Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System*

Antoine Colin, Isabelle Puaut
IRISA, Campus de Beaulieu, 35042 Rennes Cédex, France, {acolin/puaut}@irisa.fr

Abstract

An important issue in building operating systems for hard real-time *applications is to compute the worst-case execution times (WCET) of the operating system activities. Traditionally, this has been achieved by an exhaustive testing of the operating system, with a careful attention on the testing conditions to reproduce the worst-case execution scenario. In this paper, we explore the alternative approach of using* static analysis *to predict off-line the WCET of the system calls of a real-time kernel, the RTEMS kernel [11]. We give qualitative and quantitative results on the analysis of RTEMS, and draw some conclusions on the extent to which static analysis can be used on operating system code.*

1. Introduction

Real-time systems differ from other systems by a stricter criterion of correctness of their applications. Actually, the correctness of a real-time application not only depends on the delivered result but also on the time when it is produced. In *hard* real-time systems, missing task deadlines may be catastrophic. It is therefore highly recommended that such systems use *timing analysis* in order to prove, before the system execution, that all deadlines will be met. Timing analysis requires to know for every portion of code taken in isolation its worst-case execution time (WCET), i.e. the worst-case time it may take to execute on a given hardware. WCETs can be obtained either by testing, with careful attention on the testing conditions to reproduce the worst-case execution scenario, or by *static WCET analysis*, in which the WCET of a portion of code is obtained from static analysis of the program source code. Figure 1, borrowed from [2], depicts the traditional layers required for the timing analysis of a hard real-time system.

- **Level 1 - static WCET analysis of basic blocks.** A piece of code can be broken into basic blocks with a single entry/exit point, with no looping, branching or calls. Basic blocks contain no requirement of resources other than the processor. At this level, static WCET analysis computes the WCET of the basic block by *summing* the worst-case execution times of component instructions. *Summing* here means merging the WCETs of instructions by considering the processor microarchitecture (*e.g.* presence of caches that affect the duration of instructions, and pipelines that introduce parallelism between the execution of successive instructions).

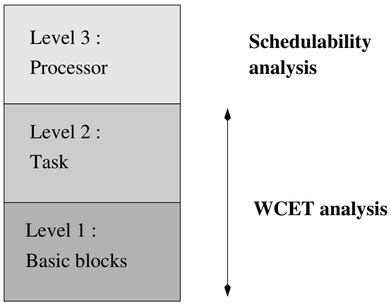


Figure 1. Levels of timing analysis

- **Level 2 - static WCET analysis of tasks.** The basic blocks identified in Level 1 may be composed into tasks by use of looping, branching and call instructions around the basic blocks. At this level, static WCET analysis considers the time requirements of an isolated task, by identifying the control-flow paths in the task and noting the WCET of basic blocks along the path. Tasks are assumed to never block (blocking is introduced at Level 3 described below).

- **Level 3 - Schedulability analysis of a processor.** At this level, multiple tasks execute on the processor and compete for resources, and thus may block while attempting to access to resources. Schedulability analysis must consider the scheduling policy in use (*e.g.* priority preemptive scheduling), and must have static knowledge about the tasks (*e.g.* tasks' periods, deadlines, synchronizations and communications between tasks).

*Appeared in the 13th Euromicro conference on real-time systems, Delft, The Netherlands, June 2001

In this paper, we are concerned with the two lowest level of timing analysis (levels 1 and 2), named hereafter *static WCET analysis*.

To be able to apply timing analysis on a complete system, all its components (application software, operating system, hardware) must be *predictable* [15], *i.e.* must provide support to predict the evolution of tasks and guarantee in advance that all deadlines will be met. To be predictable, a real-time operating system must: (i) operate in a timely fashion (all its activities must have a known WCET); (ii) it must provide mechanisms to avoid nondeterministic blocking delays in application codes (e.g. priority inheritance); (iii) it must not introduce nondeterministic delays during the execution of applications (for instance, demand paging should not be used because it introduces unpredictable delays caused by page faults and page replacement). In this paper, we do not address the whole issue of predictability of real-time operating systems, but rather focus on the first point.

