Static Timing Analysis of Real-Time Operating System Code

Daniel Sandell, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper

Dept. of Computer Science and Engineering
Mälardalen University
Box 883, S-721 23 Västerås, Sweden

Abstract. Methods for Worst-Case Execution Time (WCET) analysis have been known for some time, and recently commercial tools have emerged. However, the technique has so far not been much used to analyse real production codes. Here, we present a case study where static WCET analysis was used to find upper time bounds for time-critical regions in a commercial real-time operating system. The purpose was not primarily to test the accuracy of the estimates, but rather to investigate the practical difficulties that arise when applying the current WCET analysis methods to this particular kind of code. In particular, we were interested in how labor-intense the analysis becomes, measured by the number of annotations to explicitly constrain the program flow which is necessary to perform the analysis. We also make some qualitative observations regarding what a WCET analysis method would need in order to perform a both convenient and tight analysis of typical operating systems code. In a second set of experiments, we analyzed some standard WCET benchmark codes compiled with different levels of optimization. The purpose of this study was to see how the different compiler optimizations affected the precision of the analysis, and again whether it affected the level of user intervention necessary to obtain an accurate WCET estimate.

1 Introduction

A *Worst-Case Execution Time* (WCET) analysis finds an upper bound to the worst possible execution time of a computer program. Reliable WCET estimates are a key component when designing and verifying real-time systems, especially when real-time systems are used to control safety-critical systems like vehicles, military equipment and industrial power plants. WCET estimates are needed in hard real-time systems development to perform scheduling and schedulability analysis, to determine whether performance goals are met for periodic tasks, and to check that interrupts have sufficiently short reaction times [1]. However, WCET analysis has a much broader application domain; in any product development where timeliness is important, WCET analysis is a natural tool to apply.

Any WCET analysis must deal with the fact that a computer program typically has no 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.

The traditional way to determine the timing of a program 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 [2, 3]. This is labor-intensive and error-prone work, and even worse, it cannot guarantee that the true WCET has been found. This is because, in general, it is practically impossible to perform exhaustive testing.

Static timing analyses estimate the WCET of a program without actually running it. The analyses avoid the need to run the program by simultaneously considering the effects of all possible inputs, including possible system states, together with the program's interaction with the hardware. The analyses rely on mathematical models of the software and hardware involved. Given that the models are accurate enough, the result is a *safe* timing estimate that is greater than or equal to the actual WCET.

The static WCET analysis research area has developed during the last decade, and recently commercial WCET tools, such as aiT [4] and Bound-T [5], have reached the embedded system market. However, practical experience of WCET analysis in industry has so far been rather limited, see Section 2.

In this case study we report from experiences when using a static WCET analysis tool to analyze code from the Enea OSE Real-Time operating system [6]. This is a commercial operating system, used in applications such as mobile phones and aircrafts, and thus an example of real production code.

Real-time operating systems are important to analyze with respect to timing properties, since they often are used in time-critical applications. Tasks with hard real-time constraints may make operating system calls. If no good WCET bound for the called code is known, then it is not possible to find a good WCET bound for the calling task either. Furthermore, OS services such as task switching must have good WCET bounds in a real-time system, since they also affect the timing properties of the application code. Finally, operating systems contain *disable interrupt regions* (or DI regions for short), where the interrupts are turned off, e.g., to provide a critical section where some shared resource is protected. The WCETs of such regions need also to be bounded, since their execution can delay higher priority tasks.

In our study, we analyzed some selected system calls and DI regions. We were somewhat interested in the precision of the analysis, but more in issues like how difficult it is to analyze typical operating systems code. WCET analysis cannot be completely automatic, (or we would have solved the halting problem), and manual user directives are typically needed to provide information that the analysis is not able to derive automatically. These directives provide problems: they can be erroneous, in which case the analysis might give an underestimation of the WCET, and providing the proper directives may be laborious and require a deep understanding of the code. This means that WCET analysis methods must be tuned to handle certain important classes of code with a minimum

of needed user intervention. Our hypothesis was that operating systems code provides a particularly challenging class in this regard, and is much different in character than, for instance, signal processing code.

The second part of the study concerns how compiler optimizations affect the manual labor needed to perform an accurate WCET analysis. Optimizations may create a more unstructured, complex, program flow in the resulting code, which may make it harder to provide proper annotations for constraining the program flow. In addition, we studied how compiler optimizations for speed and size, respectively, affected the WCET itself. This is also interesting: for instance, it is not evident that an optimization for average speed will give a lower WCET. For this part of the study we were not able to use the OSE operating system code, the reason being that this code, due to its low-level nature, only compiles with a small set of compilers with certain combinations of flags set. Instead, we used a set of standard WCET benchmarks.

