Lecture: Timing Analysis

Jan Vitek

ECE/CS

Spring 2011

Slides based on material by Edward Lee, Sanjit Seshia and Reinhard Wilhelm

Reading List

- Mandatory Reading
  - Chapter 15 of CPS textbook
- Optional Reading
  - *The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools*, R. Wilhelm et al., ACM Transactions on Embedded Computing Systems, 2007
  - *Performance Analysis of Real-Time Embedded Software* Y-T. Li and S. Malik, Kluwer Academic Pub., 1999

Quantitative Analysis

- Embedded systems are often characterized by strong bounds on available resources
  - While this is the case for all computer systems, embedded systems designers are more keenly aware of the limitations of the platform
  - Errors in embedded applications can have more serious consequences because they are often critical systems
  - Embedded systems are often times difficult to patch once deployed in the field
- To be considered correct embedded system must be shown to meet all of their quantitative constraints
  - A *quantitative property* is an property of a system that can be measured
  - Example of quantitative properties include: execution time, memory usage, bandwidth, response time, latency, power consumption
  - Quantitative constraints can be imposed by the *platform* or the *problem domain*
    - the MSP430 imposes strong memory constraints on embedded application due to the small amount of on board memory
    - airbags must deploy in less than 10 msecs, thus imposing a timing constraint on the software controlling them
- Quantitative analysis answers the question wether a system meets its constraints

Quantitative Analysis

- Given a program P the goal of quantitative analysis is to compute

$$q = f_P(x)$$

where f_P is a function that compute the resource usage of program P when given input x

- *Extreme-case analysis* looks at extremal values of q

$$\max_x f_P(x)$$

is the largest value of q for any input x

$$\min_x f_P(x)$$

is the smallest value of q for any input x

- *Threshold analysis* answers the question whether q is always bounded by some threshold T

$$\forall x,\ f_P(x) \le T \quad \text{upper-bound}$$

$$\forall x,\ T \le f_P(x) \quad \text{lower-bound}$$

Quantitative Analysis

- f_P is not always computable, quantitative analysis must provide an approximation of the true value
- If the computed value is equal to the true value of f_P then it is said to be a tight bound, otherwise it is called a loose bound
- A *safe* approximation of q is a value Q such that

$$\max_x f_P(x) \le Q$$

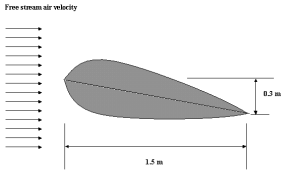
is the largest value of q for any input x

$$Q \le \min_x f_P(x)$$

Industrial Needs

Side airbag in car,
Reaction in <10 mSec

Wing vibration of airplane,
sensing every 5 mSec

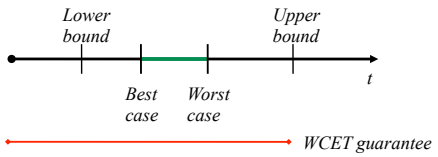


Timing Analysis

- *Latency* and *response time* are key properties of embedded systems
- These quantitative properties require *timing analysis* of the software, the hardware and the execution environment
- The most common property is *Worst Case Execution Time* (WCET)
- WCET is an extremum analysis of the timing property of the system

- Questions:
  - Is WCET related to algorithmic complexity?
  - Is WCET computed over source code or machine code?
  - Why does the architecture matter?
  - What are other factors that affect timing properties?
  - Does computing WCET of an entire application make sense?

Basic Notions



Goal

- Given the code for a software task and the platform (OS+HW) on which the task will run, determine the WCET of the task
- Upper bounds must be *safe*
  - i.e. not underestimated
- Upper bounds should be *tight*
  - i.e. not far from real execution times
- Computational effort must be *tolerable*

Program Model

10

- A **basic block** is a sequence of consecutive statements in which the flow of control enters only at the beginning and leaves at the end, without halt or the possibility of branching except at the end

- A **control-flow graph** (CFG) of a program *P* is a directed graph *G = (V,E)*, where the set of vertices *V* comprises basic blocks of *P*, and the set of edges *E* indicates the flow of control between basic blocks

modexp

11

```
1. #define EXP_BITS 32

3. typedef unsigned int UI;

5. UI modexp(UI base, UI exponent, UI mod) {
6.   int i;
7.   UI result = 1;

9.   i = EXP_BITS;
10.   while(i > 0) {
11.     if ((exponent & 1) == 1) {
12.        result = (result * base) % mod;
13.     }
14.     exponent >>= 1;
15.     base = (base * base) % mod;
16.     i--;
17.  }
18.  return result;
19. }
```

