Experiences from Industrial WCET Analysis Case Studies

Andreas Ermedahl, Jan Gustafsson, and Björn Lisper
Dept. of Computer Science and Electronics
Mälardalen University

(andreas.ermedahl, jan.gustafsson, bjorn.lisper)@mdh.se

June 29, 2005

Overview

1. Introduction, background, and motivation
2. WCET tools used
3. Case studies:
   - OSE Operating System (two case studies)
   - VCT LIN Target packace
   - CC-systems welding machine control
   - Volvo CE vehicle transmission control
4. Conclusions and further research

Introduction, Background and Motivation

WCET Analysis methods well-established academically, have yet to catch on in real life

Commercial tools exist (aiT, Bound-T), but are not yet widespread

WCET analysis research seems to focus a lot on how to deal with complex hardware, but is that the only issue?

Academic WCET papers tend to use quite artificial benchmarks for evaluation

We need to test methods on real software and systems, to find out where the real issues and problems are

Case studies can be valuable

We have made five case studies:

- Three “systems RT software”: Enea OSE, VCT LIN
- Two “RT application software” (still ongoing): CC-systems welding, Volvo CE transmission

Important to try out on different kinds of code, we’ll continue . . .

Emphasis on *usability* (effort needed to obtain good WCET bound), but also tried to estimate precision

Intentionally only quite simple architectures, without caches

Working hypothesis: automatic flow analysis can increase the usability

How the Studies are Made

We send out M.Sc. students to companies, equipped with some WCET analysis tool

They analyse some selected RT codes

They record:

- WCET bounds from the tool (including estimation of precision)
- Effort to get there (number of annotations, plus qualitative estimates)
- Statistics of code structure (like # and structure of loops), and qualitative observations about code, of relevance for automatic flow analysis
- Other observations of interest

Tools Used

One case study: early version of our own tool SWEET

This version had no automatic flow analysis

We used only the low-level analysis and calculation part, from Uppsala, with support for ARM7/9 and NEC V850E

All other case studies: aiT from AbsInt

Commercial tool, support for quite a few processors

Industrial strength

In practice, this is necessary for case studies on real production code

Some aiT Characteristics

Analyses binary executables

Pros: in some sense necessary, since this is the code being run. Otherwise dependent on compiler support, or limited to narrow class of applications where the compiler does no optimizations at all

Cons: user annotations on object code level rather than source code level. Can to some extent use debug info from certain compilers to alleviate this, but limitations

Low-level analysis is strong

Rich set of user annotations, to provide program flow constraints, and hardware info, to constrain addresses of memory references, to direct the analysis (regulate context-sensitivity), and more

Case Study 1: WCET Bounds For DI Regions in Enea OSE

Enea OSE is a major commercial RT OS

Used also in safety-critical systems

Supports concurrent processes with priorities, message-based communication, and shared resources

Available for wide range of processors (both embedded and conventional)

Several versions

We analysed parts of the “OSE delta kernel” for ARM9, using an early version of SWEET

DI Regions

Disable Interrupt regions (DI regions) are parts of the code where interrupts are turned off

Typically to protect critical data or similar

Execution time of DI regions must be kept under control

We identified 612 DI regions and selected ten for WCET analysis

A major part of the work was to actually identify the DI regions

On ARM9, interrupts are enabled/disabled by a certain instruction, according to the contents in a certain register

Our student built some simple tools to find DI regions by scanning for the instruction

However, quite often the contents of the register was not constant, then hard to know whether the interrupt was enabled or disabled

They also tend to cross function borders

Due to such problems, we could safely identify only about half of the DI regions

A dataflow analysis would have been helpful to improve on this

Results

Most DI regions were short and had at very simple control structure

Only about 5% contained loops, and only two regions contained nested loops

No function pointers

However, it was hard to find bound for the loops from the source code (dependent on unknown parameters, would have required expert knowledge to set)

Thus, loop bounds were set to large values, probably giving large WCET overestimations

Case Study 2: WCET Bounds For Function Calls in Enea OSE

A second case study of Enea OSE

This time four selected function calls from the delta kernel for ARM7, analysed with aiT

We recorded # of annotations as a measure how laborious the analysis was

Also a study (on standard WCET benchmark codes) how different levels of optimizations affected precision and laboriousness of the analysis

These WCET estimates were validated against a cycle-accurate simulator from ARM

Finally, we analysed 180 of the DI regions again

Results

Although the routines were not that big (80-150 instructions), quite a few annotations were needed (10-30)

Absolute WCET not always interesting. Could occur, say, for a path including high levels of error handling not used in normal operating modes. Rather, WCET for certain “scenarios” wanted. Possible to achieve with aiT, but only through very explicit annotations in the binary code

Loops bounds often parametrically dependent on system parameters like # of buffers, with high upper limit but typically much smaller values in actual configurations. Setting the bound to the upper limit would yield large overestimations in many cases

Much interaction with the experts at Enea was needed to set the annotations

Impact of Optimizations

To our surprise, optimised code did not seem harder to analyse

# of annotations actually tended to drop for optimized code (due to smaller code)

WCET overestimation (compared with simulations) mainly independent of optimization levels, in range 0-10%

Case Study 3: WCET Analysis of LIN Control Software

Automotive code from Volcano Communications Technologies AB

Communication control (CAN, LIN)

We analysed nine API functions of the Volcano LIN Target package

(LIN is a low-cost communication protocol for automotive electronics)

This code has some in common with operating systems code, since it deals with resource handling in networks

Analysed with aiT. Processor: Motorola Star12

Results and Observations

The code did not contain that many loops, but those present were often unstructured and complex

Again, much work needed to set all annotations (required deep understanding of the code)

Loop bounds depended on dynamic parameters, such as number of frames in the network, and their type and size

(A situation somewhat similar to the one for the OSE functions)

Estimated WCET of many loops linearly dependent on these parameters, verified by running the analysis for many combinations of parameter values

Turns out that maximal jitter also is interesting to bound for these functions. This calls for BCET analysis!

