Sistemas de Tempo Real:

Worst-Case Execution Time

Rômulo Silva de Oliveira
Departamento de Automação e Sistemas – DAS – UFSC

Romulo.deoliveira@ufsc.br
http://www.das.ufsc.br/~romulo
Outubro/2013

Reference

- The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools
  - ACM Transactions on Embedded Computing Systems, Vol. 7, No. 3, Article 36, Publication date: April 2008.
  - REINHARD WILHELM TULIKA MITRA
  - JAKOB ENGBLOM FRANK MUELLER
  - ANDREAS ERMEDAHL ISABELLE PUAUT
  - NIKLAS HOLSTI PETER PUSCHNER
  - STEPHAN THESING JAN STASCHULAT
  - DAVID WHALLEY
  - PER STENSTRÖM
  - GUILLEM BERNAT
  - CHRISTIAN FERDINAND
  - REINHOLD HECKMANN

Summary

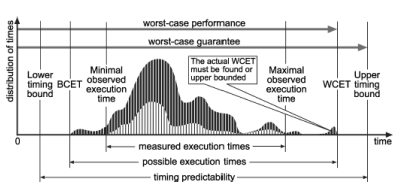
- Introduction
- Problems and Requirements
- Classification of Approaches
- Auxiliary Methods
- Static Methods
- Measurement-Based Methods
- Comparison of Static and Measurement-Based Methods
- Commercial Tools and Research Prototypes
- Experience
- Tools Comparison
- Remaining Problems and Future Perspectives
- Architectural Trends

Introduction 1/7

- Hard real-time systems need to satisfy stringent timing constraints
  - Derived from the systems they control
- Upper bounds on the execution times are needed
  - To show the satisfaction of these constraints
- It is not possible, in general, to obtain upper bounds on execution times for programs
  - Otherwise, one could solve the halting problem
- Real-time systems only use a restricted form of programming which guarantees that programs always terminate
  - Recursion is not allowed or explicitly bounded
  - Same for iteration counts of loops
- In general, the worst-case scenario is not known and hard to derive

Introduction 2/7

- A real-time system consists of a number of tasks
- A task typically shows a certain variation of execution times depending on the input data or different behavior of the environment
- In most cases, the state space is too large to exhaustively explore all possible executions



Introduction 3/7

- Today, in most parts of industry, the common method to estimate execution-time bounds is to measure the *end-to-end* execution time of the task for a subset of the possible executions – **test cases**
- This determines the *minimal observed* and *maximal observed execution times*
- This method is often called *dynamic timing analysis*
- These will, in general, overestimate the BCET and underestimate the WCET
  - It is not safe for hard real-time systems

Introduction 4/7

- Newer measurement-based approaches make more detailed measurements of the execution time

  of different *parts* of the task and combine them
  to give better estimates of the BCET and WCET for the whole task

- Still, these methods are rarely guaranteed to give bounds on the execution time

Introduction 5/7

- Bounds on the execution time of a task can be computed only by methods that consider all possible execution times
- These methods use an abstraction of the task
  to make timing analysis of the task feasible
- Abstraction loses information
  - The computed WCET bound usually overestimates the exact WCET and vice versa for the BCET
- The WCET bound represents the worst-case guarantee the method or tool can give
- How much is lost depends both on the methods used for timing analysis and on overall system properties (Timing Predictability)
  - hardware architecture
  - characteristics of the software

Introduction 6/7

- Criteria for evaluating a method or tool for timing analysis
- Safety
  - does it produce bounds or estimates?
- Precision
  - are the bounds or estimates close to the exact values?
- Performance prediction is also required for application domains that do not have hard real-time characteristics
  - Systems with soft deadlines
  - Probabilistic estimates

Introduction 7/7

- Timing Analysis
  - The process of deriving execution-time bounds or estimates
- Timing-Analysis Tool
  - A tool that derives bounds or estimates for the execution times of application tasks
- Most tools offer timing analysis of tasks in uninterrupted execution
- A task may be a unit of scheduling by an operating system, a subroutine, or some other software unit
  - This unit is mostly available as a fully-linked executable
  - Some tools assume the availability of source code

Summary

- Introduction
- Problems and Requirements
- Classification of Approaches
- Auxiliary Methods
- Static Methods
- Measurement-Based Methods
- Comparison of Static and Measurement-Based Methods
- Commercial Tools and Research Prototypes
- Experience
- Tools Comparison
- Remaining Problems and Future Perspectives
- Architectural Trends

Problems and Requirements

- Timing analysis attempts to determine bounds on the execution times of **a task** when executed on **a particular hardware**
- The time for a particular execution depends on
  - the path through the task taken by control and
  - the time spent in the statements or instructions on this path on this hardware
- The determination of execution-time bounds has to consider
  - the potential control-flow paths and
  - the execution times for this set of paths
- A modular approach to the timing-analysis problem splits the overall problem into a sequence of sub-problems
  - Some of them deal with properties of the control flow and
  - others with the execution time of instructions or sequences of instructions on the given hardware

Prob&Req: Data-Dependent Control Flow 1/5

- The task attains its WCET on one (or sometimes several) of its possible execution paths
- If the input and the initial state leading to the execution of this worst-case path were known
  - The problem would be easy to solve
  - The task would be started in this initial state with this input, and the execution time would be measured
- In general, this worst-case input and initial state are not known and hard or impossible to determine
- A data structure, the task's **control-flow graph (CFG)**, describes a superset of the set of all execution paths
  - The task's call graph usually is integrated into the CFG

Prob&Req: Data-Dependent Control Flow 2/5

- First step: the construction of the CFG and call graph of the task
  - from a source or a machine-code version of the task
- They must contain all of the instructions of the task under analysis
- Problems are created by dynamic jumps and calls with computed target address
- Dynamic jumps are mainly because of switch/case structures
  - They are a problem only when analyzing machine code, because even assembly code usually labels all switch/case branches
- Dynamic calls also occur in source code in the form of calls through function pointers and calls to virtual functions
- A component of a timing-analysis tool, which reconstructs the CFG from a machine program, is often called a **front end**

Prob&Req: Data-Dependent Control Flow 3/5

- Different paths through the CFG are taken depending directly or indirectly on input data
- Some paths in the superset described by the CFG will never be taken
  - For example, contradictory consecutive conditions
- Eliminating such paths may increase the precision of timing analysis