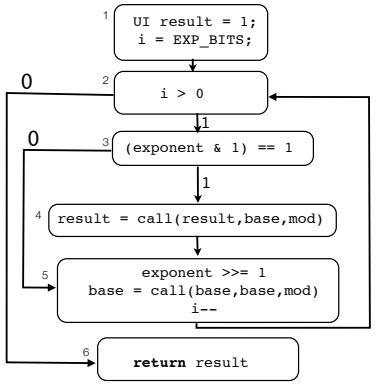
modexp - basic blocks

12

```
1. #define EXP_BITS 32

3. typedef unsigned int UI;

5. UI modexp(UI base, UI exponent, UI mod) {
6.   int i;
7.   UI result = 1;

9.   i = EXP_BITS;
10.   while(i > 0) {
11.     if ((exponent & 1) == 1) {
12.        result = (result * base) % mod;
13.     }
14.     exponent >>= 1;
15.     base = (base * base) % mod;
16.     i--;
17.  }
18.  return result;
19. }
```

modexp - CFG

13



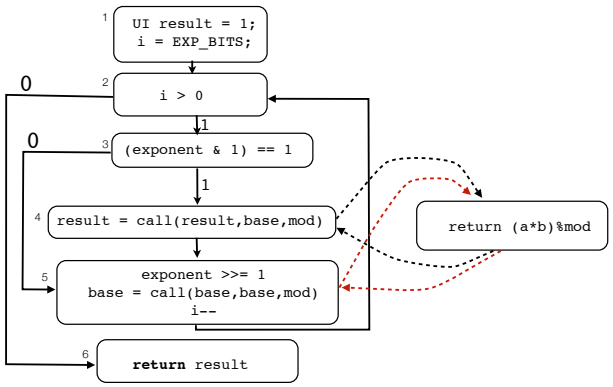
modexp-call

14

```
1. #define EXP_BITS 32

3. typedef unsigned int UI;

5. UI modexp-call(UI base, UI exponent, UI mod) {
6.   int i;
7.   UI result = 1;

9.   i = EXP_BITS;
10.   while(i > 0) {
11.     if ((exponent & 1) == 1) {
12.        result = call(result,base,mod);
13.     }
14.     exponent >>= 1;
15.     base = call(base,base,mod);
16.     i--;
17.  }
18.  return result;
19. }

21. UI call(UI a, UI b, UI mod) { return (a*b)%mod; }
```

modexp-call - CFG

15



Loop bounds

16

- Establishing loop bounds (and bounds on recursive calls) is a pre-condition to timing analysis
  - In general the problem is undecidable
- Programmers must either write code that is analyzable or provide additional information to enable analysis
  - There must be a progress measure that maps the state of the program to a well order

Loop bounds

17

```
1. void iter( int* p, int* q) {
2.     while (*p >=0) {
3.        *p--;
4.        *q++;
5.     }
6. }
```

Loop bounds

18

```
1. UI modexp1(UI base, UI exponent, UI mod) {
2.   UI result = 1; int i;
3.   for(i=EXP_BITS; i > 0; i--) {
4.     if ((exponent & 1) == 1)
5.       result = (result * base) % mod;
6.     exponent >>= 1;
7.     base = (base * base) % mod;
8.   }
9.   return result;
10. }
```

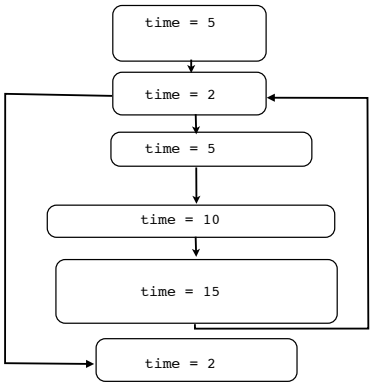
- Using a for loop with constant bounds and monotonically decreasing loop variable simplifies analysis

Loop bounds


```
1. UI modexp2(UI base, UI exponent, UI mod) {
2.   UI result = 1; int i;
3.   while(exponent!=0) {
4.     if ((exponent & 1) == 1)
5.       result = (result * base) % mod;
6.     exponent >>= 1;
7.     base = (base * base) % mod;
8.   }
9.   return result;
10. }
```

- Is the above bounded?