The rest of this paper is organized as follows. In Section 2, we give a brief introduction to WCET analysis and related work in the area. Section 3 presents our WCET project and our previous industrial experiences. In Section 4 the aiT WCET tool is described. Section 5 gives a short description of the OSE operating system. Section 6 presents the target processor for the analysis, including the associated development environment. Section 7 describes the experimental setup and the obtained results. Finally, in Section 8, we draw some conclusions and give ideas for further research.

2 WCET Analysis Overview and Related Work

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 behaviour of the program. This includes information on which functions get called, how many times loops iterate, if there are dependencies between `if`-statements, etc. Since the flow analysis does not know the execution path which corresponds to the longest execution time, the information must be a safe (over)approximation including *all* possible program executions. The information can be obtained by *manual annotations* (integrated in the programming language [7] or provided separately [8, 9]), or by *automatic flow analysis* methods [10–12]. The flow analysis is usually called high-level analysis, since it is often done on the source code, but it can equally well be done on intermediate or machine code level.

The purpose of low-level analysis is to determine the timing behaviour 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 [13–16].

The purpose of the calculation phase is to calculate the WCET estimate for a program, combining the flow and timing information derived in the previous phases. A calculation method frequently used is IPET (Implicit Path Enumeration Technique), using arithmetical constraints to model the program flow and low-level execution times [8, 17, 12]. IPET calculations normally rely on integer linear programming to solve the generated constraint system.

Studies of WCET analysis of industrial code are not common. There are some reports on application of commercial WCET tools to analyze code for space applications [12, 18, 19], and in aerospace industry [20, 21]. A recent case study describes experiences from applying WCET analysis to LIN and CAN communication software in cars [22].

An investigation of industrial embedded code has been done by Engblom [23]. He collected statistics of the number of occurrences of certain code features that may be problematic for a WCET analysis, like recursion, unstructured flow graphs, function pointers and function pointer calls, data pointers, deeply nested loops, multiple loop exits, deeply nested decision nests, and non-terminating loops and functions. In a more recent study [24], industrial code is investigated with respect to how amenable it is to a syntactical flow analysis, a method which detects certain loop patterns for whome immediate bounds can be given.

Studies of how to perform WCET analysis on operating system kernels are even more rare. We have done an earlier case study of the OSE operating system [25], where a number of DI regions were identified and analyzed. The study presented here is a followup. The only other work we know in the area is by Colin and Puaut [26]. They analyse some operating system functions of RTEMS, a small, open-source real-time kernel.

3 The Project Context

The work presented here has been carried out in the context of a project whose aim is to develop WCET analysis methods that work for real embedded software. We have developed a modular WCET analysis tool architecture [8], and an actual prototype tool named SWEET (SWEdish Execution time Tool) is in the final stage of completion [27]. Our current focus is on automatic methods to find constraints on program flows, like the maximal number of iterations for loops. The currently available WCET analysis tools only have quite crude methods for doing this, and SWEET will be used to experiment with more precise methods, like Gustafsson's interval-based flow analysis [10]. The motivation for this work is that the need for manual flow constraint annotations is believed to be a major hurdle when analyzing real embedded software. The case study presented here is an attempt to test this hypothesis for a certain class of embedded software.

Besides 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 an earlier case study with WCET analysis of the OSE operating system [25], as well as a case study with software for handling LIN and CAN bus communication in cars [22]. 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 [4]. This tool has a richer set of processor timing models than we can maintain for our prototype tool, and it also has a better user interface than what can be expected from a research prototype like ours. Therefore, it was a natural choice to use this tool once it was avaliable to us.

4 The aiT WCET Tool

The aiT tool is a commercial WCET analysis tool from AbsInt GmbH [4], which is a spinoff company from Universität des Saarlandes. aiT analyses executable binaries, and it has support for a number of target architectures including the ARM7TDMI. aiT performs the following steps in its analysis:

- a *reconstruction of the control flow graph* from the executable code,
- an analysis to *bound loop iterations*, based on a combination of an interval-based abstract interpretation and pattern-matching tuned to the compiler that generated the analyzed code [28],
- a *value analysis* to determine the range of values in registers,
- a *cache analysis* that classifies accesses to main memory w.r.t. hits and misses, if the processor has a cache,
- a *pipeline analysis*, where a model of the pipeline behavior is used to determine the execution time of basic blocks, and finally
- a *path analysis* where an IPET calculation is made 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 [29], such as the value analysis and the cache analysis [17].

The aiT ARM7 tool analyses executables stored in .ELF format. This format contains information about the code, like symbol tables, which is used by aiT. This information is to some extent vendor-specific, meaning in reality that the aiT ARM7 tool can analyze executables from only a limited set of compilers. We used the ARM compiler, see Section 6, which belongs to this set.

