Integrated Timing Analysis of Application and Operating Systems Code

Lee Kee Chong Clément Ballabriga Van-Thuan Pham Sudipta Chattopadhyay Abhik Roychoudhury
National University of Singapore
{cleekee,clementb,thuanpv,sudiptac,abhik}@comp.nus.edu.sg

***Abstract*—Real-time embedded software often runs on a supervisory operating system software layer on top of a modern processor. Thus, to give timing guarantees on the execution time and response time of such applications, one needs to consider the timing effects of the operating system, such as system calls and interrupts — over and above modeling the timing effects of micro-architectural features such as pipeline and cache. Previous works on Worst-case Execution Time (WCET) analysis have focused on micro-architectural modeling while ignoring the operating system's timing effects. As a result, WCET analyzers only estimate the maximum *un-interrupted* execution time of a program. In this work, we present a framework for RTOS aware WCET analysis - where the timing effects of system calls and interrupts can be accounted for. The key observation behind our analysis is to capture the timing effects of system calls and/or interrupts, as well as their effect on the micro-architectural states, *compositionally* via a damage function. This damage function is then composed in a controlled fashion to result in a RTOS-aware, micro-architecture-aware timing analysis of an application. We show the use of our analysis to compute the worst-case response time for a real-life robot controller software which runs several tasks such as balancing and/or navigation on top of a real-time operating system running on a modern processor.**

I. Introduction

Real-time and embedded software contains several components that need to function in the presence of timing constraints imposed by the environment. The violation of such time constraints may have serious consequences, particularly for hard real-time systems. However, providing the necessary timing guarantees are increasingly difficult due to the complex implementation of such hard real-time embedded systems - they involve pieces of application software mediated by an operating system (which acts as the supervisory software), all running on top of a modern processor. Thus, providing timing guarantees involves timing analysis of application software in presence of the operating system (taking into account *system calls* and *interrupts*), and the underlying processor (taking into account features like pipeline and cache). In this paper, we provide a *general* solution to this problem, and demonstrate an instantiation of the solution for a real-life robot controller.

We envision an application scenario that contains an autonomous robot and the robot functions in a hazardous environment, such as an area with radiation leak. The robot controller runs several tasks, including *balancing* a robot (such that the robot does not fall on ground) and *navigation* to guide a robot away from obstacles. In the case where time-critical tasks such as balancing and navigation — do not respond within an appropriate time, the robot will clearly malfunction (such as falling on the ground) or worse still, the robot may get damaged in a collision with a heavy obstacle.

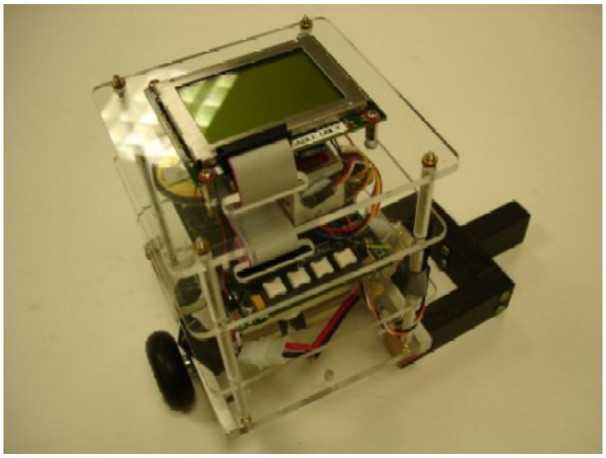


Fig. 1: Bally2 [1], a real life experimental self balancing robot in which its controller is the base for our robotic controller

The robot needs to navigate itself from a specified starting point to a specified ending point, while avoiding all obstacles coming on the way. Such a robot is supported on the ground with two wheels and the robot must maintain balance while navigating through rough terrain. The embedded controller in the robot performs several calculations based on input from different sensors and moves the wheels to keep the robot upright and to avoid obstacles. To realize our application scenario, we adapted some real life open source robotic applications written for EyeBot [4] (a microcontroller hardware designed for robotic applications) to our chosen hardware architecture and operating system.

Figure 1 shows a real life experimental robot in which its controller forms the base for the robotic controller evaluated in our work. Table I gives an outline of the time-critical tasks in our application. Table I shows that it is absolutely important to know the timing behavior of different tasks before such a robot is deployed. For example, the task `balance` *must finish* computation (or respond) within $\frac{1}{50}$ seconds, once it receives inputs from sensors. Therefore, our primary goal is to provide a guarantee on such response time, meaning that our provided response-time guarantee must be met in any scenario of the operating robot. Such a guarantee on task response time can be provided via *worst case response time* (WCRT) analysis.

Task	Real-time constraints
`balance`	Must consistently run at 50Hz to continuously adjust an upright position.
`navigation`	Must consistently run at 20Hz to safely avoid obstacles.
`remote`	Should finish processing within 100ms to react quickly to remote command.

TABLE I: Real-time constraints of robot controller tasks

Computing the WCRT of an embedded robot controller leads to several technical challenges. First and foremost, accurate computation of WCRT requires the knowledge of *worst case execution time* (WCET) of individual controller tasks. WCET of a task captures an upper bound on the uninterrupted execution time of the respective task. Since timing is extremely sensitive to the underlying execution platform, WCET of a task also depends heavily on the underlying platform. Therefore, WCET computation usually involves a micro-architectural modeling stage that analyzes the timing behavior of the underlying micro-architecture. Our application runs on a real hardware board, which is equipped with an ARM926EJ-S processor core. Therefore, to compute WCETs of different tasks, we develop analysis methodologies by modeling the micro-architecture of ARM926EJ-S processor.

Besides, any robot controller task might receive an external interrupt, which will eventually delay the response time of the controller. The delay induced by an interrupt is not *solely* limited to the execution time of the respective interrupt service routine (ISR). This is due to the fact that the micro-architectural states (*e.g.* content of a cache) get modified after executing an ISR. Such micro-architectural state changes may introduce additional delay in task response time. As an example, if the interrupt service routine evicts some cache blocks used by a controller task, the response time of the controller task will be delayed due to additional *cache misses*. We develop novel analysis methodologies which account for such delay during WCRT computation, by considering the micro-architecture used in our processor board.

However, we face the most significant challenge due to the presence of a real-time operating system (RTOS) in our application scenario. The RTOS provides several system-level supports to the robot controller, such as scheduling multiple tasks, *kernel-mode* operation via system calls and interrupt handlers. Most of the existing works in WCET analysis assume the absence of an operating system [11]. Therefore, the computed WCET *completely bypasses* the timing effects created due to interactions with an RTOS. A different stream of works [7], [10] aim to compute the WCETs of RTOS-level routines (*e.g.* system calls, interrupt handlers) in isolation (*i.e.* without considering the timing effects of RTOS-level routines on the application).

If an application uses RTOS-level routines, such interaction with the kernel may significantly change the micro-architectural state, such as modifying the content of caches. If WCET analysis of an RTOS is performed in isolation (*e.g.* using techniques proposed in [7], [10]), the application-level WCET analysis will be unaware of the micro-architectural state when the kernel returns control to the user mode. For a *sound* WCET estimation, the application-level analysis has to consider all possible micro-architectural states after a kernel-mode operation finishes. This leads to a gross overestimation. In a similar fashion, while performing the WCET analysis of an RTOS-level routine in isolation (*e.g.* analyzing the WCET of a system call using [7]), we have to consider all possible micro-architectural states at the entry of the RTOS-level routine. Due to this two-fold overestimation in the underlying WCET analysis, we believe that an integrated WCET analysis of RTOS and application code is crucial. Such an integrated analysis should consider the application context while invoking a kernel-mode operation and it should also accurately model the effect of kernel-mode operations on application code. Providing an integrated WCET analysis framework which accounts the timing effects of both RTOS and application code is the key contribution of our work.