Computing execution time



Exponential Path Space

- Execution time is a path property. Time taken by a program is a function of how conditional statements evaluate to true or false.

```
1. #define SZ 100
2. int a[SZ][SZ];
3. int p, n;
4. void count() {
5.   int o, i;
6.   for (o = 0; o < SZ; o++)
7.     for (i = 0; i < SZ; i++)
8.       if (a[o][i] >= 0)
9.         p += a[o][i];
10.       else
11.         n += a[o][i];
12. }
```

- There are 2^{10000} paths in the function above
- Enumerating them all is impractical

Path Feasibility

- A path p in program P is said to be feasible if there exists an input x to P such that P executes p on x

```
1. void altitude_control_task(void) {
2.   if (mode == 2 || mode == 3)
3.     if (vmode == 3) {
4.       float err = estimator - desired_altitude;
5.       climb = pre_climb + altitude_gain * err;
6.       if (climb < -1)
7.         climb = -1;
8.       if (climb > 1)
9.         climb = 1;
10.    }
11. }
```

- There are 11 paths in the function above
- Only 9 are feasible

Optimization Formulation

- Given a program P, let $G = (V,E)$ denote its CFG.
  - $n = |V|$ is the number of basic blocks in G,
  - $m = |E|$ is the number of edges
- We refer to the basic blocks by their index i
- Assume the CFG has a unique source node $s = 1$, and end node $t = n$
- x_i is the number of times basic block i is executed, the execution count
- $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is a vector of execution counts
- $\mathbf{x}$ is valid if its elements correspond to an execution of P

Optimization Formulation

- The problem is an instance of network flow
  - *Unit Flow at Source:* The control flow from source node to sink node is a single execution:

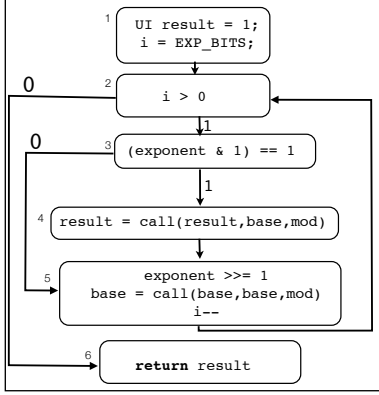
$$x_1 = 1$$
$$x_n = 1$$

  - *Conservation of Flow*: For each node i, the incoming flow from predecessor nodes equals the outgoing flow to successors
  - let d_{ij} denote the number of times the edge from node i to j is executed

$$x_i = \sum_{j \in pred(i)} d_{ij} = \sum_{j \in succ(i)} d_{ij}$$

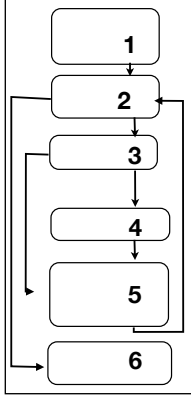
Example

25



Example

26

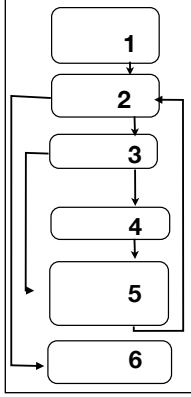


$x_1 = 1$
$x_6 = 1$

$x_1 = d_{12}$
$x_2 = d_{12} + d_{52} = d_{23} + d_{26}$
$x_3 = d_{23} = d_{34} + d_{35}$
$x_4 = d_{34} = d_{45}$
$x_5 = d_{35} + d_{45} = d_{52}$
$x_6 = d_{26}$

Example

27



$x_1 = 1$
$x_6 = 1$

$x_1 = d_{12}$
$x_2 = d_{12} + d_{52} = d_{23} + d_{26}$
$x_3 = d_{23} = d_{34} + d_{35}$
$x_4 = d_{34} = d_{45}$
$x_5 = d_{35} + d_{45} = d_{52}$
$x_6 = d_{26}$

a solution:
$x_1=d_{12}=1$, $x_2=2$, $d_{23}=1$, $x_3=1$, $d_{34}=0$, $d_{35}=1$,
$x_4=d_{45}=0$, $x_5=d_{52}=1$, $x_6=d_{26}=1$