The information present in the .ELF file and the executable itself is typically not sufficient to yield a good WCET bound for the analyzed code. In particular, information about program flow, such as bounds to loop iteration counts not caught by the loop bounds analysis, and knowledge of infeasible paths, has to be provided by the user. Therefore, aiT supports a set of *user annotations* to provide external information to the analysis [9]. Some of the more important annotations are: *loop bounds*, *maximal recursion depth*, *dead code*, and (static) *values of conditions*. The two latter can be used to exclude parts of the code, like error routines, which one may want to exclude from the WCET analysis even though their execution is feasible. In addition, there are a number of possible annotations to specify the control flow of subroutine calls, when needed, and to provide hardware-related information such as clock rate and address mapping to different kinds of memories. The annotations can be given in a separate annotation file or in the C-code in form of special comments[1].

[1] We did not use any source code annotations in our study, since the aiT tool at the time of the study did not support the ARM compiler fully.

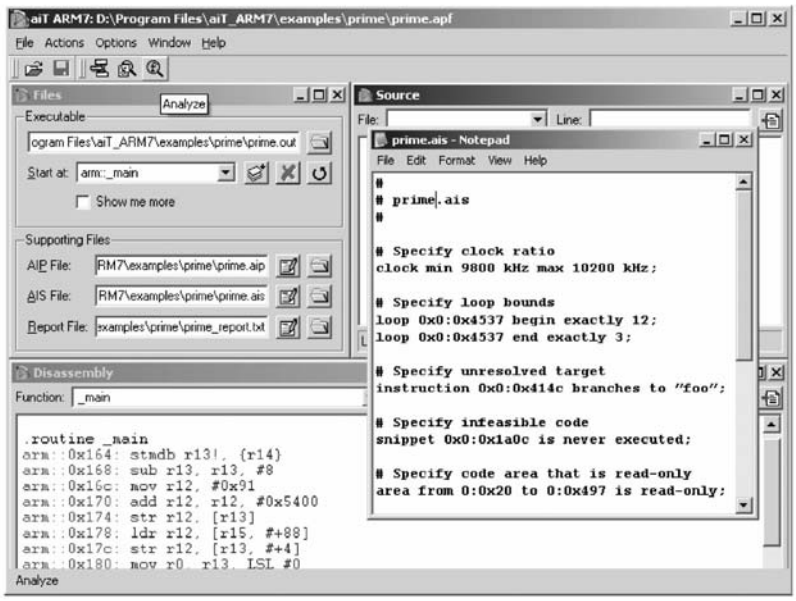

Fig. 1. aiT ARM7 WCET Tool Graphical Interface

The larger lower window in Figure 1 illustrates the graphical interface for the aiT ARM7 WCET tool, including action keys for performing the WCET analysis and a subwindow with ARM7 assembler code extracted from the .ELF file. The front window gives some illustrative examples of possible annotations, including loop bound and dead code annotations.

5 The OSE Operating System

OSE is a real-time operating system, developed by Enea Embedded Technology [6]. It is used in embedded system applications such as mobile phones and aircrafts. OSE is available for a number of target processors, mostly towards the high-end spectrum of embedded processors. The delta kernel of OSE, which has been used in this study, is available for ARM, StrongARM, PowerPC, Motorola 68k, and MIPS R3000. The OSE source code is written both in C and assembler.

The OSE kernels are process-based, fully preemptive, and provide priority-based process scheduling. Processes communicate through messages: messages are sometimes called *signals*, and are sent through buffers. Each buffer is identified by a signal number. Only a small number of system calls are needed to support most requests: for instance, basic interprocess communication can be handled by the two system calls `send` and `receive`.

In OSE, related processes can share the same memory pool. This provides a larger degree of fault tolerance, since a corrupted memory pool will affect only the processes sharing that pool. System calls like `alloc` and `free_buf` are used for memory handling by application processes.

Due to the communication and memory pool models, OSE must handle a great deal of shared memory structures. Since the process model is preemptive, this means that the operating system code must contain quite a few critical sections to keep these structures consistent during updates. These sections are typically implemented by DI regions. Executing such a DI region can thus cause a temporary priority inversion, where a higher-priority process is delayed by a lower-priority process. It is therefore important to have small WCETs for the DI regions, in order to keep these delays down.

6 The ARM Board and Development Environment

We selected the ARM7TDMI processor as target architecture since it is widely used, since OSE is implemented for it, and since there is a version of aiT for it. This is a 32-bit RISC processor, with an uncached core and a 3-stage pipeline. Most of the instructions are executed in a single clock cycle. The ARM7TDMI is forward compatible with ARM9, ARM10 and StrongARM.

