

Compiler Aided Ticket Scheduling (CATS) For Non-Preemptive Embedded Systems

Tal Anker
Marvell Technology Group
TalA@marvell.com

Yaron Weinsberg, Danny Dolev
The Hebrew University of Jerusalem
{wyaron,dolev}@cs.huji.ac.il

Udi Weinsberg
Tel-Aviv University, Israel
udiweins@post.tau.ac.il

Abstract

A key challenge in designing realtime systems for non-preemptive architectures is the timely guarantee of the execution of periodic tasks, together with non-periodic tasks. The main challenge of scheduling tasks in such systems results from the conflict between the requirement to execute periodic tasks at specific time intervals, versus the requirement to yield the CPU to non-periodic tasks. The non-periodic tasks may not relinquish the CPU, thus jeopardizing the periodic tasks deadlines. This paper describes a technique that can be used by any given non-preemptive scheduler in order to produce a finer-grained schedule for the system task set. The technique increases the CPU utilization and improves the overall system performance. Compiler Aided Ticket Scheduling (CATS), provides compiler generated primitives that are used by the scheduler at runtime, allowing it to better decide about which task to run next. The enhanced scheduler remains “safe” in the sense that the chosen task is guaranteed to yield the processor before the next periodic task must be executed. This paper also proposes a programming model which enables the developer to systematically tune the embedded system tasks to meet the scheduling requirements.

1. Introduction

A typical requirement of Real Time Operating Systems (RTOS) is scheduling a set of tasks (on a single or multiple processor) so that each task completes execution before a specified deadline. Many realtime embedded systems use non-preemptive architectures in order to decrease the overhead introduced by interrupting the CPU. Although these architectures decrease the complexity of writing an RTOS, they require extra care in scheduling tasks of arbitrary execution time. By carefully designing the tasks’ periods, a developer can guarantee that the CPU will be available for any periodic task on time, when such a schedule is feasible. However, integrating non-periodic tasks may cause the

system to fail to meet its guarantees. Note that even if non-periodic tasks have a known bound on their worst case execution time, scheduling them may yield a low CPU utilization. The reason is that algorithms that use a static worst case execution time (which is fixed throughout the life of the task) for non-periodic tasks, do not take into consideration the runtime behavior of the task. A task usually contains several execution paths with different execution times. Denote the worst case execution time among these different execution paths as a task’s *global* execution time. Thus providing the scheduler with the global execution time reduces the knowledge required by the scheduler to produce fine grained schedule. Even when a task relinquishes the CPU before its global execution time, the scheduler may only schedule a non-periodic task that its global execution time fits into the remaining time. The remaining time may be lower than the smallest global time of all the schedulable tasks but may still be longer than the “dynamic” worst case execution time which is determined by the current execution paths of the tasks.

This paper proposes a technique for enhancing a non-preemptive scheduler, using compiler generated primitives. A scheduler that utilizes these primitives can produce fine-grained schedules and increases the system’s CPU utilization. We assume the existence of a non-preemptive scheduling algorithm for the **periodic** tasks. We present an enhancement that augments this algorithm by introducing the capability to schedule additional **non-periodic** tasks. Our methodology introduces a compiler generated primitive, called a “ticket”. The compiler generates a *single* ticket per task. The ticket’s data is modified whenever a task yields the processor so a single ticket is recycled throughout the task’s execution. These tickets are used by the scheduler at runtime to decide which task to schedule for execution.

The non-periodic tasks are *logically* divided into segments using predefined execution commands that are either explicitly indicated by the programmer, or implicitly identified by the compiler. An example of such a command is the invocation of a blocking I/O operation. The ticket associated with that task is updated at runtime at each such execu-

tion command to include the information regarding the **next** segment. This way the scheduler knows the *Worst Case Execution Time* (WCET) of the next segment of a non-periodic task and can decide at runtime whether to allocate the CPU to that task at a given time. Note that the next segment's WCET (henceforth termed as the *dynamic* WCET) is not necessarily equals to the global execution time. The dynamic WCET is calculated from the current executed opcode to the next yield operation.

Typically, off-line tools are used in order to determine the WCET of non-periodic tasks. These tools can only consider the longest task segment, since they do not have any information about the actual control-flow of the non-periodic tasks. Thus, the resulting schedule is a non-optimal one. The actual runtime of the tasks is typically shorter than their calculated WCET, allowing for greater flexibility in the scheduling of non-periodic task segments. CATS enables this by providing the scheduler with the WCET of each task segment that is ready to run.

The programming model proposed, includes methodology by which the developer can declare the task set, its associated tickets and a systematic way to refine the segments in order to improve the efficiency of the resulting schedule. For example, the programmer can improve the response time of non-periodic tasks. We also discuss ways to do that dynamically at runtime.

The remainder of this paper is composed of seven major sections. Section 2 presents a common scheduling algorithm for non preemptive systems, called *Cyclic Executive*. In addition we also present some of the tools (and their limitations) available for realtime developers. Section 4 gives the necessary terms and definitions relevant for scheduling algorithms in general and for this work in particular. This section also provides some background in compilation techniques that are utilized in this algorithm. Section 5 presents the scheduling algorithm and explores some optional optimizations. Section 6 presents some experimental results, and Section 7 concludes with the current status and future work.

