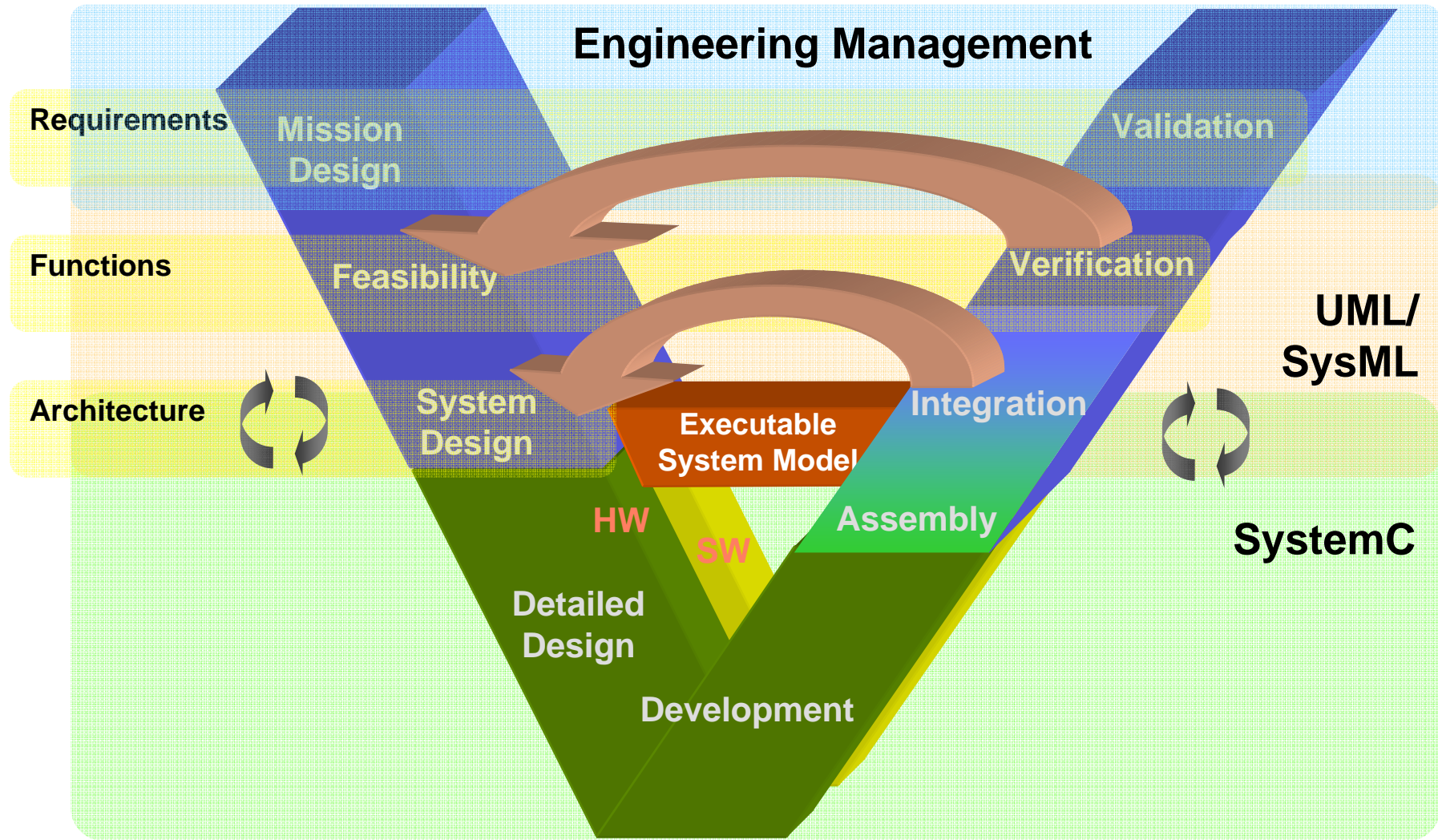
- **Early Analysis of the System Integration**

- early verification based on virtual prototypes
- exposure of integration faults using the virtual prototype

- **Seamless Transition to the Real Prototype**

- automatic “target code” generation
- co-simulation/emulation

Early Analysis of the System Integration



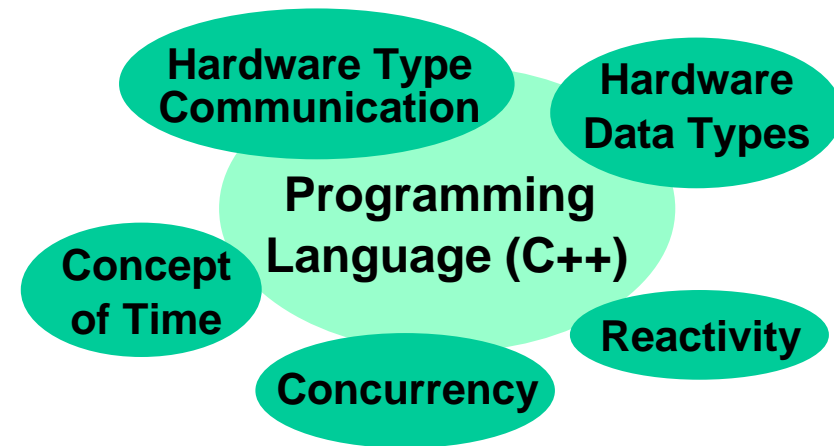
Description Language

C/C++ was *not* developed for Hardware-Design!

