WCET Analysis of the μC/OS-II Real-Time Kernel[*]

Mingsong Lv[1], Nan Guan[1], Yi Zhang[1], Rui Chen[1], Qingxu Deng[1], Ge Yu[1] and Wang Yi[1,2]

[1] Northeastern University, Shenyang, China
[2] Uppsala University, Uppsala, 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 the RTOS. Traditionally, WCET analysis mainly deals with application programs, while it is crucial to know whether the RTOS also behaves in a timely predictable manner. In this paper, we present a case study where static analysis is used to predict the WCET of the system calls of the μC/OS-II real-time kernel. To our knowledge, this paper is the first to present quantitative results on the real-time performance of μC/OS-II. The precision of applying existing WCET analysis techniques on RTOS code is evaluated, and the practical difficulties in using static methods in timing analysis of RTOS are also reported.

1. Introduction

Hard real-time systems are those systems the tasks of which 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, WCET analysis is used to obtain execution times 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. And 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 accept 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 on application programs and has achieved success in industry (e.g. aiT [2]). While complex real-time systems are composed of both applications and 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.

In this paper, we used Chronos, a static analysis WCET tool, to obtain the WCETs of the system calls (or APIs) of the μC/OS-II real-time kernel. 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 experiment, we successfully checked the WCET of 61 (out of 79) μC/OS-II 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. The difficulties in the analysis process are also analyzed. In our practice, we found that some traditional WCET analysis techniques, such as those adopted in Chronos, are far from adequate in characterizing the timing properties of an RTOS. Problems found from our research include a lack of parametric timing analysis techniques in RTOS analysis, incorrect results in the presence of context switches, etc., which are left to future work.

[*]This work was partially sponsored by the National High Technology Research and Development Program of China (863 Program) under Grant No. 2007AA01Z181, the Cultivation Fund of the Key Scientific and Technical Innovation Project of Ministry of Education of China under Grant No. 706016, and the Natural Science Foundation of Liaoning Province under Grant No. 20082032.

The rest of the paper is organized as follows. Section 2 presents background information on μC/OS-II. Section 3 elaborates the experiment methodology. Analysis results are presented and evaluated in Section 4. Section 5 lists related research, and the paper is concluded in Section 6.

2. The μC/OS-II Real-Time Kernel

μC/OS-II [1, 7] 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 meaningful to evaluate the real-time performance of μC/OS-II quantitatively due to its prevalence in industry.

3. Experiment Methodology

This section gives the detailed experimental settings applied in our experiments: the configurations of μC/OS-II, the WCET analysis tool adopted, and basically how the WCETs of the system calls are obtained.

3.1 μC/OS-II Configurations

The μC/OS-II real-time kernel has a small footprint with 11 C files and 1 header file, which contains 79 system calls spanning 9,771 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 [10] 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 experiment) 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.

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

3.2 The WCET Analysis Tool

In our research, we use Chronos [9] to analyze the μC/OS-II kernel. Chronos is open-source, which allows the users to hack into the tool to locate possible problems. 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 [8], 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 control flow graphs (CFG); then 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 configurations of the target processor in Chronos. Large overestimation is detected in 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 an 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 configurations listed in Table 1 are applied to Chronos.

3.3 How to Obtain the WCETs of System Calls

When trying to use Chronos to obtain the WCET of each individual system call, we encountered 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 it is not possible 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 Figure 1-a.

```
void main(void)
{
    // The system call to be analyzed
    OSSemCreate(5);
}
```

(a)

```
main():
...                                   \
0040d540  addiu   $4,$0,5              > Pre-Inst.
0040d548  jal     00409d70            /
              |
              └──────> OSSemCreate()
0040d550  addu    $29,$0,$30          \
0040d558  lw      $31,20($29)          |
0040d560  lw      $30,16($29)          > Post-Inst.
0040d568  addiu   $29,$29,24           |
0040d570  jr      $31                 /
end_addr
```

(b)

Figure 1. A wrapper program for system calls

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 Figure 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 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 misses of these instructions and their execution times. 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 Equation 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, so these instructions are treated similarly.

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

4. Analysis Results and Evaluation

In this section, we first give an overview of the estimated results obtained in our experiments. Then the major problems found in our experiments are detailed. Note that some of the detailed problems will not necessarily occur if different WCET tools are applied.