Overall Optimization Problem

28

- We can now formulate the WCET for a CFG
- Assume the upper bound on each block's execution time is w_i
- The WCET is

$$\max_{xi} \sum_{i=1..n} w_i x_i$$

with constraints

$$x_1 = x_n = 1$$

$$x_i = \sum_{j \in pred(i)} d_{ij} = \sum_{j \in succ(i)} d_{ij}$$

This problem is a form of linear programming solvable in polynomial time

Loop bounds and Infeasible Paths

29

- Dealing with loop bounds can be done by adding constraints of the form $x_3 < 32$ to indicate that certain blocks can be only executed less than the bound times
- Adding infeasible paths boils down to constraints on edges such as $d_{12} + d_{34} \leq 1$
  - The above is without loops. With loops you have to bound it by the loop count/
- The problem becomes an instance of integer linear programming which is NP-hard.

Bounds for basic blocks

30

- The bounds w_i for basic blocks require looking at the cost of every instruction in the block

- Use the MSP430 manual to determine cycle counts

Table 5-10. MSP430 Format I Instructions Cycles and Length

Addressing Mode		No. of Cycles	Length of Instruction	Example
Source	Destination			
Rn	Rm	1	1	`MOV R5,R8`
	PC	3	1	`BR R9`
	x(Rm)	4[1]	2	`ADD R5,4(R6)`
	EDE	4[1]	2	`XOR R8,EDE`
	&EDE	4[1]	2	`MOV R5,&EDE`
@Rn	Rm	2	1	`AND @R4,R5`
	PC	4	1	`BR @R8`
	x(Rm)	5[1]	2	`XOR @R5,8(R6)`
	EDE	5[1]	2	`MOV @R5,EDE`
	&EDE	5[1]	2	`XOR @R5,&EDE`
@Rn+	Rm	2	1	`ADD @R5+,R6`
	PC	4	1	`BR @R8+`

How to measure running time

- Several techniques, with varying accuracy:
- Instrument code to sample CPU cycle counter
  - relatively easy to do, read processor documentation for assembly instruction
- Use cycle-accurate simulator for processor
  - useful when hardware is not available/ready
- Use Logic Analyzer
  - non-intrusive measurement, more accurate

Cycle Counters

- Most modern systems have built in registers that are incremented every clock cycle
- Special assembly code instruction to access
  - On Intel 32-bit x86 machines since Pentium:
    - 64 bit counter
  - RDTSC instruction (ReaD Time Stamp Counter) sets %edx register to high order 32-bits, %eax register to low order 32-bits
- Wrap-around time for 2 GHz machine
  - Low order 32-bits every 2.1 seconds
  - High order 64 bits every 293 years

Measuring with Cycle Counter

- Add code to record start/end times for basic blocks

```
1. static unsigned cyc_hi = 0, cyc_lo = 0;
2. void start_counter() {
3.   access_counter(&cyc_hi, &cyc_lo);
4. }
```

- GCC allows inline assembly code with mechanism for matching registers with program variables (code for x86)

```
5. void access_counter(unsigned *hi, unsigned *lo) {
6.    asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
7.        : "=r" (*hi), "=r" (*lo)
8.        : /* No input */
9.        : "%edx", "%eax");
10. }
```

Measuring with Cycle Counter

- Perform double precision subtraction to get elapsed cycles
- Express as `double` to avoid overflow problems

```
1. double get_counter() {
2.   unsigned nhi, nlo, hi, lo, borrow;
3.   access_counter(&nhi, &nlo);
4.   lo = nlo - cyc_lo;
5.   borrow = lo > nlo;
6.   hi = nhi - cyc_hi - borrow;
7.   return (double) hi * (1 << 30) * 4 + lo;
8. }
```

- Pitfalls
  - Instrumentation incurs small overhead
    - measure long enough code sequence to compensate
  - Cache effects can skew measurements
    - "warm up" the cache before making measurement
  - Multi-tasking effects: counter keeps going even when the task is inactive
    - take multiple measurements and pick "k best" (cluster)
  - Ensure that task is 'locked' to a single core