Accounting the timing interaction between an application and an RTOS is not straightforward. Any naive strategy, such as the enumeration of all possible timing interactions will quickly lead to an exponential number of micro-architectural states, making the entire analysis *infeasible* in practice. To maintain scalability of our analysis, we propose to use a *compositional* analysis framework. Such a compositional analysis computes a comprehensive summary for each RTOS routine called by the robot controller. The primary goal of such summary is to capture the *change* in micro-architectural states, that might happen due to the invocation of the respective RTOS routine. For a particular execution platform, the timing behavior summary of RTOS routines can be computed only once and the computed summaries can be reused for analyzing different applications. While analyzing our robot controller, the computed summaries are used at call sites of RTOS routines. We systematically combine the micro-architectural state before the call site of RTOS routine and the summary of the RTOS routine to obtain the micro-architectural state after the call site. Finally, we show that our compositional analysis framework is generic in nature. Specifically, our compositional analysis framework can be applied in the following three scenarios: *(i)* accounting for the effect of interrupts on system call execution; *(ii)* accounting for the effect of interrupts on application execution; and *(iii)* accounting for the combined effect of system calls and interrupts on application execution. As a result, we propose a generic, yet scalable analysis framework which comprehensively models the RTOS-level timing effects on application execution.

Contributions: In summary, we propose a generic WCRT analysis framework to analyze the timing behaviour of a real-life application scenario (an embedded robot controller) in the presence of a realistic execution platform that exhibits complex timing interactions involving an application, an RTOS and a real hardware. We note that conventionally application level WCET analysis methods have ignored the timing effects of the RTOS - these works estimate the maximum uninterrupted execution time of software. In this perspective, our work can be seen as a step towards making WCET analysis methods applicable to real-life situations since most modern embedded devices (including smart-phones) employ an operating system as supervisory software.

We have implemented our entire WCRT analysis framework using Chronos [12], an open source, freely available WCET analysis tool. Our robot controller uses μC/OS-II [2], a real-time operating system (RTOS), freely available for non-commercial usage. Our analysis models ARM926EJ-S processor architecture and we take measurements on real hardware board. We have performed an extensive set of experiments to share our experience in analyzing the time-critical components of a real-life robot controller. By considering the timing effects of the operating system on application in a compositional manner, we are able to obtain a safe and reasonable bound on the WCRT of our robot controller.

II. Execution platform

A. Target processor

In our analysis, we choose an APF28-Dev board which is designed by Armadeus systems. The board is equipped with a Freescale iMX286 processor. This processor has an ARM926EJ-S core running at 454MHz with 32KB data cache and 16KB instruction cache at level 1. Both instruction and data caches are 4-way set-associative, with a cache line size of eight words. The processor supports two replacement policies for caches which are random and *first-in-first-out* (FIFO). In our analysis, we configure the hardware to use FIFO replacement policy. The processor also has a memory management unit (MMU). The MMU has a normal 2-way set-associative TLB and a fully-associative lockdown TLB. In our analysis, we lock all pages used for our application into the lockdown TLB to ensure that no page fault will occur, as our analysis tool is not modelling TLB.

ARM926EJ-S processor has a five-stage pipeline with in-order issue, execution and completion. It supports speculative, non-cacheable instruction fetches to increase performance. However, in our experiments, speculative prefetch is disabled in order to make measurements more deterministic. The processor also implements static branch prediction which predicts all branch instruction as *not taken*. If the branch is taken, the penalty for the wrong prediction is 2 cycles. In our static analysis we bound memory latency to 70 cycles, as we observed a latency of between 60-70 cycles during read or write to on board physical memory.

The processor has an interrupt controller that can manage up to a total of 128 interrupt sources. All of these interrupt sources can be configured to work as normal or fast interrupt request. The interrupt controller supports nested interrupts and we can enable (disable) nested interrupts by clearing (setting) a single bit in interrupt control register. In our analysis and measurements, nested interrupts are disabled.

B. μC/OS-II kernel

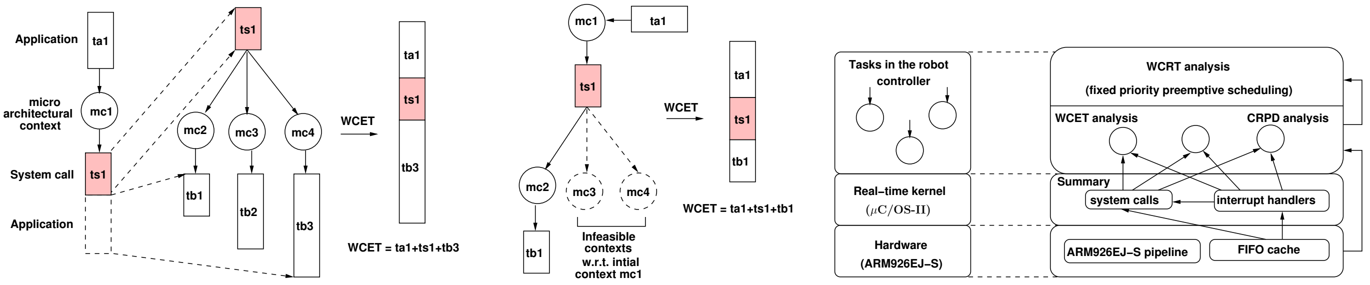
Our robot controller application runs on top of an open source operating system called μC/OS-II. μC/OS-II is a relatively small real-time kernel with 9,771 lines of C code and it supports 79 system calls [10]. The kernel implements fully preemptive scheduling policy based on fixed priority scheme. μC/OS-II supports a maximum of 250 application tasks and each task is assigned a unique priority. To support communication between tasks, the kernel implements semaphores, event flags, message mailboxes and message queues.

The kernel has been ported to different architectures such as 80x86, ARM, AVR. In our work, we port the kernel to the Freescale iMX286 processor in the APF28-Dev board. We use the official ARM port (version 2.86) provided by Micrium and we add codes for specific board support package (BSP) such as interrupt management, co-processor configuration and input/output (I/O) functions.

We choose an RTOS over a general purpose operating system for our analysis, as RTOS has several features that meet requirements of real-time applications. RTOS is designed to be consistent and deterministic regarding the scheduling of tasks. RTOS allows priority-based execution of tasks and it guarantees that a higher priority task will always be executed ahead of a lower priority task except for a few well defined scenarios (*e.g.* blocking due to shared resources). Moreover, RTOS typically has low latency for interrupt handling and task context switches. Therefore, an RTOS can respond to changes in its executing environment (interactions between tasks and handling external events) swiftly. As such, performing our analysis on RTOS like μC/OS-II allows us to bound the timing behaviour of real-time applications with less overestimation and also requires less complexity in our analysis methods.

