Applying Static WCET Analysis to Automotive Communication Software

Susanna Byhlin, Andreas Ermedahl, Jan Gustafsson and Björn Lisper
Dept. of Computer Science and Electronics, Mälardalen University
Box 883, S-721 23 Västerås, Sweden

Abstract

The number of embedded computers used in modern cars have increased dramatically during the last years, as they provide increased functionality to a reduced cost compared to previous technologies. These computers are often connected by one or more communication networks and the data traffic sent over the networks often has hard real-time requirements. To provide overall system timing guarantees, upper timing bounds need to be derived both for the data traffic and the embedded computer programs that controls the communication.

In this article, we present a case study where static Worst-Case Execution Time (WCET) analysis was used to find upper time bounds for time-critical code in products from Volcano Communications Technologies AB (VCT). The VCT company provides tools for development of real-time communication solutions for embedded network systems, mainly used within the car industry. VCT's tool suite includes support for Controller Area Network (CAN), Local Interconnect Network (LIN), FlexRay and MOST network traffic.

The primary purpose of the study was not to test the accuracy of the obtained WCET estimates, but rather to investigate the practical difficulties that arise when applying current WCET analysis methods to these particular kind of systems. A central question was if today's static WCET analysis tools can be used in the automotive software development process. In particular, we were interested in how labor-intensive the analysis becomes, measured by the number of manual annotations necessary to perform the analysis. As a result, we provide some qualitative observations on desirable research results for making static WCET analysis applicable in typical automotive software development.

1 Introduction

It is nowadays common practice to use embedded computers to control parts of automotive systems. For example, a modern car has a microcontroller controlling the engine, keeping performance up and fuel consumption down by very precise control of the ignition and fuel pump. Embedded controllers are responsible for controlling the anti-lock break (ABS) system, the antiskid control system, and the front panel display. These controllers warm (or cool) your seats and control power windows, mirrors and doors. In some cars, controllers are even used to sense tire pressure, record the mileage from the last car service, and measure fluid levels [3]. A car like Volvo S80 contains more than 30 embedded processors, communicating across several networks. Similarly, BMW 7-series and the Mercedes S-class both contain over 60 processors [28].

Consequently, in-vehicle electronics are today a major cost factor in car development, and can represent up to 30 percent of a car's total manufacturing cost [29]. Analysts estimate that more than 80 percent of all automotive innovations now stem from electronics [18]. The computers used in the cars are often standard microcontrollers, making software the key for providing the specific vehicle characteristics.

These embedded computers are often connected by one or more communication networks, and the data traffic transferred over these networks often has hard real-time requirements. However, as more computers and networks gets incorporated it becomes difficult to verify the overall system behaviour, both regarding functionality and timing. Therefore, new methods for supporting the timing verification process have the potential to offer improvements in product quality, as well as reducing development and testing time.

To give overall system timing guarantees, a key parameter is the *worst-case execution time* (WCET) of the tasks running on the different microcontrollers. Today, the common method (if any) in industry to derive WCET values is by measurements, also known as *dynamic timing analysis*. A wide variety of measurement tools are employed in industry, including emulators, logic analyzers, oscilloscopes, and software profiling tools [24]. This is labor-intensive and error-prone work, and even worse, it is difficult to guarantee that the WCET has been found.

Static WCET analysis is an alternative method to determine the WCET of a program, relying on mathematical models of the software and hardware involved. The analysis avoids the need to run the program by considering the effects of all possible inputs, including possible system states, together with the program's interaction with the hardware. Given that the models are correct, the analysis will derive a timing estimate that is safe, i.e., greater than or equal to the actual WCET. The static WCET analysis research area has developed during the last couple of years, and recently commercial WCET tools, such as aiT [1] and Bound-T [4], have reached the market. However, reports from industrial experiences of static WCET analysis are few, see Section 2.

In this case study we present experiences from using the aiT WCET analysis tool from AbsInt GmbH [1] to find time bounds for time-critical code in embedded products from Volcano Communications Technologies AB (VCT). VCT provides tools for development of real-time communication solutions for embedded network systems, principally used within the car industry. Their tool suite includes support for Controller Area Network (CAN), Local Interconnect Network (LIN), FlexRay and MOST network traffic. The VCT tool suite builds upon the concept of holistic scheduling theory [27], providing response time guarantees for individual tasks and messages as well as overall system transactions [7]. The analyzed code is part of the LIN tool suite.

The primary purpose of the study was not to test the accuracy of the obtained WCET estimates, but rather to investigate the practical difficulties that arise when applying current state-of-the-practise WCET analysis methods to this particular kind of systems. This should provide valuable input for research and WCET tool development. To make static WCET analysis industrial useful it should be possible to, in a high degree, to automate the process on a "one-click-analysis" basis. Consequently, we were interested in how labor-intense practical WCET analysis becomes, measured by the number of manual annotations necessary to perform the analysis.

Furthermore, we were interested in the characteristics of obtained WCET values. Most (holistic) scheduling theories assume that each task and message has a single fixed WCET, and we wanted find out whether this assumption is sound in a real industrial setting. Our hypothesis is that automotive software should be representative for a large class of industrial real-time code, making our results applicable to many similar kind of systems.

2 Static WCET Analysis Overview

Any WCET analysis must deal with the fact that a computer program typically has no single fixed execution time. *Variations* in the execution time occur due to the characteristics of the software, as well as of the computer upon which the program is run. Thus, both the properties of the software and the hardware must be considered in order to understand and predict the WCET of a program.

Consequently, static WCET analysis is usually divided into three phases: a (fairly) machine-independent *flow analysis* of the code, where information about the possible program execution paths is derived, a *low-level analysis* where the execution time for atomic parts of the code is decided from a performance model for the target architecture, and a final *calculation* phase where flow and timing information derived in the previous phases are combined to derive a WCET estimate.

