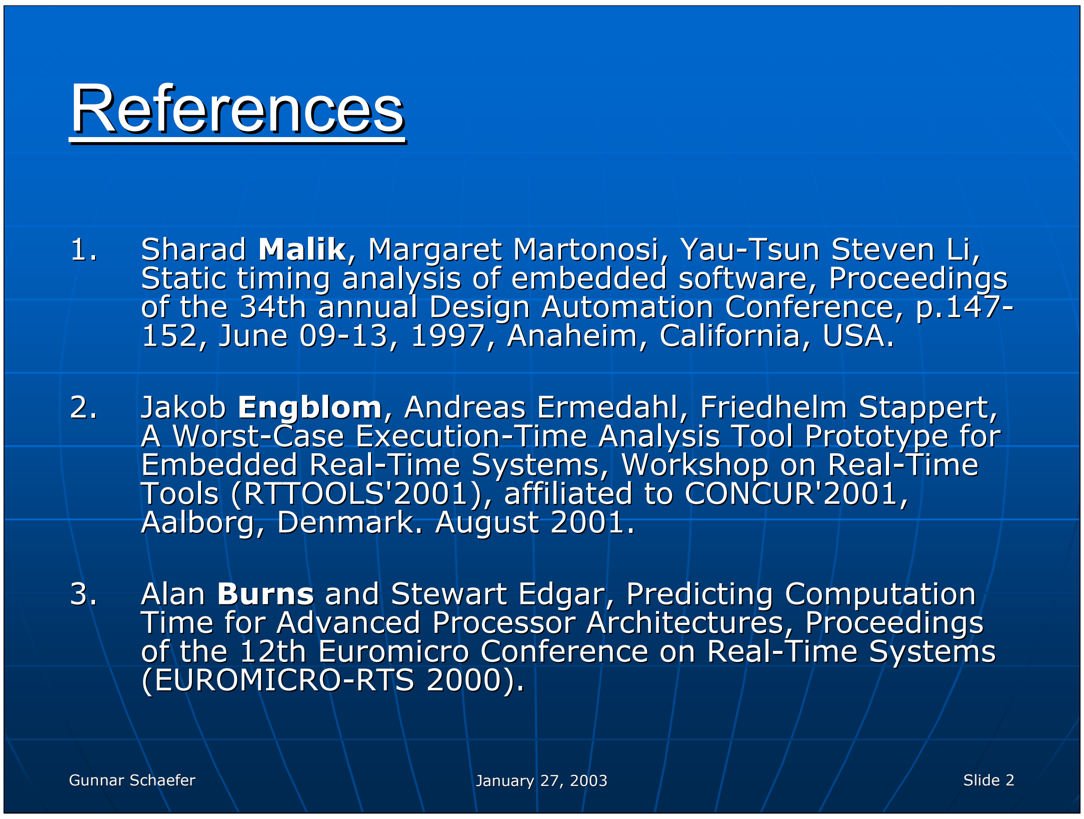
Seminar
Analyse von Programmausführungszeiten
CAU Kiel, WS 2002/03, Prof. Dr. R. v. Hanxleden

January 27, 2003

Overview

Gunnar Schaefer

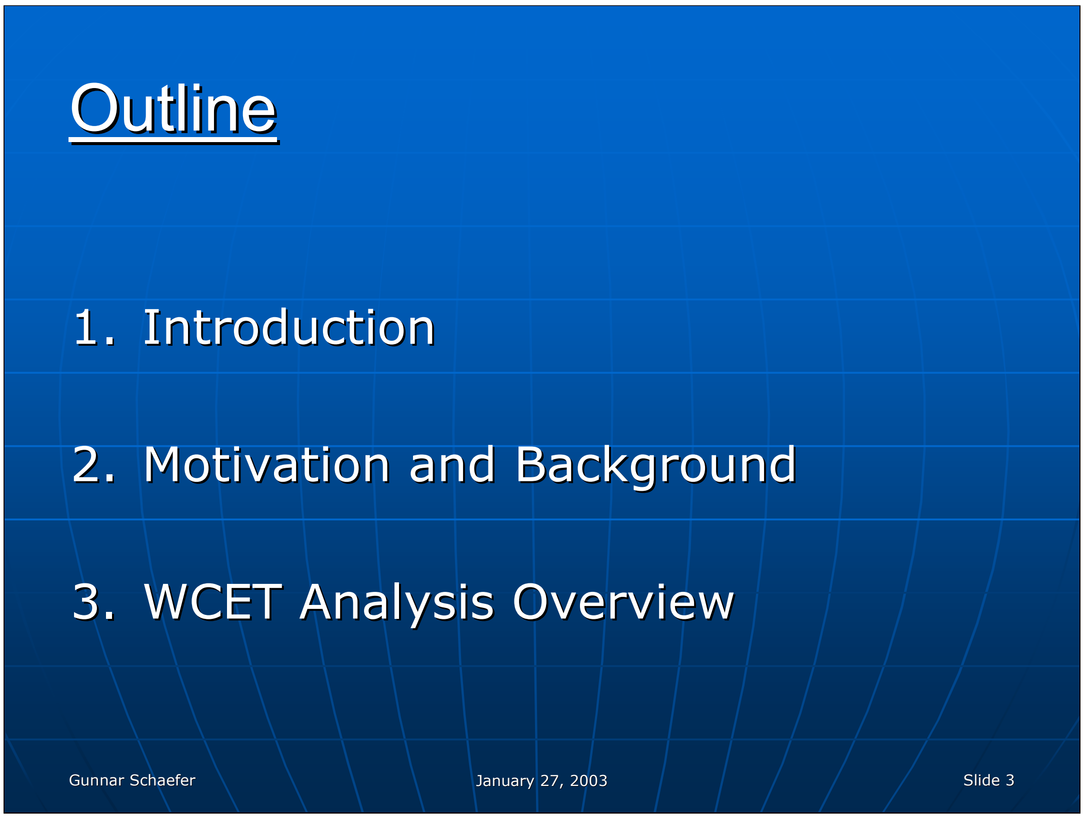


References

1. Sharad **Malik**, Margaret Martonosi, Yau-Tsun Steven Li, Static timing analysis of embedded software, Proceedings of the 34th annual Design Automation Conference, p.147-152, June 09-13, 1997, Anaheim, California, USA.

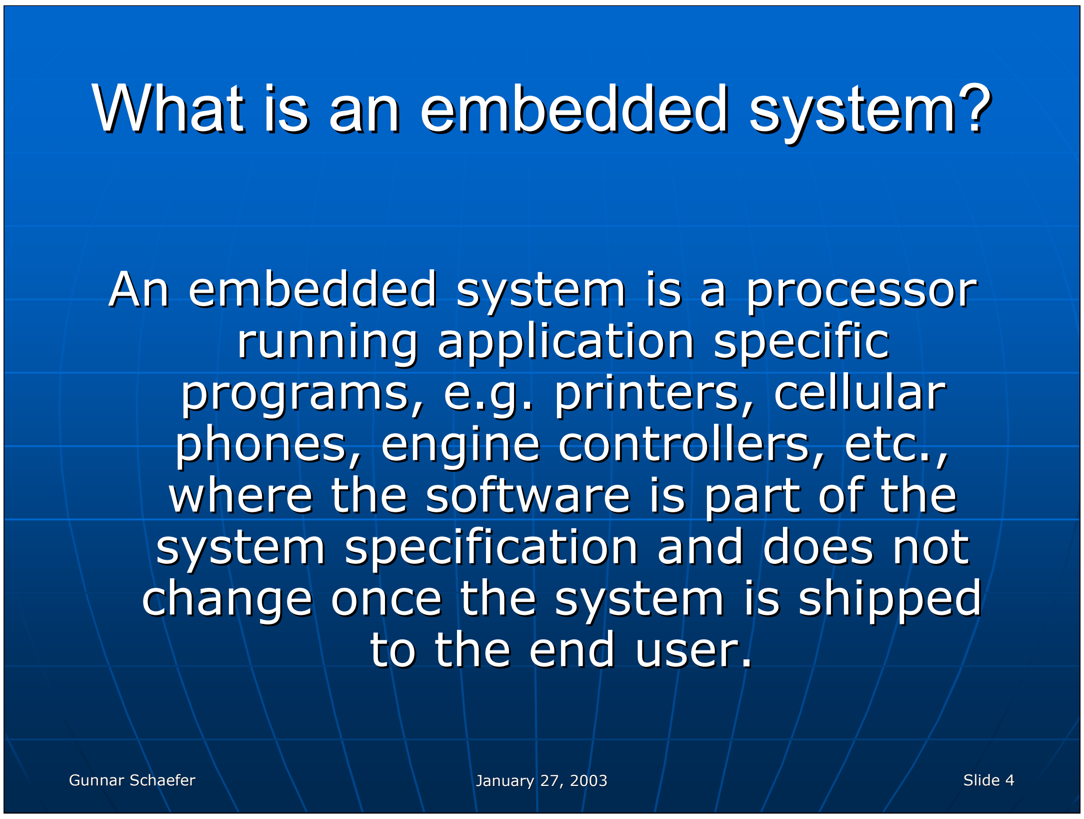
2. Jakob **Engblom**, Andreas Ermedahl, Friedhelm Stappert, A Worst-Case Execution-Time Analysis Tool Prototype for Embedded Real-Time Systems, Workshop on Real-Time Tools (RTTOOLS'2001), affiliated to CONCUR'2001, Aalborg, Denmark. August 2001.

3. Alan **Burns** and Stewart Edgar, Predicting Computation Time for Advanced Processor Architectures, Proceedings of the 12th Euromicro Conference on Real-Time Systems (EUROMICRO-RTS 2000).



Outline

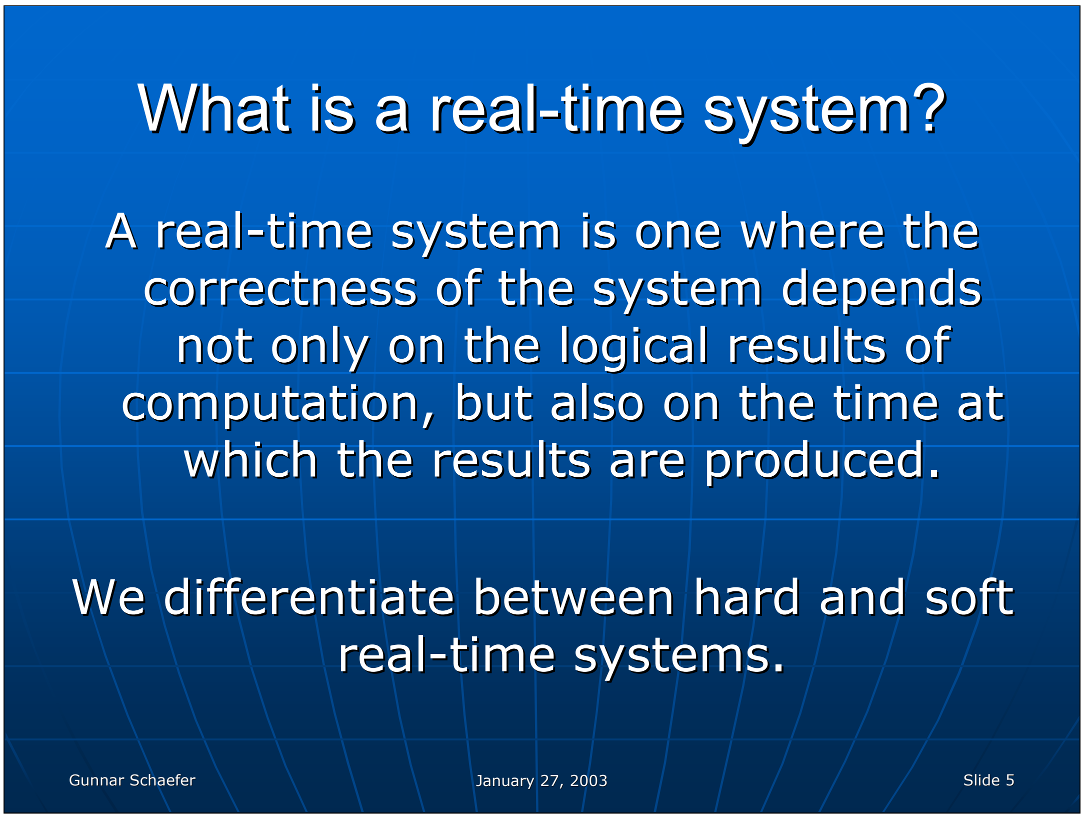
1. Introduction
2. Motivation and Background
3. WCET Analysis Overview



What is an embedded system?

An embedded system is a processor running application specific programs, e.g. printers, cellular phones, engine controllers, etc., where the software is part of the system specification and does not change once the system is shipped to the end user.

Gunnar Schaefer January 27, 2003 Slide 4



What is a real-time system?