III. Overview and general background

Why do we need an integrated analysis?: In this Section, we shall argue the potential of an integrated analysis framework. Let us consider the schematic shown in Figure 2(a). Figure 2(a) shows a simple application code fragment that invokes a system call. Specifically, the application executes for $ta1$ time units before invoking the system call. $mc1$ captures the micro-architectural state just before the system call was invoked. After the system call returns control to the user mode, the application finishes its execution without invoking any other kernel-mode operation. Let us first consider the scenario where WCET analysis of an RTOS was performed in isolation (*i.e.* similar to [7], [10]). If the analysis of RTOS was performed in isolation, we were unaware of the micro-architectural state before the invocation of a kernel-mode operation (*i.e.* $mc1$ in Figure 2(a)). As a result, the analysis of a kernel-mode operation (*e.g.* system calls) *has to conservatively assume all possible micro-architectural states* using which the same operation could be invoked. Due to this gross overestimation of possible micro-architectural states at the entry of a kernel-mode operation, the set of possible micro-architectural states at the exit of a kernel-mode operation is usually overestimated. Figure 2(a) shows one such example, where we get three possible micro-architectural states (*i.e.* $mc2$, $mc3$ and $mc4$). Note that the execution time highly depends on the micro-architectural context. Therefore, we assume that the application takes an additional time of $tb1$, $tb2$



(a) (b) (c)

Fig. 2: (a) An example shows the overestimation when the RTOS-level analysis is performed in isolation, (b) our integrated analysis framework eliminates such overestimation by performing a context-sensitive analysis, (c) the overall structure of our OS-aware timing analysis framework

or $tb3$ ($tb1 < tb2 < tb3$), if the kernel returns control with micro-architectural states $mc2$, $mc3$ or $mc4$; respectively. Since we are computing the WCET of the application, we can observe that $mc3$ leads to the worst-case finish time of the application. Therefore, using the analysis of RTOS in isolation, WCET of the application can be estimated as $ta1 + ts1 + tb3$, where $ts1$ captures the WCET of the system call.

It is worthwhile to note that in the preceding computation, we completely ignore the calling context of the system call at the application level (*i.e.* $mc1$ in Figure 2(a)). This leads us to overestimate the possible exit states of the system call (*i.e.* $mc2$, $mc3$ and $mc4$). Our integrated analysis framework takes into account this application context. Specifically, the set of possible micro-architectural states exiting a system call is computed via a micro-architectural summary of the system call and the application calling context (*i.e.* $mc1$ in Figure 2(a)). On one hand, our analysis methodology maintains the scalability of a compositional analysis by analyzing each kernel-mode operation separately and computing a micro-architectural summary for each kernel-mode operation. On the other hand, our analysis also takes into account the application context to accurately compute the effect of a kernel-mode operation on the application code. Figure 2(b) summarizes our analysis flow. It is possible that the micro-architectural state $mc1$ may only lead to the micro-architectural state $mc2$ after the system call. In such a case, the application will execute, in the worst case, only for $tb1$ time units after the system call. This leads to a more accurate WCET estimate using our framework (*i.e.* $ta1 + ts1 + tb1$).

Analysis components: In the following, we shall briefly outline our analysis framework. Figure 2(c) gives an overview of the entire methodologies used in this paper. Our robot controller contains a set of independent tasks. However, our proposed methodology does not pose restrictions on task dependencies and it can be extended by any WCRT analysis method that can handle task dependencies (*e.g.* in the form of an application task graph). The application interacts with the hardware via a real-time, multi-tasking kernel (as shown in Figure 2(c)). Our test platform is equipped with an ARM926EJ-S processor capable of supporting a full real-time operating system (RTOS). We use μC/OS-II [2] real-time kernel as the underlying RTOS. Our overall goal is to statically bound the response time of an application running on our test platform. As a result, our static analysis framework addresses the key challenges that appear due to the presence of complex timing interactions between an application, a real hardware and an RTOS.

Background: WCET analysis of each task is composed of three different phases: i) program flow analysis, ii) micro-architectural modeling and iii) WCET calculation. Program flow analysis derives useful information for WCET analysis, such as infeasible program paths and loop bounds. Micro-architectural modeling analyzes the timing behaviour of underlying hardware components (*e.g.* caches, pipeline). As an outcome of the micro-architectural modeling, we obtain the WCET of each basic block in the program control flow graph (CFG). Finally, a WCET calculation phase uses the outcome of micro-architectural modeling (*i.e.* basic block level WCET) and program flow analysis (*i.e.* infeasible paths, loop bounds) to compute the WCET of the overall program. We primarily use abstract interpretation (AI) for cache analysis and integer linear programming (ILP) for WCET calculation. Infeasible program paths and loop bounds are encoded as separate ILP constraints into our framework. The solution of the formulated ILP *soundly* over-approximates the WCET of each task.

WCET of a task captures an upper bound on the uninterrupted execution time. In the presence of multitasking, however, task interferences affect the overall timing of the application. Such task interferences could be generated due to the preemption of a low priority task (by a high priority task), servicing external interrupts and so on. Overall, there are two major sources that affect the timing of individual tasks in a multitasking environment. First, if a task T is preempted by a high priority task or an interrupt service routine (ISR), the timing of the high priority task or the ISR will directly delay the finishing time of task T. Secondly, the interruption (either by preemption or external interrupts) changes micro-architectural contexts, such as modification of cache contents, flushing pipeline and so on. Such micro-architectural changes may lead to additional delay (*e.g.* due to additional cache misses). Over the last few decades, WCET research community have investigated the problem of bounding the number of additional cache misses due to preemptions. Such additional cache misses are widely known in literature as cache related preemption delay (CRPD) [8]. Similarly, once a task resumes after an interruption, additional delay need to be considered during the analysis for flushing the pipeline. The schedulability analysis uses results from WCET analysis, CRPD analysis and any other micro-architectural delay due to an interruption (*e.g.* flushing the pipeline) to derive an upper bound on the response time of the overall application. Such an upper bound is known as *worst case response time* (WCRT). We use a fixed-priority preemptive scheduling policy to derive the overall WCRT of an application.

IV. Analysis methodologies

In this section, we shall describe our analysis methodologies in detail. The critical part of our analysis method is a compositional analysis framework as shown in Figure 3. Each task, interrupt service routine (ISR) and system call is a *component* for which we separately compute its WCET and the *summary* of its changes to the underlying micro-architectural states. The computed WCET and *summary* may in turn be used in the analysis of other *components*. The benefit of such a compositional analysis framework is two-fold. First, the compositional analysis framework accounts for the timing interaction of an application with RTOS in a generic fashion, such as accounting the timing interactions with system calls, interrupt handlers and the combined effect of system calls and interrupt handlers. Secondly, due to the compositional nature of our analysis framework, we can systematically control the number of micro-architectural states and thus maintain the scalability of the overall analysis. In the following, we shall first introduce the WCRT calculation and subsequently, we shall describe our compositional analysis framework as well as its interactions with WCRT computation.

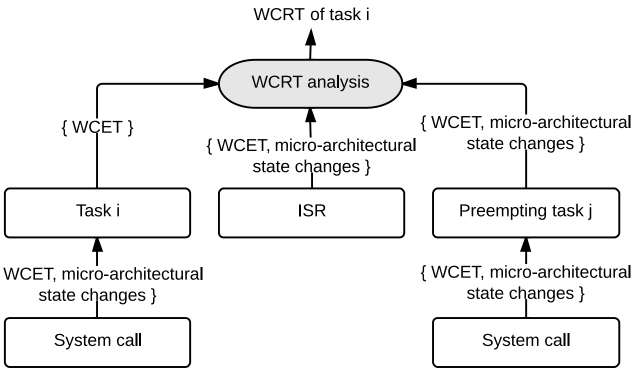

WCRT of task i
WCRT analysis
{ WCET }
{ WCET, micro-architectural state changes }
{ WCET, micro-architectural state changes }
Task i
ISR
Preempting task j
WCET, micro-architectural state changes }
{ WCET, micro-architectural state changes }
System call
System call

Fig. 3: Compositional WCRT analysis framework. Each rectangular box represents a *component* that is analyzed separately

A. WCRT computation

Our primary goal is to compute the *worst case response time* (WCRT) of individual tasks in an application, and specifically in our robot controller. WCRT of a task is defined as the worst case bound on the time between the release and completion of a task. This time bound is broadly composed of two factors: *(i)* the *worst case execution time* (WCET) of the task, and *(ii)* the cost due to interference. The interference of a task can be caused either by higher priority tasks or by interrupts. Our robot controller contains a set of *periodic* and *sporadic* tasks. In our system, all interrupt arrivals are also *periodic* or *sporadic* in nature. For a *sporadic* task or interrupt, we consider that it always arrives at its *minimum inter-arrival time* since we are only interested in the worst case scenario. An interrupt will be serviced by its assigned *interrupt service routine* (ISR) on arrival. For our computation, we treat all ISRs as *components* with the highest possible priority (as an ISR cannot be preempted by a task). For the rest of this section, we shall use the term *component* to refer to either task or ISR when they can be used interchangeably.