The purpose of the flow analysis phase is to extract the dynamic behavior of the program. This includes information on functions called, how many times loops may iterate, if there are dependencies between conditionals, etc. The information can be obtained by *manual annotations* [13, 14], or by *automatic flow analysis* methods [15, 16, 20].

The purpose of low-level analysis is to determine the timing behavior of instructions given the architectural features of the target system. For modern processors it is especially important to study the effects of various performance enhancing features, like caches, branch predictors and pipelines [12, 25, 20].

The purpose of the calculation phase is to calculate a WCET estimate, combining the flow and timing information derived in the previous phases. A common calculation method is IPET (Implicit Path Enumeration Technique), using arithmetical constraints to model program flow and low-level execution times [13, 25].

Studies of WCET analysis of industrial code are not common. There are some reports on the usage of static WCET analysis to analyze code in space applications [17, 22], and aerospace industry [25, 26]. We have performed two case studies of the OSE operating system [6, 23], obtaining timing for system calls and disable interrupt regions. Colin et al. [9] analyzed operating system functions of RTEMS, a small, open-source real-time kernel. To our knowledge this paper presents the first work where static WCET analysis has been applied to automotive software.

3 The Project Context

The work presented has been carried out within a project with the research goal to develop WCET analy-

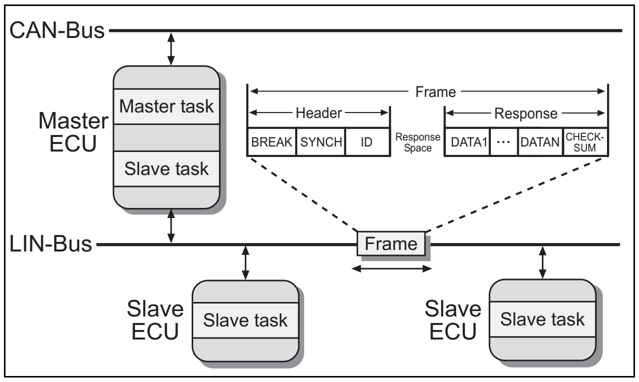


Figure 1. LIN network

sis methods for real embedded systems. Our prototype tool, named SWEET (SWEdish Execution time Tool), is in the final stage of completion [13, 15]. Our current focus is on automatic flow analysis methods, motivated by the fact that manual annotations are believed to be a major hurdle when analyzing real embedded software. This case study is a test of this hypothesis.

Besides our work on automatic flow analysis, we work with case studies involving WCET tool vendors and companies with different kinds of time-critical embedded software. As mentioned in Section 2, we have performed earlier case studies on the OSE operating system [6, 23]. In the first study we used an early version of SWEET, but in the second study as well as the one presented here, we have used the commercial WCET analysis tool aiT from AbsInt GmbH [1]. This tool has a richer set of processor timing models and a better graphical user interface than what can be expected from our research prototype.

4 Automotive Networks

A modern car contains a number of electronic control units (ECUs) interconnected to standard communication networks. In the same vehicle many different types of networks can be present, differing in the bandwidth, network topology and real-time requirements supported. Used automotive networks include low-speed and high-speed Controller Area Network (CAN), Local Interconnection Network (LIN), Media Oriented Systems Transport (MOST) and FlexRay.

CAN is currently the most widely used vehicular network, and a typical vehicle can contain two or three separate CANs operating at different transmission rates [18]. A high-speed CAN manages real-time-critical functions such as engine management, antilock brakes, and cruise control. A low-speed CAN manages a car's comfort electronics, like seat and window movement controls. MOST networks provide support for multimedia applications. FlexRay targets the future needs of in-car control, such as active chassis systems.

VCT provides tool suites for CAN, LIN, MOST and FlexRay network traffic and system management. Both the CAN and LIN tool suite were judged suitable for testing static WCET analysis upon. However, since the work was performed under a limited time period, only selected parts of the LIN tool suite targeting the LIN 2.0 protocol were included in the study.

4.1 The LIN 2.0 Protocol

LIN is a low-cost communication network used where the bandwidth and safety properties of CAN are not required. LIN is most often used for soft real-time traffic, managing devices such as window lifts, sunroofs and seat adjustment. LIN is meant to link to CAN, supporting communication between ECUs situated on different networks. As shown in Figure 1, a LIN network forms a *cluster*, composed of one master node and one or more slave nodes, all connected to a LIN bus [10]. The master node controls the communication over the network and contains a master- and a slave task. A slave node only contains one slave task.

The nodes communicate over the LIN bus by transferring messages, called *frames*, with information. Each node in the LIN cluster is responsible for transmitting a specific set of frames. The communication is initiated when the master node sends out a message *header* on the bus. One (or more) of the slaves is then activated and transmits its *response* part. A LIN frame consists of several standardized parts, including a message identifier, one to eight data fields, and a checksum. A data field can be composed of one or more *signals*, represented by either scalar values (1-16 bits) or byte-arrays (1-8 bytes). Signals are the primary information carriers within the LIN network. Frames can also include *diagnostic information*.

There are severals standardized LIN frame types allowed in the LIN network, including: *Unconditional*; the standard frame type for carrying signals, *Event-triggered*; used to allow multiple slaves to provide a response on a header, *Sporadic*; to enable a more dynamic behavior in the otherwise deterministic signal transmission scheme, and *Diagnostic*; for transmission of diagnostic information [10].

One key characteristic of the LIN protocol is the use of its schedule table(s), assuring that the LIN bus never gets overloaded, i.e., that no frames are lost. The master controls the communication on the bus by following a predefined schedule table that defines which specific frame to send at each specific time. This makes LIN a time triggered network using a Time Division Multiple Access (TDMA) mechanism, unlike CAN which is event-triggered.

