Sistemas de Tempo Real:

Worst-Case Execution Time

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

romulo@das.ufsc.br
http://www.das.ufsc.br/~romulo

Outubro/2010

Reference

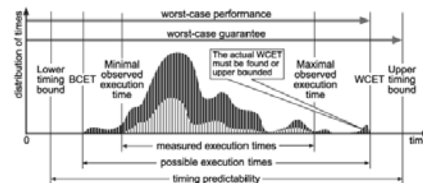
- The Worst-Case Execution-Time Problem—Overview of Methods and Survey of ToolsPrimeiros relógios
  - ACM Transactions on Embedded Computing Systems, Vol. 7, No. 3, Article 36, Publication date: April 2008.
  - REINHARD WILHELM TULIKA MITRA
  - JAKOB ENGBLOM FRANK MUELLER
  - ANDREAS ERMEDAHL ISABELLE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

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 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

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 task into a sequence of subtasks
  - 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
- A necessary ingredient for this are the 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

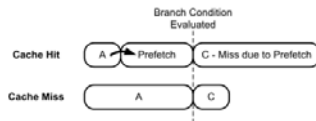
- For example, missing information about whether the next instruction will be in the cache may lead to
  - one execution starting with a cache load contributing the cache-miss penalty to the execution time
  - while another execution will start with an instruction fetch from the cache
- Intuition suggests 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*

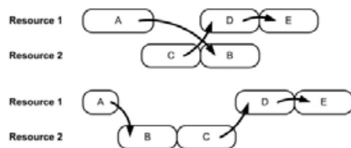


Prob&Req: Timing Anomalies 5/8

- Another type of timing anomalies are instances of well-known *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

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

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
  - Stronger guarantees can be given by equivalence checking between different abstraction levels
  - An ongoing research activity is the formal derivation of abstract processor models from concrete models

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

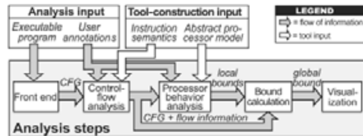
- Most WCET tools analyze software at the executable level
  - since only at this level is all necessary information available
- The first phase in timing analysis is the decoding of the executable
  - and the reconstruction of its control flow
- This can be quite involved,
  depending on the instruction 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

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
- V*alue Analysis* is able to determine many effective addresses in disciplined code statically
- 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 onthe 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, *e*body, only really occurs during one of these iterations and the others are twice as fast
- The overapproximation is $99 * 0.5 * e\text{body}$

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

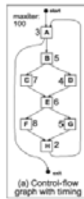
- The invariants express static knowledge about
  - The contents of caches
  - The occupancy of functional units and processor queues
  - The states of branch-prediction units
- Knowledge about cache contents is then used to classify memory accesses
  - Definite cache hits (or definite cache misses)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

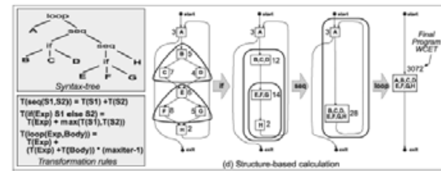
- Example of a control-flow graph



(a) Control-flow graph with timing

Static Methods: Estimate Calculation 3/9

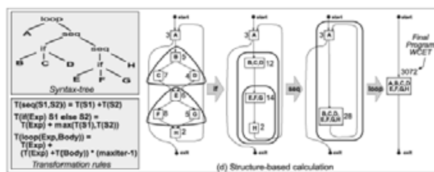
- **Structure-based calculation**
  - An upper bound is calculated in a bottom-up traversal of the syntax tree combining bounds computed for constituents of statements according to combination rules for that type of statement
- Collections of nodes are collapsed into single nodes



(d) Structure-based calculation

Static Methods: Estimate Calculation 4/9

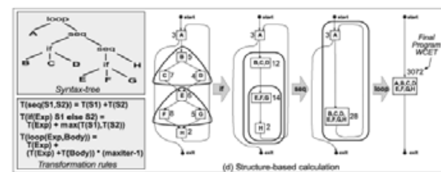
- **Structure-based calculation**
  - Precision can only be obtained if the same code snippet is considered in a number of different *flow contexts*
  - Since the execution times in different flow contexts can vary widely
  - Taking flow contexts into account requires transformations of the syntax tree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



(d) Structure-based calculation

Static Methods: Estimate Calculation 5/9

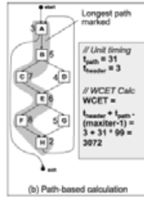
- **Structure-based calculation**
  - Not every control flow can be expressed through the syntax tree
  - The approach assumes a very straightforward correspondence between the structures of the source and the target program, not easily admitting code optimizations



(d) Structure-based calculation

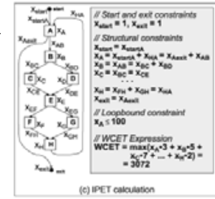
Static Methods: Estimate Calculation 6/9

- **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



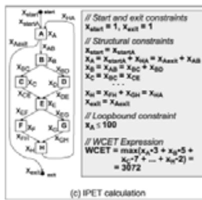
Static Methods: Estimate Calculation 7/9

- **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



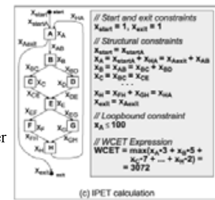
Static Methods: Estimate Calculation 8/9

- **Implicit-path enumeration (IPET)**
  - An upper bound is determined by maximizing the sum of products of the execution counts and times $\sum_{i \ \ entities} (x_i \cdot t_i)$
  - 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

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

Comparison of Static and Measurement-Based Methods 1/6

- 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 2/6

- 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 3/6

- 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 4/6

- 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

Comparison of Static and Measurement-Based Methods 5/6

- The handling of timing anomalies offers an interesting comparison of the methods

