Scheduling and resource allocation with Real-Time Linux

Ferdy T.Y. Hanssen

30th October 1998

University of Twente
Department of Computer Science

Graduation committee
Ir. P.G. Jansen
Dr. A.L. Schoute
Dr. Ir. G.J.M. Smit

Summary

This report is about enhancing Real-Time Linux by adding to it an Earliest Deadline First scheduler, which can be used instead of the existing fixed priority scheduler.

Moreover real-time scheduling has been considered in combination with resource allocation. The Earliest Deadline First algorithm is combined with the Stack Resource protocol in a special variant, where the real-time tasks have been modeled as transactions.

Concerning the implementation the interface which is used to manage real-time tasks has been redesigned.

Furthermore several tools have been developed in order to debug and test the implementation. These tools provide information in both a graphical and a textual form.

A series of tests have been executed to measure the scheduling overhead. The base overhead appears to be relatively high, but several improvements are suggested.

Samenvatting

Dit verslag behandelt een uitbreiding op Real-Time Linux door er een Earliest Deadline First scheduler aan toe te voegen, die gebruikt kan worden in plaats van de bestaande vaste-prioriteit scheduler.

Verder is real-time schedulen in combinatie met resource allocatie in overweging genomen. Het Earliest Deadline First algorithme is gecombineerd met het Stack Resource protocol in een speciale variant, waar de real-time taken als transacties zijn gemodelleerd.

Betreffende de implementatie is de interface die gebruikt wordt om real-time taken te verzorgen, herontworpen.

Daarnaast zijn er verschillende gereedschappen ontwikkeld om de implementatie van fouten te ontdoen en te testen. Deze gereedschappen verschaffen informatie in zowel grafische als textuele vorm.

Een serie tests is uitgevoerd om de scheduling overhead te meten. De basis overhead lijkt relatief hoog te zijn, maar er worden verscheidene verbeteringen voorgesteld.

Acknowledgements

This report contains the results of my graduation assignment. I would like to thank Ir. P.G. Jansen, Dr. A.L. Schoute, and Dr. Ir. G.J.M. Smit, who formed my graduation committee, for their support, understanding, and patience during the time I needed to finish this assignment.

For reviewing early versions of this report I would also like to thank Robert Boermans, Tjalling Hattink, and Koen Kornet.

Contents

Chapter 1
Introduction

This report contains the results of a graduation assignment which consisted of designing and implementing the Earliest Deadline First scheduling algorithm with the Stack Resource protocol on Real-Time Linux.

The assignment was born out of the idea that it would be desirable to have a real-time system to experiment with. Furthermore this real-time system had to be licensed in such a way, that it wouldn't be necessary to contact anyone or pay huge licensing fees when a change in the system was to be made. This led to the idea whether it was possible to take Linux and modify it in such a way that it is possible to run real-time experiments on it.

So the project goal was to create a Linux-based real-time system with which it would be possible to experiment with real-time scheduling algorithms and resource allocation policies. This report discusses the use of the Earliest Deadline First scheduling algorithm and the Stack Resource protocol for resource allocation in particular.

Chapter 2 deals with the theory behind real-time Earliest Deadline First scheduling and the Stack Resource protocol, making use of real-time transactions. And chapter 3 discusses some existing real-time extensions to Linux.

In chapter 4 the design regarding EDF scheduling in Real-Time Linux with resource allocation using the Stack Resource protocol is discussed. Chapter 5 goes into the implementation of this. The details to the tests with this implementation can be found in chapter 6.

In chapter 7 the conclusions are presented and some propositions for further research are presented in chapter 8.

Appendix A contains a list of all abbreviations used in this report, with their meanings and appendix B contains the bibliography. Appendix C contains the grammar of a language used to describe so-called scheduler events and appendices D and E contain the test results.

Chapter 2

Scheduling and resource allocation

In this chapter a background is presented to the scheduling and resource allocation algorithms used in the remainder of this report. The article [Jans98] is extensively used as a source for this information.

2.1 Earliest Deadline First scheduling method

The *Earliest Deadline First* (EDF) scheduling algorithm, presented in [Liu73], works with the following rules:

- each task is assigned a priority;
- as time proceeds, and the deadline of a task comes closer, its priority increases proportionally;
- the task with the highest priority is allowed to run.

When resources have to be shared among several tasks, and these tasks have to enter a mutually exclusive critical section to use these resources, phenomena like *blocking*, *priority inversion*, or *transitive waiting* may occur. Blocking occurs when a higher priority task has to wait for a resource held by a lower priority task to be released. Priority inversion happens when a high priority task is waiting for the release of a resource held by a low priority task which is pre-empted by a medium priority task. Transitive waiting occurs when a task is waiting for the release of a resource held by another task, which is waiting for the release of a resource held by another task, which is waiting for the release of a resource held by another task, i.e. a chain of tasks waiting for the release of a resource held by a predecessor.

