Front. Comput. Sci. China 2010, 4(1): 17–27
DOI 10.1007/s11704-009-0073-8

RESEARCH ARTICLE

Static worst-case execution time analysis of the μC/OS-II real-time kernel

Mingsong LV (✉)[1], Nan GUAN[1], Qingxu DENG[1], Ge YU[1], WANG Yi[1,2]

1 Institute of Computer Software, Northeastern University, Shenyang 110819, China
2 Department of Information Technology, Uppsala University, Uppsala S-75105, Sweden

Abstract Worst-case execution time (WCET) analysis is one of the major tasks in timing validation of hard real-time systems. In complex systems with real-time operating systems (RTOS), the timing properties of the system are decided by both the applications and RTOS. Traditionally, WCET analysis mainly deals with application programs, while it is crucial to know whether RTOS also behaves in a timely predictable manner. In this paper, static analysis techniques are used to predict the WCET of the system calls and the Disable Interrupt regions of the μC/OS-II real-time kernel, which presents a quantitative evaluation of the real-time performance of μC/OS-II. The precision of applying existing WCET analysis techniques on RTOS is evaluated, and the practical difficulties in using static methods in timing analysis of RTOS are also discussed.

Keywords worst-case execution time (WCET), real-time operating systems (RTOS), μC/OS-II, static analysis

1 Introduction

Hard real-time systems are those systems in which tasks must meet their deadlines; otherwise, there will be disastrous consequences. Timing correctness of hard real-time systems is traditionally guaranteed by a hierarchical off-line analysis framework. First, worst-case execution time (WCET) analysis is used to obtain the execution time of tasks in the worst case, and then schedulability analysis uses these results to decide whether all the tasks are schedulable. There are two important properties to describe the usefulness of an analysis technique: *safety* and *accuracy*. The results are safe if no actual execution of the program exceeds the estimated time. Also, the estimation is said to be more accurate if it is closer to the real maximal execution time of the program. Soft real-time systems do not always have safety requirements, but hard real-time systems allow no underestimation. The accuracy of the estimation is also critical, since too pessimistic estimations lead to over-design and low task acceptance ratio. Using only toy benchmarks is inadequate to justify the usefulness of the analysis techniques, so it is important to test the techniques on real-life programs.

Traditionally, WCET analysis is mainly applied to application programs and has achieved success in industry (e.g. aiT[1)]). While complex real-time systems are composed of both applications and real-time operating systems (RTOS), and the timing properties of the system are decided by both parts. In order to obtain WCET estimations for a whole system, timing analysis should be performed not only on application programs, but also on RTOS services. Although it does not make much difference between application code and RTOS code when they are compiled to binaries, (the input to static WCET analysis), analyzing RTOS is generally harder than analyzing applications with no library/system calls, since the behaviors of RTOSes are much more complex. Simply applying static analysis techniques designed for applications may yield low-quality or even incorrect results.

Received August 20, 2009; accepted December 9, 2009

E-mail: mingsong@research.neu.edu.cn

1) The absint page. http://www.absint.com, 2009

In an RTOS, certain parts of the system code are particularly time-critical. An example is the DI regions which execute with the interrupts turned off. The execution of DI regions directly affects the response of the system to external events. So it is useful to quantitatively bound the execution time of DI regions, which will further help RTOS designers to identify the bottlenecks and then improve the responsiveness of the RTOS.

This paper is an extension of our previous conference paper [1], in which Chronos, a static WCET analysis tool, is used to obtain the WCETs of the system calls (or APIs) of the μC/OS-II real-time kernel. The main enhancement of this paper is the WCET analysis of the DI regions of μC/OS-II (also using static analysis techniques). The purpose of this research is not only to test the accuracy of the estimation, but also to investigate the practical difficulties in applying static analysis techniques to real-life RTOS code. In our experiments, we successfully checked the WCET of 61 (out of 79) μC/OS-II system calls and all the DI regions residing in these system calls. Compared to simulation-based timing analysis methods, the quantitative results obtained by WCET analysis can give a safer picture of the real-time performance of an RTOS. In our practice, we find that some traditional WCET analysis techniques, such as those adopted in Chronos, are far from adequate in characterizing the timing properties of an RTOS. The problems found from our research include a lack of parametric timing analysis techniques in RTOS analysis and incorrect results in the presence of context switches, etc.

The rest of the paper is organized as follows. Section 2 gives an introduction to μC/OS-II. The experimental settings applied in our research are introduced in Section 3. Section 4 elaborates how the WCETs of the system calls and the DI regions are obtained. Analysis results and some further issues are discussed in Sections 5 and 6, respectively. Section 7 lists related research, and the paper is concluded in Section 8.

2 The μC/OS-II real-time kernel

μC/OS-II[1)] [2] is an open-source real-time kernel designed by Micriμm, Inc. The μC/OS-II kernel is designed to be efficient with a small footprint. Although it does not have as many features as other RTOSes such as RTEMS and VxWorks, it has nearly all the standard RTOS capabilities: (1) Priority-based preemptive scheduling; (2) Inter-task communications via semaphore, mutex, message queue, and message box; (3) Time management; (4) Simple memory management.