1.1. Related work

Predictability of real-time operating systems. Real-time operating systems can be divided into three main categories: real-time extensions of time-sharing operating systems, priority-based kernels for embedded applications, and research operating systems. Operating systems of the first category (e.g. real-time extensions of Unix for instance) can exhibit predictability problems, because some of the mechanisms they include were not designed with real-time applications in mind. Operating systems from the second category (e.g. pSOS, VxWorks, RTEMS, Chorus, Nucleus) have in general as objective to achieve high performance in terms of *average* response time to external events. The lack of commercial operating systems having a highly predictable timing behavior induced researchers to design new operating systems paradigms for hard real-time applications (third category of systems mentioned above). Systems in this category are for instance RT-Mach, Mars and Spring.

Most real-time operating systems only come with information on the *average* execution times of their system calls, obtained through benchmarks (*e.g.* Chorus, RTEMS [11]). Few of them (*e.g.* Nucleus [8]) come with execution times obtained through tests carefully designed to be representative of the system worst-case behavior. While this approach is better from a predictability viewpoint compared to the provision of average response times, it is error-prone. Indeed, the identification of the worst-case execution scenario is entirely under the control of the operating system designer, which may commit errors due to the complexity of the code being dealt with. To our knowledge, no previous study has been undertaken yet to apply static analysis to obtain timing information on the code of a real-time operating system.

Static WCET analysis. Together with schedulability analysis, static WCET analysis forms the basis for establishing confidence into the timely operation of a hard real-time system. While schedulability analysis is one of the traditional fields of investigation in real-time systems research, static WCET analysis caught the attention of the research community only about ten years ago. However, substantial progress has been made in the area over the last decade [12]. Two major issues have to be dealt with to achieve static WCET analysis: characterizing the possible execution paths, and computing the execution time of a given execution path in an accurate and safe manner by considering the hardware state (*hardware-level* WCET analysis). Studied architectural features are pipelines [16, 7], caches [10], and branch prediction [3].

The most common focus for static WCET analysis is user code. Moreover, most published results on static WCET analysis (except [6]) have been experimented on small scientific benchmark programs. The study described in [6] copes with more complex programs; the purpose of this study is to report static properties of (user-level) embedded programs in the areas of telecommunications, vehicular control, home and consumer electronics, in order to provide guidance for the design of static WCET analyzers. To the best of our knowledge, no prior study has dealt with the static WCET analysis of real-time operating systems.

1.2. Paper contribution and organization

This paper reports the results of the static analysis of the RTEMS kernel [11] to obtain the WCET of its system calls. RTEMS has been analyzed by a tree-based WCET analyzer we have developed, named HEPTANE [4].

The problem of analyzing statically the source code of an operating system is that it contains low level code to manage the underlying hardware (processor, peripherals). Such code can use language constructs that cannot be easily handled by most state of the art static WCET analyzers (presence of assembly language code, use of pointers, etc). The first interest of our work is to point out the problems we have encountered, and the solutions we have adopted to solve them. An interesting result is that with minor modifications to the source code of RTEMS, all system calls related to task management and semaphore management could have been analyzed. Another contribution of our study is the identification of the conditions under which the system calls of RTEMS have the worst-case execution time. The knowledge of these conditions can be used to establish worst-case benchmarks of the behavior of RTEMS, and hopefully, of other operating systems.

The remainder of this paper details the results of the study. It is organized as follows. Section 2 first gives an overview of the RTEMS kernel. Section 3 then briefly describes the static WCET analysis tool, named HEPTANE, used for the experiments. Qualitative and quantitative results of the static WCET analysis of RTEMS are detailed in Section 4. Finally, we conclude in Section 5 with a summary of the results and a perspective for future work.

2. The RTEMS real-time kernel

RTEMS (Real-Time Executive for Multiprocessor Systems) [11] is a small open-source real-time kernel for embedded applications. RTEMS includes the following features:

- multitasking capabilities,
- event-driven, priority-based, preemptive scheduling,
- intertask communication and synchronization,
- priority inheritance, used to avoid the problem of priority inversions,
- dynamic memory management,
- high level of user configurability.

RTEMS offers two Application Programmer Interfaces (API): a POSIX interface and the RTEMS interface. In the following, we focus on the RTEMS interface.