A task may have a *static* or a *dynamic* priority. Static priorities do not vary with time, dynamic ones do. Note that in EDF a deadline can be expressed as a static or a dynamic priority. Mostly a deadline interval — from release time to deadline — is associated with a static priority while, in the context of this report, an absolute deadline is associated with a dynamic priority.

Tasks are ordered by priority by the scheduler, and then assigned CPU time in the resulting order by the dispatcher. In dynamic real-time systems, including RT-Linux, these are often combined into one entity, referred to as “the scheduler”. In the remainder of this report both will be referred to as the scheduler.

The scheduler can execute several resource allocation protocols, in order to control the execution of processes in such a way that blocking, priority inversion, or transitive waiting are limited or avoided. One of these protocols is the *Stack Resource protocol* (SRP), which was presented in [Bake91].

SRP limits blocking, and avoids priority inversion as well as transitive waiting. The basic idea is to make room for a high priority task by *not* allowing pre-emption of a low priority task by any medium priority task, if the high and low priority task share at least one resource. This limits blocking to a single task only, or more precisely, to a single critical section only. This implies the impossibility of transitive waiting and, consequently, it implies the impossibility of *deadlock*.

The implementation using EDF with SRP discussed later in this report is based on the following task model.

2.2 Real-time transactions

Tasks are based on transactions. When a transaction starts, it simultaneously acquires all the resources needed to complete the transaction. During the transaction, resources can only be released. A transaction completes when it has released all its resources. Priority inheritance is applied dynamically whenever a high priority transaction has to wait for resources in use by a lower priority transaction, i.e. the lower priority transaction acquires the priority of the high priority transaction. This avoids pre-emption of low priority transactions and speeds up the release of resources.

With tasks being based on transactions the use of critical sections is simplified immensely. It makes the model straightforward with the positive consequences of a low administration overhead and a clear schedulability analysis.

The fact that real-time transactions claim all resources needed for a single run at the start of that run, even if that resource is only used near the end of that particular run, is also a disadvantage. That resource is then claimed longer than strictly necessary. But this disadvantage is compensated for by the low overhead and the clear schedulability analysis.

Transaction state

A transaction may be *sleeping* or *ready*. The *ready* state is split up into *released*, *running*, and *pre-empted*.

A transaction is put into the administration after it is admitted to the system. Upon administration entrance it is put into the *sleeping* state. At its release time, it enters the *ready* state.

In the *ready* state, a transaction is *released* while it is waiting for the CPU. When it gets the CPU, it becomes *running*. If another higher priority transaction pre-empts this transaction, it enters the *pre-empted* state. And at the end of a transaction, it becomes *sleeping* again.

When a transaction is removed from the system, it leaves the administration.

Transaction model

A transaction, denoted τ_i, is a member of the set of all transactions $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$.

Definition 1 (Transaction) *A transaction τ_i is defined by a tuple of static parameters (D_i, P_i, C_i, R_i).*

Here D_i is the deadline interval. P_i is the time interval between two successive invocations, the period. C_i is the worst-case run-time interval[1] τ_i takes to complete. And R_i is the set of mutually exclusive resources used by τ_i. If two transactions τ_i and τ_j require the same resource ($R_i \cap R_j \neq \emptyset$), then they are not allowed to pre-empt each other.

Definition 2 (Invocation) *An invocation, denoted τ_i^a, is defined by the tuple (τ_i, r_i^a, d_i^a).*

Here τ_i^a is the a^{th} invocation of τ_i, the first invocation of τ_i is τ_i^0. τ_i^a is associated with the parameters (r_i^a, d_i^a), where r_i^a is the absolute release time from which invocation τ_i^a may run, and d_i^a is the absolute deadline at which invocation τ_i^a must be completed. Note that $D_i = d_i^a - r_i^a$, $d_i^a \leq r_i^{a+1}$, and $r_i^{a+1} - r_i^a \geq P_i$ for any $i \geq 1$, $a \geq 0$.

A transaction or invocation with a priority smaller than or equal to a running invocation may not pre-empt that running invocation. With the SRP the priority is derived from the absolute deadline.

If a transaction τ_i must be executed before τ_j then there exists a precedence relation between them, denoted by $\tau_i \prec \tau_j$. Precedence relations do not form a problem for a scheduler, but they do complicate a schedulability analysis.