Prob&Req: Data-Dependent Control Flow 4/5

- A phase called ***control-flow analysis*** **(CFA)** determines information about the possible flow of control through the task to increase the precision of the subsequent analyzes
  - It excludes infeasible paths
  - It determines execution frequencies of paths
  - Tasks spend most of their execution time in loops and in (recursive) functions
- It is an essential task of CFA to determine bounds on the iterations of loops and on the depth of recursion of functions
- Necessary ingredients:
  - Values of variables, registers, or memory cells occurring in conditions tested for termination of loops or recursion

Prob&Req: Data-Dependent Control Flow 5/5

- Complex processors may actually execute an instruction stream in a different order than the one determined by CFA
- This is because of
  - Pipelining (prefetching and delayed branching)
  - Branch prediction
  - Speculative or out-of-order execution

Prob&Req: Context Dependence of Execution Times 1/2

- Early approaches to the timing-analysis problem assumed context independence of the timing behavior
- The execution times for individual instructions were independent from the execution history
  - They could be found in the manual of the processor
- Structure-based approach [Shaw 1989]
- If a task first executes a code snippet "A" and then a snippet "B"
  Worst-case bound for "A" is ub(A)
  Worst-case bound for "B" is ub(B)
  - the worst-case bound for "A;B" is ub(A;B) = ub(A) + ub(B)

Prob&Req: Context Dependence of Execution Times 2/2

- This context independence is no longer true for modern processors
  - Caches and pipelines
- The execution time of individual instructions may vary
  - By several orders of magnitude
  - Depending on the state of the processor in which they are executed
- The execution time of B can heavily depend on the execution state that the execution of A produced
  - Any tool should exploit the knowledge that A was executed before B to determine a precise upper bound for B in the context A
  - ub(A;B) = ub(A) + ub(B) ignores this information and gets imprecise results
- A phase called ***processor-behavior analysis*** gathers information on the processor behavior for the given task
  - The behavior of the components that influence the execution times
  - Memory, caches, pipelines and branch prediction

Prob&Req: Timing Anomalies 1/8

- The complexity of the processor-behavior analysis and the set of applicable methods critically depend on the complexity of the processor architecture
- Most powerful microprocessors suffer from ***timing anomalies***
  - Anomalies are contraintuitive influences of the execution time of one instruction on the (global) execution time of the whole task
- Not all input data are known
  - Parts of the execution state are missing in the analysis
  - Unknown parts of the state lead to nondeterministic behavior

Prob&Req: Timing Anomalies 2/8

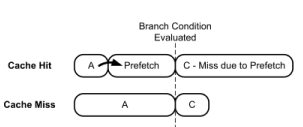
- For example, missing information about whether the next instruction will be in the cache may lead to
  - One execution starting with a cache load contributing the cache-miss penalty to the execution time
  - While another execution will start with an instruction fetch from the cache
- Intuition suggests that the latter execution would always lead to the shorter execution time of the whole task
- On processors with timing anomalies, however, this need not be true
- The latter execution may lead to a longer task execution time
- This was observed on the Motorola ColdFire 5307 processor

Prob&Req: Timing Anomalies 3/8

- The reason is that this processor speculates on the outcome of conditional branches
  - It prefetches instructions in one of the directions of the conditional branch
- When the condition is finally evaluated it may turn out that the processor speculated in the wrong direction
- All the effects produced so far have to be undone
- In addition, fetching the wrong instructions has partly ruined the cache contents
- Taken together, the costs of the misprediction exceed the costs of a cache miss
- Hence, the local worst case, the I-cache miss, leads to the globally shorter execution time
  - since it prevents a more expensive branch misprediction

Prob&Req: Timing Anomalies 4/8

- This exemplifies one of the reasons for timing anomalies, ***speculation-caused anomalies***

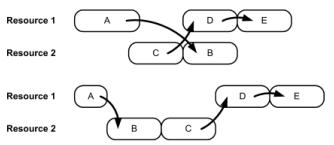


Prob&Req: Timing Anomalies 5/8

- Another type of timing anomalies are ***scheduling anomalies***
- These occur when a sequence of instructions, partly depending on each other, can be scheduled differently on the hardware resources, such as pipeline units
- Depending on the selected schedule, the execution of the instructions or pipeline phases takes different times

Prob&Req: Timing Anomalies 6/8

- Example of a scheduling-caused timing anomaly



Prob&Req: Timing Anomalies 7/8

- Timing anomalies violate an intuitive, but incorrect assumption:
  - That always taking the local worst-case transition when there is a choice produces the global worst-case execution time
- The analysis cannot greedily limit its search for upper bounds by choosing the worst cases for each instruction
- The existence of timing anomalies in a processor influences the applicability of methods for timing analysis for that processor

- The assumption that only local worst cases have to be considered to safely determine upper bounds on global execution times is unsafe
- The assumption that one could identify a worst initial execution state, to safely start measurement or analysis of a piece of code in, **is unsafe**

Prob&Req: Timing Anomalies 8/8

- The consequences for timing analysis of systems to be executed on processors with timing anomalies are as follows:

- The analysis may be forced to follow execution through several successor states, whenever it encounters an abstract state with a nondeterministic choice between successor states
  - This may lead to a quite large state space to consider

- The analysis has to be able to express the absence of state information instead of assuming some worst initial state
  - Absent information in abstract states stands for all potential concrete instances of these missing state components
  - This in order to not wrongly exclude any possible execution

Summary

- Introduction
- Problems and Requirements
- Classification of Approaches
- Auxiliary Methods
- Static Methods
- Measurement-Based Methods
- Comparison of Static and Measurement-Based Methods
- Commercial Tools and Research Prototypes
- Experience
- Tools Comparison
- Remaining Problems and Future Perspectives
- Architectural Trends

Classification of Approaches

- *Static methods*
  - Do not rely on executing code on real hardware or on a simulator
    - Take the task code itself, maybe together with some annotations,
    - analyze the set of possible control-flow paths through the task,
    - combine control flow with some (abstract) model of the hardware architecture,
    - and obtain upper bounds for this combination
  - Static methods emphasize *safety* by producing bounds on the execution time

- *Measurement-based methods*
  - Execute the task or task part on the hardware or a simulator for some set of inputs
  - Take the measured times and derive the maximal and minimal observed execution times
    or their distribution
    or combine the measured times of code snippets with those for the whole task

Summary

- Introduction
- Problems and Requirements
- Classification of Approaches
- Auxiliary Methods
- Static Methods
- Measurement-Based Methods
- Comparison of Static and Measurement-Based Methods
- Commercial Tools and Research Prototypes
- Experience
- Tools Comparison
- Remaining Problems and Future Perspectives
- Architectural Trends