2. Related Work

Most of today's realtime development for *non-preemptive* environment uses static schedulers that must be carefully tuned in order to obtain a schedule of the system's tasks. The deterministic behavior of such schedulers guarantees a great level of control, trace and debug capabilities. However, such static schedulers often result in a slow design process and difficult maintenance of the system upon minor changes. For example, a modification of a task's code may often result in the need to adjust the entire task set.

2.1. Cyclic Executive

The cyclic executive model has been used in many real-time systems [4, 5] and can be easily adapted to *non-preemptive* environments. A cyclic executive is a supervisory control program (or executive) that invokes the application's tasks of a real-time system, based on a cyclic schedule. The schedule is constructed during the system design phase and is repeatedly executed throughout the lifetime of the system.

The schedule consists of a sequence of actions to be taken (or subroutine calls to be made) along with a fixed specification of the timing for the actions. Since virtually all of the scheduling decisions are made at system design time, the executive is very efficient and very predictable. Although the cyclic executive model has been used in many real-time systems, there is no common approach. Each system is built using ad hoc techniques tuned for the specific application domain. Baker and Shaw take a formal approach to define the cyclic executive model, and they present a detailed analysis of the issues and problems with the approach [3].

The cyclic executive provides a practical means for executing a cyclic schedule, but the model does not specify how the schedule is constructed. The cyclic schedule is a timed sequence of computations which is to be repeated indefinitely, in a cyclic manner. This cyclic schedule is also known as the major schedule and the duration of the major schedule is called the *major cycle*. The major schedule is divided into minor schedules of equal duration, and the duration of these minor schedules is called the *minor cycle*. The minor schedules are also known as *frames*. The timing of the computations in the cyclic schedule is derived from the timing of the frames. The individual frames are designed to execute for at most the duration of the minor cycle, but if the execution of a frame exceeds the minor cycle, a *frame overrun* is said to occur. Frame overruns may be handled in a number of different ways, but the point here is that the timing of each frame is verified only at the end of the minor cycle. The executive has no knowledge or control of the timing of computations within a frame.

The primary advantages of the cyclic executive approach are its efficiency, simplicity (design and implementation) and predicability. It is efficient because scheduling decisions are made offline during the design process rather than during runtime. Thus context switching between tasks is very fast. Context switches may be embedded in compiler-generated code or they may be specified by a table associated with the current major schedule and frame. Resource constraints and precedence constraints can also be embedded in the pre-computed schedule, so no overhead is incurred at runtime for synchronization. The timing of the schedule is easily verified at runtime by checking for frame

overruns, but as long as the execution times of the frames were measured accurately during the design phase, the behavior of the system is predictable.

There are three areas where problems arise with the cyclic executive model: design, runtime and maintenance. In the design process, scheduling and task-splitting were already identified as problem areas. Handling frame overruns is another area where there are many choices that must be evaluated during design. These include policies such as immediate termination of the frame, suspension of the frame for later background processing, or continuation of the frame at the expense of the following frame.

System maintenance is regarded as the worst problem [10]. It is complicated by the fact that the code may reflect job splitting and sequencing details of the schedule. Organization of the code around timing characteristics instead of around functional lines makes it that much more difficult to modify. This lack of “separation of concerns” can make program modifications very difficult.

At runtime, the system is somewhat inflexible. It cannot generally adapt to a dynamically changing environment, even for fine-grain changes. Runtime efficiency may suffer in cases where the set of tasks comprising a frame were already ran and the minor cycle has remaining time (i.e. it is not fully utilized). Using a global WCET increases the internal fragmentation in the minor cycle and may result in significant unusable idle time. Compiler Aided Ticket Scheduling enable better utilization of the minor cycles by considering finer-grained execution times which change according to the task’s execution flow (i.e., the dynamic WCET).

2.2. Realtime Development ToolBox

There are few realtime tools that can help the developer or system designer tune the system. The most common useful tools are schedule simulators [7, 6, 13] and worst case execution time estimators [14, 15, 19].

2.2.1. Simulators

The usefulness of computer system simulation was recognized long time ago. Program execution can be simulated on many different abstraction levels. The usefulness of a simulator is limited by the accuracy of the model it provides. As the hardware complexity increase, it become more difficult to build complete system simulators providing useful timing models.

Most of the simulation tools provide a unique description language. This language is used by the developer to write the specific scheduler algorithm and task requirements (such as task execution times, priorities, periods, etc.). After the simulator is executed, the developer can determine if, under the given constraints, the task set can be scheduled or not. The usefulness of simulators is limited to small

systems, where the model corresponds well to the real system.

2.2.2. WCET Estimators

Determining the worst case execution time is a prerequisite for most realtime analysis. One method for calculating the WCET is to trace the execution time for a given task on the *real* target device while creating the “worst-case” conditions. Although this technique is very accurate, in most cases it is impractical to perform.

Another common way to calculate the WCET is the use of estimators. Estimators are simulation complement tools that help the developer to calculate WCET for a task. In order for a WCET estimator to determine the WCET of a task, it usually computes the longest path on a directed and weighted graph representation of the programs control-flow. The nodes of the graph represent the basic task’s blocks, while the edges represent the feasible paths. Each edge is weighted with the length (number of cycles) of the previous basic block. The WCET corresponds to the longest path of the graph.