μC/OS-II is one of the most widely used real-time kernels in industry: it has been licensed by hundreds of real-time embedded systems companies, and the products span multiple domains, including network management devices, handheld devices, and embedded monitor and control systems. μC/OS-II is also certified in avionics products by FAA for use in commercial aircrafts. We believe it is of great practical use to evaluate the realtime performance of μC/OS-II quantitatively due to its prevalence in industry.

3 Experimental settings

This section gives the detailed experimental settings applied in our experiments: the configuration of μC/OS-II, and the WCET analysis tool adopted.

3.1 μC/OS-II configuration

The μC/OS-II real-time kernel has a small footprint with 11 C files and one header file, which contain 79 system calls spanning 9771 lines of code. μC/OS-II allows developers to customize these features according to their requirements. In this paper, we exclude some features that are not the core functions of the system. Excluded features are: name management, the statistic task, task profiling, and debugging. Some trivial functions, such as the dummy function, are also excluded. We refer readers to our technical report[2)] for further details.

In order for μC/OS-II to run on a specific processor, one must first port the system to the target instruction set. μC/OS-II requires that functions to do context switch and disable/enable interrupts should be rewritten for different architectures. We find that these architecture-specific functions are all single-path functions, and they do not incur extra difficulty in the analysis. Additionally, Simplescalar the adopted simulator in the experiments, does not fully support full-functional simulation of operating systems. So without loss of generality, we replace these functions with dummy functions in our experiments.

1) http://www.micrium.com, 2009
2) Lv M, Guan N, Zhang Y, et al. Detailed results of WCET analysis of μC/OS-II. http://www.neu-rtes.org/techreport.html, 2009

3.2 The WCET analysis tool

In our research, we use Chronos [3] to analyze the μC/OS-II kernel. Chronos is open-source, which allows the users to enhance it with new functionalities. The underlying abstract processor model in Chronos is a MIPS-based architecture, and the users can configure the parameters for the instruction cache, the pipeline, and the branch predictor. The power of Chronos is the ability to model complex micro-architecture features listed above [4], which is another reason why we choose Chronos. Chronos first reads in C code and compiles them into PISA binaries; then the frontend of the tool performs data flow analysis to detect loop bounds (for the loop bounds that cannot be detected automatically, user intervention is required to set them). The core of the analyzer performs WCET analysis. It first disassembles the binary into a control flow graph (CFG); then it performs micro-architecture modeling to decide the execution time of each basic block in the CFG; at last, an Implicit Path Enumeration based technique is adopted to calculate the WCET.

In the experiments, we find that the precision of the analysis results has very close relationship with the configuration of the target processor in Chronos. Large overestimation is detected in the presence of large superscalarity of pipelines, large block size of instruction caches and large memory access latency. While the size of instruction fetch queue and reorder buffer of the pipelines, and the number of sets and associativity of the caches do not contribute much to the overestimation. Note that one of the objectives of this research is to investigate whether some characteristics of a RTOS affect the precision of the analysis. So in order to make it easy to distinguish the sources of overestimation, we need to minimize the overestimation incurred by the analysis tool. The configuration listed in Table 1 is applied to Chronos.

Table 1 Chronos configurations

	Features	Value
Pipeline configurations	Superscalarity	1
	Instruction fetch queue size	4
	Reorder buffer size	8
Instruction cache configurations	The number of cache sets	64
	Cache block size	8
	Cache associativity	8
	Main memory access latency	30
Branch prediction configurations	Branch history table size	16
	Branch history register width	1

4 How to obtain the WCETs of the system calls and the DI regions

In this section, we introduce how the WCETs of the system calls and the DI regions of μC/OS-II are obtained. The corresponding enhancements to Chronos are detailed as well.

4.1 Obtaining the WCETs of system calls

When trying to use Chronos to obtain the WCET of each individual system call, we encounter some problems. Chronos requires that any program to be analyzed should have a main() function, so that the program can be properly compiled and simulated. This means that it is impossible to analyze a standalone system call. So we have to wrap each system call in a main() function to make it analyzable, as illustrated in Fig. 1(a).

But this trick is also problematic. Even if nothing but the system call is wrapped in the main() function, the compiler automatically adds additional instructions both before and after the instructions of the system call (see Fig. 1(b)), which increases the obtained WCET value. To minimize this imprecision, we have to subtract the execution cycles of the pre- and post-instructions from