Auxiliary Methods: Static Program Analysis

- Static program analysis is a generic method to determine properties of the dynamic behavior of a given task without actually executing the task
  - These properties are often undecidable
  - Sound approximations are used
  - They have to be correct, but may not necessarily be complete
- Consider **instruction-cache analysis**
  - Attempts to determine for each point in the task which instructions will be in the cache every time execution reaches this program point
  - For straight-line programs and known initial cache contents, this is easy and can be done by a standard simulator
  - However, it is, in general, undecidable for tasks whose control flow depends on input data
- A sound analysis will compute a subset of the instructions that will definitely be in the instruction cache every time execution reaches the program point
- More instructions may actually be in the cache, but the analysis may not be able to find this out

Auxiliary Methods: Measurement

- End-to-end measurements of a subset of all possible executions produce estimates
  - Not bounds
- They may be useful for applications that do not require guarantees
  - Typically nonhard real-time systems
- They may give the developer a feeling about the execution time in common cases
  - and the likelihood of the occurrence of the worst case
- Measurement can also be applied to code snippets
  - after which the results are combined to estimates for the whole program
  - in similar ways as used in static methods
- Guarantees that safe bounds are obtained from measurement can currently only be given for rather simple architectures

Auxiliary Methods: Simulation

- Used to estimate the execution time for tasks on hardware architectures
- It is possible to derive rather accurate estimations of the execution time
  - for a task
  - for a given set of input data
  - assuming sufficient detail of the timing model of the architectural simulator
- Not all simulators can be trusted as **clock-cycle accurate simulators**
  - Results may show large differences in timing compared to the measured values
- Standard cycle-accurate simulators cannot be used off-hand in static methods for timing analysis
  - Static methods should not simulate execution for particular input data, but rather for all input data
- Input data is assumed to be unknown
  - Unknown input data leads to unknown parts in the execution state of the processor
  - And nondeterministic decisions at controlflow branches
- Simulators modified to cope with these problems are being used in several tools

Auxiliary Methods: Abstract Processor Models 1/2

- Processor-behavior analysis needs a model of the architecture
- This need not be a concrete model implementing all of the functionality of the target hardware
- A simplified model that is conservative with respect to the timing behavior is sufficient
- The construction of an abstract processor model is done at **tool-construction time**
- This approach relies on the timing accuracy of the model

Auxiliary Methods: Abstract Processor Models 2/2

- In general, computer vendors do not disclose enough information about the microarchitecture
  - Necessary to develop and safely validate the accuracy of a timing model
- Validation could be done by measurements
  - Measured execution times are compared against predicted bounds
- Another method is trace validation checking whether externally observable traces are projections of traces as predicted by the model
  - Not all events predicted by the model are externally observable
- Both methods are similar to testing
  - They can discover the presence of errors, but not prove their absence

Auxiliary Methods: Integer Linear Programming (ILP) 1/2

- *Linear programming* is a generic methodology to code the requirements of a system in the form of a system of linear constraints
  - A goal function has to be maximized or minimized to obtain an optimal assignment of values to the system's variables
- *Integer linear programming* if these values are required to be integers
- ILP is NP-hard
- The use of ILP should be restricted to small problem instances or to subproblems of timing analysis
  - Generating only small problem instances

Auxiliary Methods: Integer Linear Programming (ILP) 2/2

- The control flow of tasks is translated into integer linear programs
- Extra information about the control flow can often be coded as additional constraints
- The goal function expresses the execution time of the program under analysis
  - Its maximal value is then an upper bound for all execution times
- An escape from the exponential complexity would be to use heuristics
- These heuristics will, in general, only arrive at suboptimal solutions
- A suboptimal solution represents an unsafe estimate for the WCET
- Thus the escape of resorting to heuristics is barred

Auxiliary Methods: Annotation

- The **annotation of tasks** with information available from the developer is a generic technique to support subsequently applied automatic validation techniques
- The developer may have to supply some information that the tool needs in separate files or by annotating the task
- This information describes
  - the memory layout and any needed characteristics of memory areas
  - ranges for the input values of the task
  - information about the control flow of the task if not determined automatically
    - loop bounds, shapes of nested loops
    - if iterations of inner loops depend on iteration variables of outer loops
    - frequencies of paths or branches taken
  - deviations from the standard function-calling conventions
  - directives as to the desired precision of the result

Auxiliary Methods: Front-Ends

- Most WCET tools analyze software at the executable level
  - since only at this level is all necessary information available
- The first phase in timing analysis is the decoding of the executable
  - and the reconstruction of its control flow
- This can be quite involved,
  depending on the instruction set of the processor and
  the code-generation patterns of the compiler
- Some timing-analysis tools are integrated with a compiler,
  which emits the necessary CFG and call graph for the analysis

Auxiliary Methods: Visualization of results

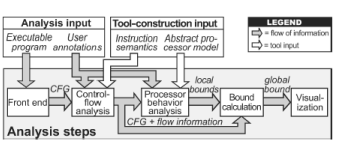
- The results of timing analysis are presented in human-readable form
- This usually shows the call and control-flow graphs
  - annotated with computed timing information and
  - possibly also information about the processor states

Summary

- Introduction
- Problems and Requirements
- Classification of Approaches
- Auxiliary Methods
- Static Methods
- Measurement-Based Methods
- Comparison of Static and Measurement-Based Methods
- Commercial Tools and Research Prototypes
- Experience
- Tools Comparison
- Remaining Problems and Future Perspectives
- Architectural Trends

STATIC METHODS

- This class of methods does not rely on executing code on real hardware or on a simulator
  - but rather takes the task code itself,
  - combines it with some (abstract) model of the system,
  - and obtains upper bounds from this combination



Static Methods: Value Analysis

- Any method for data-cache behavior analysis needs to know effective memory addresses of data
  - in order to determine where a memory access goes
- Effective addresses are only available at runtime

- ***Value Analysis*** is able to statically determine many effective addresses in disciplined code
- It does so by computing ranges for the values in the processor registers and local variables at every program point
- This analysis is also useful to determine loop bounds and to detect infeasible paths

Static Methods: Control-Flow Analysis 1/2

- The purpose of control-flow analysis is to gather information about possible execution paths
- The set of paths is always finite, since termination must be guaranteed
  - The exact set of paths can, in general, not be determined
  - Any superset of this set will be a safe approximation
  - The smaller this superset is, the better