The aim of most estimators is to provide the tightest bound to the worst case execution time. The major obstacle for predicting execution time on modern hardware is the use of caches. Caches improve overall throughput performance, but execution time prediction becomes very hard. A memory access operation resulting in a cache miss slows down the access operation by orders of magnitude. Thus, unless cache contents are known, predicted worst case execution time will be much higher than the average case. For programs with small data sets, however, it is feasible to predict cache contents, thus decreasing worst case execution time estimation significantly. Mueller et al. uses a technique called static cache simulation [17]. In this work, the control flow of a program is analyzed and fed to a cache system simulator. Prior to executing the program, a simulation of the cache is made. It turns out that a majority of the results of cache lookups may be accurately predicted in advance. Thus, it is possible to predict worst-case and best-case execution times with respect to memory hazards. The work of [20] and [19] calculate the WCET while considering modern hardware architectures that utilize pipelined execution for improving their performance. Other work in this field, such as [9, 14], tries to minimize or totally disregard the need for direct interaction with the developer (for example, when infinite loops exist in the code).

The current paper does not assume a specific WCET algorithm, thus a diversity of algorithms can be used and the one that gives the best estimations can be chosen.

3. Motivation

The motivation for this work came from the observation that the inherent limitation of simulation tools and es-

timators is that they provide off-line rather than runtime information. These tools *do not* incorporate their results into the system runtime in order to produce an efficient running schedule.

This paper’s contribution is in the integration of the ticket primitive resulting from the compilation process and its runtime usage by *any* task scheduler. The ticket contains the dynamic WCET which is the WCET of the next segment of a task. The use of such a fine-grained WCETs increases the scheduling algorithm flexibility. It increases the size of the “ready to run” task set from which the scheduler chooses the next task for execution.

4. System Model

4.1. Assumptions

Our model assumes that the periodic tasks are schedulable and a non-preemptive scheduling algorithm schedules them. We do not enforce a specific scheduling algorithm.

We also assume that all tasks run on a single processor and that tasks are independent in the sense that any task does not block a periodic task, even though non-periodic tasks might block each other.

4.2. Definitions

In this section the necessary terms and definitions for the rest of the paper are defined. These definitions supplement the definitions in [12, 11].

A task is a sequence of operations (op-codes) to be scheduled by a scheduler. A scheduler is an algorithm that determines the next task to be executed (i.e., it produces a schedule). This paper is restricted to non-preemptive scheduling, in which a scheduled task cannot be interrupted until it releases the CPU.

A task system $T = \{T_1, \dots, T_n\}$, where each task T_i is released periodically, is called a *periodic task system*. Each task T_i is defined by a tuple¹ (e_i, d_i, p_i, s_i) , where e_i is the upper bound on the task’s execution time, s_i is the first time at which the task is ready to run (also known as the start time), d_i is the deadline to complete the tasks once it is ready to run, and p_i is the interval between two successive releases of the task (see Figure 1). Thus, a task T_i is first released at s_i and periodically it is released every p_i . After each periodic release, at some time t , the task should be allocated e_i time units before deadline $t + d_i$.

A *synchronous task system* is a complete periodic task system in which all tasks are started and ready to run at time zero ($s_i = 0$). A *non-periodic* task is a task that is released

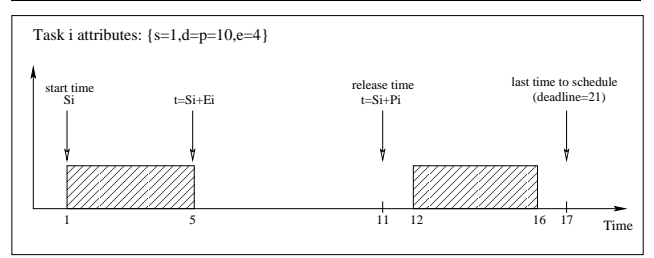


Figure 1. A Periodic Task’s Characteristics

occasionally, and at each invocation, that task may require a different execution time. A *hybrid task system* is a system that contains both periodic and non-periodic tasks. In this paper we will focus on hybrid task systems in which the periodic tasks are synchronous. To differentiate between the periodic and non-periodic tasks, a periodic task will be denoted as \tilde{T} .

This paper presents an algorithm that assumes that for each periodic task, $d_i = p_i$. To represent the runtime instance of a task the notion of a *ticket* of a task is introduced. A ticket of a periodic task, \tilde{T}_i is defined as the tuple (e_i, p_i, Pr_i) , where e_i and p_i are the execution and the period of the task, and Pr_i is the task’s priority. This assumes that any type of task scheduler used by the RTOS can be extended using this ticket. The ticket of a non-periodic task, T_j , is (e_j, Pr_j) .

4.3. EDF Related Definitions

In this paper we have chosen the non-preemptive “Earliest Deadline First” [16, 8] as our basic task scheduler. However, the algorithm is not restricted to EDF scheduling, and is designed in such a way that it can be adapted to any task scheduler.