The RTEMS interface is made of 85 system calls (*directives* in the RTEMS terminology) provided by 17 modules (namely initialization, task, interrupt, clock, timer, semaphore, message, event, signal, partition, region, dual ported memory, I/O, fatal error, rate monotonic, user extensions, and multiprocessing). All entities managed by RTEMS, such as tasks, semaphores, and messages (*objects* in the RTEMS terminology) are named and allocated in a unified manner.

3. The HEPTANE static WCET analyzer

The static WCET analysis of RTEMS was conducted using the HEPTANE tree-based static WCET analyzer. HEPTANE (*Hades Embedded Processor Timing ANalyzEr*) is a tree-based static WCET analyzer provided by the HADES environment [1] currently under construction at IRISA. HEPTANE analyzes programs written in ANSI C, and is designed to produce timing information for a Pentium 90MHz processor.

3.1. Path analysis

The computation of a program's WCET by HEPTANE highly relies on the program syntax, and uses the program syntax tree, generated from the program source code. Timing equations that extend the ones proposed by Puschner and Koza in [13] are used. To illustrate the principle of tree-based static WCET analysis, an extremely simplified example of timing equation for a conditional construct *"CO"* with code *"if (C) S; else E;"* is *"WCET(CO) = WCET(C) + max(WCET(S),WCET(E))"*. WCETs are then computed recursively in a bottom-up manner thanks to the program syntax tree.

Finding the WCET of a loop requires a knowledge of the maximum number of loop iterations. HEPTANE currently requires the user to give this information through annotations in the source program. Annotations are designed to support nested loops whose number of iterations depends on the counter variables of outer loops [3].

Some restrictions on the source code (usual in the domain of static WCET analysis) are required to identify the worst-case execution path of a program. On the one hand, the use of function pointers is forbidden in order to identify called functions statically. On the other hand, features that may cause *unstructured control flow graphs* (*i.e.* control flow graphs that are not compatible with the syntax tree) are not allowed. This restriction comes from the fact that our analysis highly relies on the program syntax. Such features are for instance the use of the *goto* construct and the inclusion of assembly code that contains branching instructions.

3.2. Microarchitecture modeling

Efficiently modeling the microarchitecture allows to reduce the pessimism of static WCET analysis. HEPTANE integrates mechanisms to take into account the effect of instruction cache, pipeline and branch prediction. The presence of pipelines is tackled by an off-line simulation of the flow of instructions in the pipelines. Considering the instruction cache uses static cache simulation [10], whose result is to classify every instruction according to its worst-case behavior with respect to the instruction cache. A similar approach is used to integrate the effect of branch prediction [3]. The modeling of the *instruction cache*, *branch predictor* and *pipeline* produce results expressed in a microarchitecture-independent formalism [4], thus allowing HEPTANE to be easily modified or retargeted to a new architecture.

4. Results of the analysis of RTEMS

The conditions in which the experiments were conducted are described in § 4.1. Then, we give qualitative results of the study by considering all sources of problems encountered during the study: unstructured control-flow graphs (§ 4.3), function calls (§ 4.4), loops (§ 4.5), blocking calls and context switches (§ 4.6), and interrupts (§ 4.7). Quantitative results are given in § 4.8.

4.1. Experimental setup

The experiments were conducted on the source code of RTEMS 4.0.0 for a i386 architecture. A monoprocessor architecture is assumed (the kernel source code is specialized to remove the code dealing with shared memory multiprocessors). In the following, we report the results of the analysis of twelve key directives of the RTEMS kernel (see table below), dealing with task management and synchronization.

Name	Description
rtems_clock_get	Get system date & time information
rtems_semaphore_create	Create a semaphore
rtems_semaphore_obtain	Acquire a semaphore
rtems_semaphore_release	Release a semaphore
rtems_task_create	Create a task
rtems_task_delete	Delete a task
rtems_task_ident	Get the identifier of a task
rtems_task_start	Start a task
rtems_task_suspend	Suspend a task
rtems_task_resume	Resume a task
rtems_task_set_priority	Set task priority
rtems_task_wake_after	Wake up after time interval

The source code that implements the twelve studied directives consists of 91 source files: 28 files containing C code, one file containing assembly code, 38 include files containing definitions of constants, macros and type declarations, and 24 files containing in-line functions. The source code contains 14532 (non-comment) lines of code, and defines 355 functions.