- The input of flow analysis consists of a task representation
  - The call graph and the control-flow graph
  - Possibly additional information, such as ranges for the input data and iteration bounds of some loops
  - Determined by a preceding value analysis or provided by the user

- The result of the flow analysis is constraints on the dynamic behavior of the task
  - Which functions may be called
  - Dependencies between conditionals
  - (In)Feasibility of paths, etc

Static Methods: Control-Flow Analysis 2/2

- There are a number of approaches to automatic flow analysis
- Some of the methods are general
  - while others are specialized for certain types of code constructs

- The methods also differ in the type of codes they analyze
  - Source code
  - Intermediate code (inside the compiler)
  - Machine code

- Control-Flow Analysis is generally easier on source than on machine code
- But it is difficult to map the results to the machine-code program
- Because of compilation may change the control-flow structure
  - Code optimization, linking

Static Methods: Processor-Behavior Analysis 1/9

- A typical processor contains several components that make the execution time context-dependent
  - Such as memory, caches, pipelines and branch prediction
- The execution time of an individual instruction, even a memory access, depends on the execution history

- To find precise execution-time bounds for a given task, it is necessary to analyze the occupancy state of these processor components
  - for all paths leading to the task's instructions

- Processor-behavior analysis determines invariants about these occupancy states for the given task

Static Methods: Processor-Behavior Analysis 2/9

- In principle, no tool is complete that does not take the processor periphery into account
  - the full memory hierarchy, the bus, and peripheral units

- The analysis is done on a linked executable

- It is based on an abstract model of the processor, the memory subsystem, the buses, and the peripherals

- The model is conservative with respect to the timing behavior of the concrete hardware
- The complexity of deriving an abstract processor model strongly depends on the class of processor used

Static Methods: Processor-Behavior Analysis 3/9

- For simpler 8- and 16-bit processors, the timing model construction is rather simple
  - But still time consuming

- Complicating factors for the processor behavior analysis include
  - Instructions with varying execution time because of argument values
  - Varying data reference time because of different memory area access times

Static Methods: Processor-Behavior Analysis 4/9

- For somewhat more advanced 16- and 32-bit processors
  - like the NEC V850E
- With a simple (scalar) pipeline and maybe a cache
- One can analyze different hardware features separately
  - since there are no timing anomalies
- Complicating factors are similar as for the simpler 8- and 16-bit processors
- But also include varying access times because of cache hits and misses and varying pipeline overlap between instructions

Static Methods: Processor-Behavior Analysis 5/9

- More advanced processors will exhibit timing anomalies
  - They have many performance enhancing features that can influence each other
- For these, timing-model construction is very complex
- The analyzes to be used are also less modular and more complex

Static Methods: Processor-Behavior Analysis 6/9

- Execution-time bounds derived for an instruction depend on the states of the processor at this instruction
- Information about the processor states is derived by analyzing potential execution histories leading to this instruction
- Different states in which the instruction can be executed may lead to widely varying execution times with disastrous effects on precision
- A loop iterates 100 times
- But the worst case of the body, *ebody*, only really occurs during one of these iterations and the others are twice as fast
- The overapproximation is **99 * 0.5 * *ebody***

Static Methods: Processor-Behavior Analysis 7/9

- Precision can be gained by regarding execution in classes of execution histories separately which correspond to ***flow contexts***
- **Flow contexts** essentially express by which paths, through loops and calls, control can arrive at the instruction
- Wherever information about the processor's execution state is missing,
  - a conservative assumption has to be made or
  - all possibilities have to be explored

Static Methods: Processor-Behavior Analysis 8/9

- Most approaches compute invariants, one per flow context, about the processor's execution states at each program point
- If there is one invariant for each program point, then it holds for all execution paths leading to this program point
- Different ways to reach a basic block may lead to different invariants at the block's program points
  - Several invariants could be computed
  - Each holds for a set of execution paths
  - The sets together form a partition of the set of all execution paths leading to this program point
  - Each set of such paths corresponds to what sometimes is called a *calling context*

Static Methods: Processor-Behavior Analysis 9/9

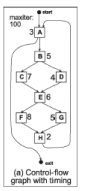
- The **invariants** express static knowledge about
  - The contents of caches
  - The occupancy of functional units and processor queues
  - The states of branch-prediction units
- Knowledge about cache contents is then used to classify memory accesses
  - Definite cache hits (or definite cache misses)
- Knowledge about the occupancy of pipeline queues and functional units is used to exclude pipeline stalls

Static Methods: Estimate Calculation 1/9

- The purpose is to determine an estimate for the WCET
- In dynamic approaches the WCET estimate can underestimate the WCET
  - Since only a subset of all executions is used to compute it
- Combining measurements of code snippets to end-to-end execution times can also overestimate the WCET
  - When pessimistic estimates for the snippets are combined
- In static approaches, this phase computes an upper bound of all execution times of the whole task
  - Based on the flow and timing information derived in the previous phases
- It is then usually called *bound calculation*
- There are three main classes of methods combining analytically determined or measured times to end-to-end estimates proposed in literature
  - *structure-based*
  - *path-based*
  - *implicit-path enumeration* (IPET)

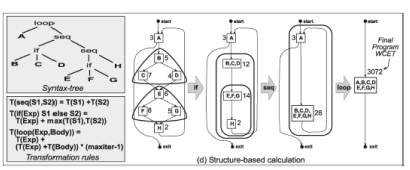
Static Methods: Estimate Calculation 2/9

- Example of a control-flow graph



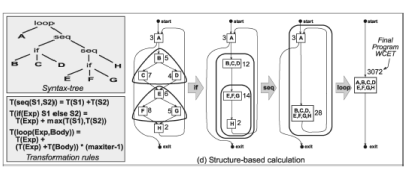
Static Methods: Estimate Calculation 3/9

- **Structure-based calculation**
  - An upper bound is calculated in a bottom-up traversal of the syntax tree combining bounds computed for constituents of statements according to combination rules for that type of statement
- Collections of nodes are collapsed into single nodes



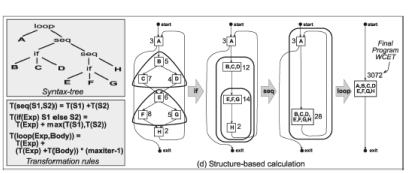
Static Methods: Estimate Calculation 4/9