5 The Volcano Tool Suite

The Volcano Tool Suite is a holistic data communications concept for design and implementation of in-vehicle networks using CAN and/or LIN [29]. It is a system engineering approach managing software development from many different suppliers. The LIN tool suite consists of several different tools.

The LIN Network Architect (LNA) is the top-level design tool. It captures all design requirements on the network, like requirements of signals, nodes and frames, and automatically generates a resulting LIN cluster configuration. LNA packs the signals into frames, assigns identifiers and generates schedule tables. The resulting cluster configuration is written to a *LIN Description File* (LDF).

Each signal is described in the LDF-file by its name, size, initial value, which node that transmits and which node(s) that receive the signal. LNA uses a timing model that specifies the time between the *generation* and the *consumption* of each signal. A signal could for example be generated in response to some external sensor event, e.g., the pressing of a button, and be consumed by some actuator action, e.g., the raising or lowering of a window.

The next tool in the LIN tool chain is the *LIN Target Package* (LTP) which takes the LDF file together with a *private* file as input. The private file is delivered by the supplier and includes hardware node details, including information on the flags used. LTP consists of a configuration tool and pre-compiled object libraries necessary for LIN network communication. LTP configures the individual LIN nodes and outputs *LIN target dependent code.*

To supply solutions for various car manufacters VCT support a large variety of target processors [29], including Fujitsu 16Lx, Hitachi H8S/SH7055/SH7058, PowerPC, NEC V85x and Motorola HC08/HC12/Star12. Since aiT supports the Star12, it was chosen as a suitable target processor for the study. VCT also have tools for testing, verifying and emulating communication on the LIN network.

6 The MC9S12DP256 Board and Development Environment

The microcontroller used in this study was a MC9S12DP256 from Motorola [21]. It includes a 16-bit Star12 CPU of the MC68HC12 family. The CPU has a three stage pipeline and several different addressing modes (indexed, inherent, immediate etc.), but does not support caches. The MC9S12DP256 memory configuration is 256K of Flash EEPROM, 4K of EEPROM and 12K of SRAM. The time for most memory accesses is one bus cycle for reading single bytes and aligned words, and two bus cycles for reading misaligned words.

The LIN target dependent code was compiled with the Hiware HC12 Compiler from Metroworks, a compiler used by VCT and supported by the aiT tool. We used the Hiware SmartLinker to link the object files into one absolute binary file which was loaded to the MC9S12DP256 using a debugger. The generated executable contains information, like symbol tables and debugger information, which is used by aiT to construct control-flow graphs and other information relevant for the WCET analysis.

7 The aiT WCET Analysis Tool

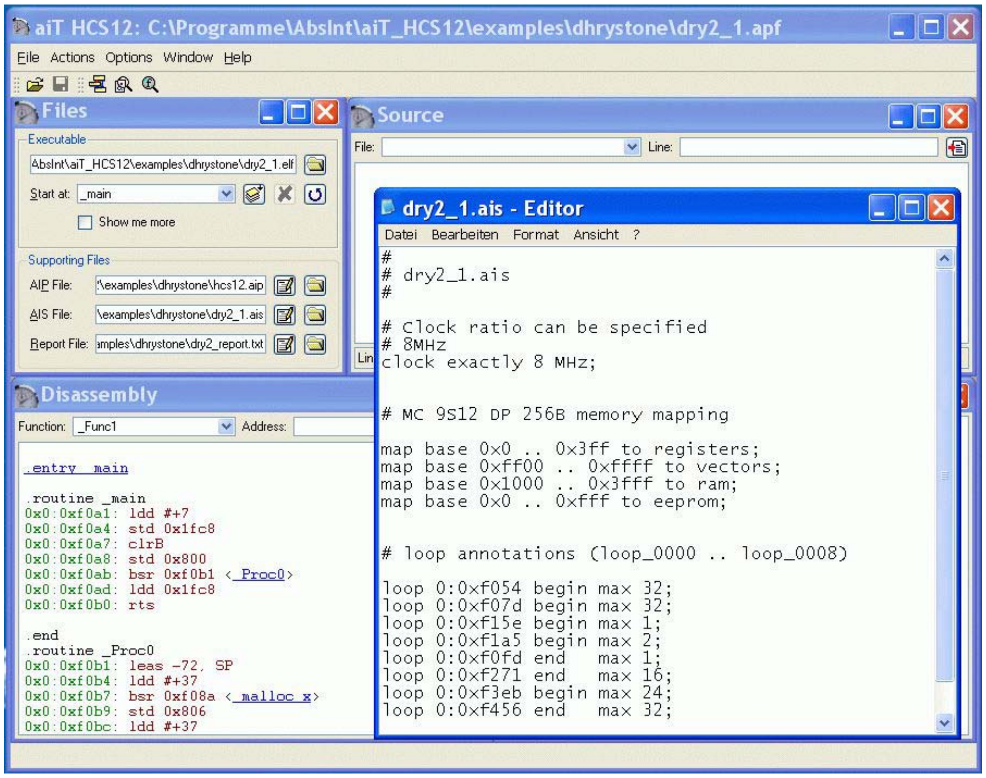
aiT is a commercial WCET analysis tool from AbsInt GmbH [1], a spinoff company from Universität des Saarlandes. aiT analyses executable binaries, and supports a number of target architectures including ARM7, PowerPC555/565/755, ColdFire5307 and HCS12/STAR12. To generate a WCET estimate the aiT tool performs the following analysis steps:

– A *reconstruction of the control flow graph* from the executable code.
– A *loop-bound analysis* to bound the number of loop iterations.
– A *value analysis* to determine the range of values in registers.
– A *cache analysis* to classify accesses to main memory w.r.t. hits and misses, if the processor has a cache.
– A *pipeline analysis*, to determine execution times of basic blocks using a model of the processor pipeline.
– A *calculation* phase where IPET (i.e., integer linear programming) is used to determine the WCET.