Interrupts are enabled (EI) and disabled (DI) by setting some bits in a status register. This is done with a move-register-to-status-register instruction. Thus, an EI or DI will be executed depending on the contents in the source register.

6.1 ARM development tools

The ARM development toolkit[2], used contains an ANSI C compiler, assembler, linker, ARMulator simulator, and an ARM development board. The C compiler produces ARM object format or assembly source output. It can be run with a variety of flags, including different levels of optimization for both space and execution time.

We used the ARMulator in our experiments. The ARMulator is a simulator, which makes it possible to evaluate the behaviour of a program for a certain ARM processor without using the actual hardware. The ARMulator model consists of four main components:

- The ARM processor core model that handles the communication with the debugger.
- The memory system. It is possible to modify the memory model, for instance w.r.t. different RAM types and access speeds.
- The coprocessor interface that supports custom coprocessor models.
- The operating system interface, which makes it possible to simulate an operating system.

It is possible to measure the number of clock cycles used by a program using the ARMulator. Bus and core related statistics can also be obtained from the debugger. There is no guarantee that the ARMulator timing model corresponds exactly with the actual hardware. However, since ARM7TDMI is an uncached and not very complex core, we expect the ARMulator to be rather cycle accurate.

[2] ARM Developer Suite Version 1.2

system call	description
`alloc`	Allocation of memory in a pool
`free_buf`	Free allocated memory
`receive`	Receive signal from another process
`send`	Send signal from process to another process

Table 1. Analyzed system calls

system call	restrictions of the analysis	assumptions
`alloc`(a)	Buffers of correct size exist	
`alloc`(b)	No buffers of correct size exist	No swap out handler is registered
`free_buf`	There are two pools in the system	
`receive`(a)	Receive all signals	The signal is first in the queue. No swap out handler is registered. A 20 bytes signal is copied. No redirection.
`receive`(b)	Receive a signal	The signal is in at second place in the queue. Max 2 buffers before in the queue. No swap out handler are registered. A 20 bytes signal is copied. No redirection.
`send`(a)	Send a signal to a process with higher priority	The call to int mask handler is not analysed. No swap out handler is registered and the analysis stops before the interrupt process is called. No redirection.
`send`(b)	Send a signal to a process with lower priority	No redirection

Table 2. Description of performed analyses and assumptions made

7 Experimental Setup and Results

We made a series of experiments. First we analyzed a set of OSE system calls, and a number of DI regions in the OSE operating system, using the aiT tool. Then we investigated the influence of code optimization on WCET analysis. For this experiment, we compiled some standard benchmark programs with different levels of optimization, and performed WCET analyses on the resulting binaries, again using the aiT tool. For these binaries, we also tried to find the exact WCET by simulating the longest path with the ARMulator. This was done in order to estimate the accuracy and safety of the WCET estimates provided by the static analysis. In all experiments, we used the ARM C compiler, and we assumed a memory model with zero wait states for both the WCET analysis and the simulation (i.e., an instruction is executed in same number of clock cycles no matter where in the memory it is stored). The estimated WCET results and simulated execution times are given in number of clock cycles.

7.1 Analysis of system calls

We analyzed the OSE system calls given in Table 1. These calls includes error checks and use advanced memory protection. They are real-time classified system calls in OSE.

A problem that we soon discovered is that the execution time of these system calls depend on many parameters, such as the number of signal buffers, or maximal message sizes. Assuming a global worst-case scenario, where all these parameters assume their "worst" values, can give very poor WCET estimates for actual configurations, where these parameters typically are bound to much smaller values. Furthermore, certain feasible paths may not be interesting to analyze since they will not be executed in normal operation. Error handling routines typically belong to this category.

system call	funcs	instr	blocks	loops	annot	WCET
`alloc`(a)	1	78	15	0	10	127
`alloc`(b)	9	390	54	0	18	433
`free_buf`	2	100	19	0	15	186
`receive`(a)	15	531	119	2	29	821
`receive`(b)	17	609	143	4	33	1469
`send`(a)	4	281	56	0	32	493
`send`(b)	5	288	62	0	33	417

Table 3. Result of system call analyses

We dealt with these problem in our experiments by assuming some "typical" scenarios for parameters affecting the WCET (after correspondance with the OSE designers). Furthermore, we excluded uninteresting execution paths from the analysis by manual annotations, setting conditions to true or false or by explicitly excluding basic blocks. For `alloc`, `receive` and `send`, we assumed two different scenarios each. They are denoted by (a) and (b), respectively, in Table 2, which summarizes the conditions under which the analyses were made.