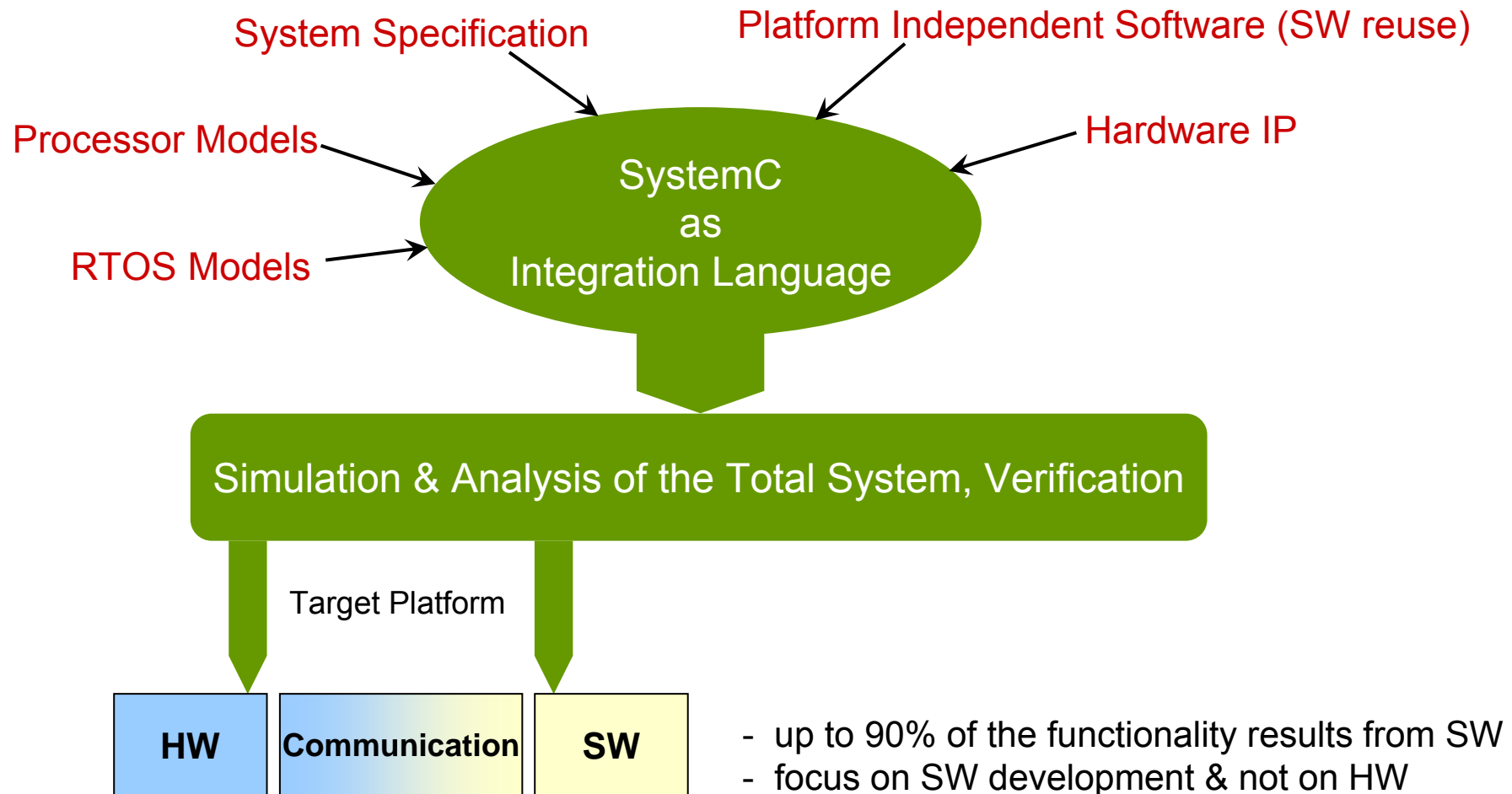
C/C++ does not support

- **Hardware Type Communication**
 - signals, protocols
- **Concept of Time**
 - clock, chronologically successive operations
- **Concurrency**
 - hardware is inherently concurrent, parallel operations
- **Reactivity**
 - hardware is inherently reactive, reacts on signals, interacts with its environment (→ interrupt problems)
- **Hardware Data Types**
 - bit, bit-vector, multi-valued logic, signed and unsigned integer, fixed-point
- **Integrated Simulation Core** for the Realization of “Executable Specifications”

Solution: Class Library

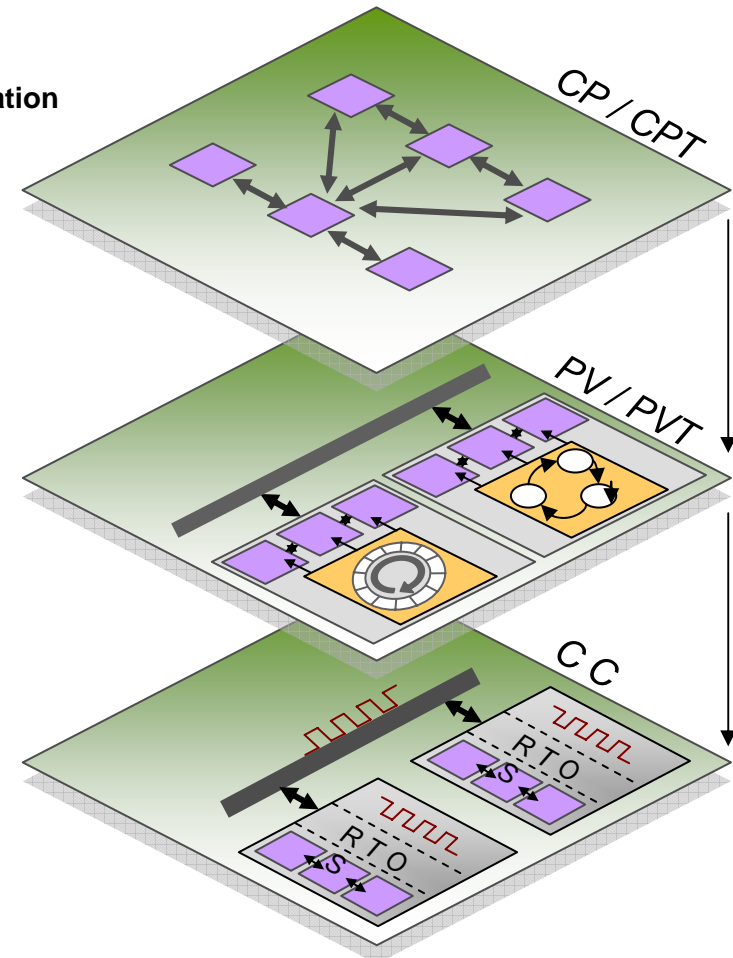
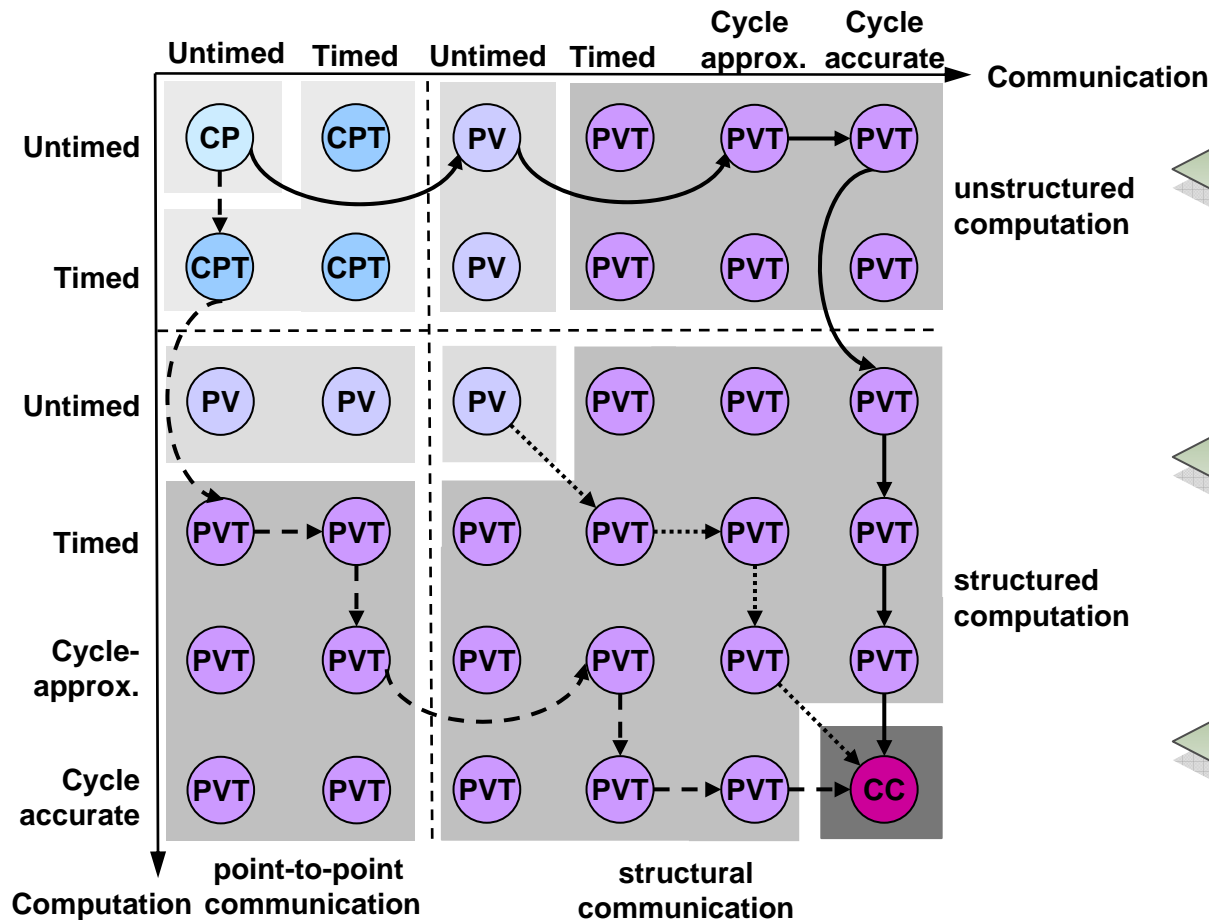