In the EDF algorithm the task with the earliest deadline is chosen for execution. In the non-preemptive version of EDF, the task runs to completion. Among tasks with the same deadlines, tasks with higher priority are given preference. It has been proven in [12] that the non-preemptive version of EDF algorithm is optimal, in the sense that if a set of tasks is schedulable, it is also schedulable using EDF.

For the implementation of the EDF scheduling algorithm the programmer should extend the ticket of a periodic task \tilde{T} to be (e, p, Nr, Nd, Pr) , where the additional fields Nr and Nd are the next release time of the task, and the latest time by which the task should begin its execution in order to meet its deadline requirement, respectively.

¹ Note that a task system with given tasks’ start times is called a *complete task system*.

4.4. Compilation Revisited

This section defines the fundamental elements of compilation theory. The general term *translator* denotes any language processor that accepts programs in some *source language* as input and produces functionally equivalent programs in another *object language* as output. A *compiler* is a translator whose source language is a high-level language and whose object language is close to the machine language of an actual computer, either being an assembly language or some variety of machine language. A *runtime* (or *execution-time*) operation runs during program execution. A *compile-time* (or *translation-time*) operation runs during program translation and prior to program execution. *Control flow analysis* is the process that enables us to represent the flow of execution of a program as a graph. A control flow analysis produces a *Control Flow Graph* (CFG) that consists of nodes and edges. Each node represents a basic block and each edge represents a flow between blocks. A *basic block* is a series of instructions that always run in sequence meaning no single instruction in a block will ever be executed without the others. The graph is used in many analysis and optimization steps in modern compilers. There are a few algorithms to construct a CFG. The simplest and probably the most known algorithm can be found in Aho et al. [2].

4.5. CATS Compiler

Compile-time information can be useful in a variety of applications. CATS algorithm uses several compile-time techniques, which provide valuable information that can be used at runtime.

The developer uses CATS specific compiler directives in order to define the system's tasks and tickets. The compiler relates to these tickets as simple data structures in which it can store the calculated WCETs.

The compiler uses the CFG in order to calculate the WCET of the periodic and non-periodic tasks. Typical periodic tasks are comprised of a single calculated WCET, while non-periodic tasks may be comprised of a set of WCETs. In our context, a WCET is defined as the worst case execution time between two successive yields (i.e., a dynamic WCET).

The ability of a compiler to modify the developer's code, at predefined places, is also utilized. By modifying the code, the ticket primitive is maintained automatically. CATS enhanced compiler updates the ticket with the task's next WCET prior to each *yield* invocation. This technique also eliminates the need to introduce a complicated runtime structure which contains all the WCETs of a given non-periodic task. A *single* ticket is recycled to represent the next task segment WCET at runtime.

5. CATS Enhanced Scheduler

This section presents the way a scheduler can utilize the compiler generated tickets in order to improve the schedulability of the hybrid task set. An EDF scheduling will be used as an example, though the same principles can be applied to a variety of other schedulers.

5.1. Scheduler Algorithm

The algorithm is designed for a non-preemptive environment. As such, every task is assumed to explicitly relinquish the CPU to enable other tasks to run. This is done using a well-known interface provided by the OS. Usually, this is done via the *Yield()* system call. Thus, whenever a task yields the CPU, it actually invokes the task scheduling algorithm. This section discusses the CATS enhancements for the task scheduling algorithm.

The task scheduling algorithm receives as an input a set of tasks and their tickets. The algorithm is invoked whenever a task relinquishes the CPU. Whenever the input is comprised of a set of periodic tasks, the scheduling algorithm results in the same schedule as the classical EDF. However, when the task set is hybrid, and the EDF scheduler selects the IDLE task (i.e., no periodic task is ready to run), the algorithm attempts to schedule a non-periodic task. The decision on which non-periodic task to schedule is based on the tasks' tickets, mainly on the next execution time and priority as specified by the ticket.

Figure 2 presents the main logic behind the scheduling algorithm that is invoked by the *Yield()* function call. Part I of the algorithm starts with the classical EDF algorithm. The algorithm selects the next periodic task T_{next} which has the earliest deadline among all periodic tasks that are ready to run.

Part II of the algorithm is invoked when no periodic task is ready to run. The algorithm uses the tickets of the non-periodic tasks in order to select the next task to run. The chosen task should be able to run without jeopardizing the deadline of the next (earliest) periodic task. The scheduler considers the subset of non-periodical tasks that are ready to run, such that their next execution time is smaller than the slack time (the time until the next periodic task is ready). Among such tasks, the algorithm can use various criteria to pick the next task to be scheduled. For instance, one can use the algorithm in [21] which chooses a set of tasks that minimizes the remaining slack time. Any such algorithm would use the next execution time (WCET) of the tasks listed in their tickets. When there is no suitable task for execution, the IDLE task is invoked until the next periodic task is ready to run (part III).