- For measurement-based methods
  - timing anomalies make it very hard to find a worst-case initial state for a measurement
  - To be safe, the measurement should now be done from all possible initial states, which is impractical
  - Measurement-based methods then use only a subset of initial states and so are not safe

Comparison of Static and Measurement-Based Methods 6/6

- For static methods
  - timing anomalies make it hard to define state abstractions that give a precise abstract interpretation of each instruction and of the execution time spent in the instruction
  - the processor behavior tends to depend on unknown aspects of the state, forcing the abstract simulation to follow many possible executions of each basic block
  - This laborious exploration is limited to basic blocks
  - the abstract simulation considers the global flow of the task for the processor-behavior analysis by propagating simulation results between basic blocks
  - No worst-case assumptions need to be made by the analysis on the basic block level

COMMERCIAL TOOLS AND RESEARCH PROTOTYPES

- Completely static tools
  - The aiT Tool – AbsInt Angewandte Informatik, Saarbrücken, Germany
  - The Bound-T Tool – Tidorum, Helsinki, Finland
  - Research Prototype from Florida State, North Carolina State and Furman Universities
  - Research prototypes of TU Vienna
  - The Chronos Research Prototype – National University of Singapore
  - The Heptane Research Prototype – IRISA, Rennes, France
- Mostly static tools with a small rudiment of measurement
  - SWEET (SWEdish Execution-Time Tool)
  - Research Prototype from Chalmers University of Technology, Göteborg, Sweden
  - SymTA/P Tool of TU Braunschweig, 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

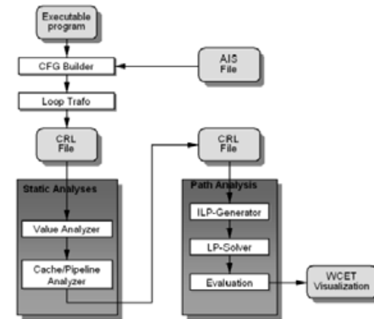
- aiT works on executables
  - the source code does not contain information on register usage and on instruction and data addresses
  - Such addresses are important for cache analysis and the timing of memory accesses
  - Specially when there are several memory areas with different timing behavior

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 Bound-T Tool – Tidorum, Helsinki, Finland

- The Bound-T tool was originally developed at Space Systems Finland
  - Under contract with the European Space Agency (ESA)
  - Intended for the verification of on-board software in spacecraft
- Tidorum Ltd. is extending Bound-T to other application domains

The Bound-T Tool – Functionality 1/2

- Determines an upper bound on the execution time of a subroutine, including called functions
- The tool can also determine an upper bound on the stack usage of the subroutine, including called functions
- The input is a binary executable program
  - with (usually) an embedded symbol table (debug information)
- It is able to compute upper bounds on some counter-based loops
  - For other loops the user provides annotations, called assertions in Bound-T
  - Annotations can also be given for variable values to support the automatic loop bounding

The Bound-T Tool – Functionality 2/2

- The output is a text file listing the upper bounds, etc. and graph files showing call and control-flow graphs
  - for display with the DOT tool
- When the task under analysis follows the ESA-specified HRT ("hard real time") tasking architecture
  - Bound-T can generate the HRT execution skeleton file
  - It contains both the tasking structure and the computed bounds
  - It can be fed directly into the ESA-developed tools for schedulability analysis and scheduling simulation

The Bound-T Tool – Methods 1/6

- The processor model is manually constructed for each processor
- Bound-T has general facilities for modeling control flow and integer arithmetic
  - but not for modeling complex processor states
- Some special-purpose static analyzes have been implemented
  - For the SPARC register-file over- and underflow traps
  - For the concurrent operation of the SPARC integer unit and floating-point unit
  - Both examples use abstract interpretation followed by ILP

The Bound-T Tool – Methods 2/6

- The CFG is often defined to model the processor's instruction-sequencing behavior
  - not just the values of the program counter
- A CFG node typically represents a certain pipeline state
- The CFG is really a pipeline-state graph
- Instruction interactions (e.g., data-path blocking) are modeled in the time assigned to CFG edges

The Bound-T Tool – Methods 3/6

- Counter-based loops are bounded by modeling the task's loop-counter arithmetic
- The computational effect of each instruction is modeled as a relation between the "before" and "after" values of the variables
  - registers and other storage locations
- The relation is expressed in Presburger arithmetic
  - as a set of affine (linear plus constant term) equations and inequalities
  - Possibly conditional
- Instruction sequences are modeled by concatenating the relations of individual instructions
- Branching control-flow is modeled by adding the branch condition to the relation
- Merging control-flow is modeled by taking the union of the inflowing relations

The Bound-T Tool – Methods 4/6

- Loops are modeled by analyzing the model of the loop body to classify variables as loop invariant or variant
- The whole loop (including an unknown number of repetitions) is modeled as a relation that
  - keeps the loop-invariant variables unchanged and
  - assigns unknown values to the loop-variant variables
- This is a first approximation
  - may be improved later when the number of loop iterations is bounded
- With this approximation
  - the computations in an entire subprogram can be modeled in one pass

The Bound-T Tool – Methods 5/6

- To bound loop iterations
  - reanalyzes the model of the loop body to find loop-counter variables
- A loop counter is
  - a loop-variant variable such that
  - one execution of the loop body changes the variable by an amount
  - that is bounded to a finite interval that does not contain zero
- If Bound-T also finds bounds on the initial and final values of the variable
  - A simple computation gives a bound on the number of loop iterations
- Loop bounds can be context-dependent
  - if they depend on scalar pass-by-value parameters
  - for which actual values are provided at the top (caller end) of a call path