4.2. Problems due to the structure of RTEMS

A large majority of called functions (89%) are declared in a different file from their caller. Thus, to be used easily, a static WCET analysis tool should be able to treat functions scattered in different files and directories. As HEPTANE currently works on a single source file, several steps were required to merge all the files needed to analyze each directive in a single file.

In addition, we observed that most files are used for the analysis of several RTEMS directives. In average, a file containing source code is used for the analysis of 6 RTEMS directives. As a consequence, since static WCET analysis is a time-consuming task (up to 10 mn at worst per RTEMS directive), we think it would be advantageous to be able to save partial WCET results and reuse them later on. An important issue to be tackled is then to define what information should be saved with a partial WCET result to obtain WCETs that are not too pessimistic: at least, the contents of the instruction cache, the branch target and the pipeline must be saved so that partial WCET results are not merged in a too pessimistic way (*e.g.* by assuming that the instruction cache and the pipelines are empty).

4.3. Unstructured control flow graphs

Only one assembly source file was needed to analyze the twelve selected RTEMS directives. It contains the code to save and restore the context of a task (*i.e.* save and restore general and floating point registers). This code did not cause any problem concerning static WCET analysis because it does not contain branching instructions.

Four loops containing *gotos* were encountered during the study. In both situations, it has been possible to rewrite the loops in a straightforward manner.

4.4. Function calls

Dynamic function calls. Calls to functions through function pointers (hereafter *dynamic* function calls) are a problem for static WCET analysis, because the called function is known only at run-time. Among the function calls found in the twelve studied RTEMS directives, we found only 15 dynamic function calls that have been classified as follows:

- **Calls to user-extensions.** RTEMS allows the application developer to augment the kernel with *extension routines*, which are invoked at critical system events, like for instance task creation or task context switch. New extension routines can be added dynamically. To deal with extension routines, one drastic approach would consist in forbidding the use of user extensions. The alternative approach we have adopted consists in requesting to the application developer an identification of all the extensions routines to be added at run-time. Note that RTEMS itself uses the user-extension facility to connect code at task creation, task deletion, task start and task restart. Fortunately, the extension routines added by the kernel are known before execution, and the dynamic calls could have been easily replaced by the actual function calls.

- **Calls to API extensions.** The POSIX interface and the RTEMS interface both rely on a common software layer. API extensions allow a given API to supply API extension routines that will be invoked by this common layer at specific points: after a task context switch, before/after a call to a device driver. Fortunately, for a given API (here the RTEMS API), called functions are known statically.

A similar situation was found for dynamic function calls that serve at calling architecture-dependent hooks (in the i386 architecture, two hooks called respectively when allocating and freeing a stack). Since the target architecture is known statically, the dynamic calls could have been easily replaced by static ones.

Recursive calls. The presence of recursion can be a problem for some static WCET analysis tools, to identify the worst-case execution path of a program. We found no recursive functions in the analyzed code. This conclusion differs with the one reached in [6] which reports statistics about user code (in this study, 0.3% of the functions are recursive).

4.5. Loops

Analyzing loops is of prime importance in static WCET analysis to identify the worst-case execution path of a program. The analyzed code contains a rather low number of loops (only 21 loops in the 14532 analyzed lines of code).

Nested loops. Nested loops can be a problem for static WCET analysis, because a pessimistic estimation of the maximum number of loop iterations of a nested loop can have a great influence on the program WCET [3]. The study found no nested loops in the analyzed code. Again, this conclusion contrasts with the one reached in [6] which reports statistics about application code, which reports in average 9% of nested loops.

Non-terminating loops. A single non-terminating loop was encountered in the whole RTEMS source code. It is the body of the *idle* task, which is executed only when no other task is ready to execute. The function in which this loop appears is never called, and thus has no influence on WCET analysis.