Notice that the scheduling algorithm strives to schedule non-periodic tasks whenever there is an available time slot

Yield() called from task T_k :

```
I:  $T_{next} = \{\tilde{T}_i | \tilde{T}_i.Nd = \min(\tilde{T}_j.Nd | \tilde{T}_j.Nr \geq t)\};$ 

/* If no periodic task is ready, then
choose from the non-periodic tasks */
II: if ( $T_{next} = NULL$ )
    SlackTime = duration until next
                periodic task is ready;
    /* Pick the next non-periodic task
    that will run at most 'SlackTime'
    time units */
     $T_{next} = PickNonPeriodicTask(SlackTime);$ 

/* if no task is ready, the Idle task
will run for the time duration until
the next periodic task is ready */
III: if ( $T_{next} = NULL$ )
     $T_{next} = Idle\_Task(Timeout)$ 

SwitchTo( $T_{next}$ );
```

Figure 2. EDF Enhanced with CATS

in the schedule. Available time slots may exist between periodic slots or whenever a task completes its execution ahead of time, which can only be determined at runtime. It can be easily shown that the sequence of periodic tasks scheduled by CATS-enhanced EDF scheduler is identical to the sequence produced by EDF. The same argument holds for any task scheduler that is enhanced by CATS.

5.2. Programming Model

The programming model which utilizes the compiler generated tickets provides the syntax for declaring the tasks and their associated tickets. The development process comprised of the following facets:

- **Tasks Declaration.** Declare tasks and tickets table. State periodic and non-periodic tasks, define priorities and periods for periodic tasks in the tasks table. The compiler uses the information provided by this table to update the tickets with the tasks' WCETs.
- **Code Writing.** Write the system's code, including the periodic and non-periodic tasks. The initial code writing can be done with minimal consideration of timing demands or scheduling decisions.
- **Compilation and Simulation.** Compile the source code in order to get the calculated WCET for each task segment². By using a simulation tool (as a stand alone tool or as a part of the compilation process) the developer can identify the maximal IDLE time provided

² We have defined the flag "--wct" for this purpose.

by a specific scheduling algorithm³. The values of the WCETs must be less than the maximal idle time, otherwise not all non-periodic tasks will be scheduled.

- **Code Adjustments.** Iteratively use the simulation phase until all tasks requirements are fulfilled. Changing the tasks' periods or splitting them into several segments (by calling yield) will usually do the work.

6. Experimental Results

In order to validate and demonstrate CATS capabilities, we have designed and implemented a realtime system that utilizes CATS. The example is based on a conceptual model of a space shuttle embedded system [18]. The task set is executed on the space shuttle, which is controlled from the ground. We begin with a short description of the system's environment. We then present the system's requirements and the task set. We conclude by applying the proposed programming model on the task set, showing that it can be used in practice to simplify the process of producing a schedulable realtime system.

6.1. Environment

Our system is running on dual MIPS R4000-based processors⁴ which share access to an external RAM. Figure 3 shows a block diagram of the hardware.

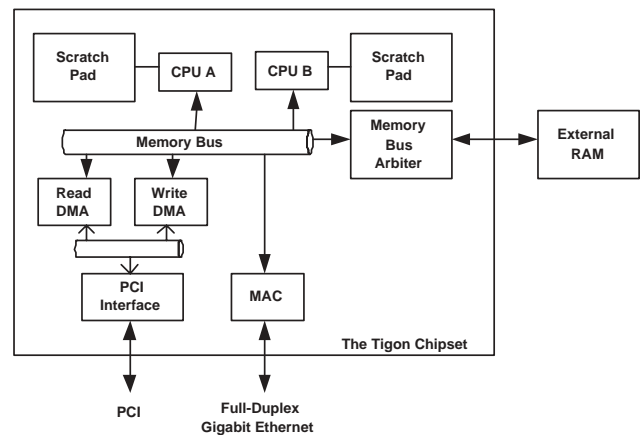


Figure 3. System's Architecture

³ We envision the integration of such a scheduling algorithm into the compiler as a plug-in.

⁴ This work applies only to scheduling on a single processor

The system does not provide a CPU interrupt mechanism thus making it suitable for implementing a non-preemptive system. The motivation behind such architectures is to increase the runtime performance by lowering the overhead imposed by interrupting the CPU each time an external event is triggered. Further, on a single processor the need for synchronization and its associated overhead is eliminated.

Our operating system is a modified version of NICOS, a lightweight non-preemptive Network Interface Card OS, which we have previously developed [1]. NICOS scheduling algorithm uses a CATS enhanced version of the non-preemptive “Earliest Deadline First (EDF)” [16, 8] algorithm.

We have also modified the compiler⁵ to use a naive WCET algorithm which counts the number of op-codes between two yields and multiplies it by the maximum execution time of the longest op-code.

6.2. System Description

The Space Shuttle uses a complex set of software and hardware modules to perform the data processing necessary for guidance, navigation, and control; payload handling and management; and performance monitoring functions. The Space Shuttle is controlled from the ground-station.

The shuttle system has the following requirements:

- The shuttle periodically performs a self-status check, collecting various data (fuel, heat etc.) and sending it to the ground station.
- The Shuttle executes a telemetry process. The telemetry process involves grouping measurements (such as pressure, speed, and temperature) and transmitting them back to the ground station.
- The ground station periodically sends “keep-alive” messages to the shuttle. If the shuttle does not receive at least one “keep-alive” for δ time units, it self-destructs.
- The shuttle should take a photograph whenever possible at its current orbit location. The shuttle photographs the image, compresses it and then sends it back to the base-station.
- In addition, there is a background requirement: the shuttle should contribute to the SETI (Search for Extraterrestrial Intelligence) effort. Whenever possible it should transmit a message and process the received signals.