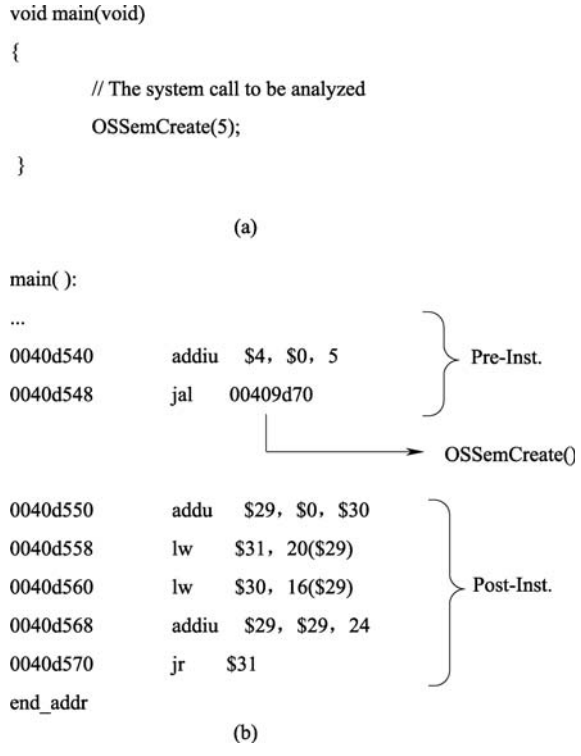

Fig. 1 A wrapper program for system calls. (a) C source code; (b) assembly code

the estimated WCET. Since the additional instructions have no loop or branch, they execute exactly once. So the additional cycles are composed of the cache miss penalties of these instructions and their execution time. Note that in the processor configuration above, the superscalarity of the pipeline is set to 1, which means at most one instruction is committed in each cycle, and the minimum execution time of any instruction is one cycle. So it is safe to subtract one cache miss penalty and one cycle for each pre- and post-instruction from the estimated WCET. The final WCET is calculated according to Eq. (1), where N_{pre} and N_{post} represents the number of pre- and post-instructions. We may also manually add some instructions before the system call to prepare a proper running context, and these instructions are treated similarly.

$$\begin{aligned} &\textit{Calculated WCET} \\ &= \textit{Estimated WCET} - (\textit{Cache\_miss\_penalty} + 1) \\ &\quad \times (N_{\text{pre}} + N_{\text{post}}). \end{aligned} \tag{1}$$

4.2 Obtaining the WCETs of the DI regions

DI regions are certain parts of the RTOS code, which execute with the interrupts turned off. Structurally, A DI region is a code segment within a certain system call, starting from interrupt disabling instructions/functions and ending with interrupt enabling instructions/functions (see Fig. 2). Chronos takes the TCFG of a program as the input of the analysis, where the TCFG is a monolithic CFG for the program with the CFGs of all subroutines merged according to function call relations. Since Chronos cannot work on partial code segments, we have to extract the DI regions from the system calls, and build TCFGs for the DI regions (we call them Sub-TCFGs), and then make these Sub-TCFGs the input to the analyzer. This is done by enhancing the Chronos work flow with the ability to process DI regions (shaded function blocks in Fig. 3). Note that the functional constraints are either obtained from "Path Analysis" or inputted by the users, and they are also useful in calculating the WCETs for DI regions. The functional constraints should be correctly mapped from the TCFG of the program to the Sub-TCFG of the DI region.

The CFG of a certain DI region is a sub-graph of the TCFG of the program. To extract DI regions from the TCFG, we first scan the TCFG to find all the entries of DI regions. For each entry of a certain DI region, we obtain the sub-graph by traversing the TCFG, which starts from the entry and stops until the exit(s) is(are) reached. In

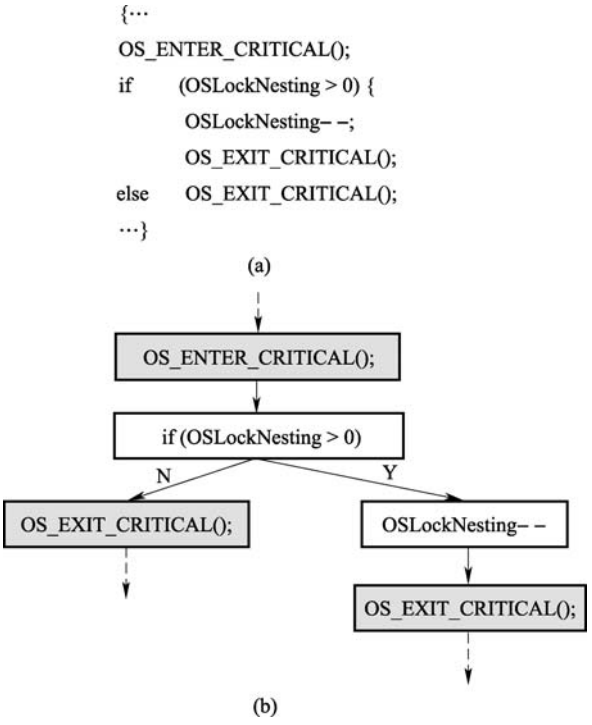

Fig. 2 The example of a DI region. (a) The code segment of a DI region; (b) the corresponding CFG

μC/OS-II, DI regions with multiple exits, as the one illustrated in Fig. 2, are very common. In this case, the CFG of such a DI region is not a standard TCFG (single entry, single exit) that Chronos can deal with, so we just simply add a dummy node and an edge from each exit to the dummy node to construct the Sub-TCFG of the corresponding DI region. This process is illustrated in Algorithm 1. It is possible that a DI region has loops inside, and the loop bound constraints are originally posed on the TCFG of the program. We correctly map the constraints to the Sub-TCFG of the DI region. The Sub-TCFGs for all the identified DI regions are fed to Chronos to generate the integer linear programming (ILP) problems, and by solving the problems we get the WCET estimations of the DI regions.