In essence, the aiT WCET analysis conforms to the general scheme presented in Section 2. Several of the analyses in the chain are based on abstract interpretation [11], such as the value analysis and the cache analysis [25]. Since the HCS12 does not support cache memories, no cache analysis was performed in this study. Figure 2 illustrates the graphical interface for the aiT WCET tool, including buttons for performing WCET analysis and some assembler code. The front window present some examples of aiT user annotations.

7.1 aiT User Annotations

The information present in the executable itself is typically not sufficient to yield a good WCET bound. Therefore, aiT supports a set of *annotations* to provide external information to the analysis [14]. In particular, annotations could be given on the *possible program flow*, like the start address of the task to analyze, targets of indirect function calls and branches,


aiT HCS12: C:\Programme\AbsInt\aiT_HCS12\examples\dhrystone\dry2_1.apf

Executable: AbsInt\aiT_HCS12\examples\dhrystone\dry2_1.elf
Start at: _main
Show me more

Supporting Files
AIP File: '\examples\dhrystone\hcs12.aip
AIS File: \examples\dhrystone\dry2_1.ais
Report File: amples\dhrystone\dry2_report.txt

Function: _Func1

```
.entry  main

.routine _main
0x0:0xf0a1: ldd #+7
0x0:0xf0a4: std 0x1fc8
0x0:0xf0a7: clrB
0x0:0xf0a8: std 0x800
0x0:0xf0ab: bsr 0xf0b1 <_Proc0>
0x0:0xf0ad: ldd 0x1fc8
0x0:0xf0b0: rts

.end
.routine _Proc0
0x0:0xf0b1: leas -72, SP
0x0:0xf0b4: ldd #+37
0x0:0xf0b7: bsr 0xf08a <_malloc_x>
0x0:0xf0b9: std 0x806
0x0:0xf0bc: ldd #+37
```

dry2_1.ais - Editor

```
#
# dry2_1.ais
#

# Clock ratio can be specified
# 8MHz
clock exactly 8 MHz;


# MC 9S12 DP 256B memory mapping

map base 0x0 .. 0x3ff to registers;
map base 0xff00 .. 0xffff to vectors;
map base 0x1000 .. 0x3fff to ram;
map base 0x0 .. 0xfff to eeprom;


# loop annotations (loop_0000 .. loop_0008)

loop 0:0xf054 begin max 32;
loop 0:0xf07d begin max 32;
loop 0:0xf15e begin max 1;
loop 0:0xf1a5 begin max 2;
loop 0:0xf0fd end   max 1;
loop 0:0xf271 end   max 16;
loop 0:0xf3eb begin max 24;
loop 0:0xf456 end   max 32;
```

File Actions Options Window Help
Files
Source
File:
Line:
Disassembly
Address:
Datei Bearbeiten Format Ansicht ?

Figure 2. Graphical interface of aiT WCET analysis tool

and iteration bounds of loops not caught by the loop bound analysis. Furthermore, *specific hardware configuration* specifications, like processor frequency and address memory mapping, are sometimes required. For example, correct memory map specifications are important when accesses to different memories take different amount of time. Most annotations are given relative the program object code. Among others, the following information can be manually given:

– *Clock rate* - If the clock rate of the microprocessor is not specified, aiT only generates the computed WCET results in cycles. `clock exactly 8 MHz;` informs aiT that clock frequency is 8 MHz.
– *Naming the compiler* - The control flow reconstruction works better if aiT knows what type of compiler that has generated the executable: `compiler "hc12-hiware";`
– *Routine entry* - A routine could be specified to start at a given address: `entry 0x0:0x414c;`[1] .
– *Stop decoding* - The analysis may be informed to end the analysis at a specific address: `end 0x0:0x414c;`.
– *Specifying control-flow* - A the target address of a call or branch can be manually given:
`instruction 0x0:0x414c calls 0x0:0x8500;`.
– *Specifying memory accesses* - The specific memory address of a memory access could be specified like:
`instruction 0x0:0x8500 accesses 0x3a:0x8100;`.
– *Known register values* - For example, `instruction 0x3a:0x9110 is entered with X = 0:0x100;` gives that the register `X` will always have a value of `0x100` at address `0x3a:0x9110`.
– *Address mapping* - The STAR12/HCS12 microprocessor has a memory addressing on the form `P:B`, where `P` is a 6 bit page number and `B` is a 16 bit base address. The base addresses should be mapped to special memory areas. The following four annotations specify the default memory mapping of MC9S12DP256B microcontroller:

```
map base 0x0 ..  0x3ff to registers;
map base 0xff00 ..  0xffff to vectors;
map base 0x1000 ..  0x3fff to ram;
map base 0x0 ..  0xfff to eeprom;
```

– *Never executed code* - The user can remove code from the analysis, e.g., infeasible code. For example: `snippet 0x0:0x1a0c is never executed;`
– *Branch condition outcome* - An constant outcome (true or false) of a branch condition, can be specified as: `condition 0x0:0452 is always true;` or `condition 0x0:0452 is always false;`
– *Recursion depth* - aiT cannot handle mutual recursion. However, direct recursion where a routine

[1]Routine entries are usually found during the control flow reconstruction phase, and their names are fetched from the symbol table in the executable.