- **Structure-based calculation**
  - Precision can only be obtained if the same code snippet is considered in a number of **different *flow contexts***
  - Since the execution times in different flow contexts can vary widely
  - Taking flow contexts into account requires transformations of the syntax tree to reflect the different contexts



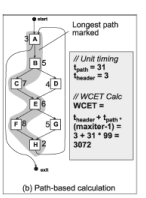
Static Methods: Estimate Calculation 5/9

- **Structure-based calculation**
  - Not every control flow can be expressed through the syntax tree
  - The approach assumes a very straightforward correspondence between the structures of the source and the target program, not easily admitting code optimizations



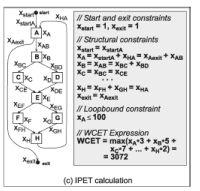
Static Methods: Estimate Calculation 6/9

- **Path-based calculation**
  - The upper bound for a task is determined by computing bounds for different paths in the task, searching for the overall path with the longest execution time
  - The defining feature is that possible execution paths are represented *explicitly*
  - The path-based approach is natural within a single loop iteration
  - But has problems with flow information extending across loopnesting levels
  - The number of paths is exponential in the number of branch points, possibly requiring heuristic search methods



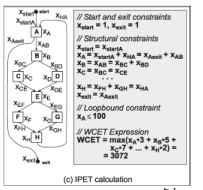
Static Methods: Estimate Calculation 7/9

- **Implicit-path enumeration (IPET)**
  - Program flow and basic-block execution time bounds are combined into sets of arithmetic constraints
  - Each basic block and program flow edge is given:
  - A **time coefficient** (*tentity*) that expresses the upper bound of the contribution of that entity to the total execution time every time it is executed
  - A **count variable** (*xentity*), corresponding to the number of times the entity is executed



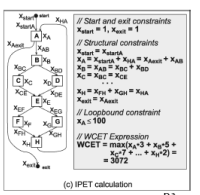
Static Methods: Estimate Calculation 8/9

- **Implicit-path enumeration (IPET)**
  - An upper bound is determined by maximizing the sum of products of the execution counts and times $\sum_{i \in entities} (x_i \,.\, t_i)$
  - The execution count variables are subject to constraints reflecting the structure of the task and possible flows
  - The result of an IPET calculation is an upper timing bound and a worst-case count for each execution count variable



Static Methods: Estimate Calculation 9/9

- **Implicit-path enumeration (IPET)**
  - The *start* and *exit constraints* state that the task must be started and exited once
  - The *structural constraints* reflect the possible program flow
  - For a basic block to be executed it must be entered the same number of times as it is exited
  - The *loop bound* is specified as a constraint on the number of times the loop-head node A can be executed
  - It uses ILP or constraint programming (CP) techniques
  - It has a complexity potentially exponential in the task size
  - The size of the constraint system grows with the number of flow facts



Static Methods: Symbolic Simulation

- Another static method is to simulate the execution of the task in an abstract model of the processor
- The simulation is performed without input
- The simulator has to be capable to deal with partly unknown execution state
- This method combines flow analysis, processor-behavior prediction, and bound calculation in one integrated phase
- Analysis time is proportional to the actual execution time of the task
- This can lead to a very long analysis
  - Simulation is typically orders of magnitudes slower than native execution

Summary

- Introduction
- Problems and Requirements
- Classification of Approaches
- Auxiliary Methods
- Static Methods
- Measurement-Based Methods
- Comparison of Static and Measurement-Based Methods
- Commercial Tools and Research Prototypes
- Experience
- Tools Comparison
- Remaining Problems and Future Perspectives
- Architectural Trends

MEASUREMENT-BASED METHODS 1/5

- These methods attack some parts of the timing-analysis problem by
  - executing the given task on the given hardware or a simulator,
  - for some set of inputs,
  - and measuring the execution time of the task or its parts
- End-to-end measurements of a subset of all possible executions
  - produce estimates or distributions,
  - not bounds for the execution times,
  - if the subset is not guaranteed to contain the worst case
- One execution would be enough if the worst-case input were known

Measurement-Based Methods 2/5

- Other approaches measure the execution times of code segments
  - Typically of CFG basic blocks
- The measured execution times are then combined and analyzed,
  - usually by some form of bound calculation,
  - to produce estimates of the WCET or BCET
- Measurement replaces the processor-behavior analysis used in static methods
- The path-subset problem can be solved in the same way as for the static methods
  - CFA to find all possible paths
  - bound calculation to combine the measured times of the code segments into an overall bound
- This solution would include all possible paths, but would still produce unsafe results if the measured basic-block times were unsafe
- Another problem is that only a subset of the possible contexts (initial processor states) is used for each measured basic block or other kind of code segment

Measurement-Based Methods 3/5

- The context-subset problem could be attacked by running more tests to measure more contexts
- It only decreases, but does not eliminate, the risk of unsafe results
- It is expensive unless intensive testing is already done for other reasons
- Exhaustive testing of all execution paths is usually impossible

Measurement-Based Methods 4/5

- The context-subset problem could be attacked by setting up a worst-case initial state at the start of each measured code segment
- It would be safe if one could determine a worst-case initial state
- However, identifying worst-case initial states is hard or even impossible for complex processors
- Measurement-based tools can compute execution-time bounds for processors with simple timing behavior
- But they produce only estimates of the BCET and WCET for more complex processors
  - as long as this problem is not convincingly solved
- Some tools collect and analyze multiple measurements to provide a picture of the variability of the execution time of the application

Measurement-Based Methods 5/5

- There are multiple ways in which measurement can be performed
- The simplest approach is by extra instrumentation code that collects a timestamp or CPU cycle counter (available in most processors)
- Mixed HW/SW instrumentation techniques require external hardware to collect timings of lightweight instrumentation code
- Fully transparent (nonintrusive) measurement mechanisms are possible using logic analyzers
- Hardware tracing mechanisms like the NEXUS standard and the ETM tracing mechanism from ARM are nonintrusive
  - but do not necessarily produce exact timings

Summary

- Introduction
- Problems and Requirements
- Classification of Approaches
- Auxiliary Methods
- Static Methods
- Measurement-Based Methods
- Comparison of Static and Measurement-Based Methods
- Commercial Tools and Research Prototypes
- Experience
- Tools Comparison
- Remaining Problems and Future Perspectives
- Architectural Trends