⁵ gcc version 2.95

6.3. Tasks Requirements and Declaration

A task is assigned for each of the above requirements. The tasks are divided to periodic tasks (self-status, telemetric and keep-alive) and non-periodic tasks (StarMap and SETI). Each task is assigned its specific timing constrains.

Id	Task Name	Period [ms]	Priority
1	Keep Alive	10	5
2	Self Status	15	4
3	Telemetric	30	4
4	StarMap		2
5	SETI		1

Table 1. Tasks’ Timing Constrains

Table 1 describes the timing requirements of these tasks. Note that tasks 1,2 and 3 are periodic while tasks 4 and 5 are non-periodic. The CATS corresponding tickets for these tasks are derived from the table. The resulting tickets are presented in Figure 4.

```

DECLARE_TICKET_TABLE(NUM_TICKETS)
  // (Method,Name,Period,Priority)
  PERIODIC_TICKET(keepAlive,
                  "keep-alive",10,5),

  PERIODIC_TICKET(selfStatus,
                  "self-status",15,4),

  PERIODIC_TICKET(send_telemetric_data,
                  "telemetric",30,4),

  // (Method,Name,Priority)
  APERIODIC_TICKET(StarMap,
                  "StarMap",2),

  APERIODIC_TICKET(SETI,"SETI",1)

END_TICKET_TABLE;

```

Figure 4. Declaring Tickets for The Shuttle Tasks

6.4. Code Writing

The following section presents code excerpts for some of the shuttle tasks. The shuttle operating system creates a dedicated queue for each task, which is used as an inter-task communication mechanism. A task usually sleeps on

the queue⁶, waiting for a specific message to arrive. Once a message with the specific type is enqueued, the task is signaled and becomes ready to run.

The *StarMap* task, which handles the process of photographing the star, is presented in Program 1. This task is an example of a non-periodic task. The task photographs the star (invoking “StarMapShoot”), compresses the picture and sends it to the ground station (invoking “StarMapZipAndSend”). The task begins its execution only when a TAKE_PICTURE message is enqueued.

```
void StarMap() {
    while (TRUE) {
        // sleep until msg arrives
        waitForMessage(TAKE_PICTURE);
        StarMapShoot();
        StarMapZipAndSend();
        yield();
    }
}
```

Program 1. StarMap Task

6.5. Compilation and Simulation

Once the ticket table is declared, the compiler is executed in WCET mode. The compilation provides the worst case execution times and time constraints for each task. Given the system’s scheduling algorithm, the developer can simulate the task’s schedule and identify the maximum possible idle time. The maximal idle time must be long enough to contain all tasks’ WCETs values, otherwise not all of the non-periodic tasks will be scheduled. Table 2 presents the calculated WCETs given by our modified compiler.

Task	WCET [ms]
StarMap	8
SETI	9

Table 2. Tasks’ WCETs

The compiler can also provide the WCETs for specific methods according to the developer preferences as shown in Table 3.

Task.Method	WCET [ms]
StarMap.StarMapShoot()	2
StarMap.StarMapZipAndSend()	6
SETI.Listen()	2
SETI.Tx_hello()	2
SETI.analyze()	1
SETI.report()	4

Table 3. Methods’ WCETs

6.6. Code Adjustments

As seen in Table 2, the “StarMapShoot” task WCET is 8ms. Simulating the task set using an EDF algorithm shows that the maximum idle time is 6ms thus the developer must split this task. Generating the WCETs for the methods that are invoked by this task (see Table 3) enables the developer to locate the appropriate splitting point. Program 2 shows the modified source code that allows the system to meet the scheduling requirements. Section 7 presents a way to use a dedicated CATS compiler directive, to allow the scheduler to optimize the schedule by deciding at runtime where to split the tasks.

```
void StarMap() {
    // endless loop
    while (TRUE) {
        waitForMessage(TAKE_PICTURE);
        StarMapShoot();
        yield(); // this is the split-point
        StarMapZipAndSend();
        yield();
    }
}
```

Program 2. Modified StarMap Task

6.7. Runtime Schedule Comparison

The Space Shuttle system has been fully implemented and was executed twice. First, by a non-preemptive EDF scheduler, and later by the CATS enhanced EDF scheduler. The first execution is given in Figure 5 and the second one is presented in Figure 6. The figure’s x-axis presents the time, in milliseconds, where the y-axis shows the name of each task (with its WCET in parenthesis) and its execution time ranges (as filled rectangles).

Comparing the execution time ranges for the IDLE task in the two graphs clearly shows that the CPU utilization of

⁶ Invoking waitForMessage API.