Function	Code properties			Annotations					WCET	
	Object code (.o)	Source code (.c)	Loops	Mem. map.	Loop bound	Proc. rate	Compiler	Total	Cycles	Time (μs)
`l_star12sci_ifc_s_connect()`	6 kb	3 kb	0	4	0	1	1	6	25	3.125
`l_star12sci_ifc_m_connect()`	6 kb	3 kb	0	4	0	1	1	6	25	3.125
`l_star12sci_ifc_s_init()`	6 kb	3 kb	0	4	0	1	1	6	159	19.875
`l_star12sci_ifc_m_init()`	6 kb	3 kb	0	4	0	1	1	6	182	22.75
`l_star12sci_ifc_s_rx()`	10 kb	14 kb	3	4	3	1	1	9	2058	257
`l_star12sci_ifc_m_rx()`	16 kb	8 kb	0	4	0	1	1	6	224	28
`l_star12sci_sch_set()`	6 kb	3 kb	2	4	2	1	1	8	3612	452
`l_star12sci_sch_tick()`	10 kb	12 kb	8	4	8	1	1	14	6823	853
`l_star12sci_sys_init()`	4 kb	2 kb	0	4	0	1	1	6	10	1.25

Table 1. Code properties, manual annotations and WCET values of some selected functions

calls itself is allowed, and a maximal recursion depth must then be specified: `recursion "foo" max 12;`

– *Loop bound* - If the loop-bound analysis fails to bound some loops, their bounds can be manually specified: `loop 0x0:0x4537 begin exactly 12;`
– *Source code annotations* - Some information can be defined direct in the source code by annotations in a specific format. Example:
`for(i=3;i*i<=n;i+=2) /* ai: loop here max 20; */`

Most of these specifications cannot be checked by aiT so care should be taken to provide correct annotations. Only a few type of annotations are mandatory, e.g., memory mapping specifications and upper bounds for loops not found by the loop-bound analysis.

8 Experiments and Results

The Volcano LIN Target package (LTP) was selected as a suitable part of the Volcano LIN tool suite to analyse. It consists of a configuration tool and some library files. The LIN API describes the interface between the network and the application program. The API consists of some C functions, each implemented in a separate C file.

Code properties of some selected LIN API functions are given in Table 1. The table shows the size of the object files (.o) and the corresponding source files (.c). The number of loops within the source code files are also given. Most of the analyzed functions were rather small, and only a few of them contained loops. Only one loop was nested, i.e., incapsulated within another loop.

To get the WCET of a function call it was usually not enough to only consider the code of the called function. The called function might call other functions, which call other functions etc. For example, an analysis with aiT on the `l_star12sci_sch_tick()` function generated a call-graph containing 19 routines, (i.e., functions or loops).

8.1 Manual Workload

Table 1 also includes information on the manual annotations required for the WCET analysis. Annotations for the memory mapping, the processor rate, and the compiler type were provided for all functions. No loop bounds were automatically found by aiT, and they therefore had to be set manually. All call and branch targets were found automatically by aiT and no function used recursion, so no such annotations were needed. Annotations for condition outcomes, never executed code, start and end points were given in some special cases, see Section 8.2.

The actual workload required for analyzing a function varied with its code size and structure. Smaller code snippets were straightforward to analyse while more complex codes became more time consuming and error-prone. The main complexity came from the need to support correct annotations, especially giving iteration bounds for loops. Setting correct annotations often required a lot of detailed system knowledge.

To investigate how some system parameters affected the WCET (see Section 8.2 below) we were sometimes required to modify some system configuration files and re-compile the system. If the code layout changed as a consequence, all annotations referring to absolute memory addresses, e.g., `loop 0x0:0x4537 begin max 37`, had to be modified accordingly. One way to reduce this hassle was to provide detailed comments about which part of the source code a given annotation was referring to. This made the first analysis of a function time consuming, but simplified later analyses. An alternative was to use aiT's possibility to use relative addressing in annotations, e.g., `loop "foo" + 1 loop begin max 37`. This turned out to be a useful option, but still required extra checks to confirm that applied compiler optimizations did not changed the code layout to make the annotations incorrect.

8.2 WCET Value Characteristics

We were able to obtain WCET values for all analyzed LIN API functions. However, these values were often not a constant single value, but instead depended on some system parameters given in the system configuration files (LDF and private files). These were:

– The numbers of frames within the current network,
– The types of the frames,

Loop	Annotation	Parameters affecting loop iterations	Cycles/ iter	Time/ iter (μs)
loop1	loop 0x0:0x456f begin max (0-7);	size of received unconditional or diagnostic frame	49	6.125
loop2	loop 0x0:0x4590 begin max (0-9);	number of flags within a frame	23	2.875
loop3	loop 0x0:0x4656 begin max 6;	sets up the sleep command	19	2.375
loop4	loop 0x0:0x46af begin max 7;	always iterates eight times	52	6.5
loop5	loop 0x0:0x4780 begin max (0-7);	size of transmitted unconditional or diagnostic frame	71	8.875
loop6	loop 0x0:0x4737 begin max 7;	size of event-triggered or sporadic frame	46	5.75
loop7	loop 0x0:0x4609 begin max 7;	size of event-triggered frame	73	9.125
loop8	loop 0x0:0x4537 begin max 7;	size of event-triggered frame	88	11

Table 2. Properties of loops in l_star12sci_sch_tick() function

CASE	Conditions	Extra Annotations		WCET	
		Never exec.	Br. cond	Cycles	Time (ms)
CASE1	-	-	-	6871	0.859
CASE2	no sporadic or event-triggered frames	3	-	3531	0.441
CASE3	no sporadic or event-triggered frames, no sleep request, no errors	7	-	1674	0.210
CASE4	unconditional received, unconditional sent, no sporadic or event-triggered	3	4	3315	0.415
CASE5	unconditional received, diagnostic sent, no sporadic or event-triggered	3	4	3308	0.414
CASE6	diagnostic received, unconditional sent, no sporadic or event-triggered	3	4	2756	0.345
CASE7	diagnostic received, diagnostic sent, no sporadic or event-triggered	3	4	2749	0.344

Table 3. Analyzed cases for l_star12sci_sch_tick() function

– The size of the frames, and
– The number of flags latched to the signals within the frames.