Comparison of Static and Measurement-Based Methods 1/4

- Static methods compute bounds on the execution time
- They use CFA and bound calculation to cover all possible execution paths
- They use abstraction to cover all possible context dependencies in the processor behavior
- The price they pay for this safety is the necessity for processor-specific models of processor behavior and possibly imprecise results
  - Such as overestimated WCET bounds
- In favor of static methods is the fact that the analysis can be done without running the program to be analyzed
  - which often needs complex equipment to simulate the hardware and peripherals

Comparison of Static and Measurement-Based Methods 2/4

- Measurement-based methods replace processor behavior analysis by measurements
- Unless all possible execution paths are measured or the processor is simple enough to let each measurement be started in a worst-case initial state
  - Some context-dependent execution-time changes may be missed
  - The method is unsafe
- For the estimate-calculation step, these methods may
  - Use CFA to include all possible execution paths, or
  - Use the observed execution paths (observed number of loop iterations, for example)
    - makes the method unsafe
- They are simpler to apply to new target processors
  - they do not need to model processor behavior
- They produce WCET and BCET estimates that are more precise
  - Closer to the exact WCET and BCET
  - Especially for complex processors and complex applications
  - There is really no way to check how precise an estimate or bound is

Comparison of Static and Measurement-Based Methods 3/4

- Both classes of methods share some technical problems and solutions
- The front ends are similar when both use executable code as input
- Control-flow analysis is similar
- Bound/estimate calculation can be similar

- For example,
  the IPET calculation is used by some static tools and by some measurement-based tools

Comparison of Static and Measurement-Based Methods 4/4

- The main technical problem for static methods is modeling processor behavior
- This is not a problem for most measurement-based methods, where the main problem is
  - to measure the execution time accurately
  - with fine granularity
  - and without perturbing the program being measured

- The solution is often processor- or platform-specific

- Implementing a measurement method for a new processor is usually less work than creating an abstract model of the processor behavior

Summary

- Introduction
- Problems and Requirements
- Classification of Approaches
- Auxiliary Methods
- Static Methods
- Measurement-Based Methods
- Comparison of Static and Measurement-Based Methods
- Commercial Tools and Research Prototypes
- Experience
- Tools Comparison
- Remaining Problems and Future Perspectives
- Architectural Trends

COMMERCIAL TOOLS AND RESEARCH PROTOTYPES

- Completely static tool
  - The aiT Tool – AbsInt Angewandte Informatik, Saarbrücken, Germany

- Mostly measurement-based tool
  - The RapiTime Tool of Rapita Systems, York, UK

The aiT Tool – AbsInt Angewandte Informatik (Germany)

- Purpose: to obtain upper bounds for the execution times of code snippets (e.g., given as subroutines) in executables
- These code snippets may be tasks called by a scheduler in some real-time application, where each task has a specified deadline

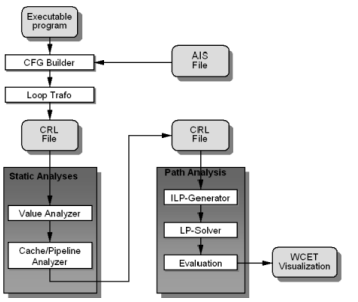
- aiT works on executables
  - The source code does not contain information on register usage and on instruction and data addresses
  - Such addresses are important for cache analysis and the timing of memory accesses
  - Specially when there are several memory areas with different timing behavior

The aiT Tool – Functionality

- aiT might need user input
  - to be able to compute a result
  - to improve the precision of the result
- User annotations
  - written into parameter files and refer to program points by
    - absolute addresses
    - addresses relative to routine entries
    - structural descriptions (like the first loop in a routine)
  - User annotations can be embedded into the source code as special comments
    - they are mapped to binary addresses using the line information in the executable
- Annotations of
  - loop bounds
  - flow facts
  - values of registers and variables (mode variables)

The aiT Tool – Architecture 1/6

- aiT architecture



The aiT Tool – Architecture 2/6

- First: the control flow is reconstructed from the given object code by a bottom-up approach
- The reconstructed control flow is annotated with the information needed by subsequent analyzes
- It is translated into CRL (control-flow representation language)
  - human-readable intermediate format designed to simplify analysis and optimization at the executable/assembly level
- This annotated CFG serves as the input for the following analysis steps

The aiT Tool – Architecture 3/6

- Next, value analysis computes ranges for the values in the processor registers at every program point
- Its results are used for
  - Loop-bound analysis
  - Detection of infeasible paths, depending on static data
  - Determination of possible addresses of indirect memory accesses
- An extreme case of control, depending on static data, is a virtual machine program interpreting abstract code given as data

The aiT Tool – Architecture 4/6

- aiT's cache analysis relies on the addresses of memory accesses as found by value analysis
- It classifies memory references as sure hits and potential misses
- It is based upon Ferdinand and Wilhelm [1999]
  - Handles LRU caches
- It had to be modified to reflect the non-LRU replacement strategies of common microprocessors
  - the pseudo-round-robin replacement policy of the ColdFire MCF 5307
  - the PLRU (Pseudo-LRU) strategy of the PowerPC MPC 750 and 755
- The deviation from perfect LRU is the reason for the reduced predictability of the cache contents in the case of these two processors compared to processors with perfect LRU caches

The aiT Tool – Architecture 5/6

- Pipeline analysis predicts the behavior of the task on the processor pipeline
- The result is an upper bound for the execution time of each basic block in each distinguished execution context

The aiT Tool – Architecture 6/6

- Finally:
  Bound calculation (called path analysis in the aiT framework)
- It determines a worst-case execution path of the task
  from the timing information for the basic blocks

The aiT Tool – Methods

- *Employed Methods*
- Value analysis and cache/pipeline analysis are realized by abstract interpretation
  - a semantics-based method for static program analysis
- Path analysis is implemented by ILP
- Reconstruction of the control flow is performed by a bottom-up analysis
- Detailed information about
  - the upper bounds
  - the path on which it was computed
  - and the possible cache and pipeline states at any program point

  are attached to the call graph/control-flow graph and can be visualized

The aiT Tool – Limitations

- *Limitations of the Tool*
- aiT includes automatic analysis to determine the targets of indirect calls and branches and to determine upper bounds of the iterations of loops
  - These analyzes do not work in all cases
  - If they fail, the user has to provide annotations
- aiT relies on the standard calling convention
  - If some code does not adhere to the calling convention the user might need to supply additional annotations describing control-flow properties of the task