Table 3 gives the results of the analyses. For each analysed system call, **funcs** is the number of analysed routines in the call graph, **instr** is the total number of assembler code instructions, and **blocks** is the number of basic blocks. All these numbers are for the system calls with the error handling excluded. The estimated WCET's are given in column **WCET**.

The most interesting information in Table 3 is the number of annotations (**annot**) needed to perform each WCET analysis. As seen, quite a few annotations are required for each system call analysis. Another observation is that excluding the error handling yields significantly smaller code to analyze. For instance, `send` with full error check uses at least 39 routines. This indicates that it really is important to identify and exclude execution scenarios that are not interesting to analyze, even if their execution is feasible.

Some of the analyzed system calls contained loops. Providing upper bounds for these loops posed a problem since they were dependent on dynamic data structures present in the system. As mention in Section 4, aiT includes a loop bound analysis. According to email discussions with the aiT developers, their loop bound analysis typically bounds 70-95% of all loops automatically (ARM7 code compiled with the Texas Instrument ARM compiler). However, the loops in the OSE kernel have a very parametrical behaviour, making the recognition rate very low for these loops.

An illustrating example is a loop appearing in `receive`. The loop iterates through an array with signal numbers, searching for a specific signal buffer. Each iteration of the loop takes 13 clock cycles: thus, each iteration has a limited impact on the total WCET. However, there are more than 32000 possible signal numbers. Using this as a loop iteration bound will give a very pessimistic WCET. In many practical situations, the actual number of signal numbers will be statically bound by a much smaller number, and the calculated WCET will be a huge overestimation.

Another interesting loop is found in `receive`(b). The loop iterates through a queue of buffers, and the number of iterations is bounded by the number of buffers searched before finding the right one. Unfortunately, the number of

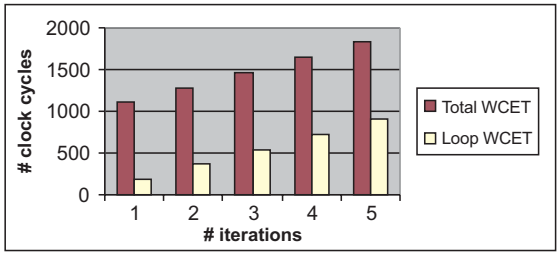

Fig. 2. `receive` WCET scaled with loop iterations

buffers in the system may be hard to know statically since it depends on the current system state. The time for an iteration was 182 clock cycles and was a significant part of the total execution time of the system call. In Figure 2 we have tabulated the total WCET of `receive`(b) against the number of iterations performed in the loop. If the loop iterates more than five times, then its total contribution to the WCET will exceed the contribution of the rest of the executed code. The WCET for `receive`(b) in Table 3 was given under the assumptions that at most two buffers were searched before the right buffer was found.

The analysis of the system calls was done by the first author, who made the work as part of his M. Sc. thesis. Thus, he may be considered a typical engineer who has not yet acquired a lot of experience using WCET analysis tools, and who is not particularly knowledgeable about the code to be analysed. The analysis was quite labor-consuming, taking in total a few weeks to perform, even if the analyzed code in the end became quite small. The main reason for this was that the author first tried to correct all the warnings that occurred in the analysis, e.g., set unresolved branches and loop bounds, before actually understanding what parts of the code that should be excluded from the analysis. Secondly, the student was forced to rely on information from the OSE designers to give feasible loop bounds.

We conclude that it is possible to apply static WCET analysis to code with properties similar to the system calls in OSE Delta kernel. However, it is hard to fully automate the WCET analysis process on a 'one-click-analysis' basis. Instead, much manual intervention, and detailed knowledge of the analyzed code, is required to perform the analysis. Furthermore, if the obtained WCET values are to be useful, they must be calculated under the actual conditions for which the system is expected to run, with stronger bounds on system parameters and input arguments to system calls.

7.2 Analysis of disable interrupt regions

In this experiment, we analysed 180 DI regions from the OSE operating system. This is a selection of the DI regions analysed in [25]. Most of the DI regions analysed were short and not so complex: 132 the regions contained five or less basic blocks, and only one of the selected regions contained a loop. Consequently, not so many annotations were needed and most DI regions needed only a few annotations: 119 of the 180 analyses needed two or less annotations. Figure 3