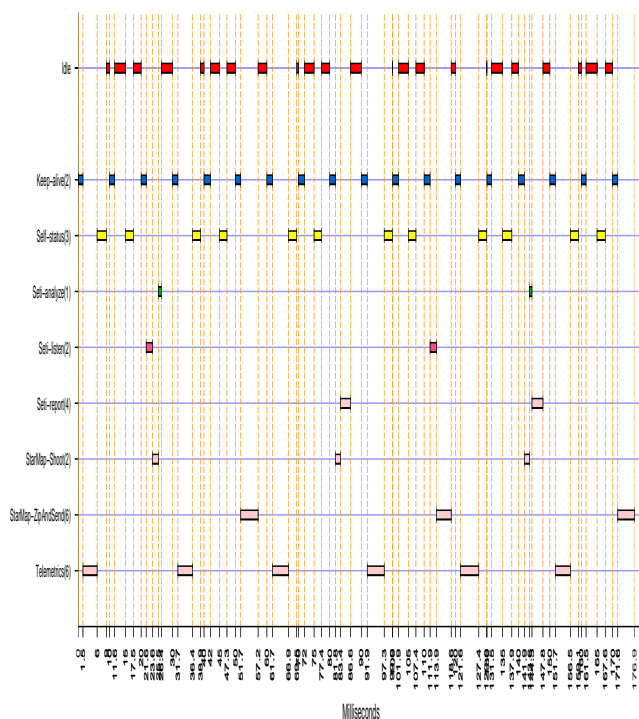


Figure 5. EDF Schedule

the CATS enhanced algorithm has been increased. Numerically, for plain EDF, the `idle` task has been executed 28.6% of the time, yielding a CPU utilization of 71.4%. For the CATS enhanced algorithm, the `idle` task ran 14.7% of the time corresponding to 85.2% CPU utilization, an increase of 20% in the system's throughput.

We have also compared the tasks' response times. Figure 7(a) and Figure 7(b) show a sequence of invocation times for each task measured from the system's start time. The x-axis shows the number of invocations where the y-axis presents the time when the specific invocation occurred. The graphs clearly show that the response times for the non-periodic tasks using the CATS enhanced scheduler are improved.

Regarding the response times, of plain EDF and CATS-enhanced schedulers respectively, calculations show that for periodic tasks, the average response time is approximately the same (1.37ms vs. 1.33ms with standard deviation of 2.49ms vs 2.39ms). Thus, the improved response for the non-periodic tasks didn't affect the response time for the periodic tasks.

The response times, in-between invocations, for the non-periodic tasks are presented in Figure 7(c) and Figure 7(b). For the `SETI` task, the average response time using

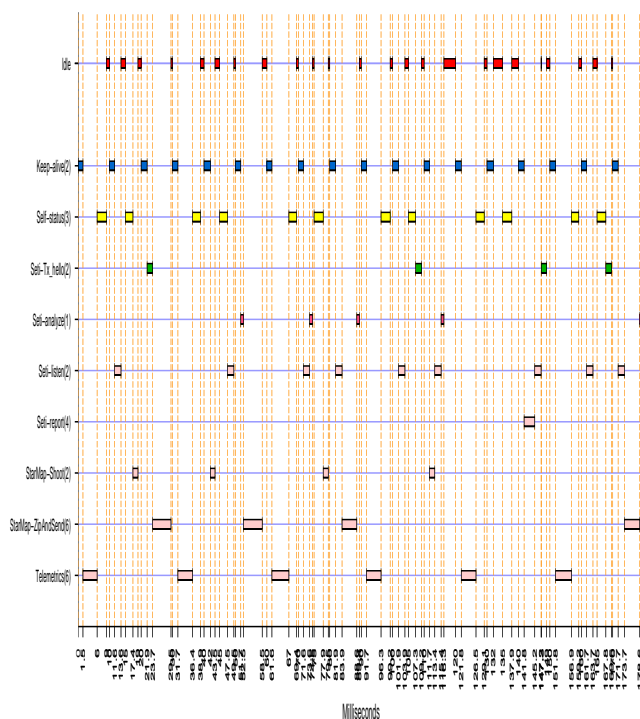


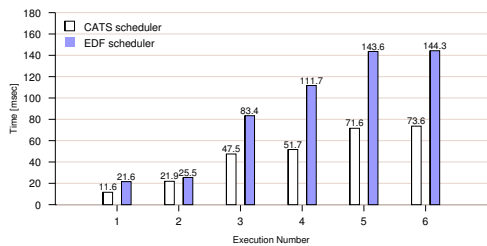
Figure 6. CATS enhanced EDF Schedule

CATS is 10.83ms with standard deviation of 8.51ms versus 22.86ms and 18.87ms using EDF (a 53% decrease in the average waiting time). For the `StarMap` task, the values are: 11.23ms and 5.78ms against 26.03ms and 2.54ms (57% decrease in the average waiting time).

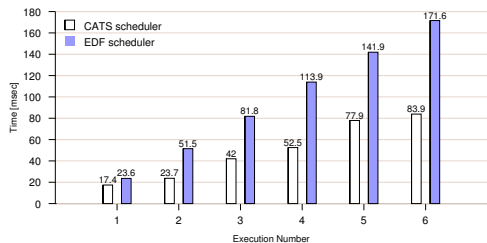
7. Discussion

The proposed CATS algorithm shows that one can significantly improve the system performance by better utilizing the available idle times. Compiler Aided Ticket Scheduling tickets can be easily integrated into any existing non-preemptive scheduler and enable it to produce finer-grained schedules.