5 Analysis of the experiment results

This section gives an intensive analysis of the estimated results of both the system calls and the DI regions obtained in our experiments.

5.1 Analysis of the estimated results of system calls

The μC/OS-II kernel has 79 system calls in all, and we successfully obtained the WCET of 61 system calls. We

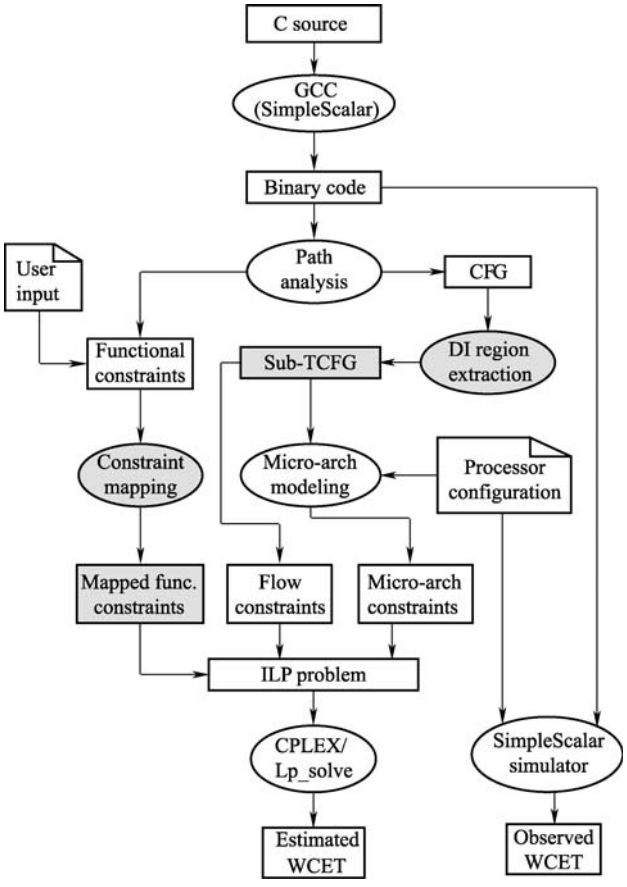

Fig. 3 The enhanced Chronos work flow

fail to analyze the 8 system calls of the timer management module, because there are dynamic function calls in the APIs, and Chronos is not able to decide the jump targets of dynamic function calls statically. The analysis of the OSTaskCreateExt() API also failed because it cannot pass simprofile (Simprofile [5] is a module of SimpleScalar to generate detailed profiles for the program, and it is one of the mandatory steps of the analysis process of Chronos).

All the analysis results are listed in Appendix A1. An average of 17.57% overestimation is reported. All the WCET values in Tables 2, 3 and the appendix is measured in terms of processor cycles. We will give a detailed evaluation of this result. First, nearly half of the μC/OS-II system calls are implemented with very simple program structures, and such programs do not pose any difficulty to the analysis tool, so the overestimation is generally under 5%.

The second type of system calls are those that try to acquire a shared resource, such as OSFlagPend(), OSSemPend(), etc. There are two main sources of the overestimation. First, these system calls share similar control flows with lots of "if-then-else" branches and

Algorithm 1 Extract the sub-TCFGs of DI regions

input: The TCFG of the program
output: A set of Sub-TCFGs for the DI regions - S

```
Scan the TCFG to find all the entries for DI regions, and
put the entry nodes e_i in a Set S_entries;
while (S_entries is not empty)
  Pick an entry e_i;
  Traverse the TCFG to find the sub-graph starting
    from e_i and ending with exit node(s);
  if (the sub-graph has multiple exits)
    merge the exits by pointing them to a dummy node;
  end if
  Add the Sub-TCFG of the DI region to S;
end while
return S
```

function calls, and in such cases, we are not able to guide the simulation to the worst-case execution path. Second, the estimation reports more cache misses than the simulation, so the analysis tool also partly contributes to the overestimation.