For a specific task i, its worst case response time $WCRT_i$ is computed as shown in Equation 1 below (the equation is derived from [15]). Note that since the computation of $WCRT_i$ requires the value of $WCRT_k$ and component k has higher priority than task i, WCRT computation has to be performed from task with the highest priority to the lowest.

$$
\begin{aligned}
WCRT_i = {} & WCET_i \ + B_i \ + \\
& \sum_{j \in hp(i)} \left(\left\lceil \frac{WCRT_i}{P_j} \right\rceil (WCET_j + MOD_{j,i} + CTX_j) \ + \right. \\
& \qquad \left. \sum_{k \in hp(i) \wedge lp(j)} \left\lceil \frac{WCRT_i}{P_k} \right\rceil * \left\lceil \frac{WCRT_k}{P_j} \right\rceil MOD_{j,k} \right)
\end{aligned} \tag{1}
$$

In Equation 1, $WCET_i$ captures the worst case execution time of task i without interference. B_i is the maximum blocking time of task i, or the maximum time a lower priority task can block the execution of one invocation of task i. We assign to B_i the maximum WCET value of all *critical sections* (code sections in which preemptions are disabled) in tasks with lower priority than task i. $hp(i)$ contains the set of all higher priority tasks and ISRs that may delay the execution time of task i. On the other hand, $lp(i)$ contains the set of all tasks with lower priority than task i. $\left\lceil \frac{WCRT_i}{P_j} \right\rceil$ bounds the number of possible preemptions by component j on task i. P_j is the period (or minimum inter-arrival time) of component j. The computation of $MOD_{j,i}$ is crucial. $MOD_{j,i}$ accounts for the additional delay inflicted by component j on task i due to the modification of micro-architectural states (*e.g.* states of caches, pipelines) after interference. CTX_j refers specifically to task context switch cost if component j is a task. If component j is an ISR, it refers to the cost of accessing the kernel's ISR entry and exit function, as well as the delay in saving or restoring CPU's context before jumping to or from some ISR code. The term $(WCET_j + MOD_{j,i} + CTX_j)$ bounds the interference cost by component j on task i for one preemption. We multiply this term by the maximum possible number of preemptions for all preempting components.

Apart from considering the $MOD_{j,i}$ cost inflicted by all components that preempt task i, we also need to take into account the modification of micro-architectural states inflicted by component j to all tasks nested between component j and task i when there are nested preemptions, as this may indirectly add additional delay to the execution time of task i. As shown in the second half of Equation 1, we sum together the total $MOD_{j,k}$ costs inflicted by component j on all nested tasks except for task i. $\left\lceil \frac{WCRT_i}{P_k} \right\rceil * \left\lceil \frac{WCRT_k}{P_j} \right\rceil$ bounds the number of possible preemptions by component j on task k during the execution of task i.

Equation 1 is essentially a fixed-point computation of $WCRT_i$, as the term $WCRT_i$ appears on both sides of the equation. The computation in Equation 1 will be repeated until the value of $WCRT_i$ reaches a fixed-point, or when it exceeds the *deadline* for task i, in which case there is no point to continue on as the task is not schedulable anymore.

B. Compositional analysis of application and RTOS code

The WCRT computation requires the WCET of individual tasks. Traditional WCET analyses ignore timing effects related to an RTOS. In the presence of an RTOS, different system-level routines (*e.g.* system calls) affect the execution time of an application. It is possible to perform WCET analysis of an RTOS as a standalone application and obtain the WCET of each system call [7], [10]. However, it is not sufficient to compute the WCET of each system call in isolation and integrate the computed WCETs into the WCET computation of a task. This is due to the fact that the execution of system calls may heavily modify the micro-architectural state and such micro-architectural state changes must be taken into account at each task for computing a *safe* WCET estimate.

In the presence of an RTOS, a straightforward approach to compute the WCET of a task is to link the application and RTOS code in a single binary. Traditional WCET analysis can be carried out on this standalone binary. However, such an approach has two primary disadvantages: *(i)* potentially huge size of the executable and *(ii)* unavailability of RTOS source code. Note that some analyses, such as computing *loop bounds* and *infeasible paths* are difficult to perform without source-level information, especially for complex system codes (*e.g.* an RTOS). Therefore, we propose to analyze RTOS system calls separately and compute a summary of each system call. Such an approach enables partial WCET result reuse, improves the scalability of overall WCET analysis and it can also leverage the source-level infeasible path and loop bound information of RTOS code. Our computed summaries for system calls are sufficient for computing a safe WCET of each task.

We compute two results for each separately-analyzed system call: (1) the WCET of the system call, and (2) a summary of the effect of the system call execution on the micro-architectural analysis. In our processor board, timing effects at micro-architectural level arise from an in-order pipeline, a static branch predictor and caches with FIFO replacement policy. Since a static branch predictor is used, we add a fixed penalty (2 cycles) for each taken branch in the WCET analysis. As a result, we do not need to compute a summary of an RTOS routine explicitly for branch predictors. Moreover, in the analysis, we assumed an empty pipeline state at system call and interrupt handler boundaries. In particular, we assume that the pipeline is empty at *(i)* the beginning of each system call and interrupt handler; and *(ii)* after the return of each system call and interrupt handler. Besides, there might be additional delay due to the dependency between application code and RTOS routines. Such dependencies include situations where the application passes data to RTOS routines and vice versa. To take into account this additional delay into our analysis, we add fixed delay for each invoked RTOS-level routine. The fixed delay ensures that all the inputs to an RTOS routine are available when it begins execution and all the outputs from an RTOS routine are available when it finishes execution. It is worthwhile to note that this delay is *bounded* by the memory latency (*i.e.* cache miss latency).

However, we cannot consider the effect of caches similar to pipeline and branch predictors. Invocation of an RTOS routine may significantly affect the cache content of the application. Due to the inherent performance gap between processor and main memory, several cache misses in the application code may significantly downgrade its performance. Therefore, static analysis of caches is needed to classify a memory access as a cache hit or a cache miss. In our experiments, we observed that without employing any cache analysis, *none of our robot controller tasks* could be guaranteed to meet their respective deadlines. Therefore, we developed a compositional analysis of instruction and data caches for deriving safe and tight bounds on the application WCRT, while taking into account system calls / interrupts.

In our summary computation, we primarily consider caches and unless otherwise stated, a summary in the following discussion will capture the cache summary of an RTOS routine. In particular, our compositional cache analysis is used to compute cache summaries for system calls as well as interrupt handlers. Such a compositional analysis also takes into account the effect of interrupts happening inside system calls. Note that, even if the time spent in interrupts is already accounted for in the WCRT computation of the task (*i.e.* using Equation 1), we need to take into account the delay caused by cache misses in a system call due to interrupts.