2.3 The Stack Resource protocol

The *Stack Resource protocol* uses *ceilings*. The essentials of a variant of this protocol using *floors* — the inverse of ceilings — and *pre-emption levels* will be described next.

In SRP the ceiling of a resource refers to the highest static priority of all tasks that may ever use that resource. In the remainder of this text *deadline interval* will be used instead of priority and consequently floor instead of ceiling.

The floor D_R of a resource R is defined as the size of the shortest deadline interval D_i of any transaction τ_i that requires R:

$$D_R = \min\{D_i | R \in R_i\}$$

The minimum of all floors of a transaction τ_i is defined as the pre-emption deadline Δ_i of a transaction τ_i. It is defined as follows:

$$\Delta_i = \begin{cases} D_i & \text{if } R_i = \emptyset \\ \min\{D_R | R \in R_i\} & \text{otherwise} \end{cases}$$

Δ_i is a static property of τ_i and can be computed off-line for a given set T. Of all currently running and pre-empted invocations, the one with the smallest pre-emption deadline is denoted with τ_r^l, and that smallest pre-emption deadline is denoted with Δ_r.

Definition 3 (SRP) *The Stack Resource protocol is defined by the following rules:*

1. *released but not yet running or pre-empted invocations are ordered to their absolute deadlines d_i^a;*

[1] Run-times are used mainly in conjunction with a quality of service schedulability analysis, they are not needed by the scheduling algorithm.

2. *the invocation* τ_i^b *with the shortest dynamic deadline, say* d_i^b*, is selected for CPU competition;*
3. τ_i^b *will pre-empt the running invocation* τ_r^l *iff* $(D_i < \Delta_r) \wedge (d_i^b < d_r^l)$.

Due to the last-in first-out property of the SRP it may be concluded that the running invocation is on top of a stack of pre-empted invocations.

Chapter 3

Linux and real-time properties

This chapter presents two existing real-time extensions to Linux, and which extension was chosen for implementing EDF with SRP.

3.1 Linux itself

Linux itself is a time-sharing multitasking operating system, capable of *soft real-time* scheduling. It provides three scheduling algorithms, as required by the POSIX standard, part 1 [IEEE96]. These three scheduling algorithms are *Round-Robin* (RR), *First-In First-Out* (FIFO), and a Linux-specific algorithm, which is the default for all processes. Processes which are scheduled according to the RR or FIFO algorithm are at best soft real-time. Although soft real-time can be useful at times, it is not always "good enough".

Take a robotic vehicle for example: when a robotic vehicle is nearing an obstacle, and it detects this, it has to be guaranteed that it doesn't collide with it, because possible damage to the robotic vehicle has to be prevented as much as possible. So it is needed to have a system which can guarantee that the time between the detection of the obstacle and the action required to prevent a collision is bound. Soft real-time systems cannot guarantee this, *hard real-time* systems are needed for this. What extensions to Linux are available which provide something better than soft real-time?

Searching on the Internet produced two real-time extensions to Linux. The first is called Real-Time Linux [Bara98], developed at the Department of Computer Science at the New Mexico Institute of Mining and Technology. The second is called KU Real-Time Linux [Srin98], developed at the Information and Telecommunication Technology Center at the University of Kansas. These will be dealt with in the following sections in the opposite order.

3.2 Kurt

The KU Real-Time Linux (Kurt) system is a set of extensions to a standard Linux system, making it possible to create a system which will respond to certain events within a predefined time. They call it a *firm real-time* system, which is something between a soft real-time system and a hard real-timesystem. It sets up the system so that it can run in two *modes*, a *normal* mode and a *real-time* mode. When the system is in normal mode, it operates just like any other Linux workstation. But when the system is operating in real-time mode, it only runs processes which