The Bound-T Tool – Methods 6/6

- The worst-case path and the upper bound for one subroutine are found by the implicit path enumeration technique applied to the CFG of the subroutine
- If the subroutine has context-dependent loop bounds, the IPET solution is computed separately for each context (call path)
- Annotations are written in a separate text file, not embedded in source code
- The program element to which an annotation refers is identified
  - by a symbolic name (subroutine, variable) or
  - by structural properties (loops, calls)

The Bound-T Tool – Limitations 1/2

- The task to be analyzed must not be recursive
- The control-flow graphs must be reducible
- A graph is reducible if and only if repeated application yields a graph with only one node
  - Replace self loop by a single node
  - Replace a sequence of nodes such that all the incoming edges are to the first node and all the outgoing edges are to the last node
- Dynamic (indexed) calls are only analyzed in special cases
  - unique target address
- Dynamic (indexed) jumps are analyzed based on the code patterns that the supported compilers generate for switch/case structures
  - but not all such structures are supported
- Bound-T can detect some infeasible paths as a side effect of its loop-bound analysis
  - There is no systematic search for such paths
- The bounds of an inner loop cannot depend on the index of the outer loop(s)
- Loop-bound analysis does not cover multiplication, division or logical bit-wise operations

The Bound-T Tool – Limitations 2/2

- The task to be analyzed must use the standard calling conventions
- Function pointers are not supported
  - statically assigned interrupt vectors can be analyzed
- No cache analysis is yet implemented
  - the current target processors have no cache or very small and special caches
- Any timing anomalies in the target processor must be taken into account in the execution time that is assigned to each basic block in the CFG
  - The currently supported, cacheless processors, probably have no timing anomalies
- Models for complex processors would be harder to implement in Bound-T

The Bound-T Tool – Platforms

- *Supported Hardware Platforms*
- Intel-8051 series (MCS-51)
- Analog Devices ADSP-21020
- ATMEL ERC32 (SPARC V7)
- Renesas H8/300
- ARM7 (prototype)
- ATMEL AVR and ATmega (prototypes)

Florida State/North Carolina State/Furman – Research Prototype

- The tool set performs timing analysis of a single task or a subroutine
- A user interacts with the timing analyzer
- The user compiles all of the files that comprise the task
- The compiler was modified to produce information used by the timing analyzer
  - Number of loop iterations
  - control flow
  - Instruction characteristics
- The number of iterations for simple and nonrectangular loop nests are supported
- The timing analyzer produces lower and upper bounds for each function and loop in the task
  - This entire process is automatic

Florida State/North Carolina State/Furman – Methods 1/2

- The tool uses data-flow analysis for cache analysis to make caching categorizations for each instruction
  - It supports direct-mapped and set-associative caches
- CFA is used to distinguish paths at each loop and function level in the task
- The pipeline is simulated to obtain the upper bound of each path
- Caching categorizations are used during this time so that pipeline stalls, and cache-miss delays can be properly integrated
- The loop analysis iteratively finds the worst-case path until the caching behavior reaches a fixed point that is guaranteed to remain the same

Florida State/North Carolina State/Furman – Methods 2/2

- Loop-bounds analysis is performed in the compiler to obtain the number of iterations for each loop
  - The timing analyzer is also able to address nonrectangular loop nests
- Parametric timing analysis support is also provided for runtime bound loops
  - producing a bounds formula parameterized on loop bounds rather than cycles
- Branch-constraint analysis is used to tighten the predictions by disregarding paths that are infeasible
- A timing tree is used to evaluate the task in a bottom-up fashion
- Functions are distinguished into instances so that caching categorizations for each instance can be separately evaluated

Florida State/North Carolina State/Furman – Limitations

- Loop bounds for numeric timing analysis are required to be statically known
  - or there has to be a known loop bound in the outer loop in a nonrectangular loop nest
- Loop bounds need not be statically known when using parametric timing analysis
- No support is provided for pointer analysis or dynamic allocation
- No calls through pointers are allowed
- No recursion is allowed
- Limited data cache support
- The timing analyzer is only able to determine execution-time bounds of applications on simple RISC/CISC architectures
- Analyzing small codes in seconds and medium-sized codes in minutes
  - Entire systems may take hours/days
  - Scalability depends on the system/target device (8-bit versus 32-bit systems)

Florida State/North Carolina State/Furman – Platforms

- *Supported Hardware Platforms*
- The hardware platforms include a variety of uniprocessors
- MicroSPARC I
- Intel Pentium
- StarCore SC100
- PISA/MIPS
- Atmel Atmega
- Experiments have been performed with the Force MicroSPARC I VME board
- The timing analyzer WCET predictions have been validated on the Atmel Atmega to cycle-level accuracy

Research Prototypes of TU Vienna

- A prototype tool for *static-timing analysis* that has been integrated into a Matlab/Simulink tool chain and can analyze C code or Matlab/Simulink models
- A *measurement-based tool* that uses genetic algorithms to direct input-data generation for timing measurements in the search for the worst case or long program execution times
- A *hybrid tool* for timing analysis that uses both measurements and elements from static analysis to assess the timing of C code

TU Vienna Static Analysis

- Performs timing analysis for software coded in WCETC
- WCETC is a subset of C with extensions
  - allow annotations about (in)feasible execution paths
- The tool cooperates with a C compiler
  - The compiler translates the WCETC code into object code and some information for the WCET analyzer
  - This object code is then analyzed to compute an upper bound
- A component has been built into the Matlab/Simulink tool chain
- This component generates code from the block set that includes all path annotations necessary for timing analysis
  - there is no need to perform any additional flow analysis or annotate the code
- The tool supports fully automatic timing analysis for the complete Matlab/Simulink block set defined within the European IST project SETTA

TU Vienna Static Analysis – Methods

- Adaptations to the Matlab/Simulink tool chain
  - the code-generation templates were modified
  - The templates for our block set were changed so that TLC generates code with WCETC macros instead of pure C code
- A GNU C compiler was adapted
  - To translate WCETC code
  - To cooperate with the WCET analyzer
- The modified C compiler uses abstract cointerpretation of path information during code translation in order to trace changes in the control structure as made by code optimization
  - Keeps path information consistent in optimizing compilers
- It computes execution-time bounds for the generated code by means of ILP adding information about infeasible execution paths