Case Study 4: WCET analysis of Welding Machine Control Code

Just finished – not yet reported

Code from CC-systems, developed for ESAB (major manufacturer of welding equipment)

Processor: Infineon C167

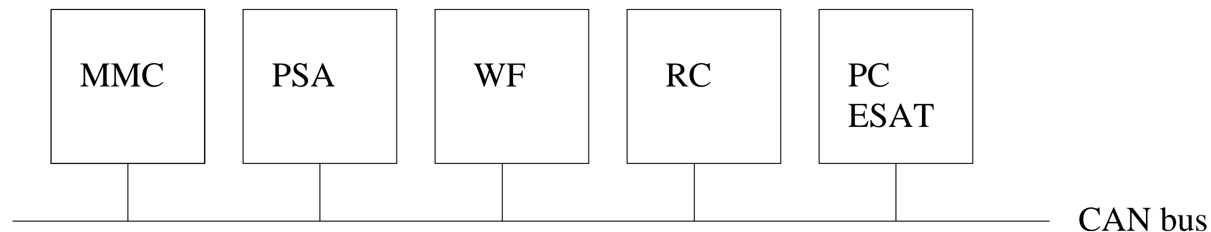
We had two students analyzing the code, in different ways:

- Static WCET analysis, using aiT
- Hardware measurements, using a logic analyser

Goal (in addition to the usual ones): to compare statically estimated WCET bounds with measured execution times

The Target System

A distributed real-time system, with nodes connected through a CAN bus



Nodes for: wire feed control, power source, remote control, man-machine interaction, and service & software upgrades

Object-oriented code (C++), each node runs a main event loop. Also time-triggered interrupts. No operating system. The code is not structured in tasks

Code Characteristics

There are for-loops, however mostly not very complex (some loop tests depends on function calls, though)

Also while loops, however about half of them are “infinite” loops

Switch cases are common (implementing state machines)

Recursion very rare, but does occur in some place

Call depth up to 22 (but usually more shallow)

We analysed:

- Routines to handle CAN interrupts
- Regulator-interrupt, state machine controlling the welding

Results

We were mostly able to obtain reasonably tight WCET bounds (3-20% above measured values, however in three cases more than 100% above)

Very labour-consuming to get there. Many annotations needed, both to constrain program flow and to give precise info about addresses of memory references

Error routines and similar were explicitly excluded by annotations

Difficult to know whether measurements covered the worst execution path. For CAN interrupts, always only one message was in queue although the queue is six elements long. The static WCET analysis was forced to analyse this case, although we don't know for sure whether this is an invariant of the system or not

Results (continued)

Program flow annotations needed for state machine code, to set execution counts equal for different parts of the code representing the same state

OO nature of code caused some complications (indirections give less precise knowledge of memory references, object initialization code has to be explicitly annotated out)

Case Study 5: WCET analysis of Vehicle Transmission Code

Ongoing work, preliminary results

Code from Volvo Construction Equipment for transmission control

Infineon C167, analysed with aiT

Task-oriented code, using the Rubus OS from Arcticus AB (commercialization of the BASEMENT concept)

Rubus tasks must have a WCET bound specified, thus WCET analysis fits very well into the design methodology

Currently measurements (high-water marking) + safety margin is used to find a “safe” WCET bound

One goal of the study is to see whether static WCET analysis can improve on the bounds obtained from measurements + safety margin

Rubus can measure task execution times, we use this to compare with static WCET estimates. Testing is done using Volvo’s real testing facilities

Code Characteristics

12 tasks analysed

Varying size of tasks, one is significantly bigger than the others

Very few loops, but still many possible paths due to coding style with many conditionals

Many infeasible paths

Results

Not so good precision of WCET estimates yet (overshooting measured times 10-60%)

We’re not sure of the cause yet

However, imprecision seems to grow with the size of tasks, which may indicate problems with false paths

Ongoing work, our student is still trying to sharpen the WCET estimates

Quite some work is needed to annotate away infeasible paths in these codes

Seems like automatic flow analysis (even simple condition propagation) should be able to find many infeasible paths

Conclusions and Further Research

Static WCET analysis can find useful WCET bounds for embedded codes

However, can be very labour-consuming to get there

Deep understanding of the code is needed

Annotations on object code level is a pain

A more powerful flow analysis (possibly parametric) would help

Better means for conditional WCET analysis are needed (specifying "execution scenarios", or system invariants not deducible from isolated code)

A powerful constraint annotation language on source code level would be good for this

BCET analysis is sometimes needed