A WCET valid under all possible system configurations would for most functions be an overly pessimistic value. Furthermore, for many code parts it was hard to directly see how these parameter values affected the execution of the code. We therefore analyzed each function in a number of specific *cases*. Each case gave a WCET for the function under some specific conditions. Indirectly, the WCET analysis also gave a better understanding of the execution characteristics of the code. We will in the following describe the cases obtained and analysis performed for two different LIN API functions. For a more detailed description we refer to [5].

The l_star12sci_sch_tick() function

The `l_star12sci_sch_tick()` function may only be called by the master node in the LIN cluster and is the function that drives the communication in the LIN network. The function is called periodically on a time basis specified in the LDF-file and follows a pre-defined schedule for frame transmissions. The specific schedule to be used is set by calling the `l_star12sci_sch_set()` function.

The `l_star12sci_sch_tick()` function contains eight non-nested loops as illustrated in Table 2. The table also gives the given loop bound annotations and the iteration cost per loop. The execution behavior for most loops are dependent on the parameters given during system configuration. For example, many of the loops iterates over the content of frames transmitted or received. This gives that their iteration bounds are implicitly given by the largest frame size allowed in the network, as given in the LDF file.

Furthermore, most of the loops are only executed under certain system conditions. For example, `loop3` will only execute if the master sends out a sleep request on the network. Some loops are only executed if a certain type of frame should be transmitted or have been received. Some of these conditions are exclusive, e.g., a frame cannot both be unconditional and sporadic, giving that some loops never can be executed together during the same call.

Table 3 presents the analyzed cases for the `l_star12sci_sch_tick()` function. All values in Table 2 and Table 3 are derived under some default assumptions, unless otherwise stated in the **Conditions** column: that sporadic and event-triggered frames may occur, sleep requests may be transmitted, errors can come up, the maximum frame size is eight, and maximum ten flags may be latched to signals within a transmitted/received frame.

In CASE1 we examined how the maximum size and the maximum latched flags allowed for a frame affect the WCET. These parameters are given values in the LDF and private files and are therefore specified by the system designer. Table 4 shows how WCET grows with the number of latched flags when the frame size is fixed. Similarly, Table 5 shows how WCET grows with the frame size when the latched flags parameter is fixed. To derive these WCET values we repeatedly run aiT on the same code. The loop bounds for `loop1` and `loop5` were systematically adjusted according to the upper frame size. Similarly, the `loop2` loop bound was systematically adjusted according to maximum number of latched flags. By combining the WCET values derived we obtained the parametrical WCET formula in Table 6, expressing the fact that the WCET is linearly dependent of both investigated parameter values.

In CASE2 we derived WCET values under the assumption that no event-triggered or sporadic frames are transmitted. Three *never executed* annotations were added to exclude code parts handling these type

Frame size	Flags	Cycles	Time (ms)
8	1	6662	0.833
8	2	6685	0.836
8	3	6708	0.839
8	4	6731	0.842
8	5	6754	0.845
8	6	6777	0.848
8	7	6800	0.850
8	8	6823	0.853
8	9	6846	0.856
8	10	6869	0.859
Diff./Flag		23	0.003

Table 4. WCET / flag CASE1

Frame size	Flags	Cycles	Time (ms)
1	10	5735	0.717
2	10	5897	0.738
3	10	6059	0.758
4	10	6221	0.778
5	10	6383	0.798
6	10	6545	0.819
7	10	6707	0.839
8	10	6869	0.859
Diff./Frame size		162	0.02

Table 5. WCET / frame size CASE1

$WCET(\text{CASE1}) =$
$5343 + Flags * 23 + Framesize * 162$
where $Flags \in \mathcal{N}, Framesize \in 1..8$

Table 6. WCET formula CASE1

$WCET(\text{CASE2}) =$
$2005 + Flags * 23 + Framesize * 162$
where $Flags \in \mathcal{N}, Framesize \in 1..8$

Table 7. WCET formula CASE2

$WCET(\text{CASE3}) =$
$994 + Framesize * 85$
where $Flags \in \mathcal{N}, Framesize \in 1..8$

Table 8. WCET formula CASE3

of frames from the WCET analysis. The WCET values were shown to be dependent on the same parameters as in CASE1, giving the resulting parametrical WCET formula in Table 7. The base cost of CASE2 (2005 cycles) is smaller than the base cost for CASE1 (5343 cycles), reflecting that code parts have been excluded from the CASE2 analysis.

In CASE3 we derived a WCET bound under the assumption that no event-triggered or sporadic frames may be transmitted and that no sleep requests or errors may occur. This allowed us to remove more code than in the two previous cases, using seven *never executed* annotations. Consequently, the WCET for CASE3 becomes much smaller that for CASE1 and CASE2, as shown in Table 3. The resulting WCET formula, given in Table 8, gives that the WCET no longer depends on the number of latched flags, but only on the frame size.

In the last four cases, (CASE4 to CASE7), we tested how different combinations of sent and received unconditional and diagnostic frames influence the WCET. To do this we had to assume that no event-triggered or sporadic frames may occur, using the same three *never executed* annotations as in CASE2. Furthermore, the conditions that triggers if a unconditional/diagnostic frame had been transmitted/received were systematically specified to be true/false using four *branch condition outcome* annotations. The experiments showed that two unconditional frames transmitted after each other, CASE4, generated the largest WCET.