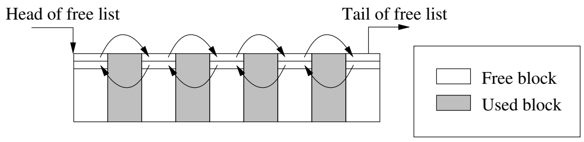
Determination of the maximum number of iterations. As said in §3.1, HEPTANE needs the source code to be annotated. Annotations provide the static WCET analyzer with the maximum number of iterations for each loop. While the number of loops is rather small, finding the maximum number of loop iterations was trivial for only four of them (*for* loops whose bounds were given directly by the loop header). The remaining 17 loops required a study of the source code of RTEMS. We report below the problems encountered, classified per type.

- *Management of object names.* Objects in RTEMS can be named using two kinds of external names: C strings or raw names made of a sequence of 32-bits values. Two similar sets of functions acting on names (name comparison, name copy, name clearing) are provided. Each function includes a loop whose number of iterations depends on the name length. Objects that are created through the RTEMS API (*e.g.* tasks, semaphores) are named with raw names made of a sequence of length *one* (type *rtems_name* in the RTEMS interface). The kernel itself does not create raw names with a sequence length longer than one. Thus, all loops managing raw names where then annotated with value *one*.

- *Name to identifier conversion.* The RTEMS directive *rtems_task_ident*, given a 32-bits raw name, returns the associated task identifier. The search for the task identifier is achieved by a loop that scans the system task table for the specified name. The upper bound of the loop index does not appear explicitly in the loop header. However, the way the system task table is managed implies that this loop iterates at worst *MAX_TASK* times, where *MAX_TASK* is the maximum number of tasks in the system at a given time. Note that RTEMS imposes that this information is known statically: it is given in a configuration table which is used by RTEMS to reserve space for task descriptors.

- *Management of user extensions.* At every system event at which a set of extension routines can be connected (*e.g.* task context switch), the code includes a loop calling all extension routines in the set. Since we request the application developer to identify all the user extensions he wishes to connect (see 4.4), and since the number of user extensions added by RTEMS itself are known, annotating this loop was straightforward.

- *Heap management.* RTEMS uses dynamic memory allocation at task creation and task deletion time to allocate and free the stack and the floating point area for the task. RTEMS uses a *first-fit* memory allocation strategy with a minimum size for allocated blocks (say *MIN_SIZE*). Free blocks are linked together using a doubly linked list. The memory allocation function contains a loop that scans forward the list of free blocks until a block with at least the requested size is found. In the worst situation, the memory is made of a succession of free blocks and busy blocks, both of size *MIN_SIZE* (see figure below). The worst-case number of iterations for this loop is then *HEAP_SIZE* / (2 * *MIN_SIZE*), where *HEAP_SIZE* is the amount of memory dedicated to dynamic memory allocation.



Clearly, the *first-fit* allocation strategy is not suited to static WCET analysis, and more generally is not suited to hard real-time systems. The overestimation of the execution time of the memory allocation function can impact seriously the worst-case execution time of the task creation routine.

- *Management of run queues.* The RTEMS scheduler allocates the processor using a priority-based, preemptive algorithm, augmented to provide round-robin characteristics for tasks having the same priority. The goal of this algorithm is to guarantee that the task which is executing on the processor at any time is the one with the highest priority among all tasks in the ready state. There are loops in the code of the scheduler that are in charge of en-queuing a task descriptor in the run-queue of the scheduler depending on its priority level. They scan the run-queue until a task descriptor with a given priority level is found.

These loops iterate at worst *MAX_READY_TASK* times, with *MAX_READY_TASK* the maximum number of ready tasks (at worst, *MAX_READY_TASK* is equal to the maximum number of tasks *MAX_TASK*). A less pessimistic figure could be obtained if the user has static knowledge on the repartitions of tasks in the run-queue.

- *Main loop of the scheduler.* The most difficult part to analyze in the source of RTEMS was the code of the context switch (function *_Thread_Dispatch*, whose simplified code is given in Figure 3).

```
void _Thread_Dispatch( void )
{
  Thread_Control   *executing;   /* the executing thread */
  Thread_Control   *heir;        /* the thread to switch to */

  executing   = _Thread_Executing;
  _ISR_Disable( level );                          (I−dis1)
  while ( _Context_Switch_necessary == TRUE ) {
     heir = _Thread_Heir;
     _Thread_Dispatch_disable_level = 1;          (D−dis)
     _Context_Switch_necessary = FALSE;
     _Thread_Executing = heir; /* Select the new thread */
     _ISR_Enable( level );                        (I−ena1)
     /* Switch to the thread */
     _Context_Switch( &executing→Registers,&heir→Registers);
     executing = _Thread_Executing;
     _ISR_Disable( level );                       (I−dis2)
  }
  _Thread_Dispatch_disable_level = 0;             (D−ena)
  _ISR_Enable( level );                           (I−ena2)
}
```

