Static WCET Analysis: Methods and Tools

Timo Lilja

April 28, 2011
Static methods do not rely on executing the code on HW or simulator but combine the code with abstract model of the system to produce an upper bound WCET.
Value Analysis

- compute possible ranges for registers and local variables
- determine loop bounds
- detect infeasible paths
Control-Flow Analysis

- collect information on possible execution paths
- the exact set of paths is finite since termination must be guaranteed
- superset of exact set is a safe approximation
- goal is to calculate upper bound
  - paths not in the WCE-path can be eliminated
- easier on source code than on machine code
  - difficult to map results to machine code due to changes to the control flow caused by optimizations and linking
- input
  - call graph and control-flow graph
  - possibly bounds for the input and loop variable data
- output
  - annotated syntax tree for structure based approaches
  - set of flow facts for other methods
Control-Flow Analysis Approaches

- pattern matching (structure-based approach)
  - find the set of instructions typically used for loop initialization and counters by the compiler
  - analyze the loop bounds with this information
  - drawback: optimizations and compiler evolution might change the patterns so that recognition fails

- computation modelling
  - use affine equations and inequalities (Presburger Arithmetic)
  - to find the variables that act as loop counters
abstract model of processor, buses, memory subsystem and peripherals

execution of an instruction depends on the context of the processor which depends on the execution history

analysis done on linked executable since it is the only thing containing all necessary information

analysis is conservative approximation never giving time less than what can be observed with actual HW

feasible for simple 8- and 16-bit CPUs but very hard on advanced CPUs exhibiting timing anomalies
execution time of an instruction dependant on the state of the CPU

history of the processor state is analyzed by seeking the execution paths leading to the instruction

consider flow context which express which paths through loops and calls arrive at the instruction

if CPU execution information is missing, a conservative approach must be made

use Data Flow Analysis with Abstract Interpretation

form invariants for each flow context

invariants hold for a set of execution paths and all sets form a partition of the executions leading to the program point

this partitions are called calling contexts

assume the worst case behavior and reduce the upper bound by consider the state of processor components (functional units, pipeline, memory)

not safe due to the timing anomalies
contra-intuitive influences to the local or global execution
unknown parts of the CPU state lead to nondeterministic behavior
example
  - instruction load assuming it is unknown if it is in cache
  - subsequent instruction load assuming the instruction is in the cache
  - on CPU with timing anomalies the latter execution may lead to longer task execution time
reason
  - CPU speculates the outcome of a conditional branch
  - CPU mispredicts it and all effects must be undone
  - misprediction exceeds the costs of a cache miss
  - this is called speculation-caused anomaly
scheduling anomalies
  - sequence of instructions can be scheduled differently on HW resources such as pipeline units
  - may lead to different execution times
incorrect assumption
- choosing the local worst-case transition produces global worst-case execution time

consequences
- state space becomes large since the analysis must consider several abstract states when a non-deterministic choice between successor states exists
- absent information expressing all potential states of missing state components must be expressed instead of assuming some initial worst state
Methods

Estimate Calculation: Structure Based (1/2)

- traverse syntax tree in bottom-up
- combine statements according to the type of the statement
- collapse nodes into a node forming a timing bound for the node
- some transformations like loop unrolling easily expressible as syntax transformations
- problems
  - not every control-flow can be expressed, code optimizations hard to describe, no support for additional flow information
Methods

Estimate Calculation: Structure Based (2/2)

Syntax-tree

\[ T(\text{seq}(S_1, S_2)) = T(S_1) + T(S_2) \]

\[ T(\text{if}(\text{Exp}) \ S_1 \ \text{else} \ S_2) = T(\text{Exp}) + \max(T(S_1), T(S_2)) \]

\[ T(\text{loop}(\text{Exp}, \text{Body})) = T(\text{Exp}) + (T(\text{Exp}) + T(\text{Body})) \times (\text{maxiter}-1) \]

Transformation rules

(d) Structure-based calculation
Estimate Calculation: Path Based

- calculate upper bounds by searching the overall path giving the largest execution time of the task
- execution paths are explicit
- natural for single loop iteration
  - exponential blow-up occurs when flow information is extended across multiple loop-nesting levels
  - might require heuristic search methods
program flow and basic-block execution time bounds combined into sets of arithmetic constraints

- **entities** are program flow edges and basic blocks
- time coefficient $t_{ent}$ express the upper bound of an entity execution time
- count variable $x_{ent}$ gives the maximum number of times the entity is executed
- upper bound is obtained by maximizing sum of products of the execution counts and times $\sum_{i \in ent} x_i \times t_i$
- IPET can handle different types of flow information
- use integer linear programming (ILP) or constraint programming (CP) and can have potentially exponential time bounds
Methods