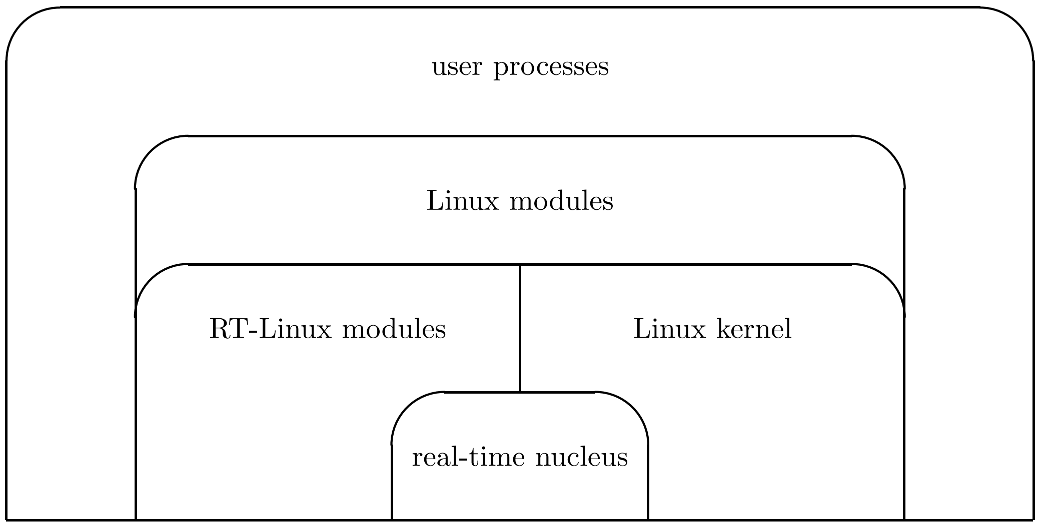


Figure 3.1: RT-Linux architecture

have been designated as real-time processes, and none other. In real-time mode the system can no longer be used as a generic workstation.

Another disadvantage of Kurt in the context of this assignment is the fact that it reads its schedule from a predefined location, be it memory or a file, and then executing that schedule while in real-time mode. It contains a static scheduler.

3.3 RT-Linux

The Real-Time Linux (RT-Linux) system follows a different approach. It lets non-real-time processes work together with real-time processes. The non-real-time processes are only allowed to run when there are no real-time tasks which need the CPU. A lot more information about RT-Linux can be found in [Bara98], [Bara97b], [Bara97a], and [Yoda95].

The architecture of RT-Linux is as follows: there is a small real-time nucleus at the heart of the system. This real-time nucleus contains the basic real-time support as real-time timing functions and real-time interrupt handlers. To use real-time tasks at least two real-time modules need to be loaded, containing the real-time FIFO handler and the real-time scheduler. Figure 3.1 shows a graphical representation of the RT-Linux architecture.

The Linux kernel runs on top of the real-time nucleus as a real-time task, with the lowest possible priority. This ensures the Linux kernel doesn't interfere with the timing for the actual real-time tasks, and the normal Linux tasks only get to run when there is no real-time task which needs CPU time, i.e. no real-time task is released.

The timer — via *IRQ #0* — controls the scheduling of real-time tasks — which is done in the function `rt_schedule()` — and when the Linux kernel is running, the normal tasks are scheduled — via the function `schedule()`. The real-time scheduler is a dynamic scheduler. Figure 3.2 depicts this.

The generic Linux kernel takes care of all the non-real-time tasks, just like it

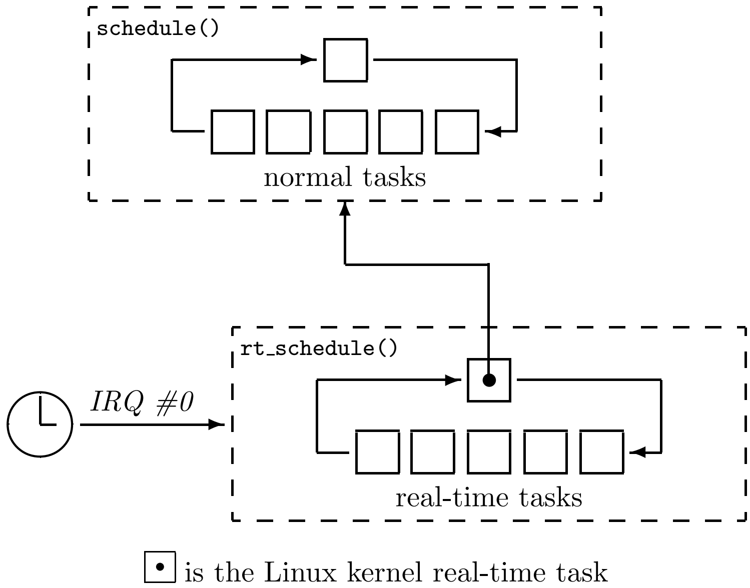


Figure 3.2: RT-Linux scheduling

normally does. All hardware is controlled by the Linux kernel as well, unless the real-time nucleus is provided with a special, real-time enabled, device driver. Only then can a peripheral be used from within a real-time task. The "normal" device drivers present in Linux cannot be used, because their designers usually didn't take a hard real-time environment into consideration.

All real-time tasks run in kernel space, which is a disadvantage. This makes it very difficult to debug real-time processes, and only the slightest error in the code of a real-time process is enough to crash or hang the entire system. The kernel doesn't enforce any limitations upon the real-time tasks, so one has to be careful when writing code in a RT-Linux environment.