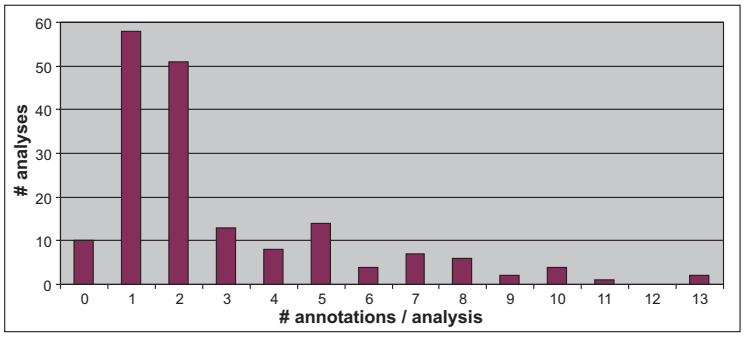

Fig. 3. Number of manual annotations per DI region

DI region	instr	blocks	loops	annot	WCET
DI92728-EI92752	6	2	0	1	12
DI74156-EI74216	16	4	0	2	29
DI82928-EI83088	28	9	1	6	331

Table 4. Properties of some example DI regions

shows in detail how the number of annotations is distributed. In Table 4 the properties of three analysed DI regions are given together with their WCETs.

For this kind of code, the annotations were used mostly to restrict the WCET analysis to the actual program paths possible between the actual EI and DI operations. This is not always a trivial task, since DI regions may span function boundaries. Two different types of annotations were used for this. The condition annotation was used to follow the paths in the basic block graph, and the dead code annotation was used to make sure that the analysis would stop at the correct instruction. The single loop found, which is looking for any changes in a signal buffer, could not have its iteration count bounded automatically by aiT. Therefore, we manually set the loop bound to 10 to be able to extract a WCET.

We conclude that DI regions are more suitable than the system calls to perform automatic WCET analysis upon. However, for some DI regions expert knowledge of the code is required to provide correct annotation and iteration bounds, making it hard to make the analysis fully automatic.

7.3 WCET analysis of optimized code

Compilers for embedded system can optimize for both speed and size. In many applications, such optimizations are important. Thus, it is interesting to study how these optimizations affect WCET analysis of the resulting code.

The benchmarks used in this experiment contain conditional constructs. They are listed in Table 5 together with their size, numbers of blocks, and number of loops, compiled with medium optimisation for space with the ARM C compiler.

Each benchmark was compiled with optimization for size and speed, respectively, and with medium and maximum optimisation levels for both. In Table 6, the results are given. For each optimization we give the WCET estimate produced by aiT (**aiT**), the simulated time obtained from the ARMulator (**armu**), the ratio in precent between these (**+%**), and the number of annotations (**ann**).

program	description	instr	blocks	loops
bs	Binary search for the array of 15 integer elements	28	10	1
crc	Cyclic redundancy check computation on 40 bytes of data	104	28	3
expint	Series expansion for computing an exponential integral function	50	18	2
isort	Insertion sort on a reversed array of size 10	42	7	2
ns	Search in a multi-demensional array	51	14	4
select	A function to select the nth largest number an array	91	34	4

Table 5. Benchmarks for evaluating WCET and compiler optimizations

pro-	speed − medium				speed − high				size − medium				size − high			
gram	aiT	armu	+%	ann	aiT	armu	+%	ann	aiT	armu	+%	ann	aiT	armu	+%	ann
bs	100	93	7.5	4	65	60	8.3	2	107	100	7.0	5	65	60	8.3	2
crc	34852	34804	0.1	7	27499	27455	0.2	7	34869	34821	0.1	7	27517	27473	0.2	4
expint	2208	1997	10.6	5	1150	1145	0.4	2	2263	2052	10.3	5	2113	1891	11.7	5
isort	1230	1190	3.4	4	1213	1190	1.9	3	969	962	0.7	4	944	919	2.7	3
ns	8518	8497	0.2	2	7228	7208	0.3	0	8601	8516	1.0	2	8603	8517	1.0	0
select	1357	1349	0.6	16	1333	1306	2.1	13	1428	1401	1.9	17	1362	1295	5.2	12

Table 6. How compiler optimizations affect WCET

We have also repeated the experiment with a ARM7 C compiler from IAR Systems with similar results, see [30] for details.

When the benchmarks were highly optimized, the structure of the programs changed a bit, but in most cases it was not so difficult to find the corresponding code and make the proper annotations. Changes that occurred were, for instance, that a function was moved inside the callers body, and the loop control could be changed to the end of the loop instead of the beginning. The most difficult changes to handle annotation-wise were when loop fission or loop fusion occurred.

Interestingly, the results indicate that it was not harder to perform an accurate WCET analysis for highly optimized code. The ratio between WCET estimate and simulated execution time stays quite constant. It was somewhat harder to produce annotations, but not much harder. (The number of annotations even drop some with increasing level of optimization, but this is mainly an effect of the code size decreasing with increasing levels of optimization.)