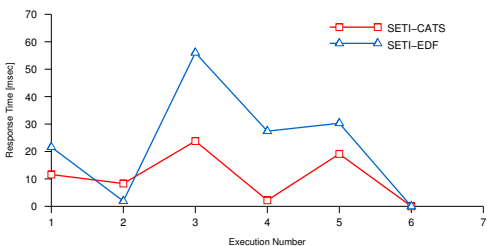
Previous solutions for such non-preemptive environments usually split the tasks by introducing multiple yields. This technique decreases the task's WCET but has several disadvantages. First, there is a trade-off between the number of yields and the overhead of the context switch. Second, task-splitting has been already identified as a major development problem - as sometimes it is impossible to split a task. Last, even if the resulting WCET is lower it is still fixed throughout the system



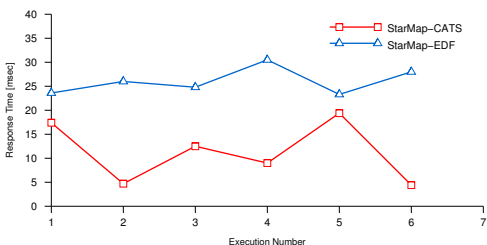
(a) SETI Invocation Times



(b) StarMap Invocation Times



(c) SETI Wait Times Between Invocations



(d) StarMap Wait Times Between Invocations

Figure 7. Non-Periodic Tasks' Responsiveness

execution. Smaller WCETs that may result from different execution flows are never tracked and utilized.

There is also an increased flexibility in the scheduling of periodic tasks. The CATS scheduling algorithm schedules a task at the earliest possible time slot after its release time. However, sometimes it is possible to postpone the task's ac-

tual activation while still keeping its deadlines intact. This observation becomes useful in the following scenario. Assume that prior to the release time of any available periodic task there is a slack time that cannot be used effectively by any non-periodic task. If, in addition, there is an expected slack time following the scheduling of the next periodic task, then the current slack time can be increased by delaying the execution of the next period task. The information can be found by studying the tickets of the next available periodic task and those right after it. Despite the fact that checking all combinations of future periodic tasks is infeasible, some first order approximation for the worse case can be considered at runtime.

8. Future Work

An important topic for future work is considering a system with multiple processors. It would be interesting to augment the ticket with CPU specific annotations and to use them in the scheduling algorithm.

Another potential enhancement of the algorithm involves the ability to dynamically modify the attributes of the task. For example, a periodic task can use an API to modify its period or to become a non-periodic task, and vice versa.

Acknowledgements

We would like to thank Maxim Grabarnik for his useful suggestions regarding the given example and for the actual implementation of the algorithm.

References

- [1] Network Interface Card Operating System (NICOS). Homepage at <http://www.cs.huji.ac.il/~wyaron/>.
- [2] *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.
- [3] T. P. Baker and A. C. Shaw. The cyclic executive model and ada. In *IEEE Real-Time Systems Symposium*, pages 120–129, 1988.
- [4] G. D. Carlow. Architecture of the space shuttle primary avionics software system. *Commun. ACM*, 27(9):926–936, 1984.
- [5] S.-C. Cheng, J.-A. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems: a brief survey. pages 150–173, 1989.
- [6] S. M. Cho and T. G. Kim. Real time simulation framework for rt-devs models. *Trans. Soc. Comput. Simul. Int.*, 18(4):203–215, 2001.
- [7] S. De Vroey, J. Goossens, and C. Hernalsteen. A generic simulator of real-time scheduling algorithms. In *The 29th Simulation Symposium*, pages 242–249, April 1996.
- [8] B. P. Douglass. *Doing Hard Time: Developing Real-Time Systems With Uml, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.

- [9] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs, 1998.
- [10] P. Hood and V. Grover. Designing real time systems in ada. Technical Report Technical Report 1123-1, SofTech, January 1986.
- [11] R. Howell and M. Venkatrao. On non-preemptive scheduling of recurring tasks using inserted idle time, 1995.
- [12] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the Twelfth IEEE Real-Time Systems Symposium*, pages 129–139, San Antonio, Texas, December 1991. IEEE Computer Society Press.
- [13] M. Jourdan and F. Maraninchi. Static timing analysis of real-time systems. *SIGPLAN Not.*, 30(11):79–87, 1995.
- [14] L. Ko, D. B. Whalley, and M. G. Harmon. Supporting user-friendly analysis of timing constraints. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 99–107, New York, NY, USA, 1995. ACM Press.
- [15] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, New York, NY, USA, 1995. ACM Press.
- [16] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
- [17] F. Mueller. Static cache simulation and its applications, 1994.
- [18] National Aeronautics and Space Administration. Biological and physical research enterprise strategy. Homepage at http://spaceresearch.nasa.gov/general_info/strat.html.
- [19] K. D. Nilsen and B. Rygg. Worst-case execution time analysis on modern processors. *SIGPLAN Not.*, 30(11):20–30, 1995.
- [20] G. Ottosson and M. Sjödin. Worst-case execution time analysis for modern hardware architectures. In *ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.
- [21] E. Shmueli and D. G. Feitelson. Backfilling with lookahead to optimize the performance of parallel job scheduling. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 228–251. Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.