TU Vienna Static Analysis – Limitations & Platforms

- If used together with the SETTA Matlab/Simulink block set
  - fully automated timing analysis
- If used to analyze C code
  - it may be necessary to annotate the code with information about (in)feasible paths
  - the quality of the computed bounds strongly depends on the quality of the annotations
- *Supported hardware platforms:*
- M68000, M68360
- C167

TU Vienna Measurement-Based Analysis

- Yields optimistic approximations to the worst-case execution times of a piece of code
- Genetic algorithms are used to generate input data for execution-time measurements
  - At the beginning, the task or program under observation is run with a number of random-input data sets
  - For each of the input data sets the execution time is measured and stored
  - The execution-time values are used as fitness values for their respective input data sets
  - The fitness values form the basis for the generation of a new population of input data sets by the genetic algorithms
- GA-based bounds can not guarantee to produce safe results
- *Supported hardware platforms:*
  C167 and PowerPC processors

TU Vienna Hybrid Analysis

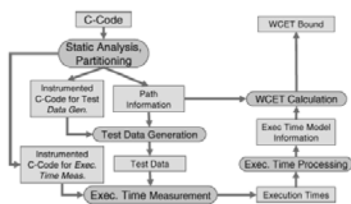
- Combines static program analysis techniques and execution time measurements to calculate an estimate of the WCET
- Automatic segmentation of the program code into segments of reasonable size
- Automatic generation of test data used to measure the execution times of all subpaths within each program segment
- The tool has been designed with a special focus on analyzing automatically generated code from Matlab/Simulink models

TU Vienna Hybrid Analysis – Architecture 1/2

- Takes a C program as input
- Partitions the program into code segments
- Extracts path information that is used during the final bounds calculation to identify infeasible paths within the program
- Automatic test data generation is used to derive the required input data for the execution-time measurements
- Measurements are performed remotely on the real target hardware
- Measurement results are processed to construct a timing model specific to the analyzed program
  - Used together with the path information to calculate a WCET estimate

TU Vienna Hybrid Analysis – Architecture 2/2

- Architecture



TU Vienna Hybrid Analysis – Methods 1/3

- The central technique is the automatic test data generation used to derive a WCET estimate by means of execution time measurements
- A formal test data-generation method is used to guarantee a full subpath coverage within each program segment
- To compensate the high computation cost of formal test data generation, a three-step approach is used

TU Vienna Hybrid Analysis – Methods 2/3

- To compensate the high computation cost of formal test data generation, a three-step approach is used
- 1. Random search is used to generate the majority of test data
  - The path coverage of these test data is derived from an automatically instrumented version of the program
- 2. Heuristic search methods, like genetic algorithms, allow to improve the segment coverage already achieved by step 1
- 3. The remaining test data are generated using formal test data generation based on *model checking*
  - The formal model of the program is automatically derived from the source code
  - Code optimizations have to be performed to improve the performance
  - It provides for a given subpath in a program segment either the test data to trigger its execution or the information that this subpath is infeasible

TU Vienna Hybrid Analysis – Methods 3/3

- Based on the measurement results and the path information of the program,
- the overall WCET estimate is calculated using ILP

TU Vienna Hybrid Analysis – Limitations

- The tool performs static program analysis at the source code level
- It has to be assured that the compiler does not significantly change the structure of the program code
- The hybrid analysis tool guarantees path coverage for each program segment
  - Every execution scenario is covered by the analysis
- The tool does not provide state coverage
  - the WCET estimate is not guaranteed to be a safe upper bound of the execution times for complex processors having pipelines or caches
- It supports program-flow annotations only at the granularity of entire program segments

TU Vienna Hybrid Analysis – Platforms

- *Supported hardware platforms*
- HCS12
- Pentium processors
- The hybrid approach does not rely on a model of a processor
- It is relatively easy to adapt it to other processors
- To port the tool to a new processor
  - Modify the instrumentation code used to measure execution times
  - Provide a mechanism to communicate the measurement results to the host computer

Chronos Research Prototype – National University of Singapore

- Input is a task written in C and the configuration of the target processor
- The front end of the tool performs data-flow analysis to compute loop bounds
  - If it fails to obtain certain loop bounds, user annotations have to be provided
  - The user may also input infeasible-path information
  - The front end maps this information from the source code to the binary executable
- The tool disassembles the executable to generate the CFG
- Performs processor-behavior analysis on this CFG
  - out-of-order pipelines
  - various dynamic branch prediction schemes
  - instruction caches
  - interaction among these different features

Chronos Research Prototype – Methods 1/2

- It determines upper bounds of execution times of each basic block under various execution contexts
  - such as correctly predicted or mispredicted jump of the preceding basic blocks
  - cache hits/misses within the basic block
- Determining these bounds is challenging for out-of-order processor pipelines because of the presence of timing anomalies
- It requires the costly enumeration of all possible schedules of instructions within a basic block
- Chronos avoids this enumeration via a fixed-point analysis of the time intervals at which the instructions enter/leave different pipeline stages

Chronos Research Prototype – Methods 2/2

- The analyzer employs ILP to bound the number of executions corresponding to each context
- This is achieved by bounding the number of branch mispredictions and instruction-cache misses

- The integration of cache and branch prediction requires analyzing the constructive and destructive timing effects because of cache blocks being "prefetched" along the mispredicted paths
  - This complex interaction is accounted for in the analyzer

- Finally, bounds calculation is implemented by the IPET technique
  - converting the loop bounds and user-provided infeasible-path information to linear-flow constraints