SystemC as Integration Language



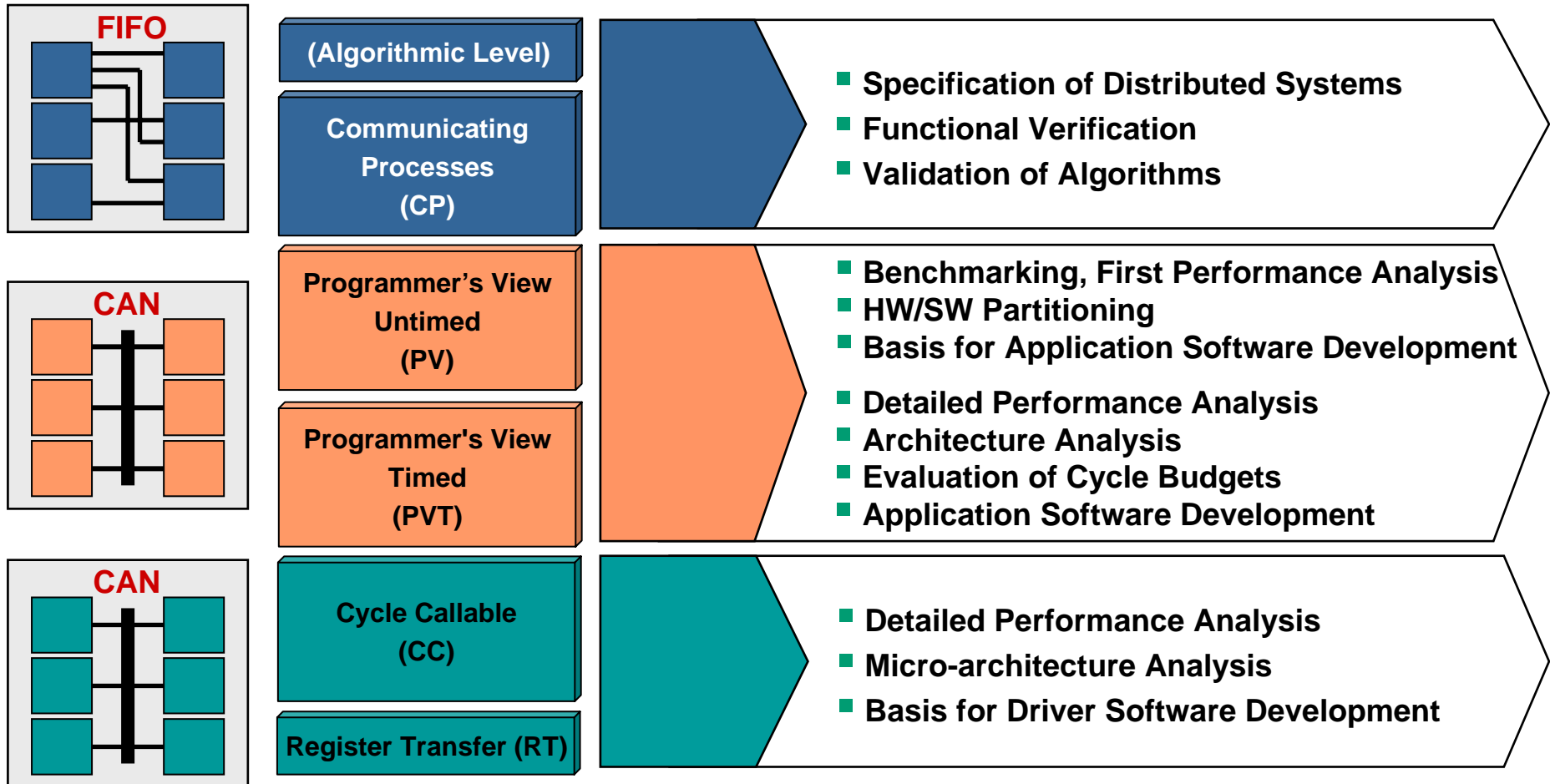
- support of existing solutions both in SW (RTOS) and in HW design flow and no creation of island solutions
- existing standard C/C++ based development environments can be used

Levels of Abstraction (TLM)



- CP = Communicating Processes; parallel processes with parallel point-to-point communication**
- CPT = Communicating Processes + Timing**
- PV = Programmers View; scheduled SW-computation and/or scheduled communication**
- PVT = Programmers View + Timing**
- CC = Cycle Callable; cycle accurate timing behavior for computation and communication**