Figure 3. Code of the RTEMS context switch

Function *_Thread_Dispatch* is called by all RTEMS directives and interrupt handlers that can modify the processor attribution. Each time the run-queues are modified, a shared variable named *_Context_Switch_necessary* is set to *TRUE* and the processor attribution is not modified yet. The processor attribution is changed exclusively by function *_Thread_Dispatch*. Interrupts are disabled during the choice of the task to be dispatched (portions of code **(I-dis1)–(I-ena1)**, **(I-dis2)–(I-ena1)** and **(I-dis2)–(I-ena2)**). A variable named *_Thread_Dispatch_disable_level* serves at delaying context switches requested by interrupt handlers that are executed during the dispatch loop (*i.e.* during code portion **(D-dis)–(D-ena)**). Each interrupt handler wishing to modify the processor attribution calls function *_Thread_Dispatch* only when variable *_Thread_Dispatch_disable_level* is null (*i.e.* if no context switch is under way). The context switch is then achieved by the interrupted *_Thread_Dispatch* function through additional loop iterations. Figure 2 illustrates a possible execution of function *_Thread_Dispatch*. In this execution, a process *P1* tries to obtain a semaphore which is already held by another process.

When *rtems_semaphore_obtain* finds out that the semaphore is not free, the scheduler run-queues and the semaphore waiting queues are modified, and variable *_Context_Switch_necessary* is set to *TRUE* to request the context switch (mark ①). Later on, function *_Thread_Dispatch* is called to achieve the actual context switch, and iterates a first time (mark ②). Assume a clock interrupt occurs during the first loop iteration and causes the processor attribution to be revised due to the expiration of a time quantum (mark ③). Since variable *_Thread_Dispatch_disable_level* is set to one, the context switch is delayed and is achieved by a second loop iteration in the interrupted *_Thread_Dispatch* execution (mark ④). The task with the highest priority (here *P2*) is then resumed.

If the only source of interrupt in the system is the clock interrupt, then, due to the time interval between clock interrupts, there can be at most one clock interrupt during the execution of function *_Thread_Dispatch*. Its loop can then be annotated with value *two*. If other sources of interrupt exist, bounding the number of loop iterations is feasible if enough information is available to bound the number of context switches requested by interrupt handlers during the execution of function *_Thread_Dispatch*.

4.6. Blocking calls and context switches

As stated in section 1, determining blocking durations is the responsibility of schedulability analysis rather that the one of static WCET analysis, which is designed to analyze code that do not include internal synchronization. Thus, during the study, we only had to verify that the kernel itself does not use blocking primitives (e.g. through semaphores). This verification was achieved through a manual analysis of the kernel source code. Fortunately, RTEMS does not use blocking primitives to control concurrent accesses to its data structures, but instead disables interrupts. If blocking calls had existed in the kernel, it would not be possible to perform WCET analysis separately from schedulability analysis, because blocking delays would have to be identified.

For RTEMS directives that can cause a context switch (*e.g.* directive *rtems_semaphore_obtain*), due to the structure of the code for the context switch (function *_Thread_Dispatch* of figure 3), the static WCET analysis tool takes into consideration the time required to save and restore the task context (function *_Context_Switch* in the code of *_Thread_Dispatch*). The WCET of a task thus includes the time required to switch to another task in the case the calling task blocks. However, the static WCET analysis tool is not aware of the fact that *_Context_Switch* actually performs the context switch, because it does not have any knowledge of the semantics of the code it analyzes. It considers that