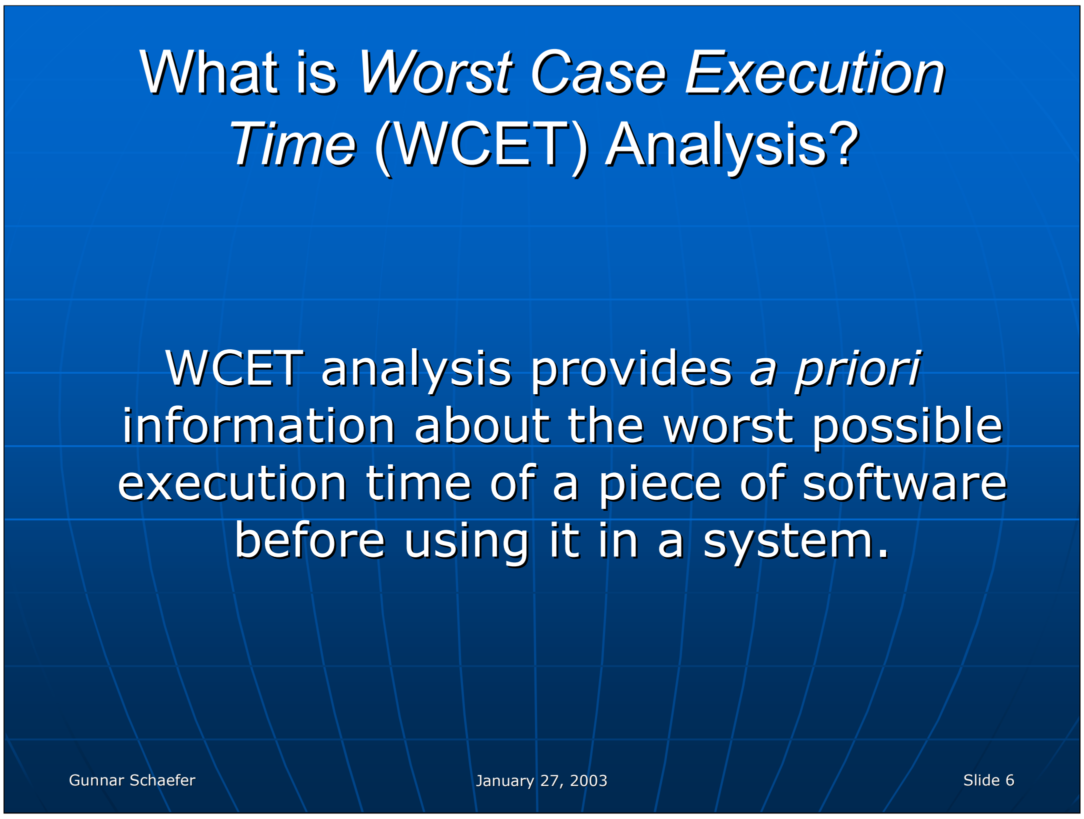
A real-time system is one where the correctness of the system depends not only on the logical results of computation, but also on the time at which the results are produced.

We differentiate between hard and soft real-time systems.

Gunnar Schaefer January 27, 2003 Slide 5

A **hard real-time system** cannot tolerate any missed timing deadlines. An example of a hard real-time system is an automotive engine control unit, which must gather data from sensors, and compute the proper air/fuel mixture and ignition timing for the engine, within a single rotation. In such systems, the response time must comply with the specified timing constraints under all possible conditions. Thus the performance metric of interest here is the extreme case performance of the software. Typically, the worst-case is of interest, but in some cases, the best-case may also be important to ensure that the system does not respond faster than expected.

In a **soft real-time system**, the timing requirements are less stringent. Occasionally missing a timing deadline is tolerable. An example of a soft real-time system is a cellular phone. During a conversation, it must be able to encode outgoing voice and decode the incoming signal in real-time. Occasional glitches in conversation due to missed deadlines are not desired, but are nevertheless tolerated. In this case, a probabilistic performance measure that guarantees a high probability of meeting the time constraints suffices.

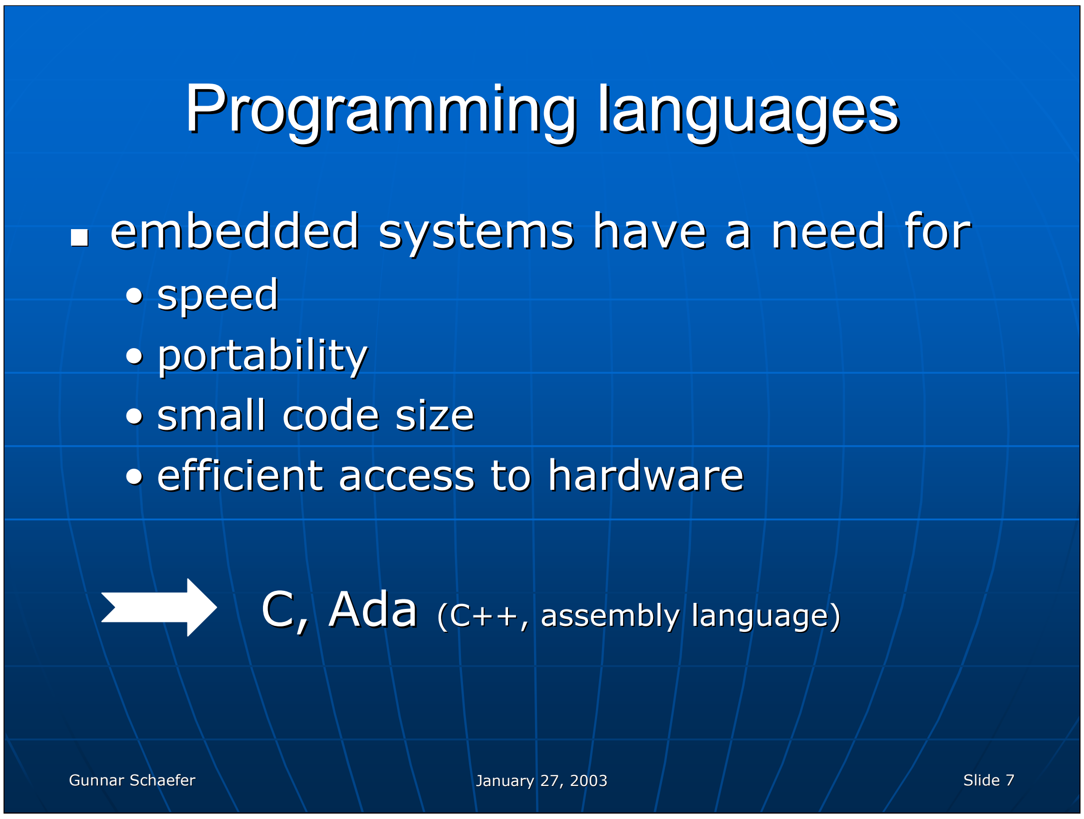


What is *Worst Case Execution Time* (WCET) Analysis?

WCET analysis provides *a priori* information about the worst possible execution time of a piece of software before using it in a system.

Gunnar Schaefer January 27, 2003 Slide 6

a priori information (here): information about the software acquired through analysis and testing, however, before (without) actual usage of that software



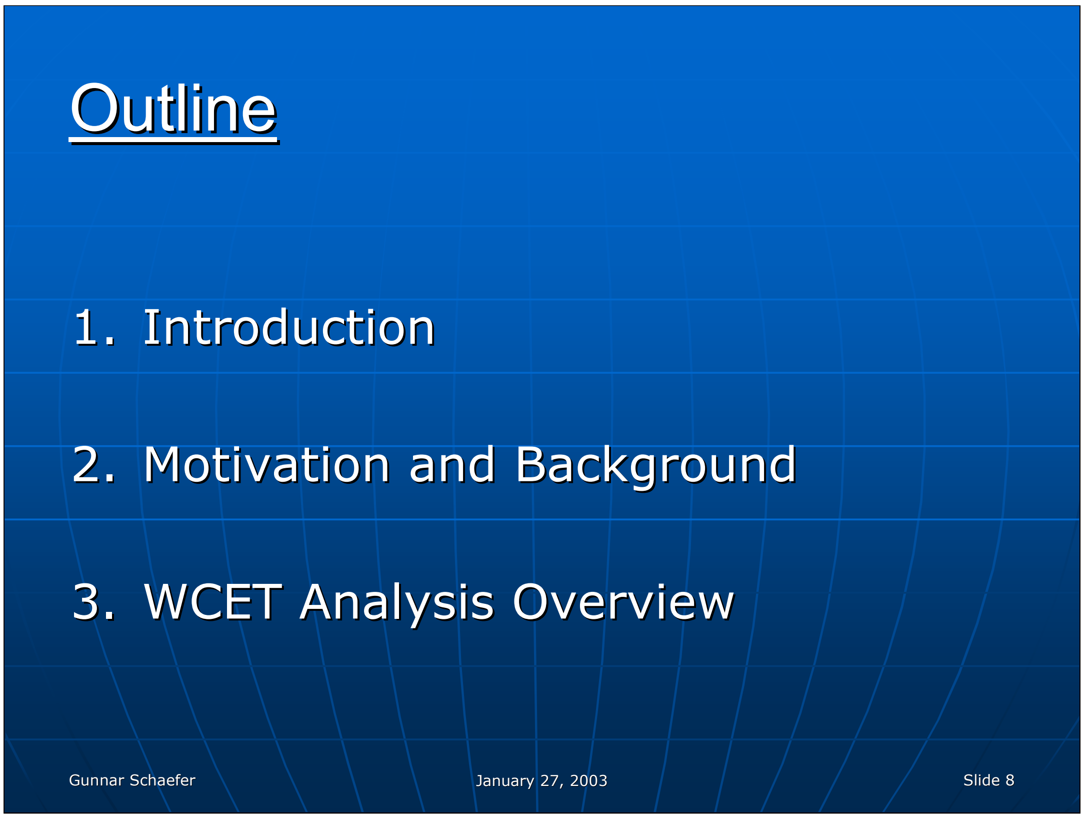
Programming languages

- embedded systems have a need for
  - speed
  - portability
  - small code size
  - efficient access to hardware

➡ C, Ada (C++, assembly language)

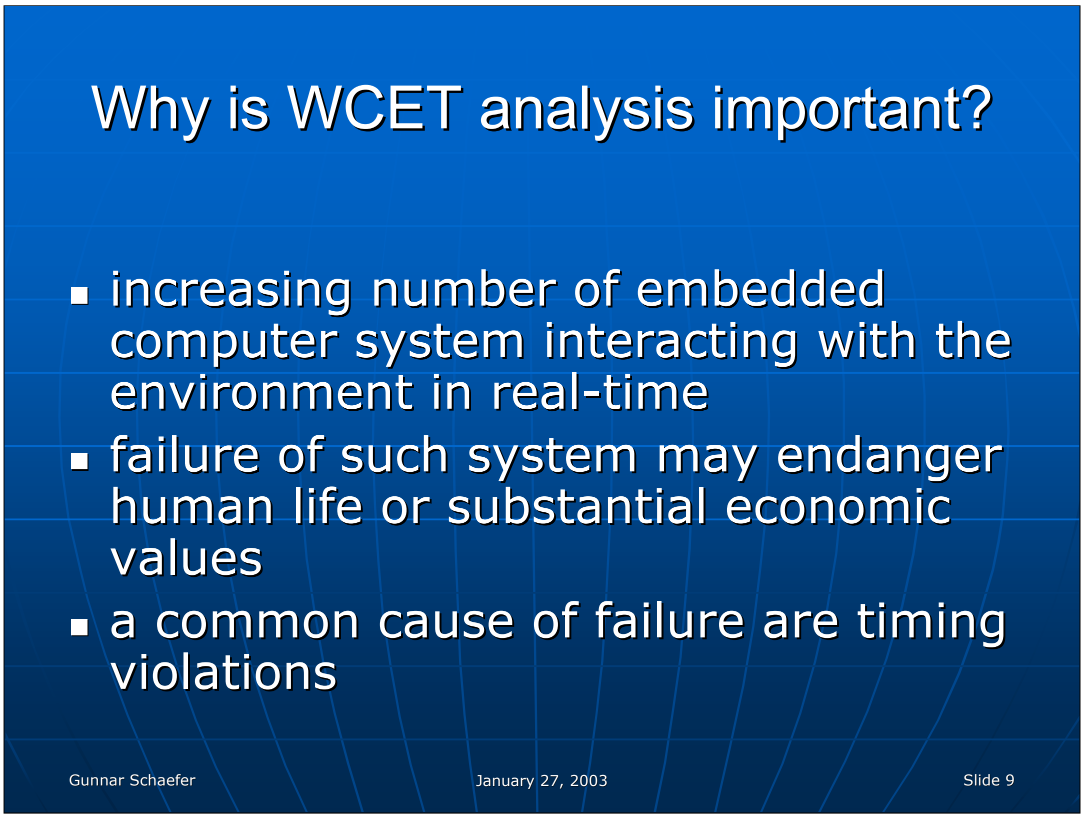
There are C compilers for more architectures than any other programming language.