The aiT Tool – Plataforms

- *Supported Hardware Platforms*
- Motorola PowerPC MPC 555, 565, and 755
- Motorola ColdFire MCF 5307
- ARM7 TDMI
- HCS12/STAR12
- TMS320C33
- C166/ST10
- Renesas M32C/85 (prototype)
- Infineon TriCore 1.3

The RapiTime Tool of Rapita Systems, York, UK

- RapiTime is the commercial quality version of the pWCET tool developed at the Real-Time Systems Research Group at the University of York
- RapiTime is a measurement-based tool
- It derives timing information of how long a particular section of code (generally a basic block) takes to run from measurements
- Measurement results are combined according to the structure of the program to determine an estimate for the longest path
- RapiTime computes the whole probability distribution of the execution time of the longest path in the program (and other subunits)
  - This distribution has an upper bound, the WCET estimate
  - A lower bound

The RapiTime Tool – Functionality

- The input of RapiTime is either
  - a set of source files (C or Ada) or
  - an executable
- The user has to provide test data for the measurements
- The output is a browsable HTML report with
  - description of the WCET prediction
  - actual measured execution times
  - split for each function and subfunction
- Timing information is captured on the running system by either
  - a software instrumentation library
  - a lightweight software instrumentation with external hardware support
  - purely nonintrusive tracing mechanisms (like Nexus and ETM)
  - traces from CPU simulators

The RapiTime Tool – Functionality

- Timing information is captured on the running system by either
  - a software instrumentation library
  - a lightweight software instrumentation with external hardware support
  - purely nonintrusive tracing mechanisms (like Nexus and ETM)
  - traces from CPU simulators
- The user can add annotations in the code to guide how the instrumentation and analysis process will be performed
  - to bound the number of iterations of loops, etc.
- The RapiTime tool supports various architectures
- Adapting the tool for new architectures requires porting the object code reader (if needed) and determining a tracing mechanism for that system

The RapiTime Tool – Limitations

- The RapiTime tool does not rely on a model of the processor
- Thus, in principle, it can model any processing unit, even with
  - out-of-order execution
  - Multiple execution units
  - Various hierarchies of caches
  - Etc
- The limitation is put on the need to extract execution traces, which require some code instrumentation and a mechanism to extract these traces from the target system
- RapiTime cannot analyze programs with recursion and with nonstatically analyzable function pointers

The RapiTime Tool – Platforms

- Motorola processors
  - MPC555
  - HCS12
  - Etc
- ARM
- MIPS
- NecV850

Summary

- Introduction
- Problems and Requirements
- Classification of Approaches
- Auxiliary Methods
- Static Methods
- Measurement-Based Methods
- Comparison of Static and Measurement-Based Methods
- Commercial Tools and Research Prototypes
- Experience
- Tools Comparison
- Remaining Problems and Future Perspectives
- Architectural Trends

EXPERIENCE 1/5

- There are extensive reports about industrial use
- Tools are under routine use in the aeronautics and the automotive industries
- They enjoy very positive feedback concerning speed, precision of the results, and usability
- The aeronautics industry uses them in the development of the most safety-critical systems (aiT)
  - The fly-by-wire systems for the Airbus A380
  - The used WCET tool will be qualified as a verification tool for Level A Code
  - International avionics standard for safety-critical software, RTCA/DO-178B (Software Considerations in Airborne Systems and Equipment Certification Requirements)

Experience 2/5

- Some comparable benchmarks
- Benchmarks published earlier offer better results regarding the degree of overestimation
  - although significant methodological progress has been made
- Due to the advancement of processor architectures
  - The divergence of processor and memory speeds
  - Both have increased the timing variability
  - Increases penalty for the lack of knowledge in analysis results

Reference	Year	Cache-miss penalty	Overestimation (%)
Lim et al. [1995]	1995	4	20–30
Thesing et al. [2003]	2002	25	15
Souyris et al. [2005]	2005	60 for accessing instructions in SDRAM 200 for access over PCI bus	30–50

Experience 3/5

- The Mälardalen University WCET-research group has performed several industrial WCET case studies as M.Sc. projects using the SWEET and aiT tools
- The case studies show that:

- It is possible to apply static WCET analysis to a variety of industrial systems
- The tools used performed well and derived safe upper timing bounds
- Detailed knowledge of the analyzed code and many manual annotations were often required to achieve reasonably tight bounds
- These annotations were necessary for
  - program flow constraints
  - constraints on addresses of memory accesses
- A higher degree of automation and support from the tools, in most cases, have been desirable
  - Automatic loop-bounds calculation

Experience 4/5

- The case studies also show that single timing bounds, covering all possible scenarios, are not always what you want
- For several analyzed systems it was more interesting to have different WCET bounds for different running modes or system configurations

- In some cases, it was possible to manually derive a parametrical formula, showing how the WCET estimate depends on some specific system parameters

- The case studies were done for processors without cache
- The overestimations were mostly in the range 5–15% as compared with measured times
  - Measurements were done with emulators or directly on the hardware

Experience 5/5

- Maximal size of tasks analyzed by the different tools vary between 10 and 80KB of code

- Analysis times vary widely
- It depends on the complexity of the processor and its periphery and the structure and other characteristics of the software

- Analysis of a task for a simple microprocessor, such as the C166/ST10, may finish in a few minutes
- Analysis of a complex software, an abstract machine, and the code interpreted by it, and a complex processor has been shown to take in the order of a day

- The size of the abstract processor models underlying some static analysis approaches range from 3000 to 11,000 lines of C code
  - This C code is the result of a translation from a formal model

Summary

- Introduction
- Problems and Requirements
- Classification of Approaches
- Auxiliary Methods
- Static Methods
- Measurement-Based Methods
- Comparison of Static and Measurement-Based Methods
- Commercial Tools and Research Prototypes
- Experience
- Tools Comparison
- Remaining Problems and Future Perspectives
- Architectural Trends

LIMITATIONS OF THE TOOLS

- Bounded iteration and recursion
- Pointers to data and to functions that cannot statically be resolved
- The use of dynamically allocated data

- Most tools will expect that function-calling conventions are observed
- Some tools forbid recursion

- Currently, only monoprocessor targets are supported
- Most tools only consider uninterrupted execution of tasks

Remaining Problems and Future Perspectives 1/7

- *Increased support for flow analysis*

- Most problems reported in timing analysis case studies relate to setting correct loop bounds and other flow annotations

- Stronger static program analyzes are needed to extract this information from the software