The overestimation of the OSTimeDlyHMSM() system call is 248.94%, which is far more pessimistic than all the other system calls. The problem comes from a piece of code depicted in Fig. 4. We find that, in the simulated results, the execution of OSTimeDly() at line 2 reports cache misses, and all the other subsequent executions of OSTimeDly() within the while loop (lines 4 and 5) report cache hits; while in the results estimated by Chronos, the executions of OSTimeDly() at lines 2, 4 and 5 are all evaluated to have cache misses, and cache hits are only identified from the second iteration of the while loop. We believe this reflects a defect in the analysis techniques of Chronos.

```
1  // some code here ···
2  OSTimeDly ( (INT16U) ticks );
3  while ( loops > 0 ) {
4          OSTimeDly ( (INT16U) 32768U );
5          OSTimeDly ( (INT16U) 32768U );
6  }
7  // some code here ···
```

Fig. 4 A chunk of OSTimeDly()

The analysis process also helps us to identify some critical bugs in Chronos. In the analysis of the OSSemDel() system call, the estimated WCET is smaller than the

simulated WCET: this means that the analysis techniques cannot guarantee safety. The problem comes from the program structure: if a while loop appears in the first line of a switch branch, the underestimation occurs. We also try to estimate this system call with different processor configurations, and find that even if all the processor features are turned off, the underestimation still exists. Then we write a very simple program with this problematic program structure, and the results are different: when all features are turned off, there is no underestimation; but when all features are turned on, the underestimation comes. The results are listed in Table 2. This should be a bug in Chronos. In this paper, we have to manually modify the source code of μC/OS-II to avoid this problem.

Table 2 Underestimations found in Chronos

Tests	Est.	Sim.
OSSemDel() + features off	329	419
OSSemDel() + features on	6487	8029
simple code + features off	187	186
simple code + features on	1373	1463

Now we make a summary of the obtained results. In the general case, an average of 17.57% overestimation is an acceptable result of WCET analysis. This shows that existing static WCET analysis techniques can be used for RTOS as long as the objective is a single WCET value for each RTOS system call. But the results are obtained with lots of manual efforts, and the human analyzer should be experienced in both the WCET tool and the analyzed RTOS. For example, some loop bounds of the system calls depend on the run-time values of system states. In our practice, we have to manually go through the μC/OS-II source code to identify all the variables that affect the loop bound, and carefully set them in the WCET analysis tool. Similar problems are also reported in the related work [6]. The degree of automation is far from acceptable in the timing analysis of RTOS using existing WCET tools.

5.2 Analysis of the estimated results of DI regions

For all the 79 system calls of μC/OS-II, we successfully analyzed 76 DI regions identified in 57 system calls. Due to some technical reasons, the branch prediction analysis is turned off in the analysis of the DI regions.

All the analysis results are listed in Appendix A2. It is possible that there are more than one DI region in a certain system call, so we just number them sequentially. For each DI region, we count how many basic blocks and branches it has, since these information gives a rough picture of the complexity of the DI regions. Loops are found in only 13 out of 76 DI regions, so the number of loops are not listed in the appendix. Note that SimpleScalar is not able to simulate partial code segments, so the simulation part is missing, and only the WCET of each DI region is given. A special case is that there exists a DI region in OS_Sched() and this function is called in may system calls; then we separately list the results of this DI region.

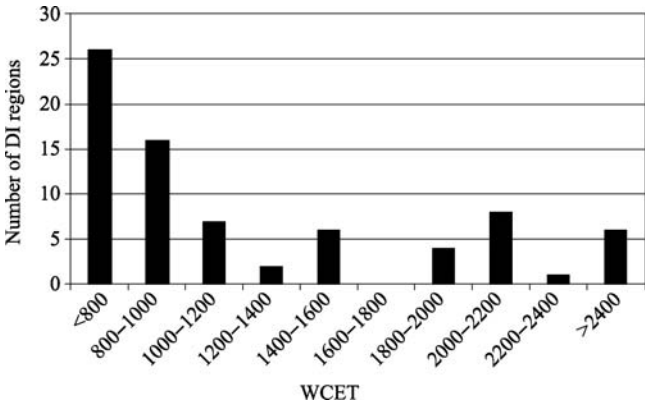

Fig. 5 The distribution of the WCETs of DI regions

Generally, the DI regions do not contain a large number of basic blocks, but some of the DI regions have many branches inside which increase their complexity. Fig. 5 illustrates the distribution of the obtained WCETs. Around 2/3 of the DI regions are relatively short. But there are 6 DI regions whose WCET values (>2400) are much greater than those of the other DI regions. Take a closer investigation of the 6 DI regions, i.e., OSSemPost(), OSMboxPost(), OSMboxPostOpt(), OSQPost(), OSQPostFront(), OSQPostOpt(), we find that they are all system calls that release some system resource. Generally, when a resource is released, the tasks waiting for the resource should be set to the "ready" state. This process contributes a lot to the execution time. Actually the μC/OS-II kernel is well tuned for performance, and there is very little space for improvement in the implementation. To remove this WCET bottleneck, the sementics of resource management probably needs to be redesigned.

6 Further issues

In this section, we present some major problems found in our experiments, which include (1) incapabilities of

single-value WCET analysis of RTOSes, and (2) analysis imprecision in the presence of context switches.

6.1 Incapabilities of single-value WCET analysis

In order to justify whether the WCET analysis techniques adopted can yield good results in static timing analysis of RTOS, we need to exclude all the other sources of overestimations. For example, in one pass of the analysis, if the simulator takes a shorter path and the analyzer takes the worst-case path, then the gap between the estimated result and the simulated result does not reflect the real overestimations introduced by the analysis techniques. In this case, the simulator must be guided to the worst-case execution path by setting the values of the variables that affect the program control flow. The biggest difficulty we encounter in our practice is preparing the correct settings that can guide the simulator to the worst-case execution path.