To do this, we need (1) to bound the number of interrupts occurring in a system call, and (2) to bound the number of additional misses during the execution of the system call. To bound the number of interrupts, we perform a fixed-point computation in a similar fashion to Equation 1 and obtain W_i, which is the execution time of the system call considering the effect of interrupts. Assume that the period of an interrupt is P_i, then the number of interrupts occurring during the execution of the system call can be bounded by $\left\lceil \frac{W_i}{P_i} \right\rceil$. Once we know the number of interrupts, additional misses occurring in the system call (due to interrupts) can be computed exactly in the same fashion as in a task (*cf.* section IV-C). Finally, the total cache delay induced by interrupts on a system call can be added to the system call WCET.

C. Cache related preemption delay

In the presence of caches, WCRT computation must take into account the additional cache misses due to preemptions. Such additional cache misses are caused by a preempting task when it evicts cache blocks used by the preempted task. The delay caused by these additional cache misses is known as *cache related preemption delay* (CRPD). CRPD is traditionally computed using a separate analysis and the outcome of CRPD analysis is integrated into the WCRT computation.

In the following, we shall primarily outline our CRPD analysis methodology for data caches. For instruction caches, we use the CRPD analysis proposed in [5] in an exactly same fashion. For a detailed description of CRPD analysis for instruction caches, we request readers to refer to [5].

CRPD analysis for data caches: Recall that our underlying processor (*i.e.* ARM926EJ-S) uses caches with FIFO replacement policy. In the presence of FIFO replacement policy, CRPD analysis for data caches poses a challenge. As shown in [5], CRPD in the presence of FIFO replacement policy cannot be computed using a similar fixed-point computation as in LRU replacement policy.

We have proposed a novel approach for analyzing CRPD. Our approach leverages the similarity between the analysis of system calls and preemptions. In both cases, we need to consider the set of possible evictions of a memory block that were not accounted during the analysis of the task in isolation. However, unlike the analysis of a system call, we do not know the exact program location where preemption will take place (*i.e.* the arrival point of an interrupt).

Our CRPD analysis revolves around the persistence analysis for data caches. We use our prior work on data cache persistence analysis [13] for this purpose. Our CRPD analysis takes two inputs as follows.

- A set of preempting tasks $\{T_1, \ldots, T_n\}$.
- Maximum number of preemptions incurred by the preempted task due to each preempting task. Let us assume PC_i captures the maximum number of preemptions due to the preempting task T_i.

Our goal is to compute the additional cache miss penalty incurred by the preempted task for all preemptions. In the following, we shall describe the approach for a single cache set. For set-associative caches, the following approach is simply repeated for each cache set. We only describe the general idea in the following. Details of the algorithm is provided in Appendix.

The basic idea behind our CRPD analysis is as follows. Our analysis first computes an upper bound on the number of additional cache misses for each loop context. A loop context is categorized by a sequence of loop iteration numbers. Formally, a loop context $\mathcal{C}$ can be captured by a k-tuple $\langle I_1, I_2, \ldots, I_k \rangle$, where I_1 represents the outermost loop iteration number and I_k represents the innermost loop iteration number. Our CRPD computation is primarily based on the following insight. *For each loop context $\mathcal{C}$, an additional cache miss for a data block should be taken into account only if the data block might be accessed in loop context $\mathcal{C}$ and the data block access is persistent in the absence of preemption.* Note that we do not need to consider the *non-persistent* data references for CRPD analysis. This is because *non-persistence* already captures the *worst case*. Therefore, for each loop context $\mathcal{C}$, the number of additional cache misses caused by preemptions cannot exceed the number of persistent data references in $\mathcal{C}$, in the absence of preemptions. For loop context $\mathcal{C}$, let us assume that this upper bound is captured by $mmc_{\mathcal{C}}$. We also observe that the number of inter-task cache conflicts by the set of preempting tasks $\{T_1, \ldots, T_n\}$ is bounded by the number $\sum_{i=0}^{n} PC_i \cdot DMG_i^{data}$, where DMG_i^{data} captures the number of unique data blocks accessed inside the preempting task T_i.

To compute the total number of additional cache misses due to the set preempting tasks $\{T_1, \ldots, T_n\}$, our analysis follows a greedy approach. We choose a loop context $\mathcal{C}$ that may lead to the *maximum* number of additional cache misses (*i.e.* $mmc_{\mathcal{C}}$), but requires the *minimum* number of inter-task cache conflicts to evict out the set of persistent blocks in $\mathcal{C}$. We continue choosing loop contexts in such a fashion until we reach the upper bound on the number of inter-task cache conflicts (*i.e.* $\sum_{i=0}^{n} PC_i \cdot DMG_i^{data}$). The resulting number of cache misses (*i.e.* the accumulation of $mmc_{\mathcal{C}}$ values) is predicted as the CRPD value. It is worthwhile to mention that the greedy heuristic is *conservative* and the precision of the computed CRPD can be improved using a more accurate technique (such as integer linear programming) and by compromising the analysis time. However, as our evaluation shows, we can still maintain a reasonable overestimation ratio with this conservative CRPD analysis.

D. Cache damage implementation

The effect of a system call, task preemption or ISR (referred to as a component C) is summarized by a *cache damage* function, represented by DMG_C. For each cache set s, $DMG_C(s)$ captures the number of unique blocks inside C mapped to s. This definition is used for both instruction and data cache persistence. The type of the damage for data (instruction) cache will be represented by $DMG_C^{data}(s)$ $(DMG_C^{inst}(s))$.

The data cache damage is used to handle preemptions and system calls, but the instruction cache damage is used only for system calls. This is because cache damage is used to handle data CRPD, however for instruction cache we use a traditional CRPD analysis [5]. We use *persistence* analysis for both instruction and data caches. In the following, we shall describe our instruction and data cache persistence analyses using *cache damage*.

To describe the instruction and data cache analyses with cache damage, we shall first introduce the concepts of *temporal scope* and *younger set* from our previous work.

Definition 4.1: (Younger set) Let B be a memory block mapped to cache set s. The younger set *of B, denoted as YS_B, is the set of memory blocks that may have smaller relative ages than B in cache set s, before the access to B.*

Definition 4.2: (Temporal scope) The temporal scope TS_B of a memory block B indicates in which loop contexts B is

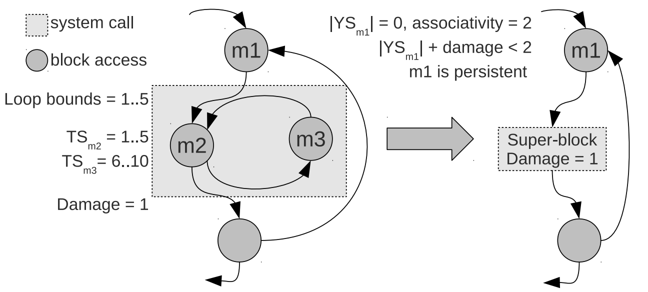


Fig. 4: Data cache damage example

accessed. It associates an integer interval $[l, u]$ to each loop L containing B. This integer interval captures the lower and upper bounds on the iterations of L during which the accesses to B can take place.