7.4 Justifying obtained WCET estimates

When comparing simulated ARMulator times and calculated aiT WCET estimates it should be noted that both methods rely on software models of the hardware, making it hard to say that one timing estimate is more correct than another one. Engblom [16] identifies several error sources in constructing a correct hardware timing model, including hardware bugs, manual writing errors and simulator implementation errors. Furthermore, for competitive reasons processor manufactures often keep the internals of their processor cores secret.

To get some justification of the quality of calculated WCET estimates we compared timing estimates from aiT and the ARMulator for a number of benchmarks (not included in this article, see [30] for details). The benchmarks contained features like system calls, loops and branches, but had only one single execution path through the program. By keeping track of the number of times each basic block was taken during a simulator run, we were able, by annotations, to provide exact bounds on the executions of each basic block for the WCET calculation. Thereby, the resulting timing discrepancies were not due to incorrect flow information, but only due to differencies in the hardware timing models.

The experiments showed that the aiT WCET estimates were on average about 5% larger than the times obtained using the ARMulator. For none of the tested benchmarks aiT gave a WCET lower than the timing produced by the ARMulator. We therefore conclude that the timing model and the timing estimates produced by the aiT tool for ARM7 should be of approximately the same quality as for the ARMulator[3].

8 Conclusions and Future Work

The results indicate that static WCET analysis is a feasible method for deriving WCET estimates for real-time operating system code. For all analyzed parts of the OSE operating system we were able to obtain WCET estimates, including both system calls and DI regions.

We note however that the static WCET analysis technique is not yet mature enough to fully automate the timing analysis process on a 'one-click-analysis' basis. Instead, detailed knowledge of the analyzed code is required and often manual intervention must be performed in terms of annotations.

We conclude that the usefulness of WCET analysis would improve with a higher level of automation and support from the tool. Especially important should be to develop advanced flow analysis methods, especially to find more loop bounds automatically. For most of the loops analyzed in OSE it was not possible to determine their bounds just by looking at the loop, and the loop iteration bounds analysis of aiT would not produce a bound either. Rather, expert knowledge was needed, and the work was time-consuming and error-prone. Similarly, better support for easy exclusion of error handling routines from the normal WCET analysis would be of great value.

Another important conclusion made is that absolute WCET bounds are not always appropriate for real-time operating system code. The reason is that the WCET often depends on dynamic system parameters, like the number of signal buffers, which however may be bounded in practical configurations or modes. Then, an absolute WCET bound, covering all possible situations, will provide a gross overapproximation.

Therefore, one would like to express the WCET conditionally, given that the system runs in a certain mode. Modes, or sets of modes, can often be encoded as value-range constraints on program variables (settings of flags, bounds on number of processes, etc.). Program flow constraints can also be expressed as value-range constraints, but on execution count variables. Thus, it seems interesting to develop means to communicate such information to the analysis in order to constrain the possible program flows for the given mode.

A parametric WCET analysis [31] may also be useful, especially for handling code like system calls. This type of WCET analysis could express how the WCET for system calls depends on, e.g., the system state and the input arguments.

[3] The aiT developers have seen overestimations up to 4% for their ARM7 timing model, (compared to measurements with a logic analyzer on a TI TMS470 bond-out chip). The main cause of overestimation are believed to be instructions with varying execution time dependent of argument values (such as mult and div) [28].

A more general conclusion is that the constant time assumed for, e.g., context switch, in many scheduling approaches, will be a gross overestimation in many cases. A conditional WCET, in terms of system state and input arguments, would lead to a much tighter value, and thus a better utilisation of the system.

Acknowledgements

This work was performed within the Advanced Software Technology competence center (ASTEC), `www.astec.uu.se`, supported by the Swedish Agency for Innovation Systems (VINNOVA), `www.vinnova.se`. We want to thank AbsInt GmbH, `www.absint.com`, for giving us access to the aiT tool, as well as Enea Embedded Technology, `www.ose.com`, for giving us access to source code and binaries for the OSE operating system.