Applications of the Levels of Abstraction



PERFORMANCE ANALYSIS OF DISTRIBUTED Embedded Systems

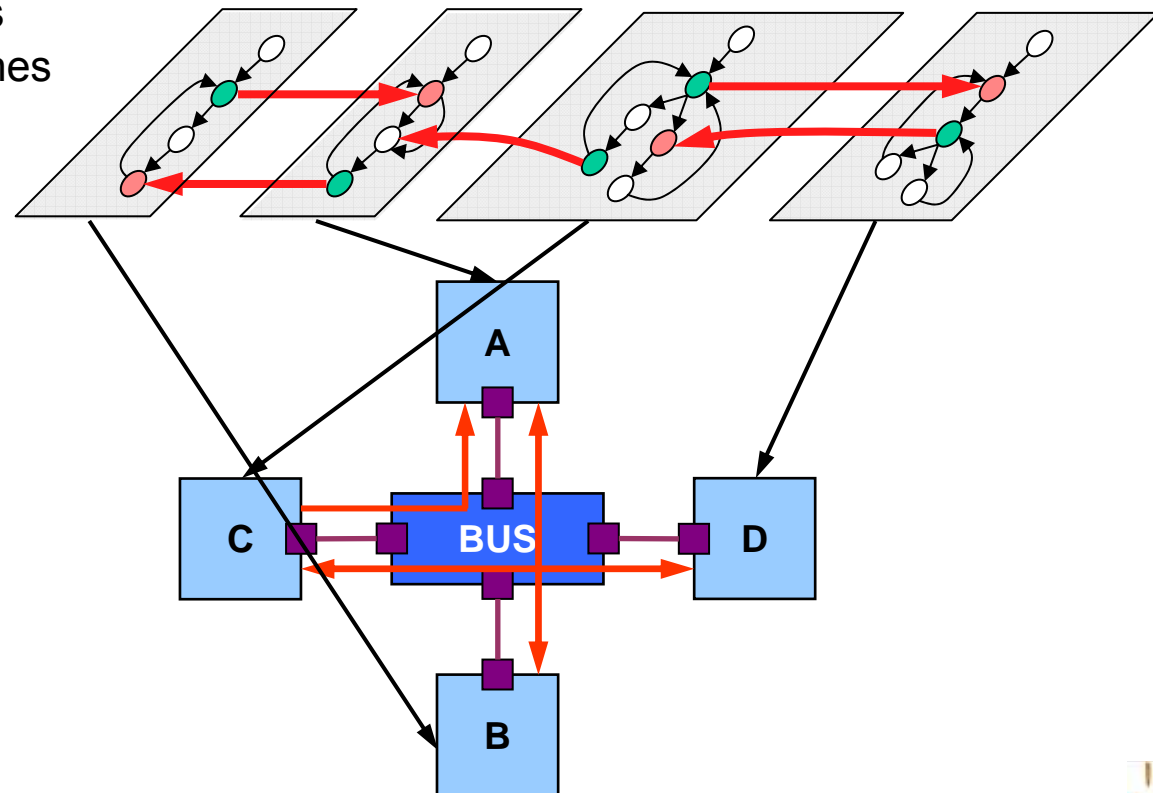
Question: How can I determine the timing behavior of my system?

- **Problem:**

- software has no timing behavior as long as the target platform is unknown
- timing behavior is determined by the target platform
- but: Specification specifies time constraints especially in real time embedded systems

- **Solution: Early Consideration of the Target Architecture**

- analytic approaches
- simulation approaches
- hybrid approaches



Analytic Approaches /1

- Formal analysis of the border cases by a system model
- **Two possible kinds of approaches:**
 - black-box approaches
 - white-box approaches
- **Black-box approaches**
 - Modeling
 - abstract task model and task activation
 - event stream, execution times on task level
 - Propagation of event streams
 - Examples for black-box approaches:
 - Real-Time Calculus: (Thiele, ETH Zurich)
 - System-level composition by event stream propagation: SymTA/S (Ernst, SymtaVision)

Analytic Approaches /2

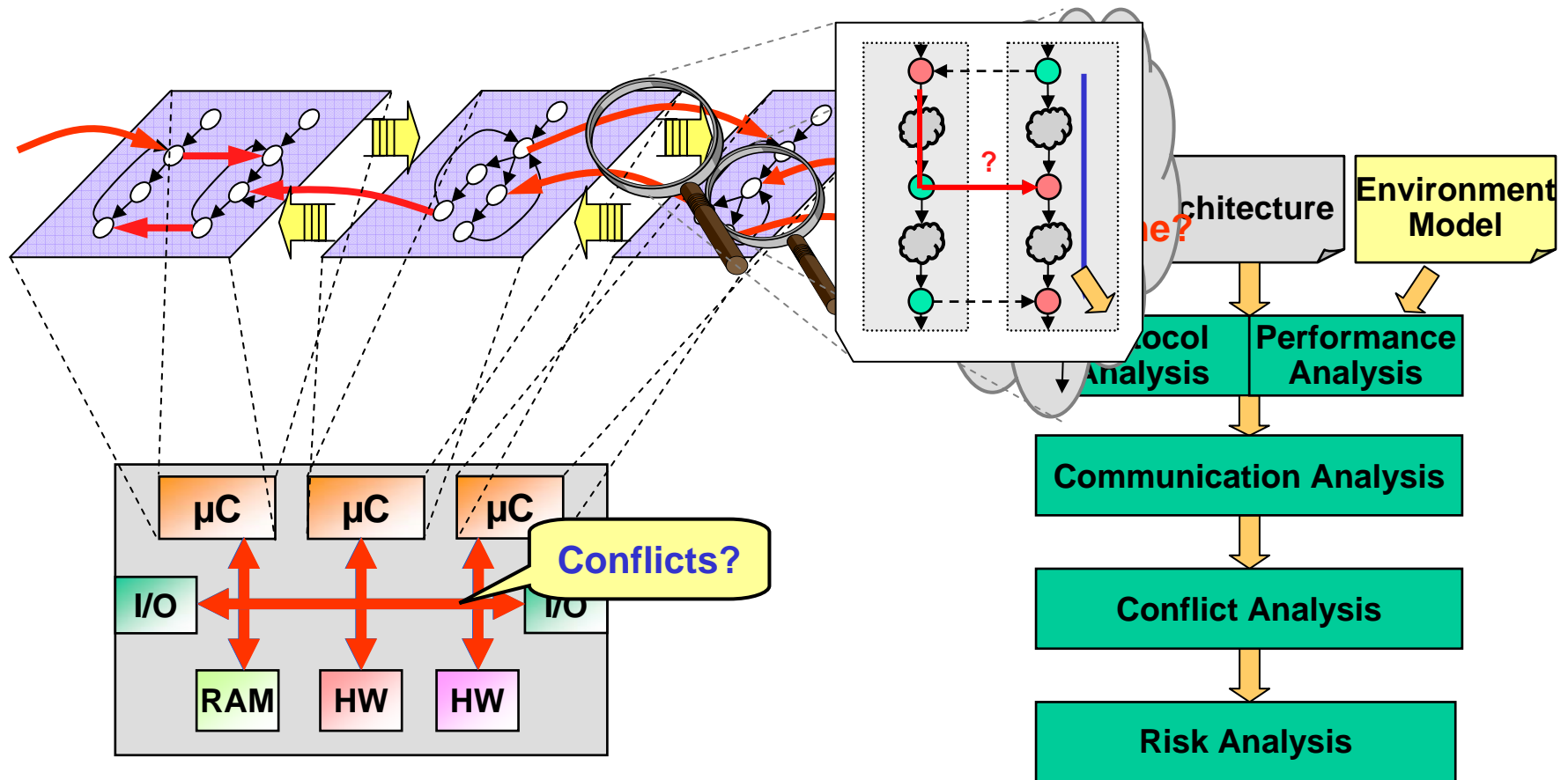
- **White-box approaches**

- Inclusion of the control flow of each process into the system model
- Global performance- and communication analysis considering (data dependent) control structures of all processes
- Extraction of the control flow out of application software or UML-Model
- Environment modeling using event models or processes
- Examples for white-box approaches:
 - Communication Dependency Analysis: SysXplorer (Bringmann, Rosenstiel, FZI)
 - Control flow based extraction hierarchical event streams: (Slomka, Oldenburg)