Cache damage for instruction persistence analysis: The instruction cache persistence analysis determines the persistence of each instruction block B by computing its respective *younger set* (YS_B). Let A be cache associativity. Block B is persistent if $|YS_B| < A$. To compute the damage for the instruction persistence analysis, we must compute for each set s, the maximum size of the *younger set* for any block. With FIFO replacement policy, the damage of a component can be computed simply by counting the number of blocks in the component mapped to s. The damage of a component C for the set s is represented as $DMG_C^{inst}(s)$. The cache damage will need to be taken into account when performing the persistence analysis of the main task. For each block B, if any path from B to B goes through the component C, then the block B is considered persistent if and only if $DMG_C^{inst}(s_B) + |YS_B| < A$ (where s_B represents the cache set to which B is mapped).

Cache damage for data persistence analysis: For data cache damage, we need to count the number of unique memory block accesses inside component C. Since component C may be analyzed in a context-sensitive way, loop bounds inside C may depend on the calling context of C (*e.g.* arguments) Therefore, depending on the calling context, some references to memory blocks inside C could never be accessed. The loop bounds inside C can be used together with the *temporal scopes* of blocks, to determine if a specific block can be accessed during a specific calling context. When performing the persistence analysis of the main task, damage is accounted similarly to the instruction cache as follows: for each block B, if any path from B to B goes through C, block B is considered persistent if and only if $DMG_C^{data}(s_B) + |YS_B| < A$ (where A is cache associativity).

An example: In the example shown in Figure 4, blocks *m1*, *m2*, and *m3* are mapped to the same cache set in a 2-way associative cache. In the considered calling context, the loop iterates at most 5 times. Only block *m2* is accessed, because the *temporal scope* of *m3* indicates that *m3* is accessed only during iterations 6 to 10. Therefore, the damage is 1.

When analyzing the main task, we try to determine the persistence status of *m1*. The *younger set* is $|YS_{m1}| = 0$, and since the path from *m1* to another access of *m1* passes through C, we need to check the condition $|YS_{m1}| + damage < 2$. The condition is true, therefore *m1* is persistent. Let us now assume a different calling context, causing the loop to iterate 10 times. Then in the component both *m2* and *m3* will be accessed, the damage will be 2, and *m1* will not be persistent.

V. Evaluation

In this section, we give a structural overview of our robot control application along with experimental results obtained from our evaluation. Based on the result, we proceed to verify if our application meet its real-time constraints.

A. Robot controller overview

As mentioned in introduction, we adapted some open source application codes written for EyeBot controller. Specifically, the source code of Bally2 [1], an experimental real life self balancing robot, is adapted as runnable tasks on μC/OS-II. To realize our application scenario, we augmented Bally2 with an obstacle detection program written for Eyebot. We also wrote some additional codes, such as low level drivers, to port the programs to our particular hardware board.

Our robot controller consists of 3 main tasks and 4 ISRs. The function of each task or ISR is briefly described in Table II. The size and code complexity of each task is given in Table III. Both `balance` and `navigation` are periodic tasks running at 50Hz and 20Hz respectively. `remote` task is sporadic and blocked on a semaphore released by `infrared_isr`.

`tick_isr` is an ISR servicing *periodic interrupt* due to OS tick. `gyro_isr`, `inclino_isr` and `infrared_isr` are ISRs servicing interrupts generated from gyroscope, inclinometer and infrared sensor respectively. These are *sporadic interrupts* generated when there are new sensor readings. As we are only interested in finding the WCRT of tasks, we assume a worst case scenario where sporadic interrupts always arrive at its *minimum interarrival time*. In the case of our application, `gyro_isr`, `inclino_isr` and `infrared_isr` all arrive at a rate of 1Khz.

Task/ISR	Priority	Description
`balance`	4	Calculation to keep balance using input from gyroscope and inclinometer.
`navigation`	6	Auto navigate to destination while avoiding obstacles.
`remote`	5	Receive remote command via infrared.
`tick_isr`	-	Periodic OS tick.
`gyro_isr`	-	Process interrupt from gyroscope.
`inclino_isr`	-	Process interrupt from inclinometer.
`infrared_isr`	-	Process interrupt from infrared sensor.

TABLE II: Tasks and ISRs in our system. Lower priority value means higher priority. (μC/OS-II reserves 4 lowest priorities)

Task	Number of instructions	Number of basic blocks	Number of loops	Number of system calls
`balance`	6914	1403	6	3
`navigation`	3899	496	11	7
`remote`	239	43	2	3

TABLE III: Code complexity of robot controller tasks

B. Issues and assumptions

One of the implementation issues for our robot controller application is that *µC/OS-II* does not have out of the box support for strictly periodic tasks. For a task to always run at some interval, *µC/OS-II* only provides OSTimeDly system call which delays execution of a task by a specified number of ticks. The system call does not take into account the execution time of the task and may result in a task running at irregular periods. Instead of delaying a task by its period, we have to delay it by the difference between its period and the execution time of the current invocation of the task.

Currently, our analysis tool does not handle task with dynamic priority. Thus, we do not allow *µC/OS-II* to change task priority in runtime and we ensure that there is no scenario in which *priority inversion* may happen.

C. WCET analysis result

We perform measurement on the execution time of each task and ISR running on our APF28-Dev board. To measure the execution time of a portion of code, the time before the execution of the code portion is written to an unused memory address on the board's RAM. Likewise, we also store the time at the end of the code portion. The stored values are read through the board's JTAG interface to minimize the effect of measurement on actual execution. To ensure that we measure the uninterrupted execution time of a task or ISR, we disable all interrupts and prevent other tasks from running.

For each task and ISR, we compare the observed WCET from our board with the estimated WCET from Chronos. The result is presented in Table IV. We also compare the WCET overestimation, which is defined as $\frac{Estimated\ WCET}{Observed\ WCET}$, between tasks and ISRs in Figure 5.

Task	Observed WCET (cycle)	Estimated WCET (cycle)	Over-estimation
balance	105120	380179	3.62
navigation	3871655	10803600	2.79
remote	17422	61124	3.51
tick_isr	3159	8056	2.55
gyro_isr	1438	3087	2.15
inclino_isr	1324	3291	2.49
infrared_isr	4256	20330	4.78

TABLE IV: WCET (in CPU cycles) of all tasks and ISRs

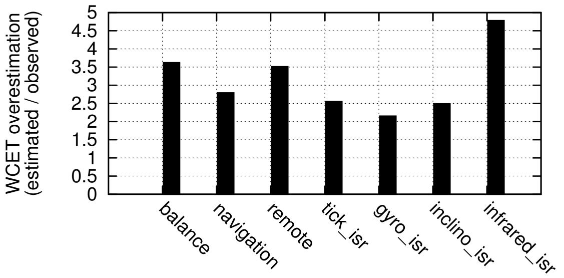


Fig. 5: WCET overestimation of all tasks and ISRs

We observe overestimation values ranging from 2.15 to 4.78 for all tasks and ISRs. The average overestimation ratio is 3.12. navigation task detects collision by processing captured image to detect the edges of a colliding object and attempts to navigate away from it. Thus, navigation task has many loops referencing data exhibiting *spatial locality* pattern. The data persistence analysis is effective in bounding the execution time of loops exhibiting such pattern and this resulted in navigation task having a relatively lower overestimation than the other tasks despite being more complex.

balance task contains many floating point operations although our ARM926EJ-S processor does not have a built-in hardware floating point unit. Thus, we have to compile our application with a software floating point library. Some of the library functions being used contain loops with loop bounds that are hard to derive. We have to put conservatively high loop bounds to some of these loops, which may have contributed to the high overestimation ratio of 3.62 for balance task.

infrared_isr has a high WCET overestimation despite being a small interrupt handler code. infrared_isr decodes infrared pulses that come in bursts, and sends signal to the robot controller once it receives sufficient pulses to decode the information. However, our WCET analysis always assumes the worst case timing scenario (*i.e.* sending signal to the robot controller) whenever infrared_isr is executed.