A typical RT-Linux application consists of a set of real-time processes and a set of normal non-real-time user processes. The real-time processes may communicate with a set of peripheral devices using either a special real-time enabled device driver or by programming the peripheral directly from the real-time tasks. Devices should not be shared between real-time tasks and the non-real-time Linux kernel. Because using a device from the non-real-time Linux kernel may destroy any real-time abilities that device has in the system.

Communication between real-time tasks and non-real-time processes is possible through so-called real-time FIFOs. The user processes can take care of storing information on disk, post-processing data, or any other non-real-time task. A graphical representation of the architecture of a typical RT-Linux application can be found in figure 3.3.

3.4 Real-time Linux extension chosen

RT-Linux was chosen for two reasons. The first reason was that RT-Linux appeared not as complicated as Kurt after a first glance, and also after a more thorough

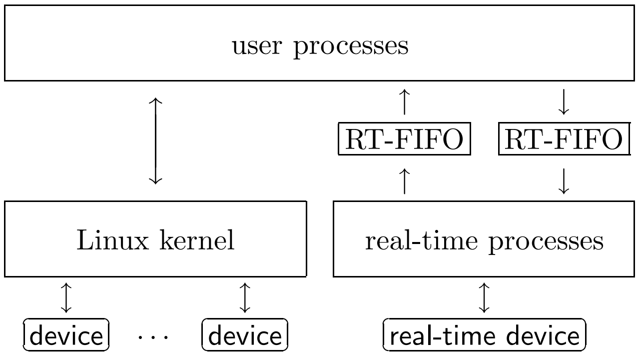


Figure 3.3: Architecture of real-time application

examination.

The second reason was the fact that RT-Linux already contained a dynamic scheduler, where Kurt only contained a static scheduler. With the assignment being to implement EDF, a dynamic scheduling technique, the choice was quickly made.

Chapter 4
Design

This chapter presents the design of enhancing RT-Linux with EDF and SRP.

Standard RT-Linux

The standard RT-Linux pre-emptive scheduler works with static priorities. When a real-time task is defined a priority has to be specified, and higher priority tasks take precedence over lower priority tasks. Both aperiodic and periodic tasks can be used, and the priority of a periodic task bears no relation to its period. Deadlines are not used, and provisions for resources are not present.

4.1 An Earliest Deadline First scheduler for RT-Linux

Ismael Ripoll has written a pre-emptive Earliest Deadline First scheduler for RT-Linux [Ripo97b]. Examination of this scheduler was started with.

It uses four task states: *none*, *dormant*, *delayed*, and *ready*. These states correspond with the states introduced in section 2.2 as displayed in table 4.1.

Initially all tasks are in the *none* state, which means they don't exist as far as the scheduler is concerned. Henceforth they will not be scheduled. When a task is set up, it is put into the *dormant* state. The task is then still not a periodic task. In the original RT-Linux scheduler this is the "resting place" for all aperiodic tasks. But the EDF scheduler does not support aperiodic tasks — the original field containing the static priority is not used in EDF — so at this moment the *dormant* state is used only as an intermediate state for the task being set up.

When the task is turned into a periodic task — i.e. its period, relative deadline and first absolute release time are set — it truly enters the scheduling mechanism, and it is put into the *delayed* state.

At the time of a process' release, that process enters the *ready* state, and stays there until it is finished with its work for this period, when it re-enters the *delayed* state. Figure 4.1 displays this, together with the RT-Linux functions which take care of each state transition.

To properly implement SRP it is needed to divide the *ready* state in the existing EDF scheduler into three states representing the states used in the transaction model. The *ready* state is split up into a *ready* state, a *running* state, and a *pre-empted* state, which correspond to respectively the *released*, *running*, and *pre-empted* states from the transaction model of section 2.2.

RT-Linux task state	transaction state
none	not in the administration
dormant	no equivalent
delayed	*sleeping*
ready	*released* or *running* or *pre-empted*