Ada is typically used for safety relevant applications.



Outline

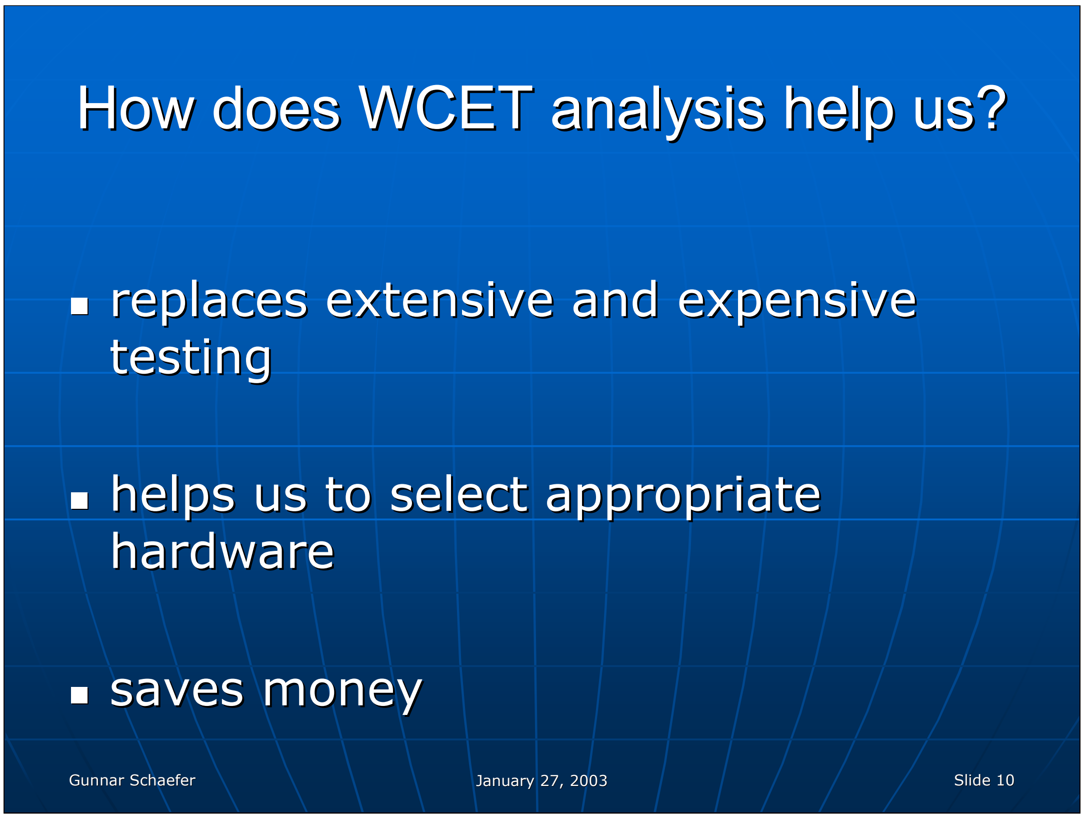
1. Introduction
2. Motivation and Background
3. WCET Analysis Overview



Why is WCET analysis important?

- increasing number of embedded computer system interacting with the environment in real-time
- failure of such system may endanger human life or substantial economic values
- a common cause of failure are timing violations

Gunnar Schaefer January 27, 2003 Slide 9



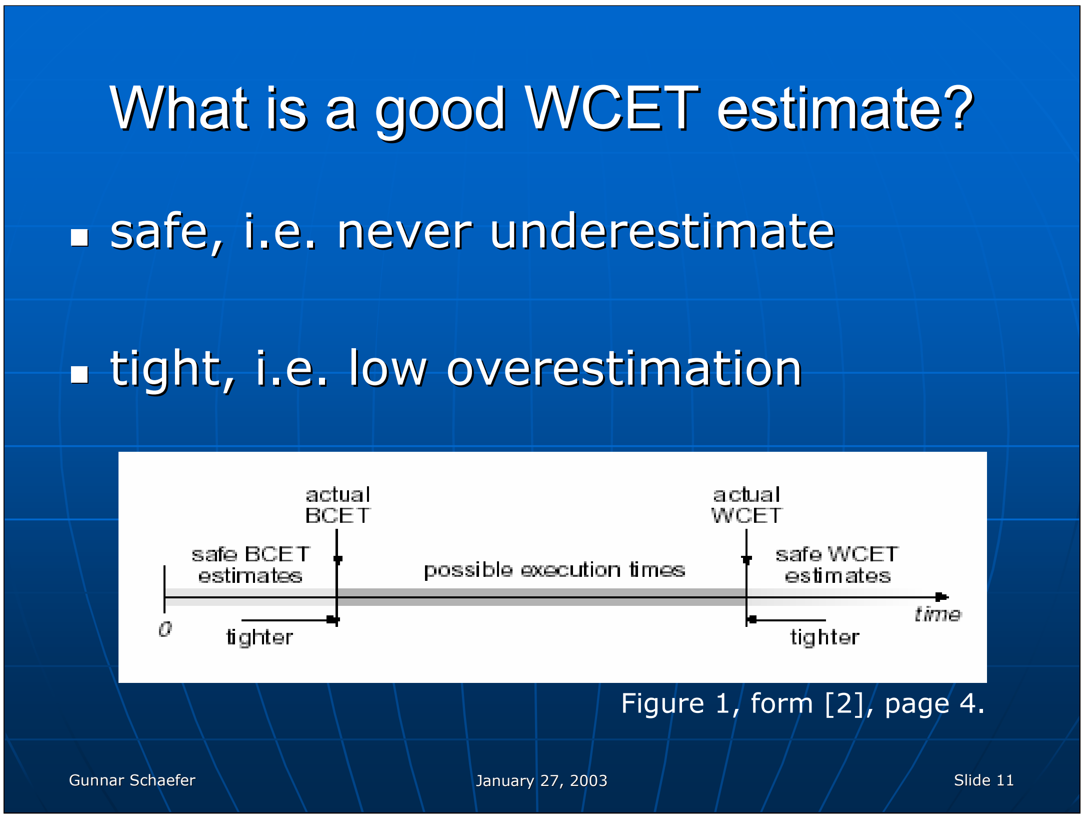
How does WCET analysis help us?

- replaces extensive and expensive testing
- helps us to select appropriate hardware
- saves money

Gunnar Schaefer January 27, 2003 Slide 10

Designing and verifying real-time systems can be much simplified by using WCET analysis instead of extensive and expensive testing. WCET estimates can be used to verify that the response time of a critical piece of code is short enough, that interrupt handlers finish quickly enough, or that the sample rate of a control loop can be kept.

WCET analysis can also be used to assist in selecting appropriate hardware. The designers of a system can take the application code they will use and perform WCET analysis for a range of target systems, selecting the cheapest (slowest) chip that meets the performance requirements.



What is a good WCET estimate?

- safe, i.e. never underestimate
- tight, i.e. low overestimation

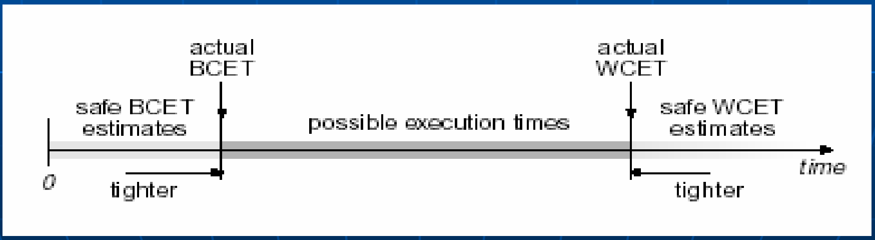
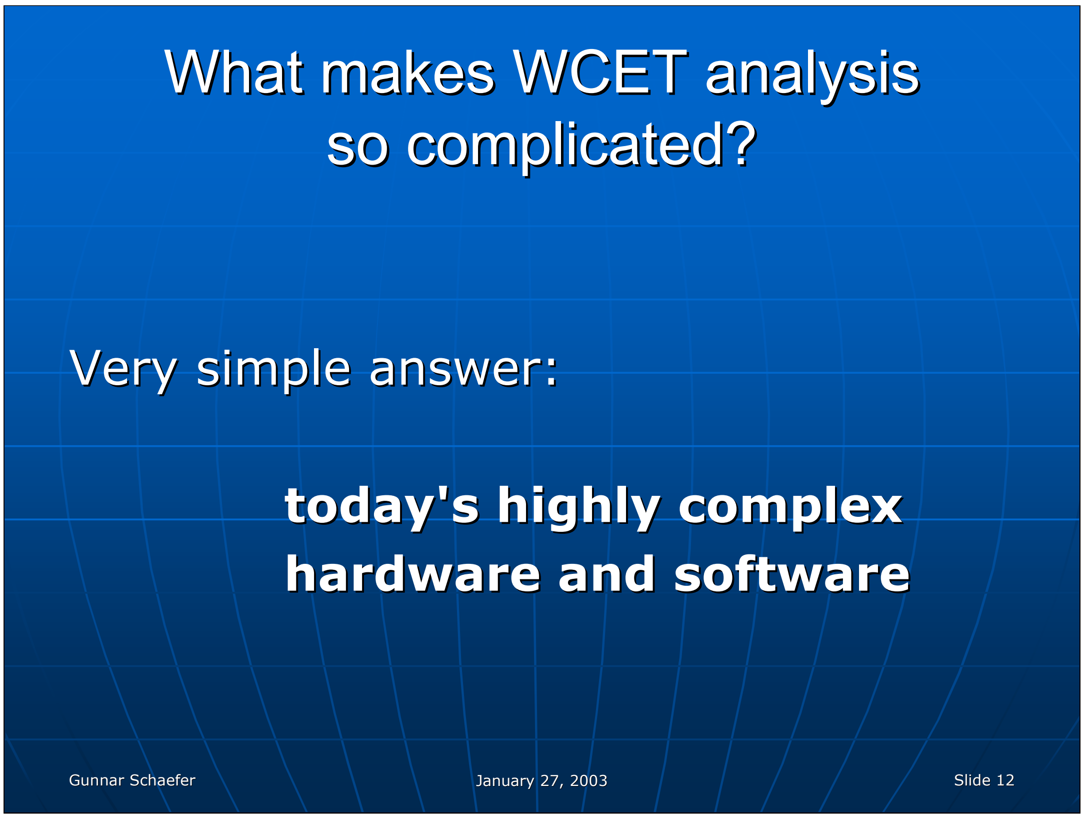


Figure 1, form [2], page 4.

Gunnar Schaefer January 27, 2003 Slide 11

The goal of WCET analysis is to generate a safe (i.e. no underestimation) and tight (i.e. small overestimation) estimate of the worst-case execution time of a program. A related problem is that of finding the Best-Case Execution Time (BCET) of a program. The figure is an illustration of WCET, BCET, tightness, and safe estimates.



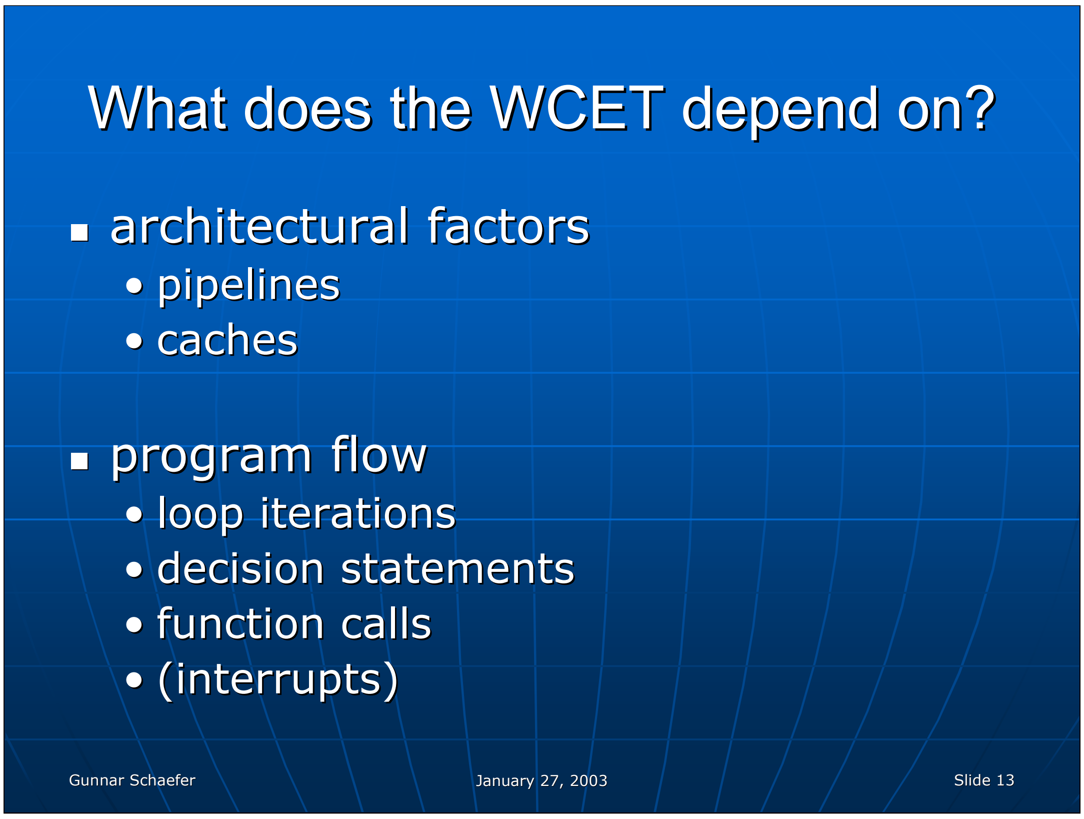
What makes WCET analysis so complicated?