Preparing the settings leads us to conduct a more intensive study of the characteristics of the control flows in the system calls. Actually, the code of the μC/OS-II kernel has been well tuned for performance, e.g., the number of loops in the system calls is minimized. But a great number of conditional branches are inevitable since the system calls should provide different services according to different system states. The execution time of the same system call in different execution scenarios may vary a lot. We will show it in a series of examples listed in Table 3.

The system call OSMutexPend() is invoked if a task tries to obtain a mutex. The execution time changes due to the availability of the resouce. If the task finds that the mutex is available, it immediately takes the mutex by marking itself as the owner; otherwise, the current task checks if the owner of the resource is ready; if ready, the owner's priority is promoted according to the Priority Inheritance Protocol; then the task puts itself in the waiting list of the mutex and invokes the scheduler to perform context switch. The execution time of the system call in the former scenario is much smaller than the latter. According to the results depicted in Table 3, using the estimated WCET value as the execution time in the former scenario, we get a large overestimation of 413%. The characteristics of OSMboxPost() are similar.

The parameters passed to the system calls also affect the execution time. The system call OSSemDel() is an example. OSSemDel() requires the caller to pass a parameter to the function indicating whether to delete the semaphore if there are tasks waiting for it. If the parameter indicates an unconditional deletion, all the waiting tasks are readied iteratively, which takes a long execution time; otherwise, the execution time is much smaller.

System calls similar to those listed in Table 3 are very common in μC/OS-II, and so do other RTOSes. Different from application code, RTOS code are intrinsically control intensive with the objective to provide different services according to different system states or user requests, so they exhibit high run-time dynamicity. Although static WCET analysis can guarantee *safe* estimations, the analysis goal limits its expressiveness to characterize the timing properties of RTOSes. It is no longer sensible to simply apply a single WCET value to each system call, (doing this implies a very pessimistic overestimation in some system states), and new techniques should be developed for better characterization of the timing properties of RTOSes.

This can be achieved in a step-by-step manner. We can first apply best-case execution time (BCET) analysis on RTOSes. A pair of BCET and WCET values can serve as a rough picture of the real-time performance and timing dynamicity of a system call. Then, parametric WCET analysis can be developed to give more detailed characterization. Bygde and Lisper recently proposed a framework to do parametric WCET analysis [7,8], the idea of which is to obtain a set of formulas representing the WCET in terms of input variables of the program. Bygde's work established a good foundation for parametric WCET analysis, but there is still a gap when applying their theories. We may need to develop methods to model "system states" of an RTOS, then build proper mappings between system states and related variables. Powerful data flow analysis should be accompanied to explore how the

Table 3 Execution times in different scenarios

System Calls	Sensitivity	Est.	Sim.1	Est./Sim.1	Sim.2	Est./Sim.2
OSMutexPend()	Availability of resources	12461	2428	5.13	9251	1.35
OSMboxPost()	Waiting tasks	7440	1560	4.77	6347	1.17
OSSemDel()	Calling parameters	8548	1963	4.35	7437	1.15

"states" affect the control flows. Putting them all together, we may have a comprehensive framework to do parametric timing analysis of RTOSes.

6.2 Imprecision due to context switches

In many system calls of μC/OS-II, the scheduler OS_Sched() may be called, possibly causing context switches. For example in Fig. 6, task T1 wants to release a semaphore during execution, then it calls the OSSemPost() system call. It is possible that there are tasks blocked on this semaphore, so the task with the highest priority is readied. Then OSSemPost() explicitly calls OS_Sched() to invoke the scheduler. The scheduler may find that currently task T2 is ready and its priority is higher than task T1, then the scheduler performs a context switch to T2. After T2 finishes, T1 is the ready task with highest priority, then it resumes execution.

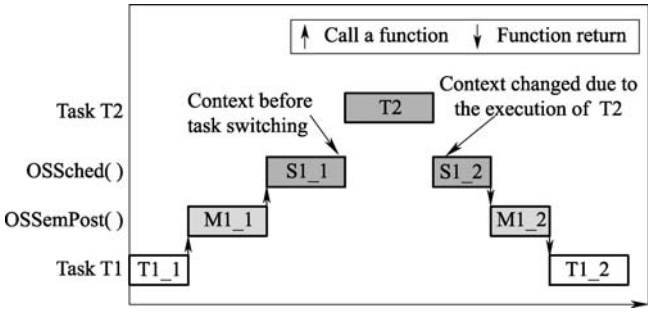

Fig. 6 An example of context switch in μC/OS-II

In this case, the execution of T1, OSSemPost(), and OS_Sched() are split into two parts. Since T2 may modify the contents in the pipelines and caches, the execution times of S1_2, M1_2, and T1_2 are different from those when no context switch is performed. For example, if some cached data of T1_2 is replaced by T2, then the execution time of T1_2 will be larger since it has to reload the data from main memory. But the WCET analysis tool is not aware of the semantics of OS_Sched(), and just treats it as an ordinary function in the analysis. The effects of prolonged execution time due to context switches are neglected in this way, and the execution time may be underestimated.