Dealing with Modern Hardware

- Modern processors increase performance by using:
  - *Caches*
  - *Pipelines*
  - *Branch Prediction*
  - *Speculation*
  - ...
- These features make WCET computation difficult:
  Execution times of instructions vary widely
  - Best case: everything smooth
    - no cache miss, operands ready, needed resources free, branch correctly predicted
  - Worst case: everything wrong
    - all loads miss the cache, resources needed are occupied, operands are not ready
    - Jitter may be several hundred cycles

Timing Accidents and Penalties

- *Timing Accident*
  - cause for an increase of the execution time of an instruction
- *Timing Penalty*
  - the associated increase

- Types of timing accidents
  - Cache misses
  - Pipeline stalls
  - Branch mispredictions
  - Bus collisions
  - Memory refresh of DRAM
  - TLB miss

Penalties for Memory Access

37

cache miss	cm = 40
cache miss + write back	cm + wb = 80 (wb = 40)
TLB-miss and loading	tlb = 12 cm + 1 wb = 520
Memory-mapped I/O	mm = 800
Page fault	pf = 2000

Penalties have to be assumed for uncertainties!
Tendency increasing, since clocks are getting faster faster than everything else

How to Deal with Murphy's Law?

38

- Three answers:
- Accept
  - Every timing accident that may happen will happen
- Fight
  - Bound timing accidents
- Cheat
  - Monitor enough runs to get a good feeling

Accepting Murphy's Law

39

like guaranteeing a speed of 4.07 km/h for this car

because of the variability of execution times on modern processors

Cheating to deal with Murphy's

- Measuring "enough" runs to feel comfortable
  - How many runs are enough?
  - Example: Testing vs. Verification
    - AMD was offered a verification of the K7.
    - They had tested the design with 80 000 test vectors, considered verification unnecessary.
    - Verification attempt discovered 2 000 bugs!

The only remaining solution: Fighting Murphy's Law!

Execution Time is History-Sensitive

- Contribution executing an instruction to a program's execution time depends on the execution state, i.e., on the execution so far, i.e., cannot be determined in isolation

Deriving Run-Time Guarantees

- Static Program Analysis derives Invariants about all execution states at a program point.
- Derive Safety Properties from these invariants:
  - Certain timing accidents will never happen.
  - Example: At program point p, instruction fetch will never cause a cache miss.
- The more accidents excluded, the lower the upper bound
  - (and the more accidents predicted, the higher the lower bound).

Natural Modularization

43

- Processor-Behavior Analysis:
  - Uses Abstract Interpretation [Cousot&Cousot, 77]
  - Excludes as many Timing Accidents as possible
  - Determines upper bound for basic blocks (in contexts)
- Bounds Calculation
  - Maps control flow graph to an integer linear program
  - Determines upper bound and associated path

Abstract Interpretation (AI)

44

- Semantics-based method for static program analysis
- Basic idea of AI:
  - *Perform the program's computations using abstract values in place of the concrete values, start with a description of all possible inputs*
- AI supports correctness proofs

Abstract Interpretation

45

- abstract domain
  - complete semilattice,
    - related to concrete domain by abstraction and concretization functions,
    - e.g. intervals of integers (including -∞, ∞) instead of integer values
- abstract transfer functions for each statement type
  - abstract versions of their semantics,
    - e.g. arithmetic and assignment on intervals
- a join function combining abstract values from different control-flow paths
  - lub on the lattice
    - e.g. “union” on intervals

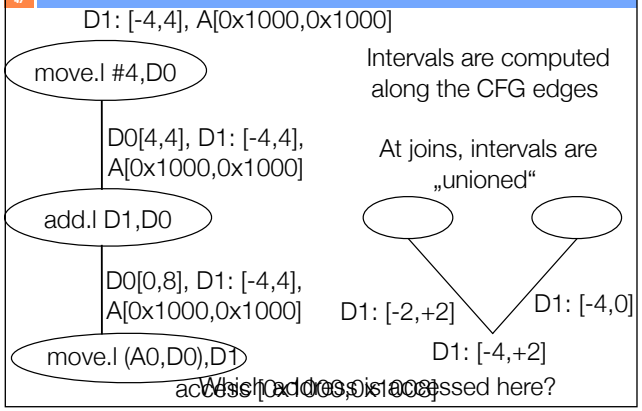
Value Analysis

46

- Motivation:
  - Provide access information to data-cache/pipeline analysis
  - Detect infeasible paths
  - Derive loop bounds
- Method:
  - calculate intervals, i.e. lower and upper bounds for the values occurring in the machine program (addresses, register contents, local and global variables)

Value Analysis

47



Intervals are computed along the CFG edges

At joins, intervals are „unioned“

Which address is accessed here?