Very simple answer:

today's highly complex hardware and software

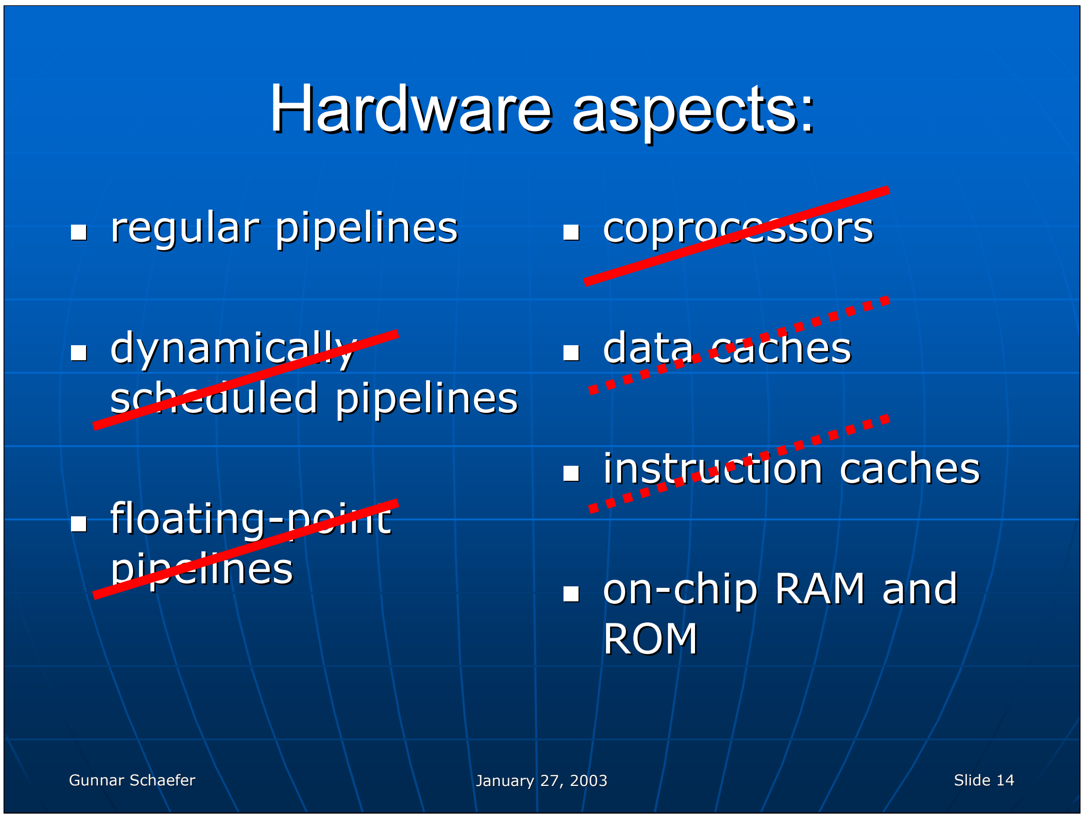
Gunnar Schaefer January 27, 2003 Slide 12

Additionally, worst-case analysis is in general undecidable since it is equivalent to the halting problem. To make the problem decidable, several approximations and generalizations must be made.



What does the WCET depend on?

- architectural factors
  - pipelines
  - caches
- program flow
  - loop iterations
  - decision statements
  - function calls
  - (interrupts)

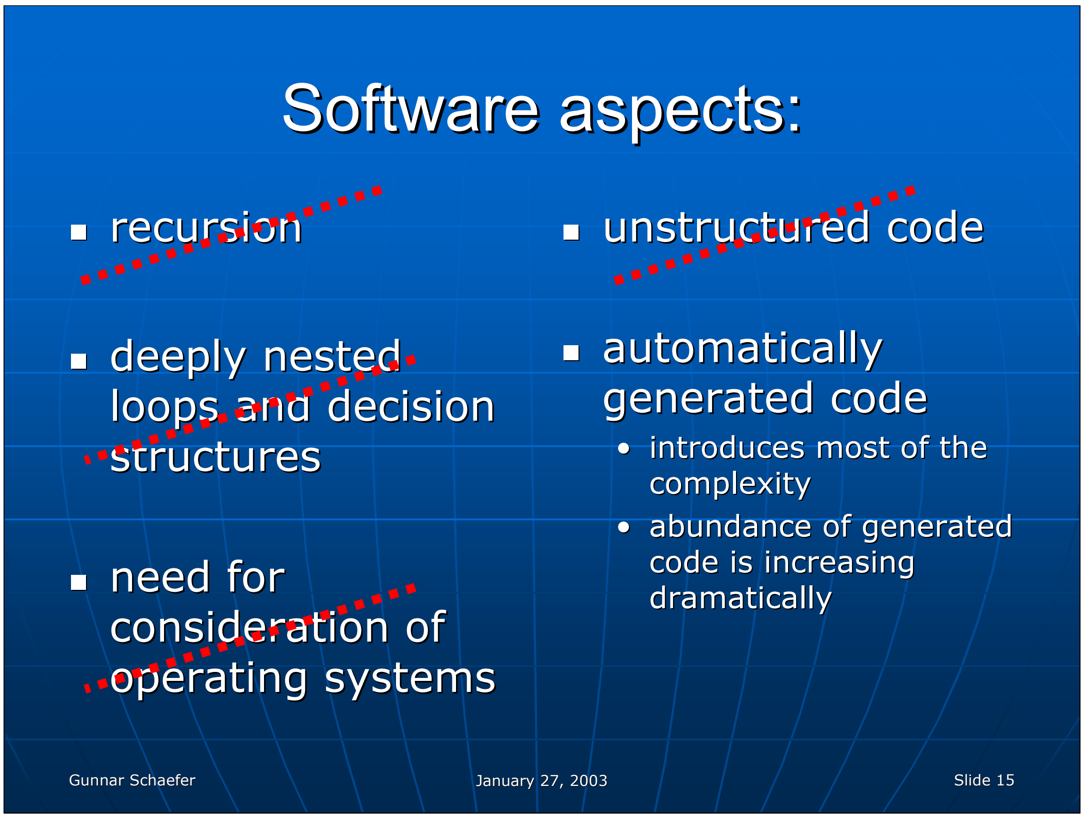


Hardware aspects:

- regular pipelines
- ~~dynamically scheduled pipelines~~
- ~~floating-point pipelines~~
- ~~coprocessors~~
- ~~data caches~~
- ~~instruction caches~~
- on-chip RAM and ROM

Gunnar Schaefer January 27, 2003 Slide 14

With focus on embedded systems, regular pipelines and on-chip RAM and ROM are common, data and instruction caches were rare in the past, but are becoming more and more common, dynamically scheduled pipelines, floating-point pipelines and coprocessors are extremely rare.

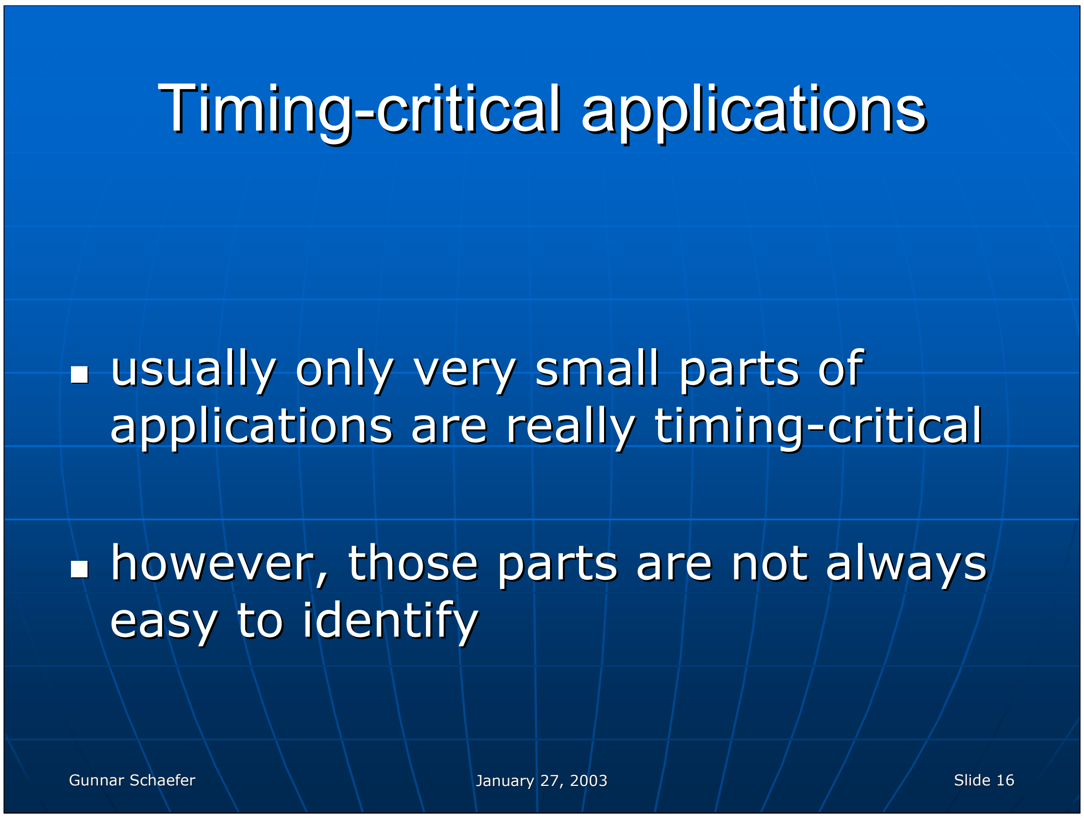


Software aspects:

- ~~recursion~~
- ~~deeply nested loops and decision structures~~
- ~~need for consideration of operating systems~~
- ~~unstructured code~~
- automatically generated code
  - introduces most of the complexity
  - abundance of generated code is increasing dramatically

Gunnar Schaefer January 27, 2003 Slide 15

With focus on embedded software, recursion, deeply nested loops and decision structures and unstructured code are rare. There is also no real need for considering the operating systems. However, automatically generated code introduces a lot of complexity and is becoming dramatically more abundant.