In our research, the imprecision due to context switches has not been successfully removed yet. A simplest way is to assume worst-case processor states (basically empty pipeline and cache, were there no timing anomaly) at the context switch boundary. But this is too pessimistic: first, context switch may not necessarily occur in OS_Sched() if the task is the highest priority task at that time, and to accurately predict the occurrence of context switches requires runtime information; second, assuming worst-case processor states is too pessimistic. Note that interrupts may also pose similar problems since they introduce context switches, too. Here we omit the detailed discussion on interrupts. To handle context switches is hard, but important in analyzing multi-tasking systems. This will be one of the focuses in our future work.

7 Related work

Research on static timing analysis of RTOS have been conducted by several groups. Colin and Puaut are the first to conduct research on static WCET analysis of RTOS [6]. The RTEMS real-time kernel was analyzed using the HEPTANE tool. Only 12 out of 85 system calls were analyzed. Experiments showed that it is not easy to apply static timing analysis on RTOS. Problems existed in bounding loops, handling irreducible program structures, handling dynamic function calls, and analyzing blocking system calls.

The WCET analysis of the Enea OSE kernel was conducted by Carlsson [9] and Sandell [10] conjunctively. Carlsson's work centered on timing analysis of the Disable Interrupt regions of the OSE kernel. Sandell used the aiT tool to analyze the entire OSE kernel with the objectives of finding out how hard it is to analyze operating systems code and how compiler optimizations affect the manual labor needed to perform an accurate WCET analysis.

Singal and Petters performed WCET analysis of the L4 real-time kernel with the objective of exploring the degree of automation in WCET analysis of RTOS [11]. The analysis tool uses a hybrid design with a tree representation of the CFG and the execution time of each basic block obtained by measurement. Efforts were made to analyze the whole L4 kernel and some obstacles were reported in their paper, including irregular code structures, irregular loops, inlined asembly, dynamic function calls, context switches, etc.

Schneider [12] pointed out that pessimistic estimations in WCET analysis of RTOS mainly come from the lack of application information, and the analysis precision can be improved by considering the applications, and vise versa. Later in Ref. [13], Schneider proposed a framework for combined WCET and schedulability analysis. It is the first

research to consider the interdependence between WCET analysis and schedulability analysis. But the framework was not completely implemented, and no evaluation on its utility in real-life systems was given.

Related work in Refs. [14–17] conducted research on system level timing analysis by considering cache related preemption delays, which are possible solutions to deal with the effects of context switches.

We refer interested readers to Ref. [18] for a survey of research on WCET analysis of RTOS and Ref. [19] for a survey of general WCET research problems and related tools.

8 Conclusion

In this paper, we present a case study where static analysis is used to obtain the WCET of both the system calls and the DI regions of the μC/OS-II real-time kernel. We give a quantitative evaluation of the real-time performance of μC/OS-II by analyzing 61 out of 79 system calls. By applying static analysis techniques on code from real-life systems, we find that traditional WCET analysis cannot properly characterize the timing properties of RTOSes and parametric WCET analysis is highly desirable. Existing techniques may also yield incorrect results in presence of context switches. Our future work will focus on (1) enforcing data flow analysis to identify RTOS system states; (2) developing BCET and parametric WCET analysis techniques to better characterize RTOSes; (3) finding methods to safely and precisely bound the effect of context switches.

Acknowledgements This work was partially sponsored by the National High Technology Research and Development Program of China (863 Program) (2007AA01Z181), the National Natural Science Foundation of China (Grant No. 60973017), and the Natural Science Foundation of Liaoning Province (20082032).