The l_star12sci_s_rx() function

The `l_star12sci_s_rx()` function is the largest function executed by a slave node in a LIN cluster. The function is executed whenever the node receives one character of data from the network. The function calls several other functions, like `id_received()` and `frame_received()` to perform its tasks, many which contain one or more non-nested loops. The execution of most loops were dependent on some LDF file parameters, like the maximum frame size and the number of frames in the network. Loop bounds for these loops had to be manually provided. Again, we were able to distinguish a number of interesting cases, (CASE8 to CASE10), to be investigated in more detail.

In CASE8 we tested how the WCET of `l_star12sci_s_rx()` depends on the maximum frame size and the maximum number of latched flags allowed. We were able to derive a parametrical WCET formula, were the WCET was linearly dependent on the two investigated parameters. Each increase in frame size gave a WCET increase of 36 cycles and each extra latched flag gave a WCET increase of 49 cycles. The WCET base cost were 1306 cycles, giving a WCET of 2058 cycles or 157 ms, for a maximum frame size of eight and up to ten latched flags as shown in Table 1.

In CASE9 we tested how the WCET depends on number of frames within the network. Two loops in two different called functions, iterated according to this parameter, and we had to modify their loop bounds in a systematic way to obtain the dependency. Again, we were able to generate a parametrical formula giving that one extra frame increases the WCET with 198 cycles or 25.1 ms.

In the last case, CASE10, we tried to obtain the longest path executed from when a frame ID is received in a slave, until the slave sends out the first byte in its response. This required extra *routine entry* and *stop decoding* annotations to specify where the analysis should start and end. We also had to give three *branch condition outcome* annotations to specify the outcome of certain conditions to force the execution to take a certain path.

The provided annotations gave a large reduction in the amount of code to analyze. As an illustration, consider the difference between Figure 3, which shows the original call graph and Figure 4, which shows the reduced call graph after the annotations were added. Both graphs were generated by aiT's graphical visualization tool aiSee. The WCET for the original graph

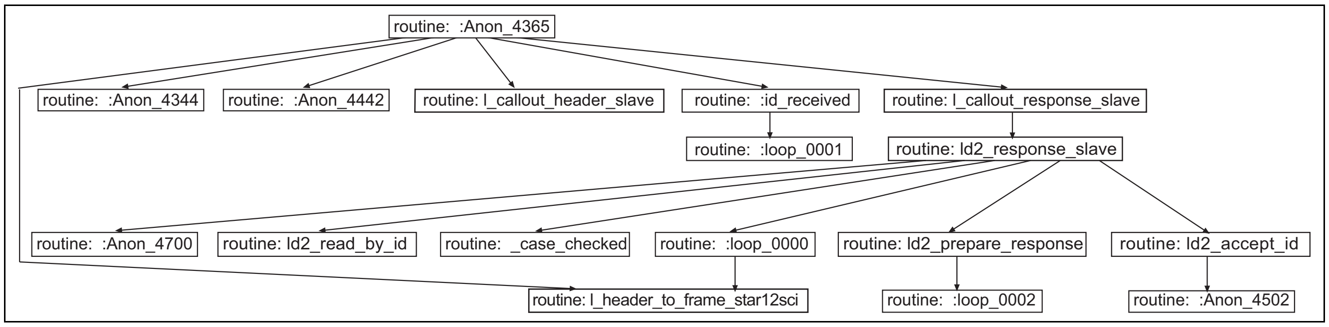


Figure 3. Entire call graph for l_star12sci_s_rx() function

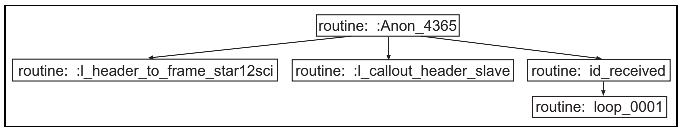

routine: :Anon_4365
routine: :l_header_to_frame_star12sci
routine: :l_callout_header_slave
routine: id_received
routine: loop_0001

Figure 4. Reduced call graph CASE10

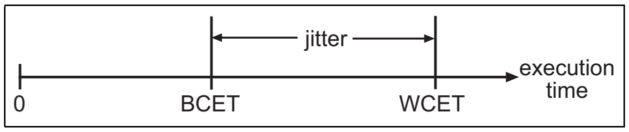


Figure 5. Function jitter

became 11174 cycles while the reduced graph gave a much tighter WCET estimate of 325 cycles. We conclude that it is possible to use aiT to get the WCET for a specific execution path, but that extra annotations are required to constrain the analysis to include only relevant code.

9 Conclusions and Future Work

We were able to obtain WCET estimates for all analyzed parts of the Volcano LIN Target Package. The overall conclusion is therefore that static WCET analysis is a feasible method for deriving WCET estimates for this kind of automotive communication software. We note, however, that static WCET analysis is not yet mature enough to fully automate work on a "one-click-analysis" basis. Instead, detailed system and code knowledge is often required and manual intervention is needed in terms of annotations. Consequently, the usefulness of static WCET analysis would improve with a higher level of automation and support. Especially important should be to develop better methods for loop bound analysis.

Another important conclusion is that absolute WCET bounds (in terms of cycles or milliseconds) often were insufficient for the analyzed system. Instead, the WCET often depended on some specific system parameters, like the number of frames in the system. An absolute WCET bound, covering all possible situations, was often a gross over-approximation. A mode- and input-sensitive WCET analysis would have the possibility to get better resource utilization and provide better understanding of the system's timing characteristics. Thus, it seems important to develop methods to support such analyses.