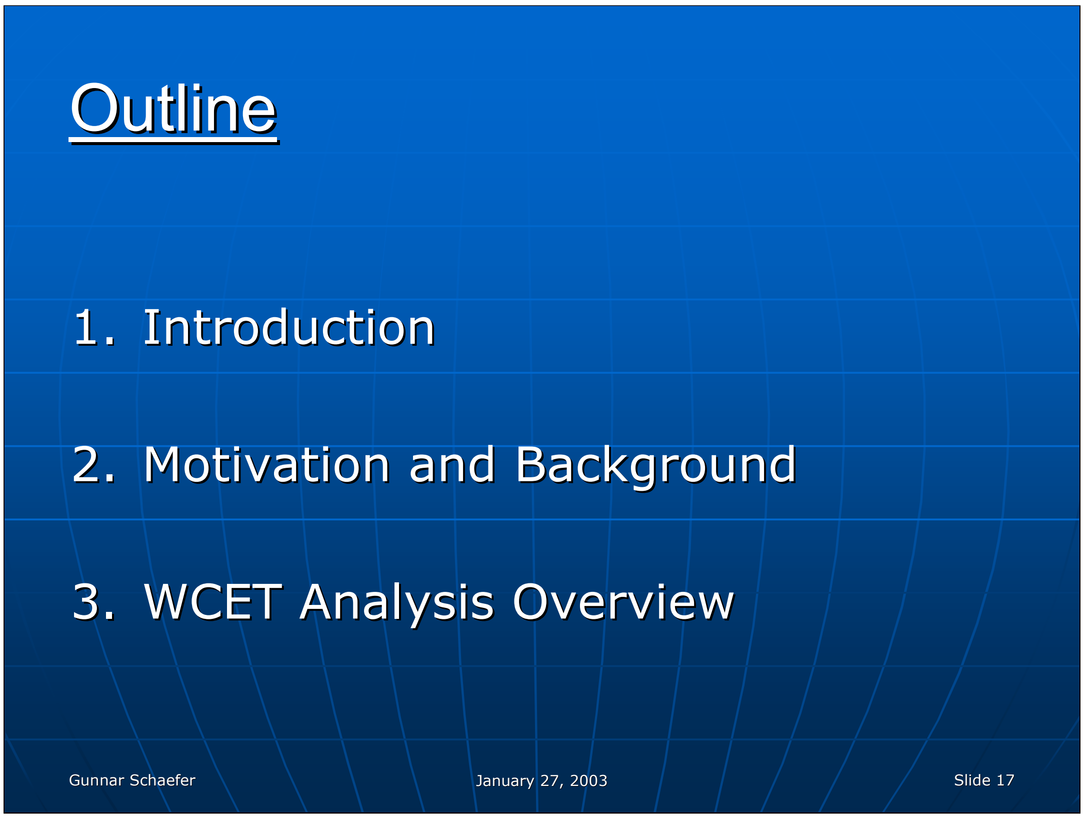


Timing-critical applications

- usually only very small parts of applications are really timing-critical
- however, those parts are not always easy to identify

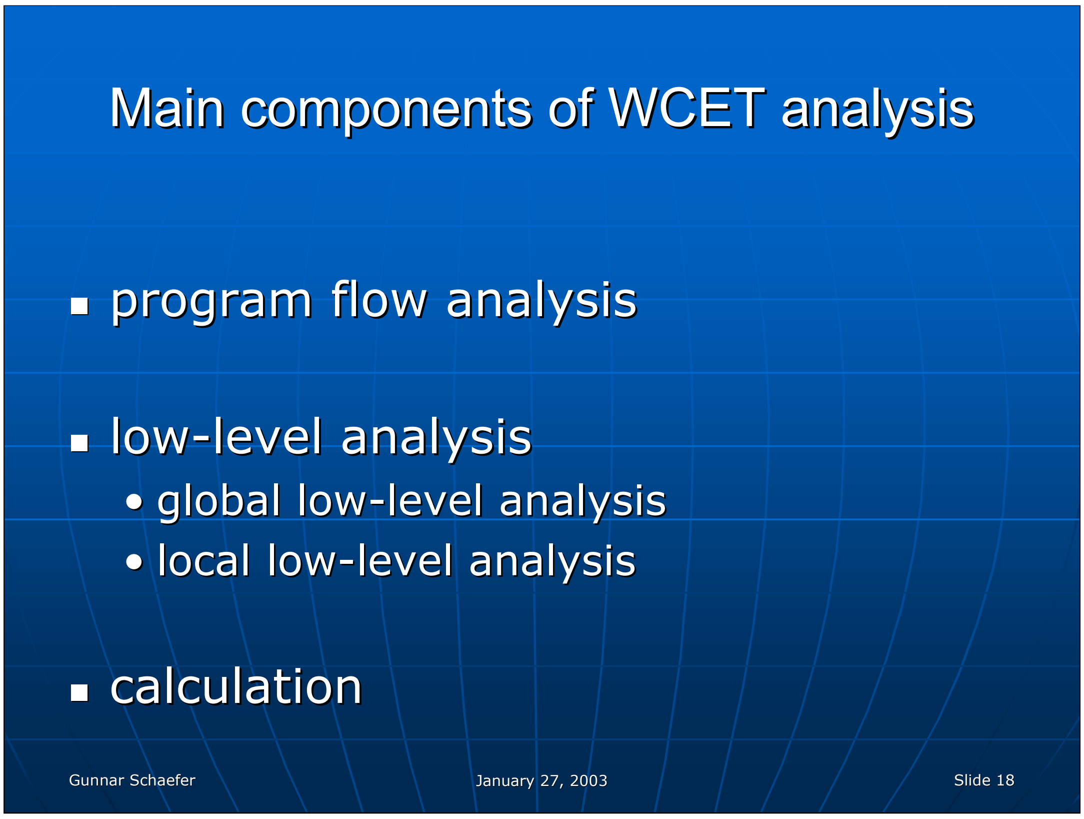
Gunnar Schaefer January 27, 2003 Slide 16

For example, in a GSM mobile phone, the GSM code is very small compared to the non-real-time user interface.



Outline

1. Introduction
2. Motivation and Background
3. WCET Analysis Overview



Main components of WCET analysis

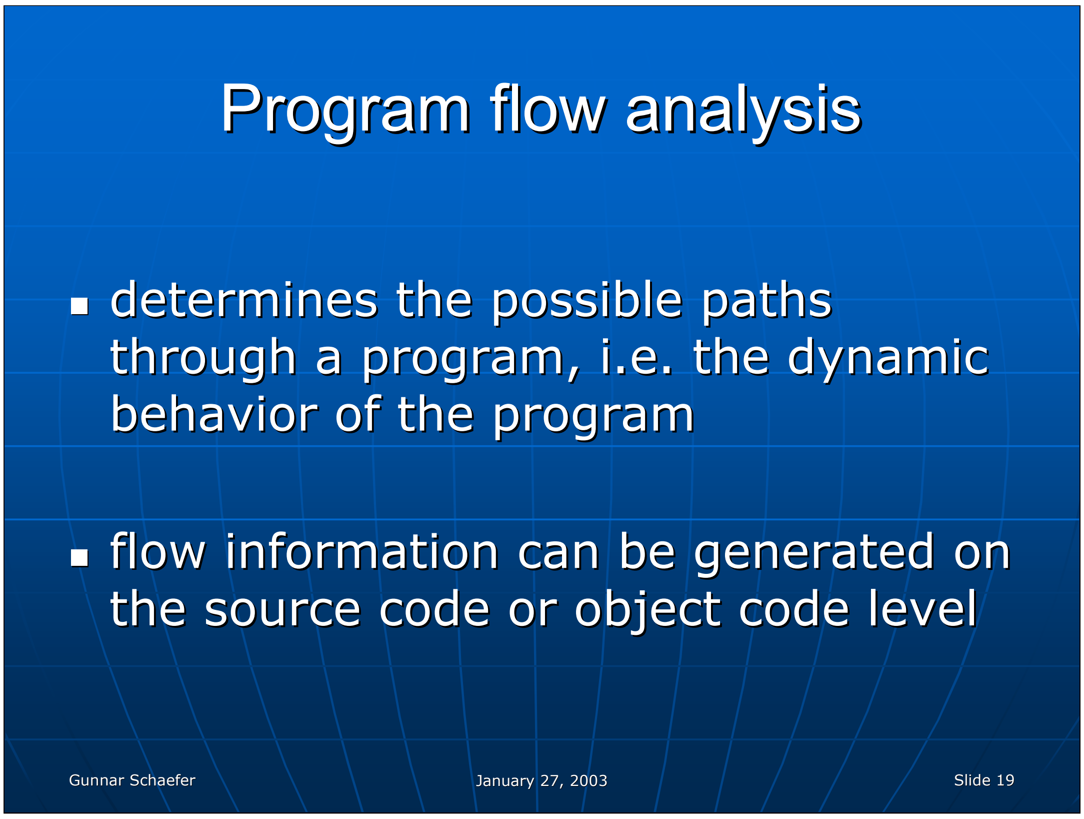
- program flow analysis
- low-level analysis
  - global low-level analysis
  - local low-level analysis
- calculation

Gunnar Schaefer January 27, 2003 Slide 18

The **program flow analysis** calculates the possible flows through the program.

The **low-level analysis** calculates the execution time for the instructions.

The **calculation** combines the above results into a WCET estimate.

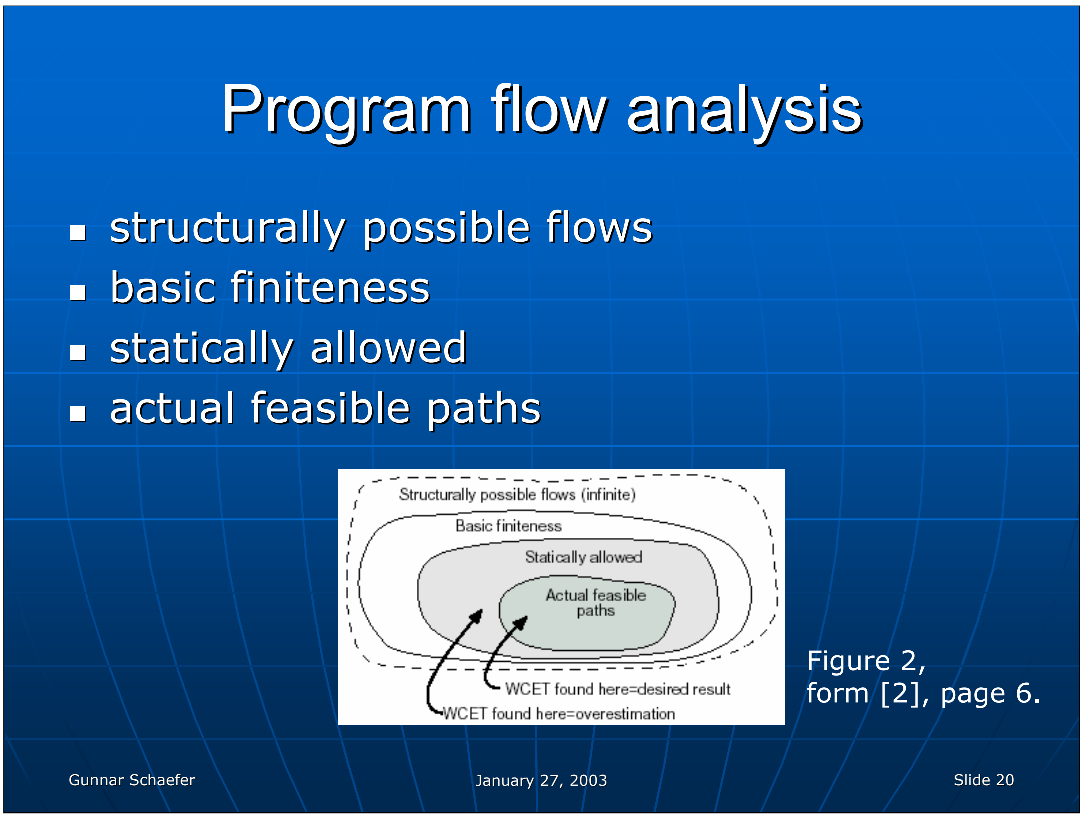


Program flow analysis

- determines the possible paths through a program, i.e. the dynamic behavior of the program
- flow information can be generated on the source code or object code level

Gunnar Schaefer January 27, 2003 Slide 19

Flow analysis provides information about which functions get called, how many times loops iterate, if there are dependencies between different if-statements, etc. Since the problem is computationally intractable in the general case, a simpler, approximate analysis is normally performed. This analysis must yield safe path information, i.e. all feasible paths must always be covered by the approximation.



Program flow analysis

- structurally possible flows
- basic finiteness
- statically allowed
- actual feasible paths

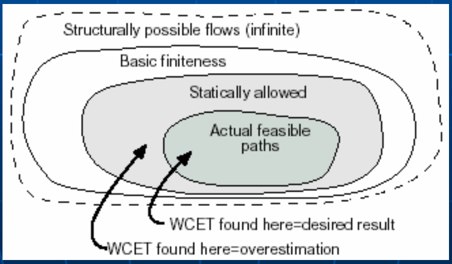
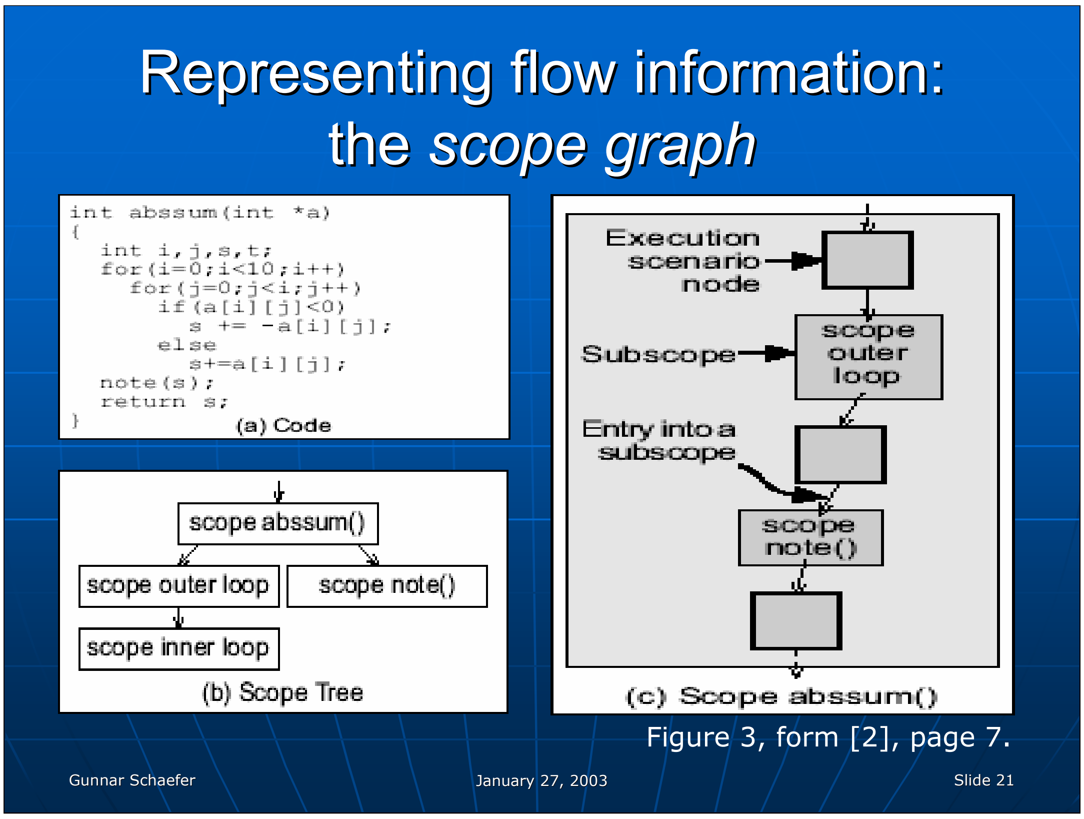

Structurally possible flows (infinite)
Basic finiteness
Statically allowed
Actual feasible paths
WCET found here=desired result
WCET found here=overestimation

Figure 2,
form [2], page 6.