4.1 An Evaluation of the Estimated Results

The μC/OS-II kernel has 79 system calls in all, and we have successfully obtained the WCET of 61 system calls. We failed 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` [3] 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 A. An average of 17.57% overestimation is reported. All the WCET values in Table 2, 3 and Appendix A are 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 for 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 function calls, and in such cases, we were 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 other

```
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 ···
```

Figure 2. A chunk of OSTimeDly()

system calls. The problem comes from a piece of code depicted in Figure 2. We found 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 (line 4 and 5) report cache hits; while in the results estimated by Chronos, the executions of `OSTimeDly()` at line 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.

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

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 tried to estimate this system call with different processor configurations, and found that even if all the processor features are turned off, the underestimation still exists. Then we wrote 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 had to manually modify the source code of μC/OS-II to avoid this problem.

Now we make a summarization 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 had 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 related work [6]. The degree of automation is far from acceptable in the timing analysis of RTOS using existing WCET tools.

4.2 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 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 encountered in our practice is preparing the correct settings that can guide the simulator to the worst-case execution path.

Preparing the settings led us to conduct a more intensive study of the characteristics of the control flows in RTOS APIs. 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()` is 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

Table 3. Execution times in different scenarios

System Calls	Sensitivity	Est. WCET	Sim. S-1	Est./Sim.	Sim. S-2	Est./Sim.
OSMutexPend()	Availability of resources	12,461	2,428	5.13	9,251	1.35
OSMboxPost()	Waiting tasks	7,440	1,560	4.77	6,347	1.17
OSSemDel()	Calling parameters	8,548	1,963	4.35	7,437	1.15

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 recently proposed a framework to do parametric WCET analysis [4], 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 "states" affect the control flows. Putting them all together, we may have a comprehensive framework to do parametric timing analysis of RTOSes.

4.3 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 Figure 3, 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 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.

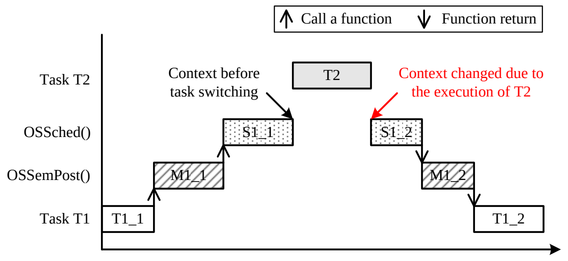

Figure 3. An example of context switch in μC/OS-II

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 data that T1_2 uses is replace 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.

5. 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 [5] and Sandell [13] 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 [16]. 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 in [14] 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 [15], 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 [17, 12] 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 [11] for a survey of research on WCET analysis of RTOS and [18] for a survey of general WCET research problems and related tools.

6. Conclusion

In this paper, we presented a case study where static analysis is used to obtain the WCET of the system calls of the μC/OS-II real-time kernel. We gave a quantitative evaluation of the real-time performance of μC/OS-II by analyzing 61 out of 79 system calls. Applying static analysis on code from real-life systems helps us to find some defects in the analysis tool. We found that traditional WCET analysis cannot properly characterize 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 in Chronos 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.

References

[1] http://www.micrium.com. 2009.

[2] AbsInt. The absint page. *http://www.absint.com*, 2009.

[3] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 1997.

[4] S. Bygde and B. Lisper. Towards an automatic parametric wcet analysis. In *WCET 2008*.

[5] 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 2nd International Workshop on Real-Time Tools*, 2002.

[6] A. Colin and I. Puaut. Worst-case execution time analysis of the rtems real-time operating system. *ECRTS*, 2001.

[7] J. J. Labrosse. *MicroC/OS-II The Real-Time Kernel, Second Edition*. CMP Books, 2002.

[8] X. Li. Microarchitecture modeling for timing analysis of embedded software. *Ph.D. Thesis of NUS*, 2005.

[9] X. Li, Y. Liang, T. Mitra, and A. Roychoudury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67.

[10] M. Lv, N. Guan, Y. Zhang, and R. Chen. Detailed results of WCET analysis of μC/OS-II. *http://www.neurtes.org/techreport.html*, 2009.

[11] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang. A survey of WCET analysis of real-time operating systems. *In ICESS 2009*.

[12] F. Nemer, H. Casse, P. Sainrat, and J. Bahsoun. Inter-task wcet computation for a-way instruction caches. In *SIES 2008*.

[13] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static timing analysis of real-time operating system code. *In ISoLA*, 2004.