D. WCRT analysis result and deadline verification

In Section III, we argue that using an integrated timing analysis approach can more accurately estimate the timing bound of a real-time application running on top of an RTOS. We aim to compare the result between *(i)* analyzing the RTOS in isolation and *(ii)* using our integrated analysis. We compute the WCRT of each task (with the fixed-point computation formula in Equation 1) through both approaches and compare them with the WCRT that we observe on the APF28-Dev board. For approach *(i)*, we flush both the instruction and data caches after each context switch from OS to application. Note that flushing the caches may not necessarily lead to worst-case micro-architectural state changes, due to the possible presence of timing anomaly. However, we will at least obtain an *optimistic* WCRT overestimation using approach *(i)*. The experimental result is presented in Table V. WCRT overestimation of both approaches is compared in Figure 6.

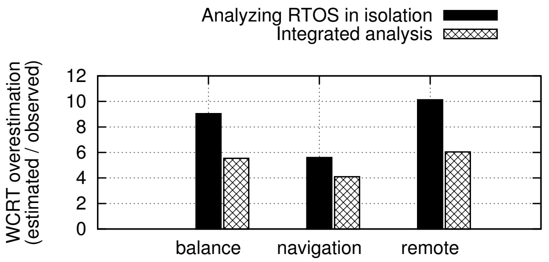


Fig. 6: WCRT overestimation of all tasks

1) Comparison of both approaches: From Table V, we observe that for all the robotic application tasks, our integrated analysis (*i.e.* approach *(ii)*) has between 1.50 to 4.09 less overestimation compared to approach *(i)*. The higher overestimation of approach *(i)* is mainly due to the overestimation

Task	Deadline (ms)	Observed WCRT (ms)	Estimated WCRT of isolated analysis (ms)	Over-estimation	Estimated WCRT of integrated analysis (ms)	Over-estimation
balance	20	0.240	2.172	9.04	1.331	5.54
navigation	50	9.013	50.442	5.60	36.936	4.10
remote	100	0.245	2.480	10.13	1.478	6.04

TABLE V: WCRT (in milliseconds) of all tasks

of cache misses from context switching between OS and application code. It is more apparent in smaller tasks (*e.g.* balance, remote) due to a higher OS overhead ratio compared to actual application execution time.

2) Integrated analysis result: For our integrated analysis, the average WCRT overestimation ratio is 5.23. We attribute the high overestimation ratio (compared to WCET analysis) to the difficulty in observing the WCRT of tasks on the hardware. As we cannot control the exact program points in which task preemptions or interrupts occur, it is hard to guide a task towards its worst case execution scenario while taking measurement for observed WCRT. We rely on repeated measurements and choose the maximum response time over a large set of response time values. It is possible that the *actual WCRT* is much higher than our observed WCRT.

3) Deadline verification: With our result, we can verify whether our system meet the real-time constraints presented in Table I. For balance and navigation tasks, since they are required to always run at a specific frequency, each task invocation is required to finish execution within its period. In this case, its deadline is equivalent to its period. balance task has a period of 20ms since the task is required to run at 50Hz. The computed WCRT for the task (refer to Table V) is 1.331ms, which is less than the task's period. Thus balance task meets its real-time constraint. Likewise, navigation and remote tasks also meet their respective deadlines.

Note that all tasks in our application meet their deadlines under any environmental conditions. Thus, we can conclude that our robot controller *can never fail* to balance itself while navigating rough terrain, and still be able to receive and process remote commands without missing any signal.

E. Discussion

1) Effect of application on WCET of system calls: We observed that WCET of many system calls are significantly affected by the application due to these factors:

- **Calling parameters** Parameters passed to system call from application may affect the control flow within the system call. For example, OSTaskSuspend takes a priority value as input argument, and suspends the task with the associated priority value. If the task calling OSTaskSuspend is the suspended task itself, then a context switch will happen to switch execution to a pending task. Otherwise if the calling task is not the suspended task, no context switch happens and the computed WCET for OSTaskSuspend will be much smaller.
- **Application dependent configuration** Static system configuration parameters which depends on the currently running application (*e.g.* task stack size) may affect the execution time of some system calls. For example, OSTimeTick has a loop bounded by the number of tasks in the application. Thus, WCET of OSTimeTick can be refined if we have information about the number of tasks.

We have listed some system calls in *µC/OS-II* which have application dependent WCET in Table VI. By considering the user application when performing timing analysis on system calls, we reduce the pessimism in the estimated WCET of these system calls, which in turn reduce the overestimation for WCET of each task. For example, when analyzing OSTimeTick, given that there are 3 tasks in our robot controller, we estimate the WCET to be 8027. If we ignore information from application level and just assume the maximum possible number of tasks, the estimated WCET would instead be 78056 (refer to Table VI), which is a gross overestimation.

System call	Range of estimated WCET (cycle)
OSTaskSuspend	2660 – 5210
OSTaskResume	4189 – 4942
OSTimeTick	3946 – 78056

TABLE VI: System calls with WCET which can be refined to a lower value with information on the application

2) Impact of pipeline flushes compared to cache evictions: A real-time application running on top of an RTOS introduces overhead due to supervisory code from the OS. This overhead includes both the execution time of OS code and the impact on hardware due to context switching between OS code and application code. In this work, we mainly focus on the effect on caches. This is because of the significant overhead due to cache misses. In Table VII, we show the impact on hardware pipeline and caches due to interruptions from system calls and ISRs, as a fraction of the estimated WCRT of each application task. The impact on hardware due to OS overhead is around 2–6% of the total estimated WCRT in our robotic application.

Task	Estimated WCRT (cycle)	Pipeline cost (cycle)	Cache eviction cost (cycle)
balance	604329	4200	25900
navigation	16769164	65660	410760
remote	671051	4480	33040

TABLE VII: Impact on pipeline and caches (in CPU cycles) due to OS overhead

3) Applicability for other applications: Our analysis tool, Chronos, takes in the following inputs to perform analysis:

- Binary of the embedded application
- Processor configuration (e.g. cache size and associativity)
- List of tasks and interrupts with their respective periods and priorities

Given these inputs, our analysis techniques can be applied to other embedded applications running on ARM9 family processors. Chronos can also be extended to support other similar architectures.

VI. Related work

Research on worst case execution time (WCET) analysis of embedded software has been started two decades ago. A comprehensive survey describing different techniques and tools for WCET analysis has appeared in [11]. Existing WCET analysis techniques assume a direct interaction between an application and the underlying hardware. However, most real-life embedded software are developed in the presence of a supervisory software (*e.g.* an RTOS). Our work analyzes the WCET of an embedded software in the presence of an RTOS, which in turn interacts with the underlying hardware. Therefore, our work extends *state-of-the-art* WCET analysis via analyzing a realistic execution platform that consists of an RTOS as well as a real hardware.

Several research activities [7], [10] have leveraged the progress in WCET analysis to analyze the WCET of an RTOS. A more comprehensive survey of such RTOS analysis techniques can be found in [9]. However, works in [7], [10] analyze RTOS as a standalone application, meaning that timing interactions between an application and the RTOS are not taken into account while computing the WCET. Existing work [16] has discussed the potential *unsound* WCET computation of an application without considering RTOS effect. However, works in [7], [10] do not consider the micro-architectural state changes in an application due to kernel mode operations. Therefore, such works cannot be directly used to compute a *sound* WCET of an application in the presence of an RTOS. In contrast to the approaches proposed in [7], [10], we have devised novel compositional methodologies that systematically combine timing interactions between an application and an RTOS to obtain a *sound* WCET of the application. Therefore, our proposed framework has established a direction where the technical problems discussed in [16] can be solved for a real-life application on a realistic execution platform.