Gunnar Schaefer January 27, 2003 Slide 20

The set of **structurally possible** flows for a program, i.e. those given by the structure of the program, is usually infinite, since, e.g. loops can be taken an arbitrary number of times. The executions are made finite by bounding all loops with some upper limit on the number of executions (**basic finiteness**). Adding even more information, e.g. about the input data, allows the set of executions to be narrowed down further, to a set of **statically allowed** paths. This is the "optimal" outcome of the flow analysis. Figure 2 provides an illustration of the different levels of approximation. Note that the set of **actual feasible paths** might be smaller than the statically allowed paths, due to approximations.



Representing flow information: the *scope graph*

```
int abssum(int *a)
{
    int i,j,s,t;
    for(i=0;i<10;i++)
       for(j=0;j<i;j++)
          if(a[i][j]<0)
             s += -a[i][j];
          else
             s+=a[i][j];
    note(s);
    return s;
}
```

(a) Code

scope abssum()
scope outer loop
scope note()
scope inner loop

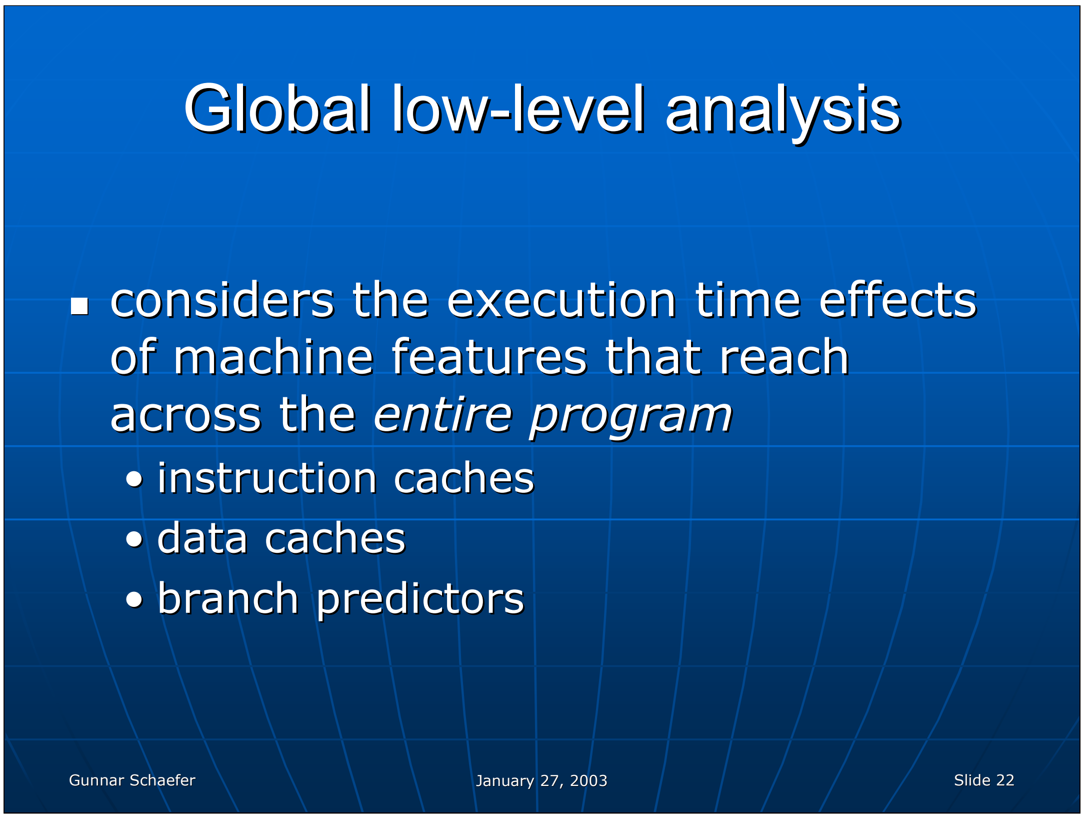
(b) Scope Tree

Execution scenario node
Subscope
scope outer loop
Entry into a subscope
scope note()

(c) Scope abssum()

Figure 3, form [2], page 7.

The **scope graph** is a hierarchical representation of the dynamic structure of a program. Each scope corresponds to a certain repeating or differentiating execution context in the program, e.g. loops and function calls, and describes the execution of the object code of the program within that context. Each scope is assumed to iterate, and has a header node. A maximal number of iterations must be given for each scope, and a new iteration is defined to start each time the header node is passed. Scopes are allowed to only iterate once, i.e. not loop.

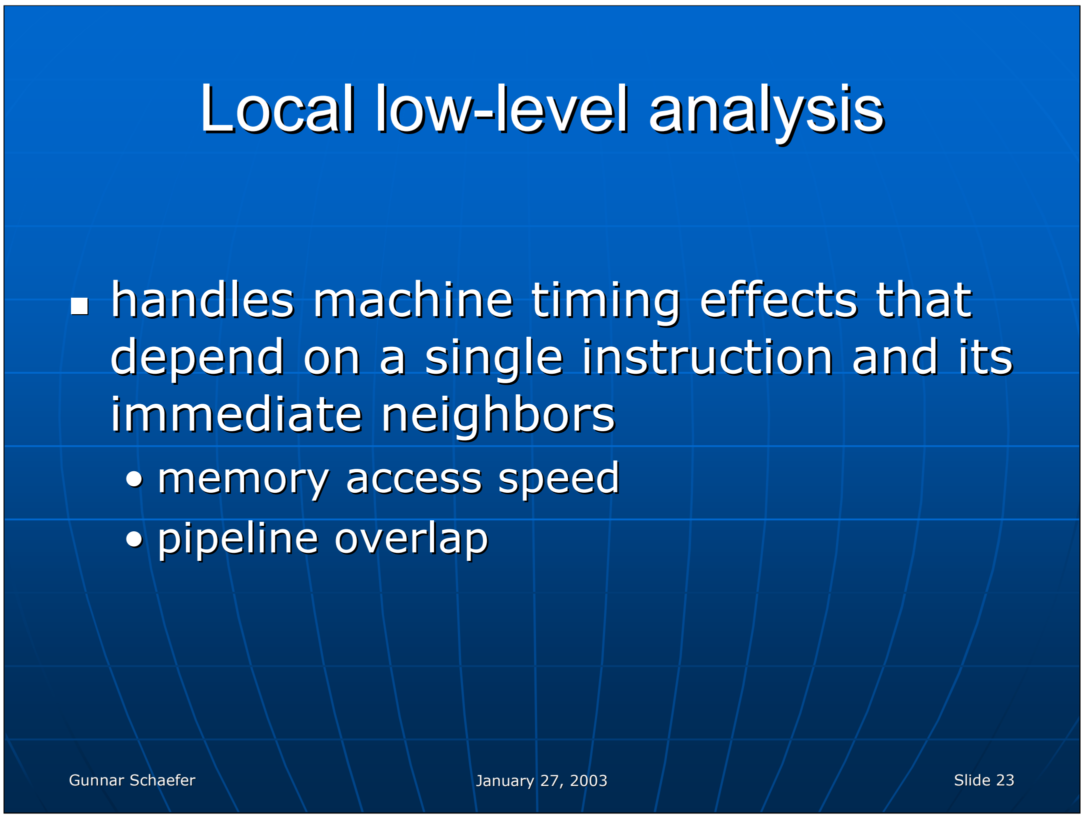


Global low-level analysis

- considers the execution time effects of machine features that reach across the *entire program*
  - instruction caches
  - data caches
  - branch predictors

Gunnar Schaefer January 27, 2003 Slide 22

Since exact analysis is normally impossible, an approximate but safe analysis is necessary. For example, when an attempt is made to determine whether a certain instruction is in the cache, a cache miss is assumed unless we can be absolutely sure of a cache hit. This might be pessimistic but is definitely safe.



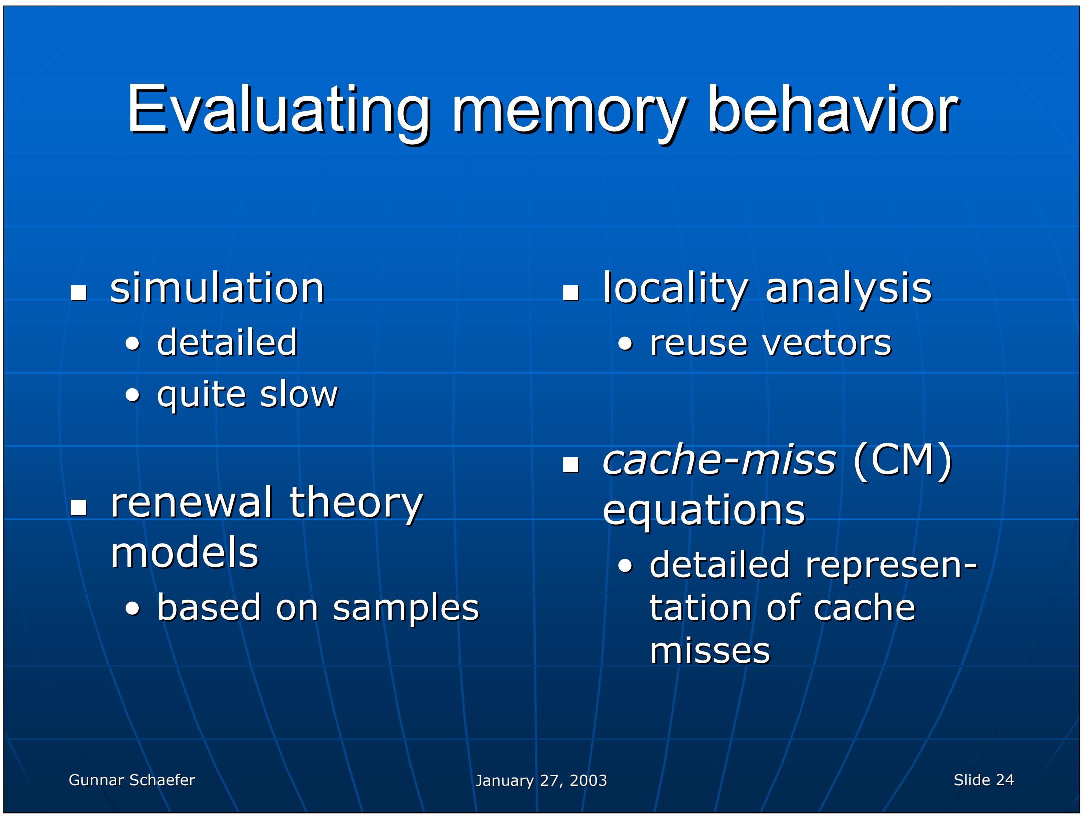
Local low-level analysis

- handles machine timing effects that depend on a single instruction and its immediate neighbors
  - memory access speed
  - pipeline overlap

Gunnar Schaefer January 27, 2003 Slide 23

With processor speeds improving at a much faster rate than memory speeds, the relative importance of memory behavior on program performance has increased significantly in recent years.