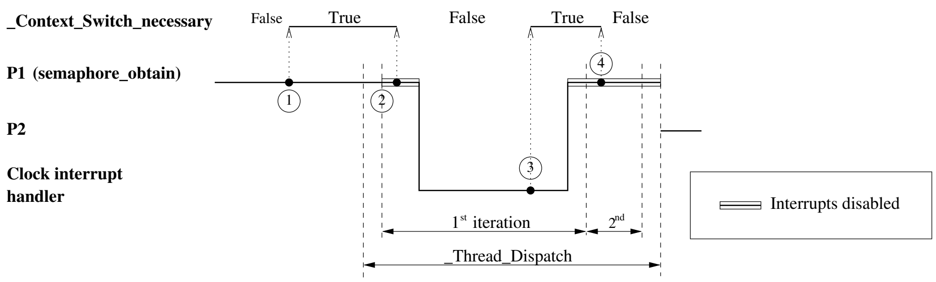


Figure 2. Illustration of function *_Thread_Dispatch*

_Context_Switch is an ordinary function and thus cannot integrate the overhead of cache, branch target buffer and pipeline reconstruction due to the context switch. At worst, it must be assumed that the entire cache and branch target buffer are flushed at every context switch; the corresponding overhead must be added to the WCET of function *_Context_Switch* that has been computed by the static WCET analyzer.

4.7. Influence of interrupts

An interrupt can occur at almost any time during the execution of a task. The presence of interrupts has a direct influence on the WCET of a task, because it conditions the number of iterations of the main loop of the scheduler (see § 4.5). This is the only impact of interrupts on static WCET analysis. All other influences are relevant to schedulability analysis, which is outside the scope of this paper. Examples of integration of interrupts in schedulability analysis can be found in [9] and [14]. Note however that WCET analysis can be used to compute the WCET of interrupt handlers.

4.8. Quantitative results

This paragraph gives quantitative results concerning the static WCET analysis of RTEMS. It compares results obtained through static analysis with results collected during the actual execution of RTEMS directives.

Experimental results were obtained on code compiled with no optimization. For each directive, we have reproduced its worst-case execution scenario. For instance, for the directives that use dynamic memory allocation (e.g. *rtems_task_create*), we have introduced code to fragment the heap before calling the primitive. The Pentium on-chip monitoring features (Pentium event counter registers) allowed us to count executed instructions, branch mispredictions and cache misses. Since our analysis does not model the data cache and the second pipeline of the Pentium processor, in the measurements these features are disabled.

Table 1 gives the results. Columns labeled *"est."* are the results obtained by static analysis and columns labeled *"exec."* are the results obtained by executing the directives.

Column II gives the number of instruction-block reads that cause a cache miss, and column III shows the number of mispredicted conditional branches. The average ratio concerning cache misses is about 1.44, whereas the average branch misprediction ratio is 2.65. Column IV gives the estimated and measured WCETs, expressed in clock cycles (on a 90Mhz Pentium). The ratio between estimated and actual WCET is never less than 1 (otherwise the analyzer would be incorrect), the average ratio is 1.86.

An in-depth analysis of the source code of the twelve analyzed RTEMS directives allowed us to identify two main sources of pessimism of the analysis. The first one is the multiple call in the same directive of a function, which is clearly executed at most once: this function is the *Thread_Dispatch* function, which achieves a context switch. It has a large WCET and appears up to 3 times in the call-graph of some directives.

The second source of pessimism comes from some functions that can be called both indirectly by the user and by RTEMS directives. In the latter case, it appears that the majority of the parameters of these functions are constant. For instance, the RTEMS queue management functions have a parameter named *timeout* which is NULL when the queue management functions are called by RTEMS itself. In this situation, a part of the function code is not used, which leads to an overestimation of the function WCET, due to infeasible paths taken into account to compute the WCET estimate.

5. Conclusion

We have successfully used static WCET analysis to obtain the WCETs of the system calls of a real-time operating system, the RTEMS kernel. We found that the restrictions on the source language made by HEPTANE are reasonable: most dynamic calls could have been replaced by static ones, and gotos were easily replaced by loops. Finding the maximum number of iterations for loops was trivial for only 25% of them. Finding the maximum number of iterations for the remaining loops required an in-depth study of the source code of RTEMS. We found rather pessimistic bounds for the loop of the dynamic memory allocation rou-