Chronos Research Prototype – Limitations

- Chronos currently does not analyze data caches
- Since the focus is mainly on processor-behavior analysis, the tool performs limited data-flow analysis to compute loop bounds
- The tool also requires user feedback for infeasible program paths

Chronos Research Prototype – Platforms

- Chronos supports the processor model of SimpleScalar sim-outorder simulator
- It is a popular cycle-accurate micro-architectural simulator
- The tool targets a simulated processor model so that the processor can be easily configured with different
  - Pipeline, branch prediction and instruction-cache options

- This allows the user to evaluate the efficiency, scalability, and accuracy of the WCET analyzer for various processor configurations without requiring the actual hardware

- The source code of the entire tool is publicly available
  - Allowing the user to easily extend and adapt it for new architectural features and estimation techniques

The Heptane Research Prototype – IRISA, Rennes, France

- HEPTANE is an open-source static WCET analysis tool released under GPL license
- The purpose is to obtain upper bounds for the execution times of C programs by a static analysis of their code
  - source code and binary code

- The tool analyzes the source and/or binary format depending on the calculation method is selected

The Heptane Research Prototype – Methods 1/3

- HEPTANE embeds a timing schema-based and an ILP-based method for bound calculation

- Timing-schema based
  - Produces quickly and safe, albeit overestimated, upper bounds
- ILP-based
  - Requires more computing power, but yields tighter results

- The two calculation methods cannot be used simultaneously

- The timing-schema method operates on the task's syntactic structure
  - Extracted from the source code
- The ILP-based method exploits the task's CFG
  - Extracted from the task's binary

The Heptane Research Prototype – Methods 2/3

- Finding the upper bound of a loop requires the knowledge of the maximum number of loop iterations
- HEPTANE requires the user to give this information through *symbolic annotations* in the source program
- Annotations are designed to support nonrectangular and even nonlinear loops
  - nested loops whose number of iterations arbitrarily depends on the counter variables of outer loops

- The final bound is computed using an external evaluation tool
  - Maple and Maxima, for the method based on timing schemata
  - lp solve and CPLEX, for the method based on ILP

The Heptane Research Prototype – Methods 3/3

- HEPTANE integrates mechanisms to take into account the effect of *instruction caches*, *pipelines*, and *branch prediction*
- Pipelines are tackled by an off-line simulation of the flow of instructions through the pipelines
- It classifies every instruction according to its worst-case behavior with respect to the instruction cache
  - Instruction categories take into account loop-nesting levels
  - The HEPTANE tool takes as input the memory map of the code
    - cacheable versus uncacheable code regions, address range of scratchpad memory
  - as well as the contents of locked cache regions if a cache-locking mechanism is used
- An approach derived from static cache simulation is used to integrate the effect of branch predictors
  - based on a cache of recently taken branches

The Heptane Research Prototype – Limitations

- No automatic flow analysis
  - loop bounds are given manually as annotations at the source code level
- No detection of mutually exclusive or infeasible paths
  - resulting in pessimistic upper bounds for some tasks
- The bound-calculation method based on timing schemata currently does not support compiler optimizations that cause a mismatch between the task's syntax tree and CFG
- No support for data-cache analysis
- Limited number and types of target processors (currently limited to scalar processors with in-order execution)
- Limited to the gcc compiler

The Heptane Research Prototype – Platforms

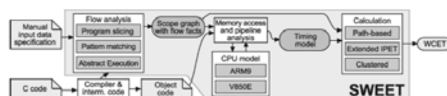
- HEPTANE is designed to produce timing information for in-order monoprocessor architectures
- Pentium 1 — accounting for one integer pipeline only
- StrongARM 1110
- Hitachi H8/300
- MIPS as a virtual processor with an overly simplified timing model

SWEET (SWEdish Execution-Time Tool)

- Previously developed in Mälardalen University, C-Lab in Paderborn, and Uppsala University
  - The development has now fully moved to Mälardalen University
- Modular fashion
  - Different analyzes and tool parts work rather independently
- It consists of three major phases:
  - *flow analysis*
  - *processor-behavior analysis*
  - *estimate calculation*
- The analyzes communicate through two well-defined data structures
  - the *scope graph with flow facts*
    - representing the result of the flow analysis
  - and the *timing model*
    - representing the result of the processor behavior analysis

SWEET (SWEdish Execution-Time Tool) – Architecture

- Architecture



SWEET (SWEdish Execution-Time Tool) – Functionality

- Automatic flow analysis on the intermediate code level
- Integration of flow analysis and a research compiler
- Connection between flow analysis and processor-behavior analysis
- Instruction cache analysis for level-one caches
- Pipeline analysis for medium-complexity RISC processors
- A variety of methods to determine upper bounds based on the results of flow and pipeline analysis

SWEET (SWEdish Execution-Time Tool) – Methods 1/6

- Unlike most WCET analysis tools, SWEET's flow analysis is integrated with a research compiler
- The flow analysis is performed on the intermediate code (IC) of the compiler
  - after structural optimizations
- The control structure of the IC and the object code is similar
- The flow-analysis results for the IC are valid for the object code as well

SWEET (SWEdish Execution-Time Tool) – Methods 2/6

- SWEET's flow analysis is based on a multiphase approach
- A *program slicing* is used to restrict the flow analysis to only those parts of the program that may affect the program flow
- A value analysis, combined with *pattern-matching* catches "easy cases"
  - such as simple loops
- More complicated codes are handled by the *abstract execution*
  - a form of symbolic execution based on abstract interpretation
- The analysis uses abstract interpretation to derive safe bounds on variables values at different points in the program
- However, rather than using traditional fixed-point iteration, loops are "rolled out" dynamically and each iteration is analyzed individually in a fashion similar to symbolic execution
- The abstract execution is able to automatically calculate both loop bounds and infeasible path information