- **Problems of the analytic approaches**

- often to pessimistic
- risk estimation for real scenarios difficult

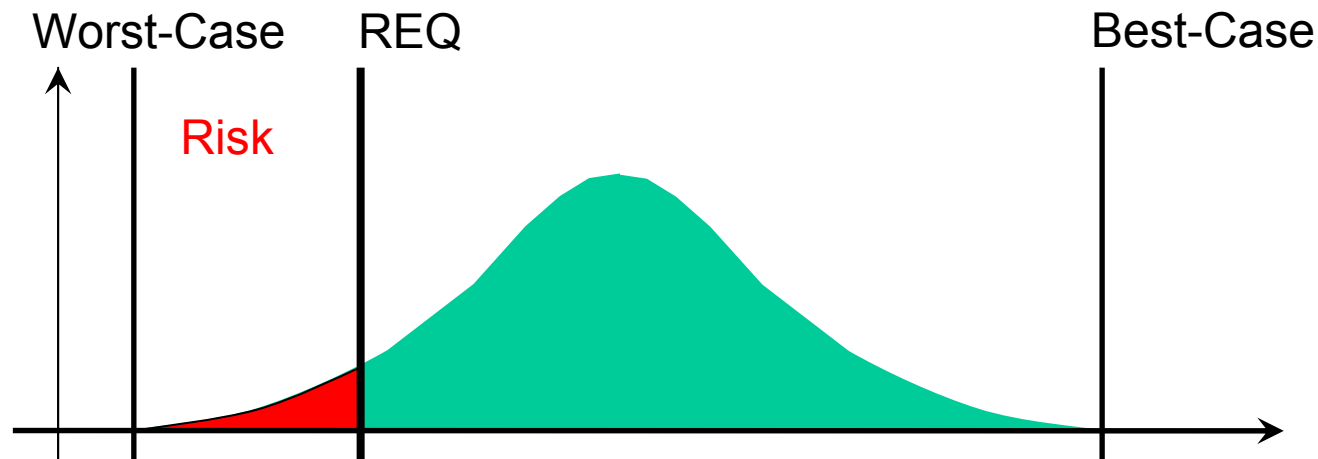
White-Box Analysis Flow



- global performance analysis of HW/SW Systems
- automatic detection of bottlenecks as well as time and resource conflicts
- application-oriented optimization of the system architecture
- → Bringmann, Rosenstiel: [CODES/ISSS 05], [ASPDAC 07]

Risk Assessment

- **Up to now:** Determination of the worst-case and best-case characteristics of parameters
- **Goal:** Performance numbers with a probability distribution for risk assessment
- **Approach:** Include probabilities in the formal model
- Alignment of requirements with the determined distributions leads to risk assessment
- e.g. QoS of an MPEG-decoder



PERFORMANCE ANALYSIS BY Simulation

- **Simulation of the software execution on the target processor**
 - Accurate but fast simulation for
 - Architecture exploration and
 - Validation of timing constraints
 - Software Debugging
- **Instruction-Set-Simulation**
 - Modeling of the instruction execution, no bus modeling
 - **interpretation** or binary code
- **Processor model**
 - includes the complete processor (FUs, Pipeline, Cache, Register, Counter, ...)
 - Transaction-Level-Model
 - Register-Transfer-Model

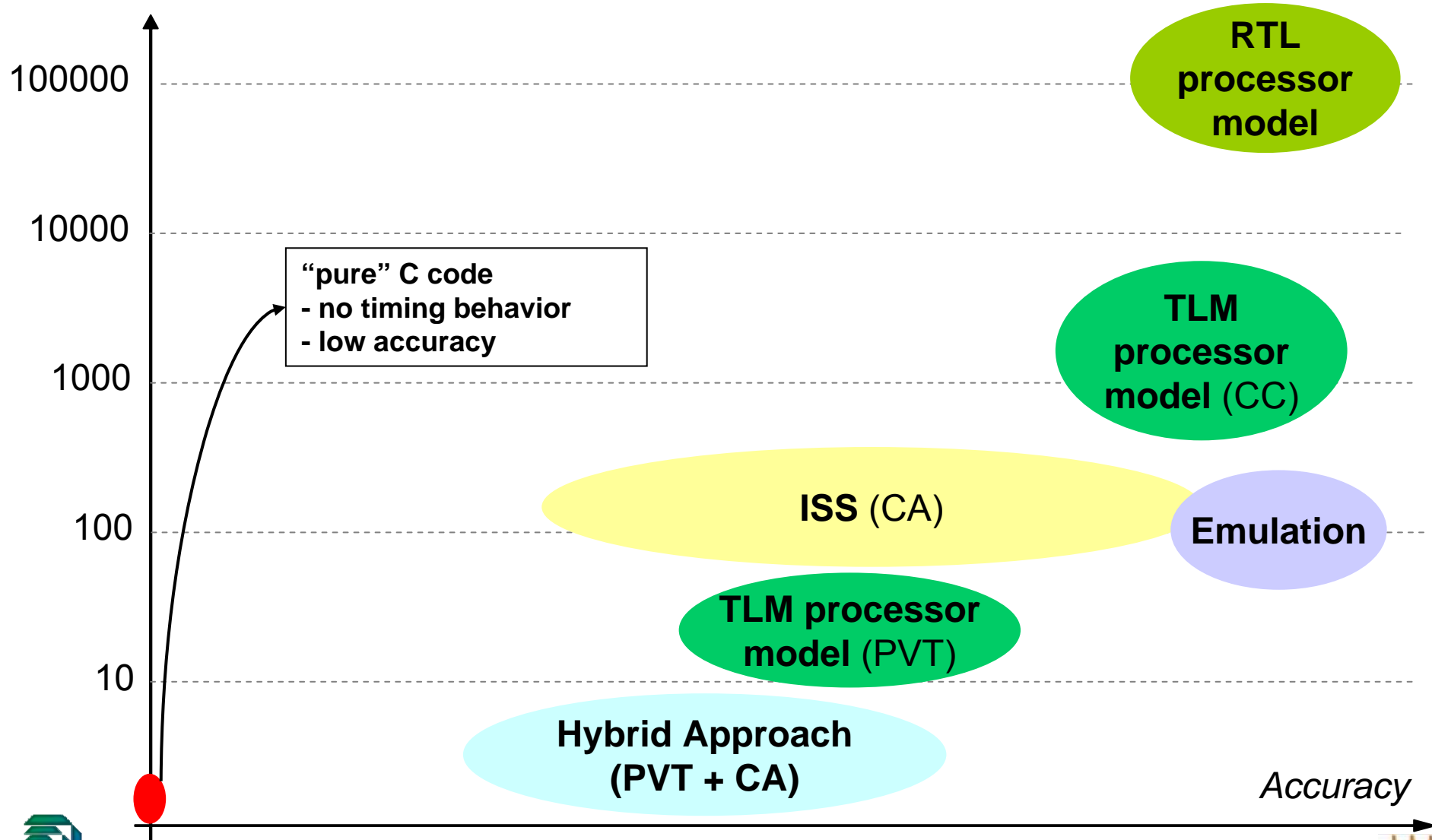
Hybrid Approaches

- **Combination of the analytic and simulation approaches**
 - to gain simulation speed
- **Observation**
 - Static WCET/BCET-Analysis delivers very good results inside a basis block,
 - but is too pessimistic across basis block boundaries.
 - exact simulation requires time-consuming interpretation of binary code
- **Two alternative new concepts:**
 - Translation of binary code into annotated SystemC code
 - Back-annotation of WCET/BCET values determined statically on basic block level in SystemC source code (communicating software-threads)
 - Followed in both cases by translation and simulation of the SystemC code
- **Problem**
 - Misprediction at basic block boundaries (branch prediction, cache)
- **Solution**
 - Embedding of a dynamic calculation of correction cycles
 - Approach: branch behavior and cache accesses are known at run-time