For many parts of the LIN API it was possible to manually create parametrical WCET formulas. This allowed us to test how certain parameter values affect the WCET and gave a good understanding of the execution behavior of the analyzed code. Thus, it seems interesting to develop methods to automatically derive these parametrical dependencies and formulas [8, 19]. Even if such methods require some manual annotations, like limits of input parameter values, we believe that these annotations are fewer, and easier to give, than e.g., loop bounds and exact execution counts. Compared to earlier work where we analyzed operating system code [6, 23], we note that the VCT code is of less parametrical nature, but still more than what we initially expected.

After discussions with the VCT employees it turned out that not only the WCET, but also the *jitter* that a function or a task can experience is of large interest. The jitter is the largest execution time variation a function can experience, i.e., the difference between the best-case execution time (BCET) and the WCET. Figure 5 gives an illustration of the concept. To support jitter calculation today's WCET tools need to be extended with BCET analysis. We do not believe this to be very complicated, but it requires adaptation of the analysis methods to do safe underestimations for BCET (instead of safe overestimations of WCET).

We believe that static WCET analysis could be integrated into VCTs development environment. However, VCT uses a large variety of processors and compilers (see Section 5), and is often required to quickly adapt its tool suite to new target platforms. A full integration of a static WCET analysis tool in VCTs development chain may pose a problem, since WCET tools for new processors and compilers may not be available or developed quickly enough. However, we expect that the amount of platforms supported by static WCET analysis to increase, making this a less problematic issue.

The employed at VCT had a positive attitude towards aiT and static WCET analysis in general. They thought that the tool was very user-friendly, and gave results on a detailed level. Future work includes to compare static WCET values with timing values derived using more dynamical analyses, such as hardware measurements. We would also like to perform complementary studies on the Volcano CAN target package.

Acknowledgements

This work was performed within the Advanced Software Technology competence center (ASTEC) [2], supported by the Swedish Agency for Innovation Systems (VINNOVA) [30]. We especially want to thank AbsInt GmbH [1], for giving us access to their HCS12 WCET analysis tool, as well as Volcano Communications Technologies AB [29], for giving us access to source code and binaries of the Volcano Tool Suite.

References

[1] AbsInt company homepage, 2005. `www.absint.com`.

[2] ASTEC homepage, 2005. `www.astec.uu.se`.

[3] S. Barret and D. Pack. *Embedded Systems - Design and Applications with the 68HC12 and HCS12*. Pearson Education, Inc, 2005. ISBN: 0-13-140141-6.

[4] Bound-T tool homepage, 2005. `www.tidorum.fi/bound-t/`.

[5] Susanna Byhlin. Evaluation of Static Time Analysis for Volcano Communications Technologies AB. Master's thesis, Mälardalen University, Västerås, Sweden, Sept 2004.

[6] M. Carlsson, J. Engblom, A. Ermedahl, J. Lindblad, and B. Lisper. Worst-case Execution Time Analysis of Disable Interrupt Regions in a Commercial Real-Time Operating System. In *Proc. 2nd International Workshop on Real-Time Tools (RT-TOOLS'2002)*, 2002.

[7] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano – A Revolution in On-Board Communications. *Volvo Technology Report*, 1:9–19, 1998.

[8] A. Colin and G. Bernat. Scope-tree: a Program Representation for Symbolic Worst-Case Execution Time Analysis. In *Proc. 14th Euromicro Conference of Real-Time Systems, (ECRTS'02)*, pages 50–59, 2002.

[9] A. Colin and I. Puaut. Worst-Case Execution Time Analysis for the RTEMS Real-Time Operating System. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*, June 2001.

[10] LIN Consortium. LIN Specification Package, Revision 2.0, Sept 2003. `http://www.lin-subbus.org/`.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, Los Angeles, January 1977.

[12] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden, Apr 2002. ISBN 91-554-5228-0.

[13] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Box 325, Uppsala, Sweden, June 2003. ISBN 91-554-5671-5.

[14] C. Ferdinand, R. Heckmann, and H. Theiling. Convenient User Annotations for a WCET Tool. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, 2003.

[15] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a Flow Analysis for Embedded System C Programs. In *10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005)*, Feb 2005.

[16] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding Loop Iterations for Timing Analysis. In *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, June 1998.

[17] N. Holsti, T. Långbacka, and S. Saarinen. Using a Worst-Case Execution-Time Tool for Real-Time Verification of the DEBIE software. In *Proc. of the DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457)*, Sep 2000.

[18] G. Leen and D. Hefferman. Expanding Automotive Electronic Systems. *IEEE Computer*, 35(1):88–93, Jan 2002.

[19] B. Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, June 2003.

[20] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 2002.

[21] MC9S12DP256 Advance Information, Rev. 1.1, Dec 2002.

[22] M. Rodriguez, N. Silva, J. Estives, L. Henriques, and D. Costa. Challenges in Calculating the WCET of a Complex On-board Satellite Application. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, 2003.

[23] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static Timing Analysis of Real-Time Operating System Code. In *Proc. 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, Oct 2004.

[24] D. B. Stewart. Measuring Execution Time and Real-Time Performance. In *Proceedings of the Embedded Systems Conference (ESC SF) 2002*, Mar 2002.

[25] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Universität des Saarlandes, 2004.

[26] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretetation-Base Timing Validation of Hard Real-Time Avionics Software. In *Proc. of the IEEE International Conference on Dependable Systems and Networks (DSN-2003)*, June 2003.

[27] K. Tindell and J. Clark. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, 1994.

[28] J. Turley. Embedded processors. In Extremetech.com, Jan 2002.

[29] Volcano Technologies Communications AB homepage. `www.volcanoautomotive.com`, 2005.

[30] Vinnova homepage, 2005. `www.vinnova.se`.