SWEET (SWEdish Execution-Time Tool) – Methods 3/6

- Processor-behavior analysis is decoupled from the flow analysis
- It is based on a two-phase approach
- *memory-access analysis*
  - memory areas accessed by different instructions are determined
  - If the target hardware has an instruction cache, an instruction-cache analysis is performed
  - The result of the analysis is a set of "execution facts," which are used in the pipeline analysis
  - Such facts specify the memory area(s) that an instruction may reference
    - or if the instruction may hit and/or miss the cache
  - Execution facts can also specify other factors like
    - assumptions on branch prediction outcomes

SWEET (SWEdish Execution-Time Tool) – Methods 4/6

- *pipeline analysis*
- Performed by simulating object code sequences through a trace-driven cycle-accurate CPU model
- The CPU model needs to be controllable
  - execution facts can correctly be accounted for in each instruction
  - the instruction trace is followed as provided
- The execution facts are used to enforce worst-case timing behavior of the instructions
- For data-dependent instructions, the worst-case timing is assumed, unless execution facts specify otherwise
- The pipeline analysis has been explicitly designed to allow standard CPU simulators to be used as CPU models
- However, this requires that the simulator
  - is clock-cycle accurate
  - can be forced to perform its simulation according to given instruction sequences
  - can be forced to perform its simulation according to given execution facts
  - does not suffer from timing anomalies

SWEET (SWEdish Execution-Time Tool) – Methods 5/6

- Consecutive simulation runs
  starting with the same basic block in the code
  are combined
  to find timing effects across sequences of two or more blocks
- The analysis assumes that there is a known upper bound on the length of block sequences that can exhibit timing effects
- This value can be greater than two even on quite simple processors

SWEET (SWEdish Execution-Time Tool) – Methods 6/6

- The estimate calculation phase support three different type of calculation techniques
  - all taking the same two data structures as input
- A *fast path-based* technique
- A *global IPET* technique
- A hybrid *clustered* technique
- The clustered calculation can perform both local IPET and/or local path-based calculations
  - the decision on what to use is based on the flow information available
- DOT from GraphViz is used to graphically visualize the results

SWEET (SWEdish Execution-Time Tool) – Limitations

- The flow analysis can handle ANSI-C programs including pointers, unstructured code, and recursion
  - But the program must be compiled with the integrated research compiler
  - otherwise flow facts must be manually given
- The use of dynamically allocated memory is currently not supported
  - annotations will be needed in such cases
- The current memory access analysis does not handle data caches
- Only one level instruction caches are supported
- Limited to in-order pipelines with bounded long timing effects and no timing anomalies
  - out-of-order pipelines are not handled
- The path-based bound calculation requires the code to be well structured
  - The IPET-based and clustered calculation methods can handle arbitrary task graphs

SWEET (SWEdish Execution-Time Tool) – Platforms

- SWEET's processor-behavior analysis currently supports:
- ARM9
- NEC V850E
- The V850E model has been validated against actual hardware,
  - which the ARM9 has not

Research Prototype from Chalmers University of Technology, Göteborg, Sweden

- The developed tool is capable of automatically deriving safe upper bounds for tasks' binaries using a subset of the Power-PC instruction-set architecture
- It is sometimes possible to derive the exact WCET, in case the worst-case input of the task is known

Chalmers UT Prototype – Functionality

- Integration of path and timing analysis through symbolic cycle-level execution of the task on a detailed architectural simulation model extended to handle unknown input data values
- Binary code-transformation techniques that eliminate timing anomalies
- A data cache analysis method that can identify data structures that can be safely cached (predictable) so as to improve the worst-case cache performance
  - data structures with unpredictable access patterns are identified and tagged as non-cacheable
- A method that can determine the worst-case data-cache performance for data accesses to predictable data structures whose exact location in the address space is statically unknown
  - This method can be applied to improve worst-case cache performance
  - Procedures whose input parameters are pointers to data structures whose access patterns are predictable,
    but whose locations in the memory space are not known until runtime

Chalmers UT Prototype – Methods 1/2

- Loop bounds, branch conditions, etc
  - which are not input data dependent
  - will be calculated as the task is executed symbolically on the architectural simulator
- In order to handle unknown input data, the instruction-set simulator is extended with the capability of handling unknown parts of execution states
- For example,
  - if a branch condition depends on unknown input data
  - both paths are executed
- Inevitably, this may result in path-explosion problem in program loops with a large number of iterations

Chalmers UT Prototype – Methods 2/2

- A path-merging approach excludes the paths that may not be part of the worst-case execution path through the task
- The analysis that determines which paths to exclude must take into account what timing effect they may have in the future
- This involves analysis of worst-case performance taking micro-architectural features, such as pipelining and caching, into account
- Methods have been developed and integrated to perform this analysis for a range of architectural features, including
  - multilevel instruction and data caches
  - multiple-issue pipelines with dynamic instruction scheduling
- How often path merging is carried out is a trade-off between computational speed and accuracy of the analysis

Chalmers UT Prototype – Limitations

- The user has to provide annotations for loops with unknown termination properties
- The computational complexity of the analysis
- While the path-merging method partly addresses this concern
  it also introduces overestimation of the WCET

Chalmers UT Prototype – Platforms

- It currently supports a fairly rich subset of the PowerPC instruction set architecture
- Also have integrated architectural timing models for some implementations of the PowerPC featuring
  - dual-instruction issue
  - a dynamically scheduled pipeline
  - a parameterized cache hierarchy with split first-level set-associative instruction and data caches
  - a unified second-level cache
- The cache and block size as well as the associativity of each cache is parameterized

SymTA/P Tool of TU Braunschweig, Germany

- SymTA/P is an acronym of SYMbolic Timing Analysis for Processes
- The purpose of SymTA/P is to obtain upper and lower execution time bounds of C programs running on microcontrollers
- The key idea of SymTA/P is to combine
  - Platform independent path analysis on source code level
  - Platform-dependent measurement methodology on object code level using an actual target system
- The main benefit is that this hybrid analysis can easily be retargeted to a new hardware platform