References

1. Ganssle, J.: Really Real-Time Systems. In: Proceedings of the Embedded Systems Conference San Fransisco (ESC SF) 2001. (2001)
2. Ive, A.: Runtime Performance Evaluation of Embedded Software. Presented at the 8^{th} Nordic Workshop on Programming Enviroment Research (1998)
3. Stewart, D.B.: Measuring Execution Time and Real-Time Performance. In: Proceedings of the Embedded Systems Conference (ESC SF) 2002. (2002)
4. AbsInt: Absint company homepage (2004) URL: `http://www.absint.com`.
5. Bound-T: Bound-t tool homepage (2004) URL: `http://www.bound-t.com`.
6. Enea: Enea Embedded Technology homepage (2004) URL: `http://www.enea.com`.
7. Kirner, R., Puschner, P.: Transformation of Path Information for WCET Analysis during Compilation. In: Proc. 13^{th} Euromicro Conference of Real-Time Systems, (ECRTS'01), IEEE Computer Society Press (2001)
8. Ermedahl, A.: A Modular Tool Architecture for Worst-Case Execution Time Analysis. PhD thesis, Uppsala University, Dept. of Information Technology, Box 325, Uppsala, Sweden (2003) ISBN 91-554-5671-5.
9. Ferdinand, C., Heckmann, R., Theiling, H.: Convenient User Annotations for a WCET Tool. (In: Proc. 3^{rd} International Workshop on Worst-Case Execution Time Analysis, (WCET'2003))
10. Gustafsson, J.: Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation. PhD thesis, Department of Computer Systems, Information Technology, Uppsala University (2000)
11. Healy, C., Sjödin, M., Rustagi, V., Whalley, D.: Bounding Loop Iterations for Timing Analysis. In: Proc. 4^{th} IEEE Real-Time Technology and Applications Symposium (RTAS'98). (1998)
12. Holsti, N., Långbacka, T., Saarinen, S.: Worst-Case Execution-Time Analysis for Digital Signal Processors. In: Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference). (2000)
13. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The Influence of Processor Architecture on the Design and the Results of WCET Tools. IEEE Proceedings on Real-Time Systems (2003)
14. Engblom, J.: Analysis of the Execution Time Unpredictability caused by Dynamic Branch Prediction. In: Proc. 8^{th} IEEE Real-Time/Embedded Technology and Applications Symposium (RTAS'03). (2003)

15. Healy, C., Arnold, R., Müller, F., Whalley, D., Harmon, M.: Bounding Pipeline and Instruction Cache Performance. IEEE Transactions on Computers **48** (1999)
16. Engblom, J.: Processor Pipelines and Static Worst-Case Execution Time Analysis. PhD thesis, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden (2002) ISBN 91-554-5228-0.
17. Ferdinand, C., Martin, F., Wilhelm, R.: Applying Compiler Techniques to Cache Behavior Prediction. In: Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97). (1997)
18. Holsti, N., Långbacka, T., Saarinen, S.: Using a Worst-Case Execution-Time Tool for Real-Time Verification of the DEBIE software. In: Proceedings of the DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457). (2000)
19. Rodriguez, M., Silva, N., Estives, J., Henriques, L., Costa, D.: Challenges in Calculating the WCET of a Complex On-board Satellite Application. (In: Proc. 3^{rd} International Workshop on Worst-Case Execution Time Analysis, (WCET'2003))
20. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and Precise WCET Determination for a Real-Life Processor. In: Proc. 1^{st} International Workshop on Embedded Systems, (EMSOFT2000), LNCS 2211. (2001)
21. Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R., Ferdinand, C.: 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). (2003)
22. Byhlin, S.: Evaluation of Static Time Analysis for Volcano Communications Technologies AB. Master's thesis, Mälardalen University, Västerås, Sweden (2004)
23. Engblom, J.: Static Properties of Embedded Real-Time Programs, and Their Implications for Worst-Case Execution Time Analysis. In: Proc. 5^{th} IEEE Real-Time Technology and Applications Symposium (RTAS'99). (1999)
24. Sandberg, C.: Inspection of Industrial Code for Syntactical Loop Analysis. In: Proc. 4^{th} International Workshop on Worst Case Execution Time Analysis. (2004)
25. Carlsson, M., Engblom, J., Ermedahl, A., Lindblad, J., Lisper, B.: Worst-case Execution Time Analysis of Disable Interrupt Regions in a Commercial Real-Time Operating System. In: Proc. 2^{nd} International Workshop on Real-Time Tools (RT-TOOLS'2002). (2002)
26. Colin, A., Puaut, I.: Worst-Case Execution Time Analysis for the RTEMS Real-Time Operating System. In: Proc. 13^{th} Euromicro Conference of Real-Time Systems, (ECRTS'01). (2001)
27. Gustafsson, J., Lisper, B., Sandberg, C., Bermudo, N.: A Tool for Automatic Flow Analysis of C-programs for WCET Calculation. In: 8^{th} IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003). (2003)
28. Lisper, B.: Personal communication with C. Ferdinand at AbsInt (2004)
29. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. 4^{th} ACM Symposium on Principles of Programming Languages, Los Angeles (1977)
30. Sandell, D.: Evaluating Static Worst Case Execution Time Analysis for a Commercial Real-Time Operating System. Master's thesis, Mälardalen University, Västerås, Sweden (2004)
31. Lisper, B.: Fully Automatic, Parametric Worst-Case Execution Time Analysis. Technical report, Mälardalen Real-Time Research Centre, Mälardalen University, Sweden (2003) MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-97/2003-1-SE.