Table 4.1: State correspondence between RT-Linux EDF and transactions

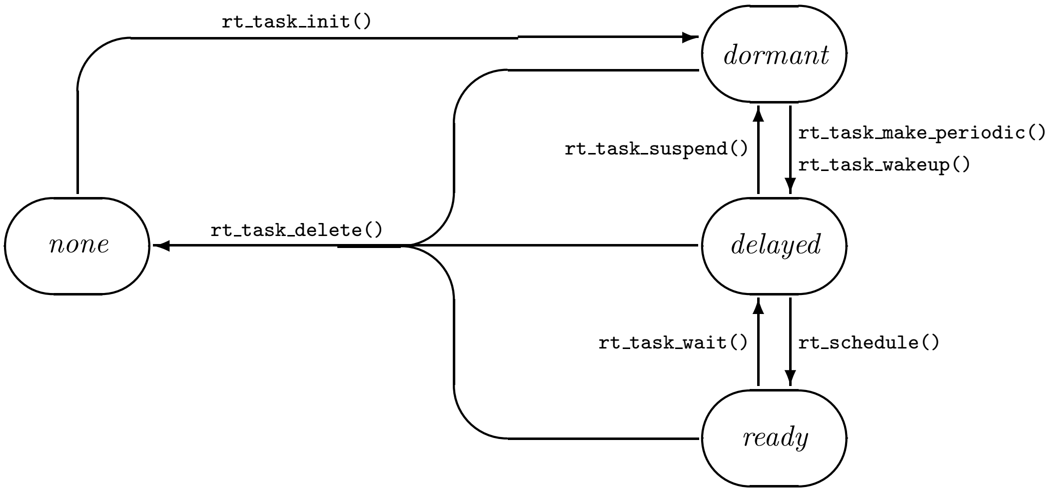


Figure 4.1: RT-Linux task state transition diagram

4.2 Using resources with RT-Linux

Already having a scheduler, it was only necessary to find a way to use resources. To the scheduler resources are nothing more than a mutually exclusive critical section. When SRP is used with transactions, the protocol already guarantees mutual exclusion, so the functions to actually enter and leave a critical section do nothing more than checking if a critical section is indeed not entered twice and providing information on when a task actually enters and leaves the critical section. The following functions were selected to perform the necessary operations:

- a function to initialize a critical section, to be called `rt_cs_init()`;
- a function to clean up a critical section when it is no longer needed, which would be called `rt_cs_delete()`;
- a pair of functions to modify the state of which task is using which resource: one to indicate that a real-time task uses a certain resource, to be called `rt_cs_used_by_task()`, and one to indicate that a real-time task no longer uses a certain resource, which would be called `rt_cs_no_longer_used_by_task()`, these functions should also take care of recomputing all necessary pre-emption deadlines Δ_i;
- a function to let a real-time task enter one or more critical sections simultaneously, to be called `rt_cs_enter()`;

- a function to let a real-time task leave one or more critical sections simultaneously, which would be called `rt_cs_leave()`.

4.3 Task and resource structures location

The original schedulers all depend on the programmer to provide adequate memory for placing the structures to use for task management, and keep a single linked list which contains the active task structures. But this makes the system depend upon the programmer of the application to provide memory which will then be used explicitly for system management. Memory used explicitly for system management should be managed explicitly by the system.

So it was decided to use an array to store the structures necessary for the management of tasks and resources. But to avoid unnecessary slow-downs at least a single linked list would be needed for use inside the scheduler, so it wouldn't have to look at all task structures, but only at those actually used. During the implementation it was found to be much easier for certain operations — notably `rt_task_delete` — to keep a double linked list.

4.4 Set functionality

It was also needed to have a data structure to be able to store which process is using which resource and which resource is being used by which process. These two sets of information are redundant, because the one can be got from the other, but having both speeds up the process of resource reservation detection and simplifies the signalling of errors to processes calling the `rt_cs_*` functions.

A set data structure is ideal for this. Unfortunately the programming language that was going to be used, C, doesn't provide sets. So a new data type would be needed, including a set of operations to work with it. The operations which would certainly be needed, are:

- *adding an element to a set*,
- *removing an element from a set*,
- *clearing a set*,
- *checking if a set is empty*, and
- *checking if a set contains a certain element*.

This data type is also very useful when a schedulability analysis algorithm has to be implemented like the one in [Laan97].

4.5 Debugging functionality

The last item that surely would be needed was a way to debug the system. The EDF scheduler by Ismael Ripoll already contained a debugging system using a real-time FIFO. The scheduler put some information about the actual schedule — absolute process release time, absolute process deadline, absolute process start of execution, and absolute process end of execution — into the real-time FIFO, and a user process read the real-time FIFO and put the information into a format, which is readable by both man and machine. He also provided a program to display that data in a graphical manner.

This setup was taken as a basis, particularly because it is not possible to use conventional methods to debug a real-time process running in kernel space. So the man-machine-readable format he used was enhanced to better suit this assignment — a bit more information from the scheduler was required — and the result of that is presented in appendix C. The man-machine-readable format is called a *schedule events listing*. This was done because it is easier to derive quantifiable data from a well-defined file. To understand the grammar better, a fictional, and commented, example is given in figure 4.2. Note that the actual events do not have to appear in chronological order.

A few programs to transform the schedule events listing into a form more easily interpretable by humans would also be needed. At least a verbose textual listing of events and a graphical representation would have to be generated. More information about these programs is given in section 6.1.