SymTA/P Tool of TU – Functionality

- The execution time measurement can be obtained
  - by an off-the-shelf cycle-accurate processor simulator or
  - by an evaluation board
- Instruction cache as well as data-cache behavior is analyzed for a single uninterrupted task execution
- The cache-related preemption delay for fixed priority preemptive real-time systems for direct mapped and associative caches is analyzed and integrated in a cache-aware response time analysis
- To be efficient, cache analysis requires that a task execution trace can be generated that is uninterrupted by cache misses because
  - the task is small enough to fit in the cache or
  - the evaluation system offers sufficient on-chip fast memory (scratchpad RAM)
- If that is not available
  - then an appropriate simulator must be used instead

SymTA/P Tool of TU – Methods 1/7

- The hybrid approach consists of
  - a static program path analysis and
  - a measurement for the execution time of program segments
- In most static analyzes of execution times,
  a basic block has been assumed as the smallest entity
- However often a program consists of a single feasible execution path (SFP) only
- Such a SFP is a sequence of basic blocks where the execution sequence is invariant to input data

SymTA/P Tool of TU – Methods 2/7

- SymTA/P uses symbolic analysis on the abstract syntax tree to identify such single feasible paths (SFP) at the source code level
- The result is a CFG with nodes containing
  - single feasible paths or
  - basic blocks that are part of a multiple feasible path
- A single feasible path can reach beyond basic block boundaries, for example, a fast Fourier transformation or a FIR filter
  - The program contains loops with several if-then-else statements, which are input independent
  - This means that the branch direction depends only on local variables with known values
    - for example, the loop iteration count
  - Therefore, the entire loop represents an SFP and is represented by a single node
  - The main benefit of SFPs is a smaller number of instrumentation points

SymTA/P Tool of TU – Methods 3/7

- In a second step, the execution time for each node is estimated
- Off-the-shelf processor simulators or standard cost-efficient evaluation boards can be used
- The C source code is instrumented with measurement points that mark the beginning and the end of each node
- Such a measurement point is linked to a platform-dependent measurement procedure, such as accessing the system clock or internal timer
- For a processor simulator the instrumentation can use a processor debugger command to stop the simulation and store the internal system clock

- The entire C-program with measurement points is compiled, linked, and executed on the evaluation board (or simulated on the processor simulator)
- During this measurement, a safe initial state cannot be assured in all cases
  - An additional time delay is added that covers a potential underestimation during such a measurement

SymTA/P Tool of TU – Methods 4/7

- At this step, SymTA/P assumes that each memory access takes constant time
- For timing analysis, input data for a complete branch coverage must be supplied
- A complete branch coverage means that with a given set of input data all branches and other C statements are at least executed once
- This criterion requires the user to specify a considerably fewer number of input data than a full path coverage
  - because combinations of execution paths need not be considered
- This advantage comes with the drawback that it adds a conservative overhead to cover pipelining effects between nodes

SymTA/P Tool of TU – Methods 5/7

- The constant memory access time assumption is revised by analyzing the instruction-cache and the data-cache behavior
- When using a processor simulator, the memory-access trace for each node is generated using a similar methodology as the execution time measurements
- The traced memory accesses are annotated to the corresponding node in the CFG
- A dataflow analysis technique is used to propagate the information
  - which cache lines are available at each node
- Pipelining effects between nodes are not directly modeled
- Instead, a conservative overhead corresponding to starting with an empty pipeline is assumed

SymTA/P Tool of TU – Methods 6/7

- The longest and shortest path in the CFG are found by IPET
- The time for each node is given by the measured execution time and the statically analyzed cache-access behavior
- This framework has also been used to calculate the power consumption of a program
- If the loop condition is input-dependent, loop bounds have to be specified by the user

SymTA/P Tool of TU – Methods 7/7

- If preemptive scheduling is used, cache blocks might be replaced by higher priority tasks
- SymTA/P considers the cache behavior of the preempted and preempting task to compute the maximum cache-related preemption delay
- This delay is considered in cache-aware response time analysis

- SymTA/P uses a static analysis approach for data-cache behavior
  - Combines symbolic execution to identify input-dependent memory accesses
  - Uses ILP to bound the worst-case data-cache behavior for input-dependent memory accesses

SymTA/P Tool of TU – Limitations

- The measurement on an evaluation board is more accurate if the program paths between measurements points are longer
- If many basic blocks are measured individually
  - the added time delays to cover pipelining effects would lead to an overestimation of the total worst-case execution time

- Data-dependent execution times of single instructions are not explicitly considered
- It is assumed that input data covers the worst-case regarding data-dependent instruction execution time
- Input data has to be provided that generates complete branch coverage
  - Such patterns are usually available from a functional test
- The precision of the final analysis depends on the measurement environment
- Especially for evaluation boards, the interference of the measurement instrumentation has to be a constant factor to obtain sufficiently accurate results
- The approach assumes a sequential memory access behavior where the CPU stalls during a memory access

SymTA/P Tool of TU – Platforms

- C programs on the following microcontrollers have been analyzed:

- ARM architectures (ARM9)
- TriCore
- StrongARM
- C167
- I8051

- The software power analysis has been applied to SPARClite
- An open interface for processor simulators and a measurement framework for evaluation boards is provided

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 – Methods

- It works on a tree representation of the program
- The structure is derived from either
  - the source code
  - or from the direct analysis of executables
- The timing of individual blocks is derived from extensive measurements extracted from the real system

- The WCET estimates are computed using an algebra of probability distributions

- The timing analysis of the program can be performed on different contexts
  - allowing to individually analyze, for instance, each different call to a function

- The level of detail and how many contexts are analyzed is controlled by annotations