Value Analysis (Airbus Benchmark)

48

Task	Unreached	Exact	Good	Unknown	Time [s]
1	8%	86%	4%	2%	47
2	8%	86%	4%	2%	17
3	7%	86%	4%	3%	22
4	13%	79%	5%	3%	16
5	6%	88%	4%	2%	36
6	9%	84%	5%	2%	16
7	9%	84%	5%	2%	26
8	10%	83%	4%	3%	14
9	6%	89%	3%	2%	34
10	10%	84%	4%	2%	17
11	7%	85%	5%	3%	22
12	10%	82%	5%	3%	14

Good means less than 16 cache lines

Caches: Fast Memory on Chip

49

- Caches are used, because
  - Fast main memory is too expensive
  - The speed gap between CPU and memory is too large and increasing
- Caches work well in the average case:
  - Programs access data locally (many hits)
  - Programs reuse items (instructions, data)
  - Access patterns are distributed evenly across the cache

Caches: How the work

50

- CPU wants to read/write at memory address a,
  - sends a request for a to the bus
- Cases:
  - Block m containing a in the cache (hit):
    - request for a is served in the next cycle
  - Block m not in the cache (miss):
    - m is transferred from main memory to the cache,
    - m may replace some block in the cache,
    - request for a is served asap while transfer still continues
  - Several replacement strategies: LRU, PLRU, FIFO,... determine which line to replace

Cache Analysis

51

- How to statically precompute cache contents:
- Must Analysis:
  - For each program point (and calling context), find out which blocks are in the cache
- May Analysis:
  - For each program point (and calling context), find out which blocks may be in the cache
  - Complement says what is not in the cache

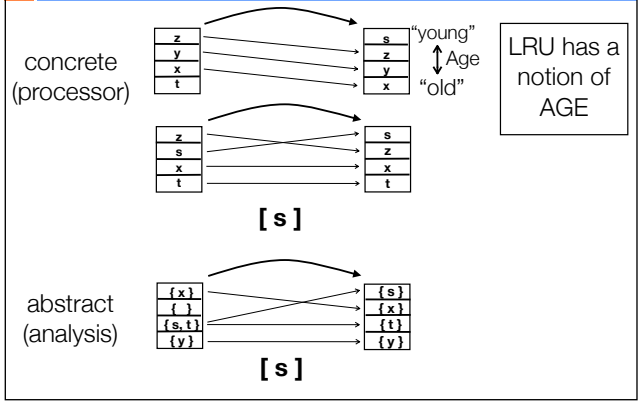
Must-/May- Cache Information

52

- Must Analysis determines safe information about cache hits
  - Each predicted cache hit reduces upper bound
- May Analysis determines safe information about cache misses
  - Each predicted cache miss increases lower bound

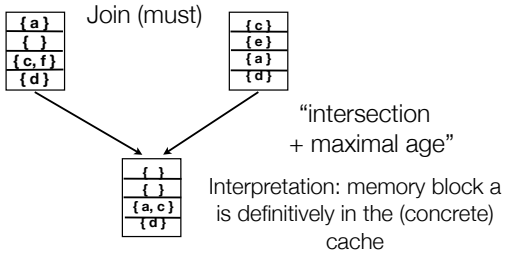
Cache with LRU (Must)

53



Cache Analysis: Join (must)

54



"intersection
+ maximal age"

Interpretation: memory block a is definitively in the (concrete) cache
=> always hit

Timing Predictability

55

- Experience has shown that the precision of results depend on system characteristics both
  - of the underlying hardware platform and
  - of the software layers
- System characteristics determine
  - size of penalties – hardware architecture
  - analyzability – HW architecture, programming language, SW design
- Many "advances" in computer architecture have increased average-case performance at the cost of worst-case performance

Conclusion

56

- The determination of safe and precise upper bounds on execution times by static program analysis essentially solves the problem
- Usability, e.g. need for user annotation, can still be improved
- Precision greatly depends on predictability properties of the system
- Integration into the design process is necessary