[14] J. Schneider. Why you can't analyze RTOSs without considering applications and vice versa. *WCET 2002*.

[15] J. Schneider. Combined schedulability and WCET analysis for real-time operating systems. *Ph.D. thesis of Saarland University, Germany*, 2002.

[16] M. Singal and S. M. Petters. Issues in analysing L4 for its WCET. *in MIKES 2007*.

[17] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *EMSOFT '04*.

[18] R. Wilhelm, J. Engblom, A. Ermedahl, and et al. The worst-case execution-time problem—overview of methods and survey of tools. *Transaction on Embedded Computing Systems*, 2008.

Appendices

A. The Results of WCET Analysis of μC/OS-II

System Call	**Est.**	**Sim.**	**Overest.**	**System Call**	**Est.**	**Sim.**	**Overest.**
Task Management (9)							
OSTaskChangePrio	8,134	6,847	18.80%	OSTaskCreate	15,813	13,724	15.22%
OSTaskCreateExt	Cannot pass simprofile			OSTaskDel	10,198	8,476	20.32%
OSTaskDelReq	1,975	1,868	5.73%	OSTaskResume	5,146	4,053	26.97%
OSTaskStkChk	3,201	3,014	6.20%	OSTaskSuspend	5,558	4,367	27.27%
OSTaskQuery	4,265	4,035	5.70%				
Memory Mangement (4)							
OSMemCreate	3,423	3,335	2.64%	OSMemGet	1,749	1,740	0.52%
OSMemPut	1,780	1,773	0.39%	OSMemQuery	1,994	1,988	0.30%
Event Flags Management (7)							
OSFlagCreate	2,183	2,174	0.53%	OSFlagDel	7,836	6,638	18.05%
OSFlagPend	9,758	7,453	30.93%	OSFlagPendGetFlagsRdy	1,056	1,052	0.38%
OSFlagPost	8,865	7,236	22.51%	OSFlagQuery	1,439	1,432	0.49%
OSFlagAccept	2,601	2,550	2.00%				
Mutual Exclusion Semaphore Management (6)							
OSMutexAccept	2,409	2,393	0.67%	OSMutexCreate	3,503	3,490	0.37%
OSMutexDel	12,029	11,499	4.61%	OSMutexPend	12,461	9,251	34.7%
OSMutexQuery	2,907	2,798	3.90%	OSMutexPost	11,575	11,092	4.35%
Semaphore Management (7)							
OSSemAccept	1,594	1,585	5.68%	OSSemCreate	2,728	2,651	2.9%
OSSemDel	8,548	7,437	14.94%	OSSemPend	8,472	5,467	55.00%
OSSemPost	7,344	6,252	17.47%	OSSemQuery	2,461	2,450	0.45%
OSSemSet	1,724	1,711	0.76%				
Message Mailbox Management (7)							
OSMboxAccept	1,343	1,337	0.45%	OSMboxCreate	2,635	2,558	3.01%
OSMboxDel	8,548	7,435	14.97%	OSMboxPend	8,565	5,808	47.47%
OSMboxPost	7,440	6,347	17.22%	OSMboxPostOpt	11,428	6,721	70.03%
OSMboxQuery	2,461	2,450	0.45%				
Message Queue Management (9)							
OSQAccept	2,251	2,116	6.38%	OSQCreate	4,188	3,658	14.49%
OSQDel	8,796	7,685	14.46%	OSQFlush	1,591	1,585	0.38%
OSQPend	8,658	5,903	46.67%	OSQPost	7,375	6,283	17.38%
OSQPostFront	7,375	6,283	17.38%	OSQPostOpt	11,363	6,816	66.71%
OSQQuery	2,960	2,946	4.75%				
Time Management (5)							
OSTimeDly	4,378	3,303	32.55%	OSTimeDlyHMSM	16,199	6,507	248.94%
OSTimeDlyResume	5,363	4,272	25.54%	OSTimeGet	1,027	1,023	0.39%
OSTimeSet	996	992	0.40%				
Miscellaneous (8)							
OSInit	29,086	25,758	12.92%	OSIntEnter	286	279	2.51%
OSIntExit	1,619	1,490	8.66%	OSSchedLock	627	618	1.46%
OSSchedUnlock	2,115	1,740	21.55%	OSStart	999	992	0.71%
OSTimeTick	4,253	3,496	21.65%	OSVersion	283	279	1.43%

*For more details on the results, please refer to [10].