References

1. Lv M, Guan N, Zhang Y, et al. WCET analysis of the μC/OS-II real-time kernel. In: Proceedings of International Conference on Computational Science and Engineering, 2009
2. Labrosse J. MicroC/OS-II the Real-Time Kernel. 2nd ed. CMP Books, 2002
3. Li X, Liang Y, Mitra T, et al. Chronos: A timing analyzer for embedded software. Science of Computer Programming, 2007, 69 (1–3): 56–67
4. Li X. Microarchitecture modeling for timing analysis of embedded software. Ph.D. Thesis of National University of Singapore, 2005
5. Burger D, Austin T M. The Simplescalar Tool Set. 2nd ed. SIGARCH Computer Architecture News, 1997
6. Colin A, Puaut I. Worst-case execution time analysis of the rtems real-time operating system. In: Proceedings of the 13th Euromicro Conference on Real-Time Systems, 2001
7. Lisper B. Fully automatic, parametric worst-case execution time analysis. In: Proceedings of the 3rd International Workshop on Worst-Case Execution Time (WCET) Analysis, 2003
8. Bygde S, Lisper B. Towards an automatic parametric wcet analysis. In: Proceedings of the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis, 2008
9. Carlsson M, Engblom J, Ermedahl A, et al. Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system. In: Proceedings of the 2nd International Workshop on Real-Time Tools, 2002
10. Sandell D, Ermedahl A, Gustafsson J, et al. Static timing analysis of real-time operating system code. In: Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods, 2004
11. Singal M, Petters S M. Issues in analysing L4 for its WCET. In: Proceedings of the 1st International Workshop on Microkernels for Embedded Systems, 2007
12. Schneider J. Why you can not analyze RTOSs without considering applications and vice versa. In: Proceedings of the 2nd International Workshop on Worst-Case Execution Time (WCET) Analysis, 2002
13. Schneider J. Combined schedulability and WCET analysis for real-time operating systems. Ph.D. Thesis of Saarland University, Germany, 2002
14. Staschulat J, Ernst R. Scalable precision cache analysis for real-time software. ACM Transactions on Computer Systems, 2007
15. Lee C, Lee K, Hahn J, et al. Bounding cache-related preemption delay for real-time systems. IEEE Transactions on Software Engineering, 2001, 27(9): 805–826
16. Nemer F, Casse H, Sainrat P, et al. Inter-task wcet computation for a-way instruction caches. In: International Symposium on Industrial Embedded Systems, 2008
17. Burguière C, Reineke J, Altmeyer S. Cache-related preemption delay computation for setassociative caches. In: Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis, 2009
18. Lv M, Guan N, Zhang Y, et al. A survey of WCET analysis of real-time operating systems. In: The International Conference on embedded Software and Systems, 2009
19. Wilhelm R, Engblom J, Ermedahl A, et al. The worst-case execution-time problem—overview of methods and survey of tools. Transaction on Embedded Computing Systems, 2008

Appendixes

Table A2 The results of WCET analysis of μC/OS-II DI regions

DI region	# BB	# Branch	WCET	DI region	# BB	# Branch	WCET
Task management (15)				OSSemPend-1	9	2	872
OSTaskChangePrio	19	8	810	OSSemPend-2	9	2	2143
OSTaskCreate-1	7	2	624	OSSemQuery	7	1	1576
OSTaskCreate-2	3	0	779	OSSemSet	8	2	934
OSTaskResume	13	5	872	Message mailbox management (8)			
OSTaskCreateExt-1	7	2	624	OSMboxAccept	3	0	717
OSTaskCreateExt-2	3	0	779	OSMboxCreate	5	1	779
OSTaskCreateExt-3	5	1	779	OSMboxDel	24	7	903
OSTaskCreateExt-4	5	1	1802	OSMboxPost	14	4	4220
OSTaskDel-1	25	11	2019	OSMboxPend-1	9	2	779
OSTaskDel-2	10	2	1058	OSMboxPend-2	9	2	2143
OSTaskDelReq-1	3	0	655	OSMboxPostOpt	26	8	8073
OSTaskDelReq-2	7	2	872	OSMboxQuery	7	1	1576
OSTaskStkChk	11	4	1058	Message queue management (11)			
OSTaskSuspend	14	5	1182	OSQAccept	8	2	1523
OSTaskQuery	15	4	1058	OSQCreate-1	5	1	779
Memory mangement (4)				OSQCreate-2	5	1	779
OSMemCreate	5	1	779	OSQDel	24	7	903
OSMemGet	5	1	1089	OSQFlush	3	0	965
OSMemPut	5	1	748	OSQPend-1	11	3	1430
OSMemQuery	3	0	1182	OSQPend-2	9	2	2143
Event flags management (9)				OSQPost	16	5	4220
OSFlagCreate	5	1	1027	OSQPostFront	16	5	4220
OSFlagDel	31	9	903	OSQPostOpt	32	11	8073
OSFlagPend-1	28	10	748	OSQQuery	10	2	2072
OSFlagPend-2	19	7	1895	Time management (5)			
OSFlagQuery	3	0	655	OSTimeDly	5	1	1368
OSFlagPendGetFlagsRdy	3	0	655	OSTimeDlyResume	14	5	872
OSFlagPost-1	53	7	717	OSTimeGet	3	0	624
OSFlagPost-2	3	0	655	OSTimeSet	3	0	624
OSFlagAccept	18	6	748	OSTimeDlyHMSM	5	1	1368
Mutual exclusion semaphore management (6)				Miscellaneous (10)			
OSMutexAccept	7	2	1492	OS_Sched*	10	3	2112
OSMutexCreate	7	2	810	OSIntExit	12	4	2298
OSMutexDel	29	10	903	OSInit-1	7	2	624
OSMutexQuery	10	2	1948	OSInit-2	3	0	779
OSMutexPend-1	26	11	1430	OSInit-3	5	1	779
OSMutexPend-2	9	2	2143	OSInit-4	5	1	1802
Semaphore management (8)				OSSchedLock	6	2	841
OSSemAccept	5	1	934	OSSchedUnlock	9	3	872
OSSemCreate	5	1	779	OSTimeTick-1	3	0	686
OSSemDel	24	7	903	OSTimeTick-2	10	4	2112
OSSemPost	14	4	4220				