- RapiTime also allows to analyze different loop iterations
  - Loop bounds are derived from actual measurements (or annotations)

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

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

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

TOOL COMPARISON 1/4

Table II. Analysis Methods Employed

Tool	Flow	Processor behavior	Bound calculation
aiT	Value analysis	Static program analysis	IPET
Bound-T	Linear loop-bounds and constraints by Omega test	Static program analysis	IPET per function
RapiTime	n.a.	Measurement	Structure-based
SymTA/P	Single feasible path analysis	Static program analysis for I/D cache, measurement for segments	IPET
HEPTANE	—	Static prog. analysis	Structure-based, IPET
Vienna S.	—	Static program analysis	IPET
Vienna M.	Genetic algorithms	Segment measurements	n.a.
Vienna H.	Model checking	Segment measurements	IPET
SWEET	Value analysis, abstract execution, syntactical analysis	Static program analysis for instr. caches, simulation for the pipeline	Path-based, IPET-based, clustered
Florida		Static program analysis	Path-based
Chalmers		Modified simulation	
Chronos		Static prog. analysis	IPET

Tool Comparison 2/4

- Data caches need a resolution of effective memory addresses at analysis time
- Out-of-order execution almost unavoidably introduces timing anomalies
  - which require integrated analyzes of the cache and the pipeline behavior

Table III. Support for Architectural Features

Tool	Caches	Pipeline	Periphery
aiT	I/D, direct/set associative, LRU, PLRU, pseudo round robin	In-order/out-of-order	PCI bus
Bound-T	—	In-order	—
RapiTime	n.a.	n.a.	n.a.
SymTA/P	I/D, direct/set-associative, LRU	n.a.	n.a.
HEPTANE	I-cache, direct, set associative, LRU, locked caches	In-order	—
Vienna S.	Jump-cache	Simple in-order	—
Vienna M.	n.a.	n.a.	n.a.
Vienna H.	n.a.	n.a.	n.a.
SWEET	I-cache, direct/set associative, LRU	In-order	—
Florida	I/D, direct/set associative	In-order	—
Chalmers	Split first-level set-associative, unified second-level cache	Multi-issue superscalar	—
Chronos	I-cache, direct, LRU	In-order/out-of-order, dyn. branch prediction	—

Tool Comparison 3/4

Tool	Language level	Result representation	Maximal analyzed	Integration
aiT	Object	Text, graphical	80 KB	Some versions adapted to code generated by STATEMATE, Ascet/SD, Scade, MATLAB/Simulink, or Targetlink
Bound-T	Object	Text, graphical	30 KB	Sack-extent analysis, HRT schedulability analyzer
RapiTime	Source (C, Ada), object	Graphical, html based	50 KLOC	Matlab/Simulink, Targetlink, SimpleScalar simulator
SymTA/P	Source (C)	Text, graphical	7 KLOC	Tasking compiler, schedulability analysis (cache-related preemption delay)
HEPTANE	Source, object	Graphical	14 KLOC	
Vienna S.	Source, object	Text, graphical		Matlab/Simulink, optimizing compiler
Vienna M.	Object	Text		
Vienna H.	Source, object	Text, graphical		Matlab/Simulink, Targetlink
SWEET	Flow analysis on intermediate, proc. beh. anal. on object	Text, graphical		IAR, SUIF, and LCC compilers
Florida	Object			Cycle-accurate simulators, power-aware schedulers, compiler
Chalmers	Object			
Chronos	Object	Graphical	10 KB	GCC compiler, SimpleScalar simulator

Tool Comparison 4/4

Table V. Supported Hardware Platforms

Tool	Hardware platform
aiT	Motorola PowerPC MPC 555, 565, and 755, Motorola ColdFire MCF 5307, ARM7 TDMI, HCS12/STAR12, TMS320C33, C166/ST10, Renesas M32C/85, Infineon TriCore 1.3
Bound-T	Intel-8051, ADSP-21020, ATMEL ERC32, Renesas H8/300, ATMEL AVR, ATmega, ARM7
RapiTime	Motorola PowerPC family, HCS12 family, ARM, NecV850, MIPS3000
SymTA/P	Various ARM (RealView Suite), TriCore, i8051, C167
Heptane	Pentium1, StrongARM 1110, Hitachi H8/300
Vienna S.	M68000, M68360, C167
Vienna M.	C167, PowerPC
Vienna H.	HCS12, Pentium
SWEET	ARM9 core, NEC V850E
Florida	MicroSPARC I, Intel Pentium, StarCore SC100, Atmel Atmega, PISA/MIPS
Chalmers	PowerPC
Chronos	SimpleScalar out-of-order processor model with MIPS-like instruction-set architecture (PISA)

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
- Strong guarantees for correctness can be given by equivalence checking between different abstraction levels
- 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 also 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
- A new scheduling approach considering the combination of timing analysis and preemptive scheduling will have to be developed
- SymTA/P provides an integration of WCET calculation and cache related preemption delay for schedulability analysis

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.
- Anantaraman et al. [2003] propose a virtual simple architecture (VISA)
- A VISA is the pipeline timing specification of a hypothetical simple processor
- Upper bounds for execution times are derived for a task assuming the VISA
- At runtime, the task is executed speculatively on an unsafe complex processor
  - Its progress is continuously gauged
- If continued safe progress appears to be in jeopardy, the complex processor is reconfigured to a simple mode of operation that directly implements the VISA
  - thereby explicitly bounding the task’s overall execution time
- VISA shifts the burden of bounding the execution times of tasks, in part, to hardware by supporting two execution modes within a single processor

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

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

- Examples are the Sun Niagara chip
  - combines eight in-order four-way multithreaded cores on a single chip
- the IBM-designed PowerPC for the Xbox 360
  - using three two-way multithreaded in-order cores
- 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 many 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

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