Remaining Problems and Future Perspectives 2/7

- *Verification of abstract processor models*
- Static timing-analysis tools based on abstract processor models may be incorrect if these models are not correct
- Ongoing research activities attempt the formal derivation of abstract processor models from concrete models
  - Progress in this area will not only improve the accuracy of the abstract processor models
  - It will also reduce the effort to produce them
- Measurement-based methods can be used to test the abstract models

Remaining Problems and Future Perspectives 3/7

- *Integration of timing analysis with compilation*
- An integration of static timing analysis with a compiler can provide valuable information available in the compiler to the timing analysis
- Improves the precision of analysis results

Remaining Problems and Future Perspectives 4/7

- *Integration with scheduling*
- Preemption of tasks causes large context-switch costs on processors with caches
- The preempting task may ruin the cache contents
  - The preempted task encounters considerable cache-reload costs when resuming execution
- These context-switch costs may be even different for different combinations of preempted and preempting task
- These large and varying context-switch costs violate assumptions underlying many real-time scheduling methods
- Scheduling approaches considering the combination of timing analysis and preemptive scheduling have to be developed

Remaining Problems and Future Perspectives 5/7

- *Interaction with energy awareness*
- This may concern the trade off between speed and energy consumption
- Jayaseelan et al. [2006] presents a static analysis technique to estimate the worst-case energy consumption of a task on complex microarchitectures
- Estimating a bound on energy is nontrivial
  - It is unsafe to assume any direct correlation with the bound on execution time
  - Information computed for WCET determination is of high value for the determination of energy consumption
    - cache behavior

Remaining Problems and Future Perspectives 6/7

- *Design of systems with time-predictable behavior*
- Several trends in system design make systems less and less predictable
- Any proposal increasing predictability plainly at the cost of performance will most likely not be accepted by developers.

Remaining Problems and Future Perspectives 7/7

- *Extension to component-based design*
- Timing-analysis methods should be made applicable to
  - Component-based design
  - Systems built on top of realtime operating systems
  - Systems using real-time middleware

Summary

- Introduction
- Problems and Requirements
- Classification of Approaches
- Auxiliary Methods
- Static Methods
- Measurement-Based Methods
- Comparison of Static and Measurement-Based Methods
- Commercial Tools and Research Prototypes
- Experience
- Tools Comparison
- Remaining Problems and Future Perspectives
- Architectural Trends

Architectural Trends 1/8

- The hardware used in creating an embedded real-time system has a great effect on the ease of predictability of the execution time
- The simplest case are traditional 8- and 16-bit processors
  - with simple architectures
- In such processors,
  each instruction basically has a fixed execution time
- Such processors are easy to model from the hardware timing perspective
- The only significant problem in WCET analysis is how to determine the program flow

Architectural Trends 2/8

- There is also a class of processors with simple in-order pipelines
- They are found in cost-sensitive applications requiring higher performance than that offered by classic 8- and 16-bit processors
- Examples are the ARM7 and the recent ARM Cortex R series
- Over time, these chips can be expected to replace the 8- and 16-bit processors for most applications
- They have typically simple pipelines and cache structures
- Relatively simple and fast WCET hardware analysis methods can be applied

Architectural Trends 3/8

- At the high end of the embedded real-time spectrum
  performance requirements for applications
  like flight control and engine control
  force real-time systems designers to use complex processors
- Processors with caches and out-of-order execution
- Examples are the PowerPC 750, PowerPC 7448, and ARM11 families of processors
- Analyzing such processors requires more advanced tools and methods
  - especially in the hardware analysis

Architectural Trends 4/8

- The mainstream of computer architecture is steadily adding complexity and speculative features in order to push the performance envelope
- Architectures such as the AMD Opteron, Intel Pentium 4 and Pentium-M, and IBM Power5 use
  - multiple threads per processor core
  - deep pipelines
  - several levels of caches
- to achieve maximum performance on sequential programs

Architectural Trends 5/8

- This mainstream trend of ever-more complex processors is no longer as dominant as it used to be
- In recent years, several other design alternatives have emerged in the mainstream, where the complexity of individual processor cores has been reduced significantly
- Many new processors are designed by using several simple cores instead of a single or a few complex cores
- This design gains throughput per chip by running more tasks in parallel
  - at the expense of single-task performance
- These designs are cache-coherent multiprocessors on a chip
  - Fairly complex cache and memory system
  - The complexity of analysis moves from the behavior of the individual cores to the interplay between them as they access memory

Architectural Trends 6/8

- Another very relevant design alternative:
  to use several simple processors with private memories
  - instead of shared memory
- This design is common in mobile phones
  - an ARM main processor combined with one or more DSPs on a single chip
- Outside the mobile phone industry
  - the IBM–Sony-Toshiba Cell processor is a high-profile design using a simple in-order PowerPC core along with eight synergistic processing elements (SPE)
  - The SPEs in the Cell are designed for predictable performance and use local program-controlled memories rather than caches, just like most DSPs
  - This type of architecture provides several easy-to-predict processors on a chip as an alternative to a single hard-to-predict processor

Architectural Trends 7/8

- Field-programmable gate arrays (FPGAs) are another design alternative for some embedded applications
- Several processor architectures are available as “soft cores” that can be implemented in an FPGA together with application-specific logic and interfaces
- Such processor implementations may have application-specific timing behavior
  - This may be challenging for off-theshelf timing analysis tools
  - But they are also likely to be less complex and thus easier to analyze than general-purpose processors of similar size
- Some standard processors are now packaged together with FPGA on the same chip for implementing application-specific logic functions
  - The timing of these FPGA functions may be critical and need analysis
  - Separately or together with the code on the standard processor

Architectural Trends 8/8

- There is also work on application-specific processors or application-specific extensions to standard instruction sets
  - again creating challenges for timing analysis
- Here there is also an opportunity for timing analysis:
  to help find the application functions that should be speeded up by defining new instructions

Final Remark

- This paper was submitted in february 2004, revised june 2006
  - Any one working in the area should check for newer advances in this area
- Certainly some new tools and features appeared since then
- But there is no reason to think that the general framework has changed dramatically
- The problems and requirements are still the same
- The main methods presented here are still used
- aïT is the leader commercial tool

Summary

- Introduction
- Problems and Requirements
- Classification of Approaches
- Auxiliary Methods
- Static Methods
- Measurement-Based Methods
- Comparison of Static and Measurement-Based Methods
- Commercial Tools and Research Prototypes
- Experience
- Tools Comparison
- Remaining Problems and Future Perspectives
- Architectural Trends