I	II			III			IV		
	Instruction cache misses			Branch mispredictions			WCET (clock cycles)		
RTEMS function	*est.*	*exec.*	ratio	*est.*	*exec.*	ratio	*est.*	*exec.*	ratio
rtems_clock_get	10	10	1.00	8	7	1.14	1420	1302	1.09
rtems_semaphore_create	106	88	1.20	204	112	1.82	27050	11878	2.28
rtems_semaphore_obtain	380	180	2.11	983	363	2.71	288484	97417	2.96
rtems_semaphore_release	227	137	1.66	271	131	2.07	40414	17025	2.37
rtems_task_create	393	231	1.70	852	283	3.01	435550	309618	1.41
rtems_task_delete	315	213	1.48	630	262	2.40	72679	47275	1.54
rtems_task_ident	23	35	1.21	229	31	7.39	150093	98439	1.52
rtems_task_resume	93	90	1.03	138	83	1.66	19045	11521	1.65
rtems_task_set_priority	164	140	1.17	182	139	1.31	24728	20179	1.23
rtems_task_start	473	214	2.21	730	149	4.90	138940	36532	3.80
rtems_task_suspend	111	100	1.11	149	110	1.35	20106	16237	1.24
rtems_task_wake_after	156	116	1.34	250	122	2.05	19823	16923	1.17
Average			1.44			2.65			1.86

Table 1. Analysis and experimental results

tine, which is clearly not well suited for static WCET analysis. The maximum number of iterations of the other loops (*e.g.* the scheduler loops) depends on the operational conditions, such as the arrival law of interrupts or the actual number of active tasks at a given priority. Some knowledge from the user could avoid considering the most pessimistic bound on the number of loop iterations.

Concerning our current work, we are working on the use of *partial evaluation* before static WCET analysis (*i.e.* specialization of programs by exploiting partial knowledge of its inputs) to improve static WCET analysis accuracy. We have successfully used Tempo [5], a partial evaluator for C, to specialize the code of RTEMS, and currently study the gains with respect to the precision of static WCET analysis.

Acknowledgments

This work is partially supported by the French Departement of Defense (DGA/DSP), under contract number #198C4980031303011. We would like to thank Philippe Selo, student with INSA, who actually achieved the study of RTEMS during its summer internship. We thank Éric Valette from Canon research France, Pascal Chevochot and David Decotigny from IRISA and the anonymous reviewers for their comments and helpful suggestions.

References

[1] E. Anceaume, G. Cabillic, P. Chevochot, and I. Puaut. A flexible run-time support for distributed dependable hard real-time applications. In *Proceedings of the 2nd International Symposium on Object-oriented Real-time distributed Computing*, pages 310–319, St Malo, France, May 1999.

[2] N. Audsley and M. Burke. Distributed fault-tolerant avionic systems - a real-time perspective. In *Proceedings of IEEE Aerospace*, pages 8.104.1 – 8.104.18, Aspen, U.S.A., 1998.

[3] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, May 2000.

[4] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proc of the 13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[5] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys*, 30(3), Sept. 1998.

[6] J. Engblom. Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. In *Proceedings of the 1999 Real-Time technology and Applications Symposium*, pages 46–55, Vancouver, CA, June 1999.

[7] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS96)*, pages 288–297, Dec. 1996.

[8] A. T. Inc. Nucleus PLUS embedded real-time kernel. Accelerated technology, $http://atnucleus.com$, 1999.

[9] K. Jeffay and D. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS93)*, pages 212–221, Raleigh-Durham, North Carolina, Dec. 1993.

[10] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2):217–247, May 2000.

[11] On-Line Applications Research Corporation, Huntsville, AL, USA. *RTEMS Applications C User's Guide*, 4.0 edition, Oct. 1998. (http://www.oarcorp.com/RTEMS/rtems.html).

[12] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2-3):115–128, May 2000. Guest Editorial.

[13] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, Sept. 1989.

[14] K. Sandström, C. Eriksson, and G. Fohler. Handling interrupts with static scheduling in an automotive vehicle control

system. In *Proceedings of the 1998 Real-Time technology and Applications Symposium*, Hiroshima, Japan, Oct. 1998.

[15] J. Stankovic and K. Ramamritham. What is predictability for real-time systems ? *Real-Time Systems*, 2:247–254, 1990.

[16] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, Oct. 1993.