Timo Lilja ()

Static WCET Analysis: Methods and Tools

April 28, 2011 15 / 23

(a) Control-flow graph with timing

(b) Path-based calculation

(c) IPET calculation

maxiter: 100

\[ t_{\text{path}} = 31 \]
\[ t_{\text{header}} = 3 \]

\[ \text{WCET} = 3 + 31 \times 99 = 3072 \]

\[ x_{\text{start}} = 1, x_{\text{exit}} = 1 \]

\[ x_{\text{start}} = x_{\text{start}} \]
\[ x_{A} = x_{\text{start}} + x_{HA} = x_{Aexit} + x_{AB} \]
\[ x_{B} = x_{AB} = x_{BC} + x_{BD} \]
\[ x_{C} = x_{BC} = x_{CE} \]
\[ \vdots \]
\[ x_{H} = x_{FH} + x_{GH} = x_{HA} \]
\[ x_{\text{exit}} = x_{Aexit} \]

\[ \text{Loopbound constraint} \]
\[ x_{A} \leq 100 \]

\[ \text{WCET Expression} \]
\[ \text{WCET} = \max(x_{A} \times 3 + x_{B} \times 5 + x_{C} \times 7 + \ldots + x_{H} \times 2) = 3072 \]
developed by AbsInt Angewandte Informatik

input
- executable
- annotations (loop bounds, flow facts, specifying values of registers and variables)

analysis
- control-flow analysis
- value analysis (abstract interpretation)
- cache analysis (abstract interpretation)
- pipeline analysis (abstract interpretation)
- bound calculation (path analysis, uses ILP)

output
- WCET upper bound
- visualization of the call graph, cache and pipeline behavior
- a report file that can be integrated with other tools
- screen shots and more information [1]
aiT Limitations

- as described in [1]
- WCET analyzable task must be sequentially executed with no threads, parallelism, external events, interrupts, ...
- code must be generated by restricted subset of ANSI C, and even more limited on C++ and ADA
- no support for dynamic allocation (malloc and such)
- setjmp/longjmp not supported
- code must follow standard ABI calling conventions
- function return addresses must not be modified
- host platforms: Windows and Linux
- target platforms: Motorola PowerPC MPC 555/565/755, ColdFire MCF 5307, ARM7 TDMI, HCS12/STAR12, TMS320C33, C166/ST10, Renesas M32C/85, Infineon TriCore 1.3
Bound-T

- Originally developed by Space Systems Finland and currently developed by Tidorum Ltd [2]

Input:
- Executable with debugging symbols
- Annotations (loop bounds, some loop bounds can be deduced, written to a separate file)

Analysis:
- Control flow
- Loop bounds (Presburger Arithmetic and set of linear transformations)
- Bound calculation (IPET applied to the CFG of the subroutine)

Output:
- Text file listing upper bounds
- Call-graph and control-flow graph displayed with the DOT tool
Bound-T Limitations

- task must not be recursive
- CFG must be reducible
- dynamic calls analysis only if an unique target address found
- weak aliasing analysis
- bounds of an inner loop cannot depend on the index of outer loop
  - "rectangular" upper bound can be constructed
- loop-bound analysis doesn’t cover multiplication, division, bitwise ops
- task must use standard calling conventions
- no cache analysis
- host platforms: PC x86 Linux, Windows, Mac OS X
- target platforms: Analog Devices ADSP21021, ARM7 TDMI, Atmel AVR 8-bit, Intel 8051, Renesas H8/300, Sparc V7/ERC32
Chronos

- research prototype by the National University of Singapore [3]
- input
  - task written in C
  - configuration of the target processor
- analysis
  - uses ILP to bound the number of executions
  - bound calculations are done with IPET
- output
  - statically estimated WCET and SimpleScalar simulated WCET
- doesn’t support any real HW platform
- doesn’t analyze data caches
- notes on building
  - GPL licensed but requires “registering”
  - considerable build-depends (LP solve, old gcc, simplesim)
  - simplesim doesn’t compile on x86_64
open-source tools don’t seem to be in a mature state
  e.g. GPL HEPTANE by IRIS seems to be abandoned
commercial tools well documented and seem to be mature enough
  should test them though, we could get an evaluation license?
are there any independent benchmark suites (like TPC in SQL)
  3rd-party reviews?
References

- aiT Worst-Case Execution Time Analyzers.
  http://www.absint.com/ait/.

- Bound-T time and stack analyzer.
  http://www.tidorum.fi/bound-t/.

- Chronos.