On embedded systems, there are usually several different memory areas, each with different timing. On-chip RAM and ROM are fast, while off-chip memory typically takes several extra cycles to access. Pessimistic (approximate but safe) approaches are common, e.g. assuming there is a pipeline speed-up effect only when enough pipeline content information is available to guarantee the effect.



Evaluating memory behavior

- simulation
  - detailed
  - quite slow
- renewal theory models
  - based on samples
- locality analysis
  - reuse vectors
- *cache-miss* (CM) equations
  - detailed represen-tation of cache misses

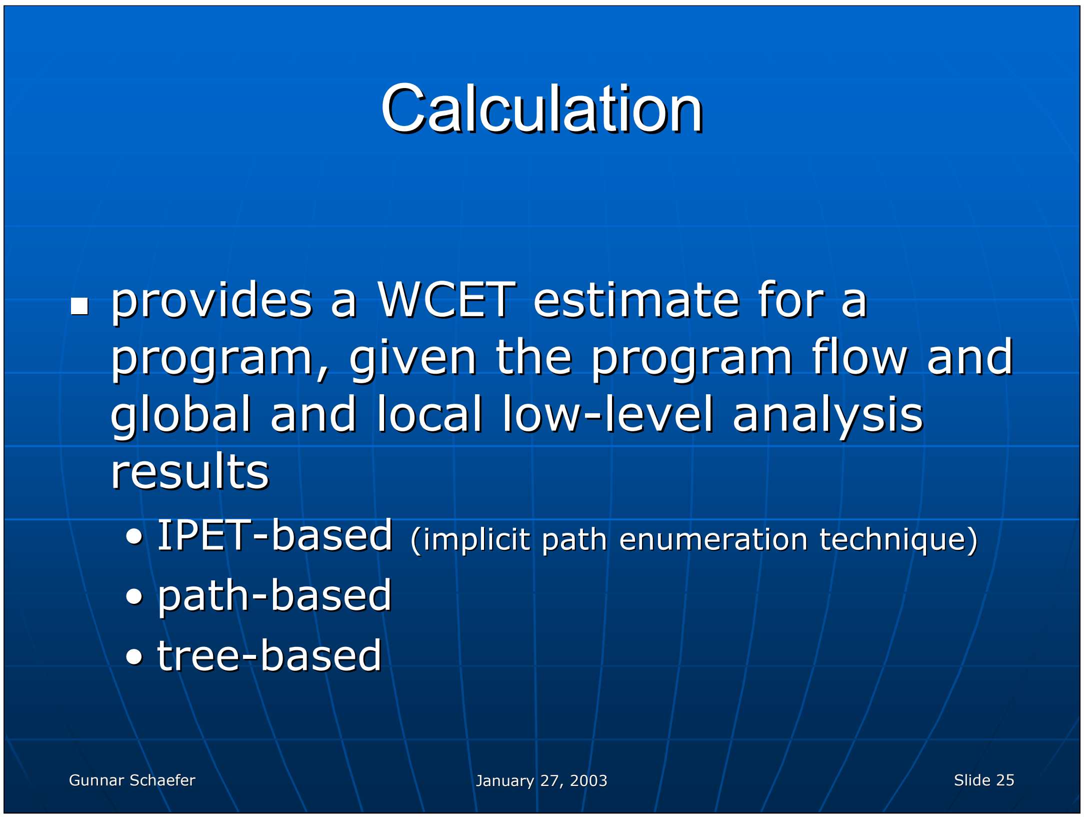
Gunnar Schaefer January 27, 2003 Slide 24

All four methods focus and rely on **loop-oriented scientific code**.

In **renewal theory models** the cache state is unknown at the beginning of each sample. Time is divided into "live time", where a cache-block will be referenced again before it is replaced, and "dead time", where a cache-block will not be referenced again before it is replaced. Then the expected miss rate for each block is equal to the dead time divided by the total time.

Locality analysis relies on computing a set of reuse vectors that summarize how a loop accesses memory locations and cache lines.

CM equations are linear Diophantine equations summarizing each loop's memory behavior. Then each solutions of these equations corresponds to a potential cache miss. Diophantine equations are equations with whole-numbered coefficients, where one is only interested in whole-numbered solutions.



Calculation

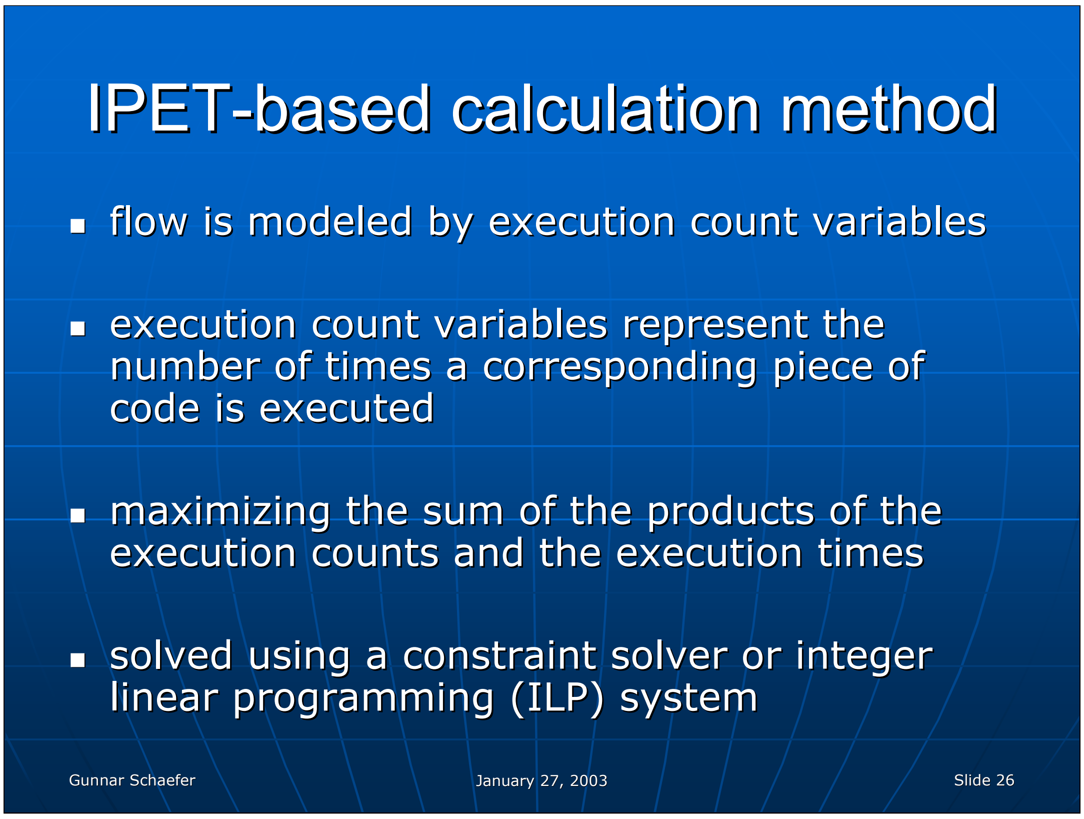
- provides a WCET estimate for a program, given the program flow and global and local low-level analysis results
  - IPET-based (implicit path enumeration technique)
  - path-based
  - tree-based

Gunnar Schaefer January 27, 2003 Slide 25

IPET-based methods express program flow and atomic execution times using algebraic and/or logical constraints. The WCET estimate is calculated by maximizing an objective function, while satisfying all constraints.

In a **path-based** calculation, the final WCET estimate is generated by calculating times for different paths in a program, searching for the path with the longest execution time. The defining feature is that possible execution paths are represented *explicitly*.

In **tree-based** methods, the final WCET is generated by a bottom-up traversal of a tree representing the program. The analysis results for smaller parts of the program are used to make timing estimates for larger parts of the program.

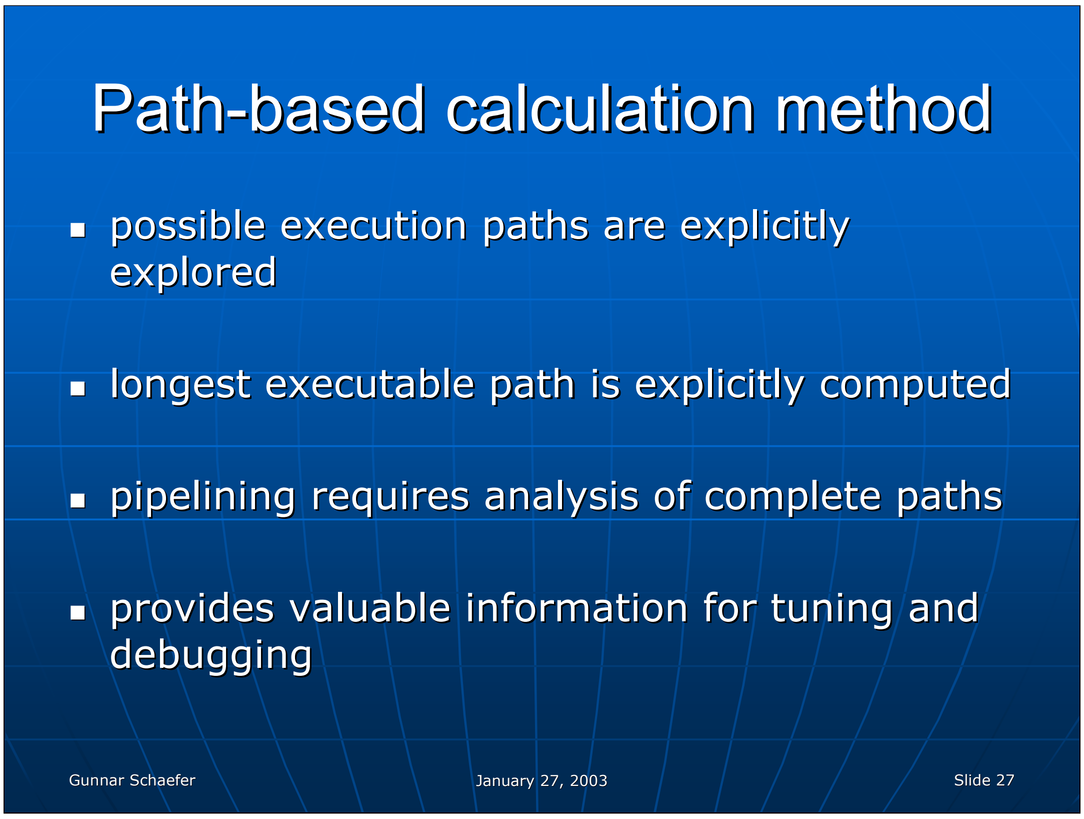


IPET-based calculation method

- flow is modeled by execution count variables
- execution count variables represent the number of times a corresponding piece of code is executed
- maximizing the sum of the products of the execution counts and the execution times
- solved using a constraint solver or integer linear programming (ILP) system

Gunnar Schaefer January 27, 2003 Slide 26

Observe that **IPET** will not explicitly find the worst case execution path, i.e. the precise order in which all nodes are executed, since paths are not explicitly represented. However, the execution counts can be interpreted as an execution count profile of the worst-case execution, which is very useful to identify hot spots and bottlenecks in the program.

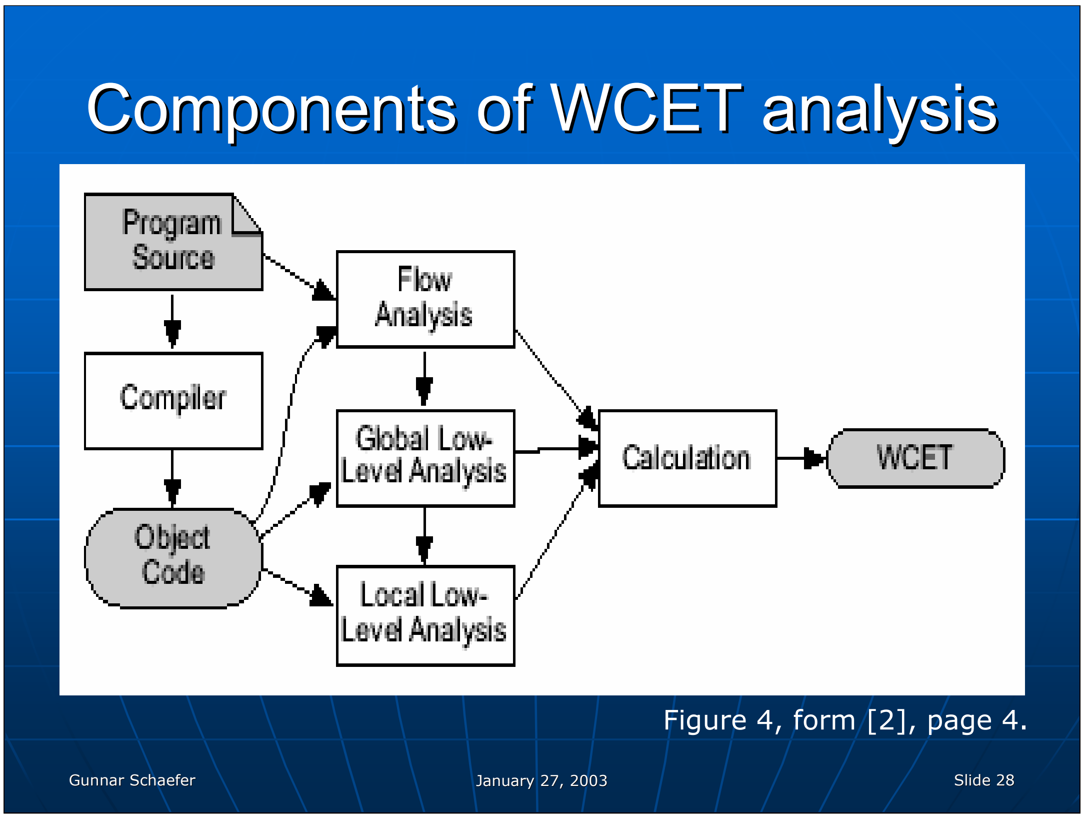


Path-based calculation method

- possible execution paths are explicitly explored
- longest executable path is explicitly computed
- pipelining requires analysis of complete paths
- provides valuable information for tuning and debugging

Gunnar Schaefer January 27, 2003 Slide 27

The need to handle complete paths arises from the use of pipelining in modern processors: to get a tight timing estimate, one must account for the overlap between basic blocks, and this can only be done by analyzing all the basic blocks in a path in a continuous sequence.