To debug resource allocation and usage another real-time FIFO would have to be used to track a little bit more information — the actual state of use of the mutually exclusive semaphore — concerning resource claim and release events. A new grammar was, however, not designed for this. A program was written to simply transform the data coming out of the real-time FIFO into human readable text, as it would only be used for debugging. All interesting statistical data — time of resource claim and release — is already provided for in the schedule events listing.

```
# Start of header, containing definitions

# Define three tasks
TASK 1 "First task"
TASK 2 "Second task"
TASK 3 "Third task"

# Define two resources
RESOURCE 1 "First resource"
RESOURCE 2 "Second resource"

# End of header

# Start of body, containing actions

:BODY

ACTI 1 0 250            # Task 1, release at 0, deadline at 250
ACTI 2 0 500            # Task 2, release at 0, deadline at 500
ACTI 3 100 200          # Task 3, release at 100, deadline at 200
RSCL 1 1 10             # Task 1 claims resource 1 at 10
EXEC 1 0 100            # Task 1 runs from 0 to 100
RSCL 3 2 100            # Task 3 claims resource 2 at 100
RSFR 3 2 140            # Task 3 frees resource 2 at 140
EXEC 3 100 160          # Task 3 runs from 100 to 160
ACTI 3 300 400          # Task 3, release at 300, deadline at 400
RSFR 1 1 190            # Task 1 frees resource 1 at 210
EXEC 1 160 220          # Task 1 runs from 160 to 220
RSCL 2 2 250            # Task 2 claims resource 2 at 250
REDEF DEADLINE 2 300 350 # Deadline of task 2 is redefined to be 350 at 300
RSFR 2 2 350            # Task 2 frees resources 2 at 350
EXEC 2 220 350          # Task 2 runs from 220 to 350
RSCL 3 2 350            # Task 3 claims resource 2 at 350
RSFR 3 2 390            # Task 3 frees resource 2 at 390
EXEC 3 350 395          # Task 3 runs from 350 to 395

# End of body
```

Figure 4.2: A sample schedule events listing

Chapter 5

Implementation

In this chapter the actual implementation of EDF with SRP on RT-Linux is discussed. The previous chapter discussed the design, this chapter will discuss several implementation-specific aspects of that design.

First the important data structures are explained, followed by an explanation of the scheduler, the critical sections, the process stack handling functions, and finally some functions to provide some information about the task and resource sets are discussed.

In the remainder of this report the terms task, process, and transaction will be used alternately, meaning a task from the transaction model.

The version of RT-Linux used as a basis for this implementation is RT-Linux version 0.6 upon Linux version 2.0.35.

5.1 Important data structures

The set data type

The declaration of the `set` data type is given in figure 5.1. The bits of a 64-bit word are simply used to designate the elements of a set. Consequently, a set may contain maximally 64 elements, and henceforth, at most 64 real-time tasks — including the Linux kernel task — and 64 resources can be defined. The compiler that was used — `gcc` version 2.7.2.1 — cannot work with a simple data type of more than 64 bits on a CPU of the Intel 80x86 class. If more than 64 real-time tasks or resources are required, the set data type has to be changed, and will then get considerably more complicated. And more complicated data structures take more time to work with.

The declarations of the operations which function on this data type are given in figure 5.2, and table 5.1 tells which operation exactly does what.

The real-time task data type

Figure 5.3 contains the declaration of the `RT_TASK` data type. The element `stack` is the stack-pointer of the real-time task — each real-time task has to have its own stack —, and `uses_fp` indicates if the real-time task uses floating point arithmetic or not. With an Intel 80x86 class processor it is important to know whether a real-time task uses the floating point unit of the CPU, because if it does not, the real-time task switch overhead is lower, as the floating point state doesn't have to be saved. The `magic` field is set to a special predefined value if the task structure is in use, and is set to something else if it is not.

The `state` part contains the state of the real-time task, which is one of `RT_TASK_NONE`, `RT_TASK_DELAYED`, `RT_TASK_DORMANT`, `RT_TASK_PREEMPTED`, `RT_TASK_READY`, and `RT_TASK_RUNNING`. These task states have already been explained in section 4.1. The pointer `stack_bottom` points to the bottom of the stack, and is used when the task

```
typedef __u64 __set;

typedef __set set;
```