Simulation Speed vs. Simulation Accuracy

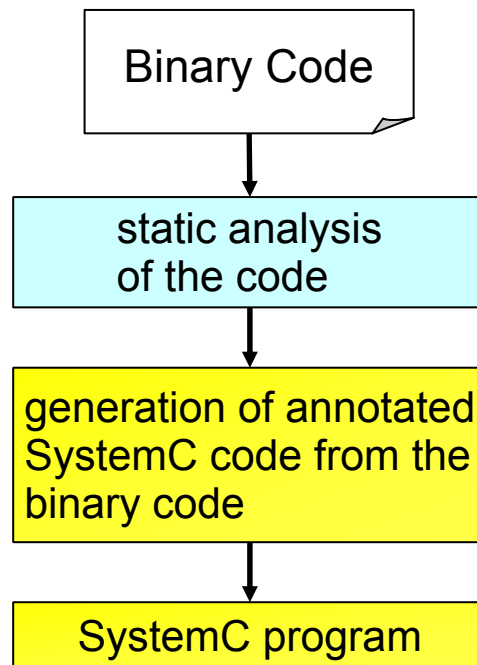
Accuracy

Medium simulation time

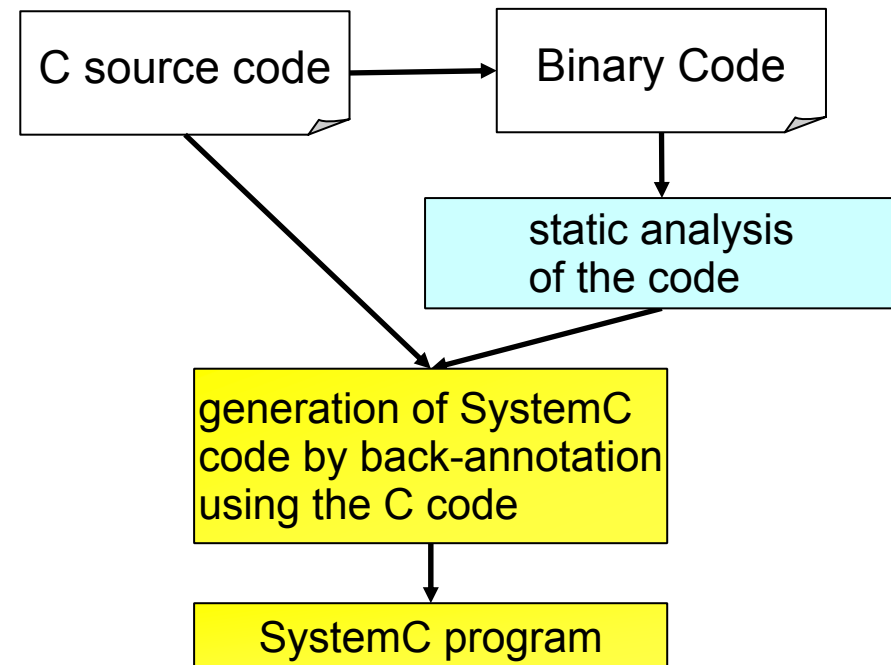


Basic Concepts

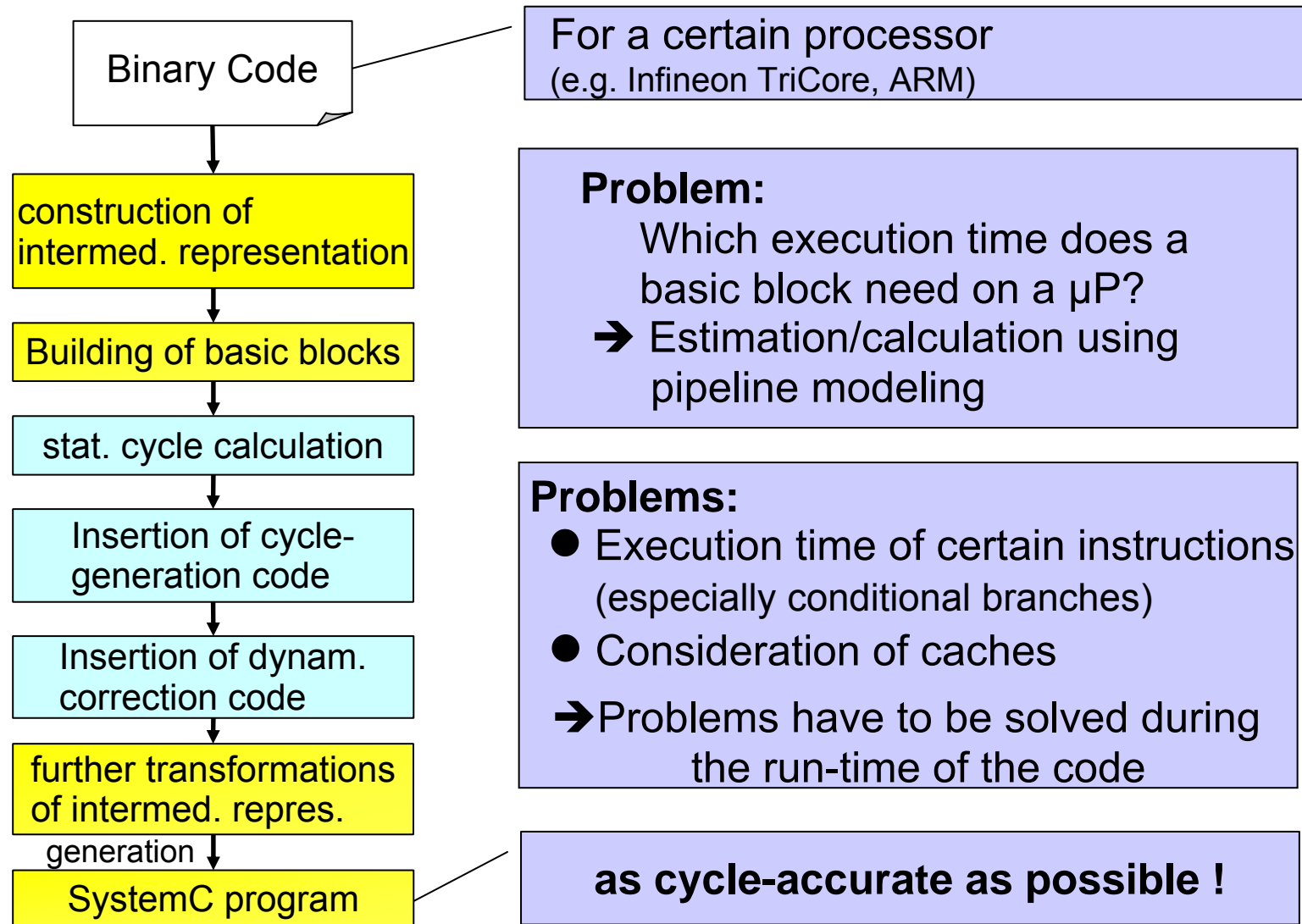
Translation of binary code into annotated SystemC code