Research on compositional cache analysis has been performed, among others, in [14], [6], [3]. However, such compositional cache analyses have only targeted the reuse of timing summary for *commercial off-the-shelf components* (COTS), such as library codes. On the contrary, we propose a generic compositional framework for cache analysis, where the compositional strategy can be applied to handle any system-level timing effect. Such system level timing effects can be arbitrarily complex, such as individual timing effects due to system calls and external interrupts, as well as their complex combinations. Moreover, our compositional analysis framework has been designed, implemented and evaluated on a real hardware running an RTOS.

In summary, our work takes a first step forward to bridge the gap between WCET research activities at application level and RTOS level. To accomplish our goal, we leverage the concept of compositional analysis and we have built a generic timing analysis framework in the presence of supervisory software.

VII. Conclusion

We have proposed a general solution for analyzing real-time, embedded application in the presence of a supervisory software (*e.g.* operating system). Our proposed analysis method models the micro-architecture of a real-life processor. Such an analysis methodology enables us to verify that individual tasks constituting the robot controller finish within the required deadline and such a verified controller is thus perfectly safe to use on the respective execution platform.

Apart from the technical novelties in our analysis - we propose a generic compositional WCRT analysis framework and we consider the effect of RTOS routines on the underlying application via a reusable summary of micro-architectural state changes, our work also involved very substantial system building effort. These include porting an RTOS to run on ARM926EJ-S architecture, construction of a real-time robot controller application from open source code and writing of low level drivers for our APF28-Dev board to support the robotic controller. To the best of our knowledge, ours is the first work in WCET analysis that considers an application in the presence of an operating system. We also plan to make our *entire robot control infrastructure* (including our ARM port of *μC/OS-II* and the robot controller application) available to the research community to spur more research which is based on real-life implementations of real-time embedded systems.

Acknowledgement

This work was partially supported by A*STAR Public Sector Funding Project Number 1121202007 - "Scalable Timing Analysis Methods for Embedded Software".

References

[1] Ballybot balancing robots. http://robotics.ee.uwa.edu.au/eyebot/doc/robots/ballybot.html.

[2] μc/os-II real-time kernel. http://micrium.com/rtos/ucosii/overview/.

[3] C. Ballabriga, H. Cassé, and P. Sainrat. An improved approach for set-associative instruction cache partial analysis. In *SAC*, 2008.

[4] Thomas Braunl. Eyebot: a family of autonomous mobile robots. In *Neural Information Processing, 1999. Proceedings. ICONIP'99. 6th International Conference on*, volume 2, 1999.

[5] C. Burguière, J. Reineke, and S. Altmeyer. Cache-related preemption delay computation for set-associative caches - pitfalls and solutions. In *WCET*, 2009.

[6] A. Rakib et al. Component-wise instruction-cache behavior prediction. In *ATVA*. 2004.

[7] B. Blackham et al. Timing analysis of a protected operating system kernel. In *RTSS*, 2011.

[8] C.G. Lee et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6), 1998.

[9] M. Lv et al. A survey of WCET analysis of real-time operating systems. In *ICESS*, 2009.

[10] M. Lv et al. WCET analysis of the μc/os-II real-time kernel. In *CSE (2)*, 2009.

[11] R. Wilhelm et al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[12] X. Li et al. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 2007. http://www.comp.nus.edu.sg/~rpembed/chronos.

[13] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *RTAS*, 2011.

[14] K. Patil, K. Seth, and F. Mueller. Compositional static instruction cache simulation. In *ACM SIGPLAN Notices*, volume 39, 2004.

[15] Jörn Schneider. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. In *RTSS*, 2000.

[16] Jörn Schneider. Why you cant analyze RTOSs without considering applications and vice versa. In *WCET*, 2002.

Appendix

A. CRPD Analysis for data caches

Algorithm 1 CRPD analysis for data caches with FIFO replacement policy

```
1: **Input:**
2: $T$: the preempted task
3: $\{T_1, \ldots, T_n\}$: Set of preempting tasks
4: Say $PC_i$ captures the number of preemptions by task $T_i$
5: Say $DMG_i^{data}$ is the ageing due to task $T_i$
6: $\mathcal{L}$: The set of innermost loops in $T$
7: $ctx(L)$: the list of contexts for loop $L$
8: $set$: the currently analyzed cache set
9: **Output**:
10: $crpd(set)$: the CRPD of task $T$ for $set$
11:
12: /*MDDP = Maximum Damage Due to Preemption */
13: $MDDP \leftarrow \sum_{i=1}^{n} PC_i \cdot DMG_i^{data}(set)$
14: /*Compute possible misses for each loop context */
15: let $\lambda$ be an empty list of integer pairs
16: $ctxlist \leftarrow \{(c, L) \mid L \in \mathcal{L} \wedge c \in ctx(L)\}$
17: **for** $(c, L) \in ctxlist$ **do**
18:     let $\mathcal{B}$ be the set of memory references within $L$ or any
19:     loop enclosing $L$
20:     $mmc \leftarrow |\{b \mid b \in \mathcal{B} \wedge |YS_b| < A \wedge c \in TS_b\}|$
21:     $mna \leftarrow \min(\{x \mid b \in \mathcal{B} \wedge x = A - |YS_b| \wedge c \in TS_b\})$
22:     insert pair $(mmc, mna)$ into $\lambda$
23: **end for**
24: /*Compute maximum total misses */
25: $misscount \leftarrow 0$
26: **while** $(\lambda \neq \emptyset) \wedge (MDDP > 0)$ **do**
27:     select $(mmc, mna)$ from $\lambda$ such that $\frac{mmc}{mna}$ is highest
28:     remove $(mmc, mna)$ from the list $\lambda$
29:     $u \leftarrow \min(MDDP, mna)$
30:     $misscount \leftarrow misscount + u \times \frac{mmc}{mna}$
31:     $MDDP \leftarrow MDDP - u$
32: **end while**
33: $crpd(set) \leftarrow \lceil misscount \rceil \times misspenalty$
```

Algorithm 1 describes the CRPD computation. It can be briefly summarized as follows. Recall that the *younger set* (*cf.* Definition 4.1) of a memory block b is captured by YS_b and the *temporal scope* (*cf.* Definition 4.2) of block b is captured by TS_b.

- We first compute $MDDP$, the upper bound on the number of inter-task cache conflicts due to preemptions (line 13).
- For each loop iteration context c, mna captures the *minimum needed ageing* for any additional misses to occur. More precisely, if the amount of inter-task cache conflicts is below mna, no additional cache miss can occur in loop iteration context c. The variable mmc captures an upper bound on the additional misses in loop iteration context c. Specifically, mmc is the number of data blocks that are accessed in context c and *persistent* in the absence of preemption. It is worthwhile to note that the additional misses in context c will always be bounded by mmc, irrespective of the number of preemptions. This is computed in lines 15-22.
- Our goal is to maximize the number of additional misses. The general idea is to assign preemption-related ageing in a greedy fashion. More precisely, we always pick a loop context having the highest $\frac{mmc}{mna}$ ratio (*i.e.* causing the most additional misses, while needing the less preemption-related ageing). This computation has been performed in lines 25-31.