Components of WCET analysis

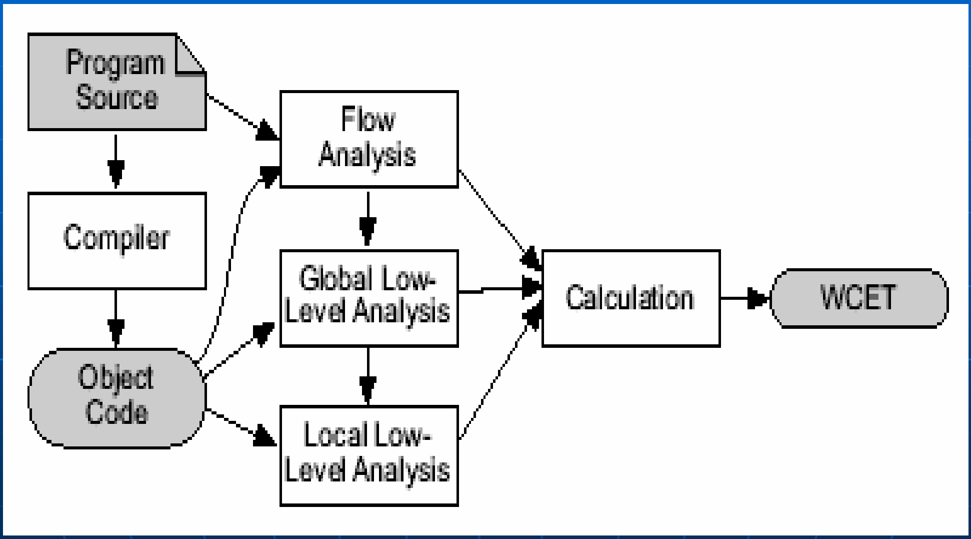
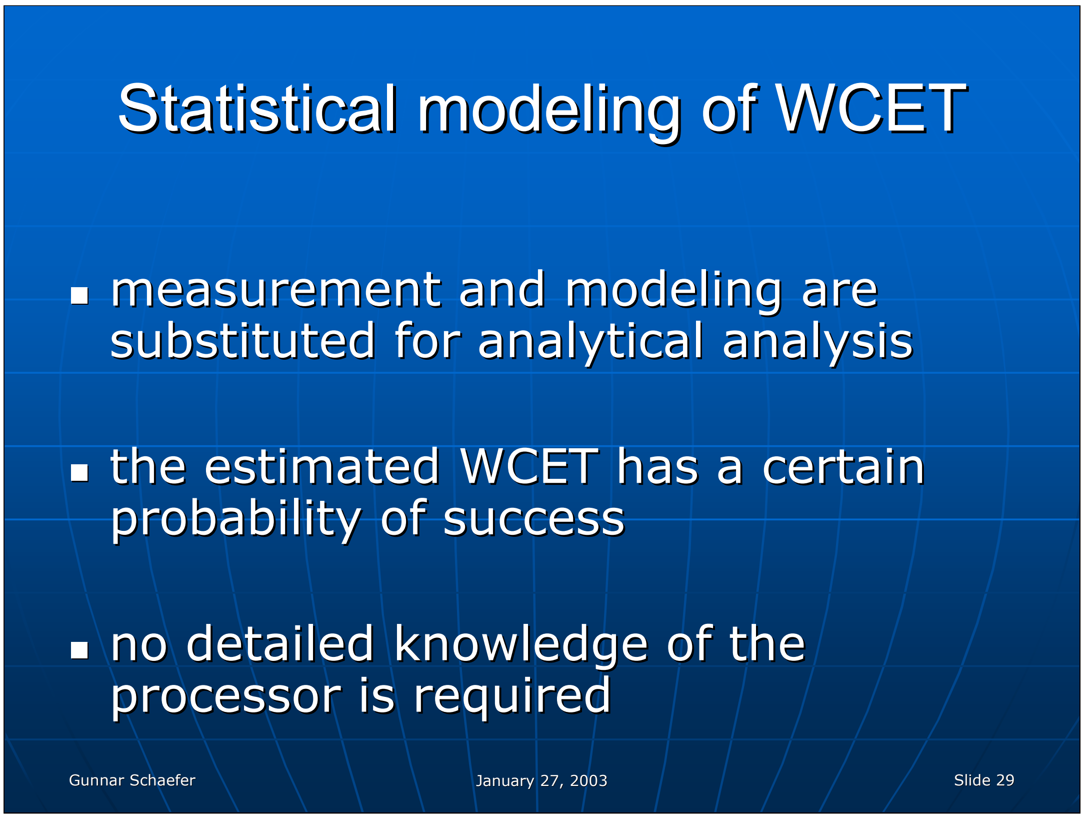


Figure 4, form [2], page 4.

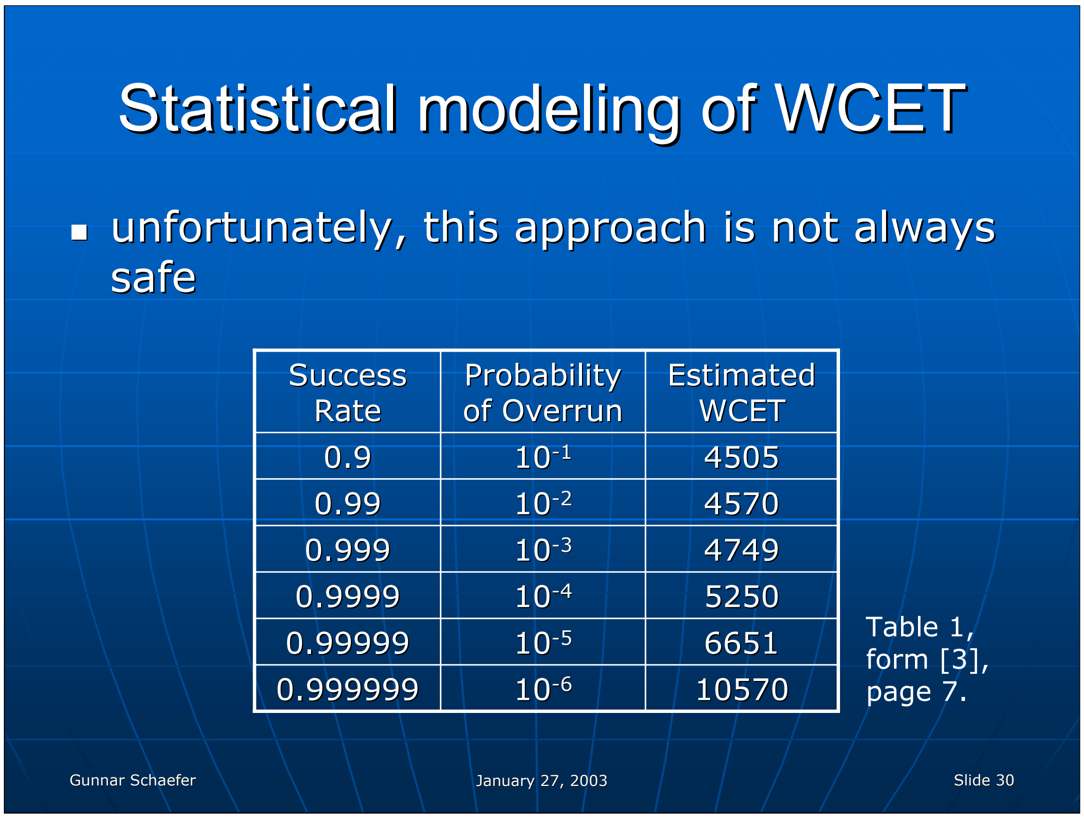


Statistical modeling of WCET

- measurement and modeling are substituted for analytical analysis
- the estimated WCET has a certain probability of success
- no detailed knowledge of the processor is required

Gunnar Schaefer January 27, 2003 Slide 29

There are three major steps when using statistics in any field. Firstly, data has to be obtained. Secondly, a suitable mathematical model has to be found to represent the data. Thirdly, this model is used to generalize the behavior of a system beyond the existing data.

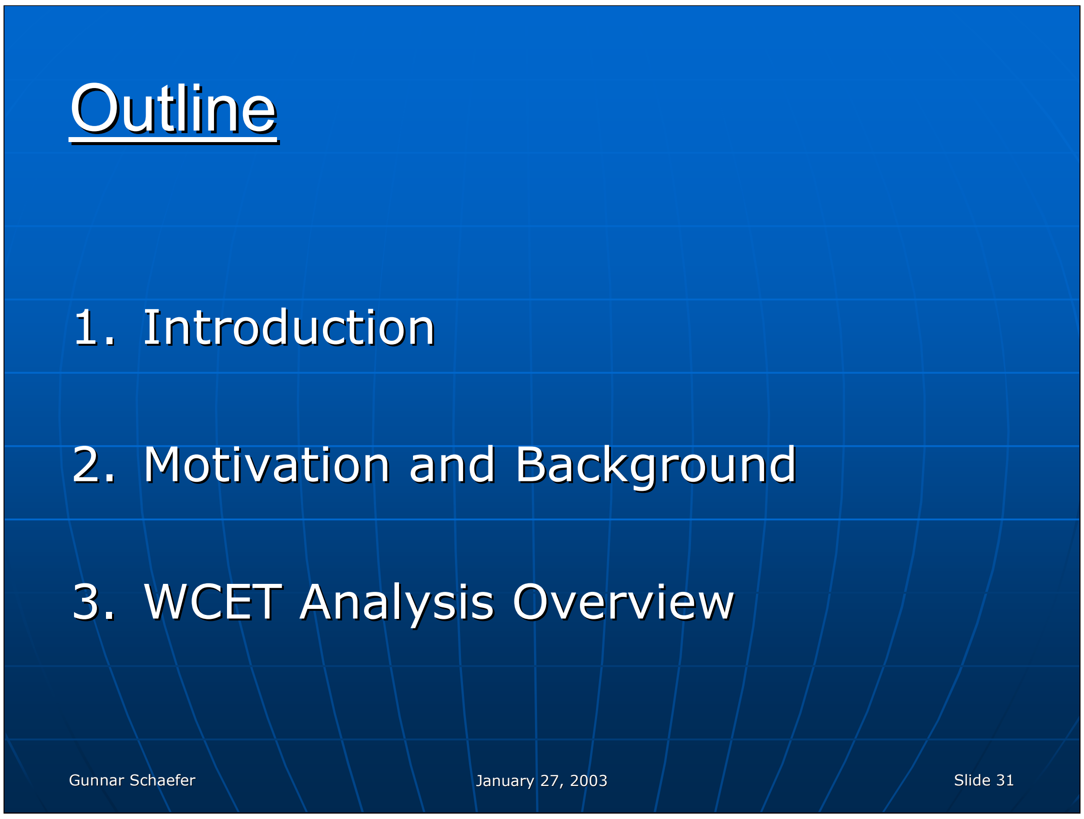


Statistical modeling of WCET

- unfortunately, this approach is not always safe

Success Rate	Probability of Overrun	Estimated WCET
0.9	10^{-1}	4505
0.99	10^{-2}	4570
0.999	10^{-3}	4749
0.9999	10^{-4}	5250
0.99999	10^{-5}	6651
0.999999	10^{-6}	10570

Table 1, form [3], page 7.



Outline

1. Introduction
2. Motivation and Background
3. WCET Analysis Overview