Back-annotation of WCET/BCET values determined statically on basic block level in SystemC source code



Idea 1: Translation of Binary Code into annotated SystemC Code /1



Idea 1: Translation of Binary Code into annotated SystemC Code /2

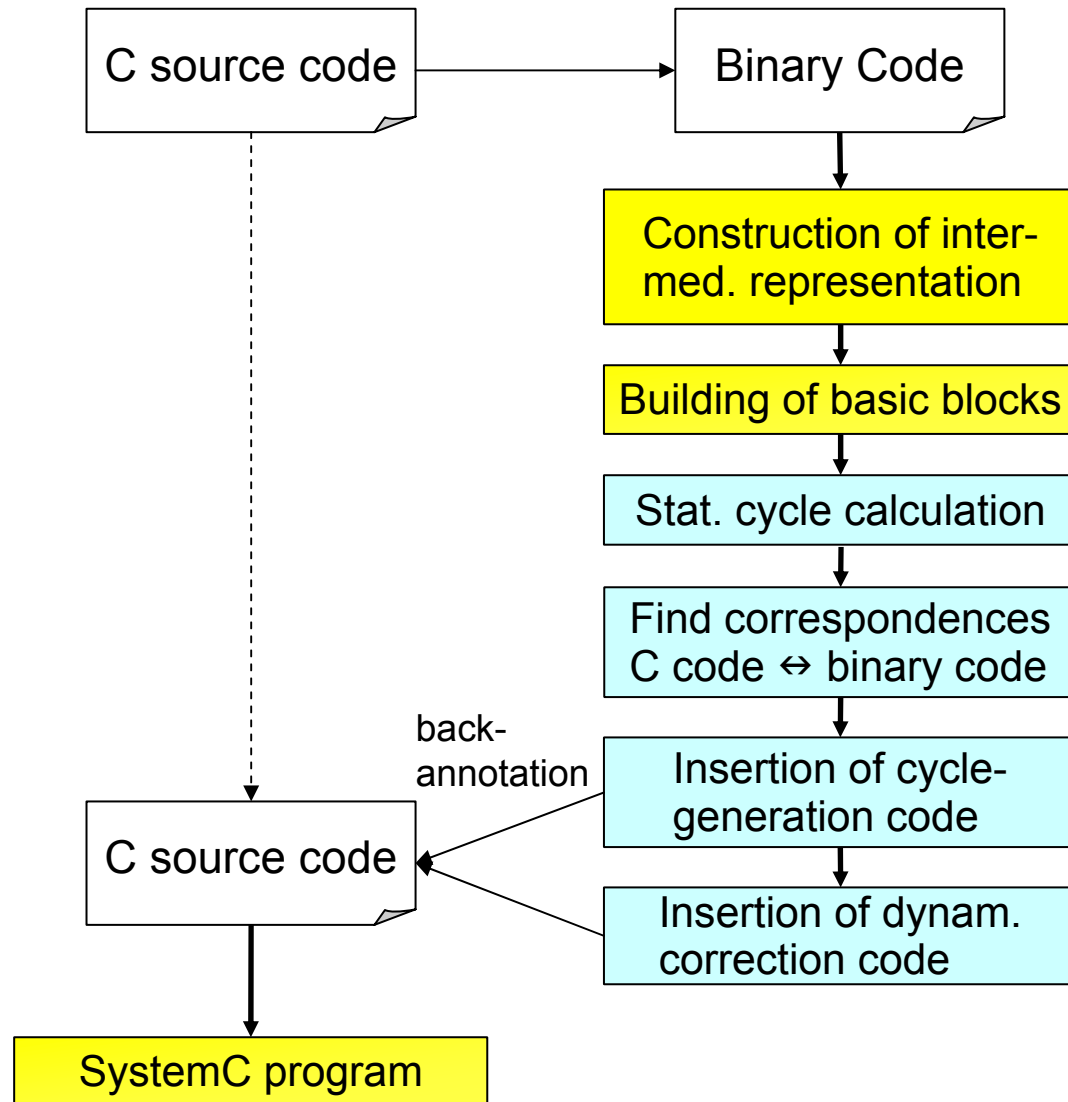
- **Advantages:**

- no source code needed
- Binary code can be used to find out cycle counts and to generate SystemC code
- Translation of binary code into SystemC code generates fast code compared to an interpreting ISS, as no decoding of instructions is needed
- Generated SystemC code can be easily used within a SystemC simulation environment

- **Disadvantages:**

- The same problems that have to be solved in static compilation (binary translation) have to be solved here,
 - **e.g.** addresses of calculated branch targets

Idea 2: BACK-Annotation of Cycle Information into C Program /1



Main Problem:

Finding corresponding parts of the binary code in the C code for the annotation.

- can be solved by letting the compiler generate debuggable code
- then it is possible to find the correspondence between the source code and the generated binary code

Idea 2: BACK-Annotation of Cycle Information into C Program /2

- **Advantages:**

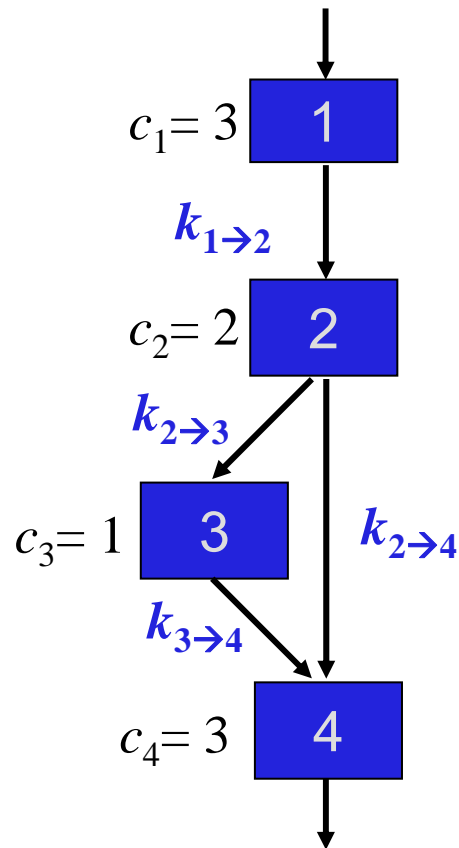
- Fast execution of the annotated code
- C source code does not have to be changed much for the annotation
- Generated SystemC code can be easily used within a SystemC simulation environment

- **Disadvantages:**

- C source code is needed for annotation
- Finding corresponding parts of the binary code in the C source code is difficult if the compiler optimizes or changes the structure of the binary code too much
 - then recompilation techniques have to be used

Cycle Calculation

Control Flow Graph of a Program



Static cycle prediction

c_i cycle count of basic block i

Dynamic correction

$k_{i \rightarrow j}$ Correction cycle count for the transition from basic block i to basic block j

conditional branch or loop instruction
certain instructions
cache

dynamic correction cycles

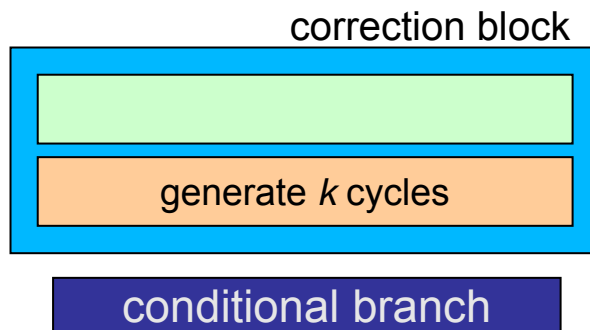
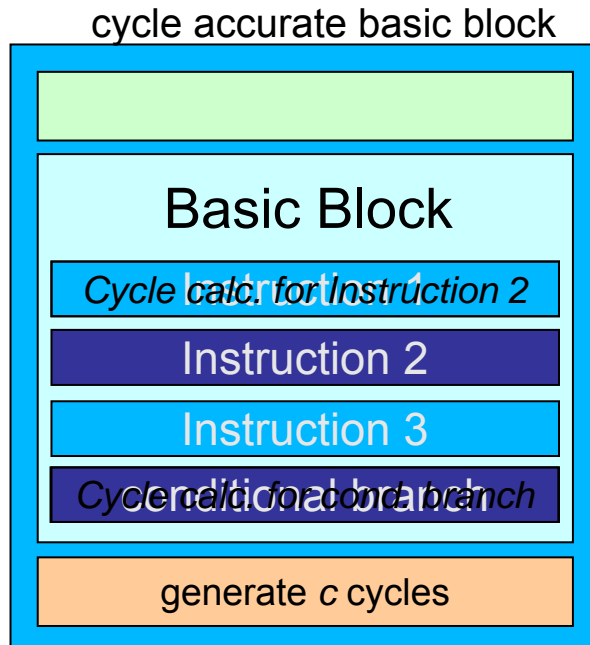
$- g_{i \rightarrow j}$

Three detail levels:

- ❶ pure static calculation
- ❷ additional dynamic correction
- ❸ additional dynamic inclusion of caches

Dynamical Correction of the Cycle Calculation

Instructions without statically known execution time



Problem:

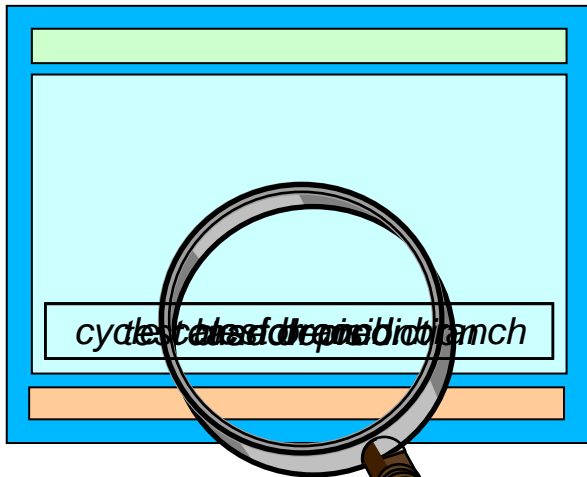
Some instructions do not have a statically predictable execution time.

- Code has to be added to find additional cycles during execution
- A correction block has to be added at the end of the basic block. This block generates these correction cycles.

For conditional branches the branch prediction has to be considered additionally.

Dynamical Cycle Correction for Conditional Branches

Consideration of the Branch Prediction



- Find out, if branch is taken according to the branch prediction
- Find out, if branch is really taken
- **Case decision** with help of these results

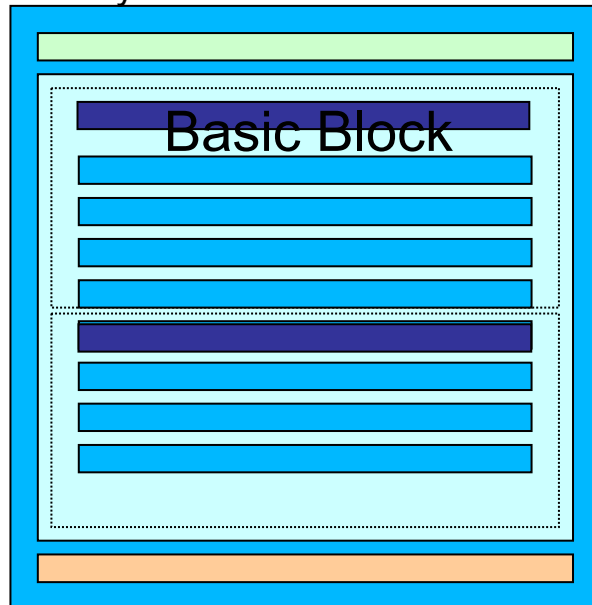
$$k = k + \text{appropriate corr. cycles}$$

- **For processors having a statical branch prediction:**
 - it is known during translation time, if a branch is predicted as taken or not
- **For processors having a dynamical branch prediction:**
 - a model of the branch predictor that is used by the processor has to be included into the translated program

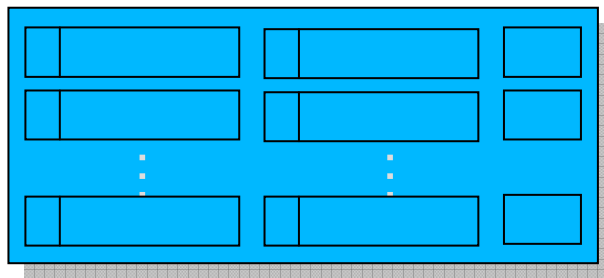
Caches

Instruction Caches

cycle accurate basic block



cache model



Problem:

Caches can have an important influence on the execution time.

- **Splitting** the basic block according to the cache architecture
- Adding of **cache cycle calculation code**
- A **cache model** is added at the end of the translated program

Conclusion

- **Methodical Design of Distributed Embedded Systems**
 - comprehensive specification of the hardware- and software parts
 - early model-based system integration on the basis of virtual prototypes
 - platform-independent development of hardware-near software
- **Performance Analysis of the Application Software**
 - analytical approaches
 - simulative approaches
 - hybrid approaches
- **Two Ideas for Hybrid Approaches**
 - generation of annotated SystemC code from **binary code**
 - generation of SystemC code out of the **original C code** and back-annotation of cycle-information into the generated code



Thank you very much for your attention!

Questions?

Wolfgang Rosenstiel

FZI Forschungszentrum Informatik

Microelectronics System Design

Email: rosenstiel@fzi.de

Web: www.fzi.de/sim

University of Tübingen

Wilhelm-Schickard-Institut

Technische Informatik

Web: www-ti.informatik.uni-tuebingen.de