Figure 5.1: The data type `set`

```
/* Macro to inquire about max. nr. of elements a set can contain */
#define SET_MAX_ELEMENTS        (sizeof(__set) * 8)
#define SET_BOTTOM_ELEMENT      0
#define SET_TOP_ELEMENT         SET_MAX_ELEMENTS

/* Constant sets */
#define SET_EMPTY               0
#define SET_FULL                (~SET_EMPTY)

/* Macros to be used _outside_ expressions (s must be lvalue) */
#define SET_CLEAR(s)            (__set) s = SET_EMPTY
#define SET_ADDALL(s)           (__set) s = SET_FULL
#define SET_ADD(s,i)            (__set) s |= ((__set) 1 << (i))
#define SET_REMOVE(s,i)         (__set) s &= ~((__set) 1 << (i))

/* Macros to be used _inside_ expressions */
#define SET_CONTAINS(s,i)       ((s) & ((__set) 1 << (i)))
#define SET_UNION(s1,s2)        ((s1) | (s2))
#define SET_INTERSECTION(s1,s2) ((s1) & (s2))
#define SET_DIFFERENCE(s1,s2)   ((s1) & ~(s2))
```

Figure 5.2: Operations for the `set` type

Operation	Explanation
`SET_ADD(s,i)`	Assign $s \cup \{i\}$ to s
`SET_ADDALL(s)`	Assign $\mathbb{U}$ to s
`SET_BOTTOM_ELEMENT`	Return value of lowest numerical value which can be stored in a set
`SET_CLEAR(s)`	Assign $\emptyset$ to s
`SET_CONTAINS(s,i)`	Return $i \in s$
`SET_DIFFERENCE(s1,s2)`	Return $s_1 - s_2$
`SET_EMPTY`	Return $\emptyset$
`SET_FULL`	Return $\mathbb{U}$
`SET_INTERSECTION(s1,s2)`	Return $s_1 \cap s_2$
`SET_MAX_ELEMENTS`	Return maximum amount of numbers which can be stored in a set
`SET_REMOVE(s,i)`	Assign $s \cap (\mathbb{U} - \{i\})$ to s
`SET_TOP_ELEMENT`	Return value of highest numerical value which can be stored in a set
`SET_UNION(s1,s2)`	Return $s_1 \cup s_2$

Table 5.1: Explanations of the `set` operations

```
enum {RT_TASK_NONE, RT_TASK_READY, RT_TASK_DELAYED, RT_TASK_PREEMPTED,
      RT_TASK_RUNNING, RT_TASK_DORMANT};

struct rt_task_struct {
   int *stack;                  /* HARDCODED in rt-task switch code! */
   int uses_fp;                 /* THIS ONE IS TOO! */
   int magic;                   /* Used to indicate this task is in use */
   int state;
   int *stack_bottom;
   void (*fn)(int);             /* Function to execute periodically */
   int id;
   RTIME P;                     /* Period */
   RTIME resume_time;           /* Absolute release time */
   RTIME D;                     /* Deadline interval */
   RTIME d;                     /* Absolute deadline */
   RTIME Delta;                 /* Preemption deadline */
   set R;                       /* Set of resources used by task */
   struct rt_task_struct *next; /* Pointer to next active task_struct */
   struct rt_task_struct *prev; /* Pointer to prev. active task_struct */
};

typedef struct rt_task_struct RT_TASK;
```

Figure 5.3: Data type `RT_TASK`

is deleted. The pointer `fn` points to the code for the task, and `id` contains the task identifier.

The `P` field contains the period, and `resume_time` contains the next absolute release time r_i^a. Field `D` contains the relative deadline D_i, and `d` contains the next absolute deadline d_i^a. The `Delta` element contains the pre-emption deadline Δ_i, and field `R` contains the set of resources R_i this task uses. The fields `next` and `prev` are used for the double linked list of tasks which are present in the system.

The critical section data type

The data type used for resources, or critical sections, is depicted in figure 5.4. The `magic` field is again used to define when a particular structure is in use and when it is not. The `used` field indicates whether the resource is currently being used, and is used for checking if a resource is not used twice at the same time. The field `id` contains the critical section identifier.

The set `used_by` contains the set of tasks that have indicated to use this resource. The `D_R` field contains the floor D_R of the resource.

The pointers `next` and `prev` are used for the double linked list of all resources in use. The `event` field, finally, is used to transfer debugging information about this resource to user space through a real-time FIFO.

5.2 The scheduler

The scheduler itself consists of one function: `rt_schedule()`. It is invoked when a task is finished, suspended, woken up, or when the timer goes off. It does not work with a release time, but with a release interval. The idea behind such an interval is

```
typedef struct rt_cs_tp {
   int magic;
   int id;
   int used;             /* Indicates resource is in use (for debugging) */
   set used_by;          /* Set of indexes */
   RTIME D_R;            /* Floor */
#ifdef DEBUG
   rt_event_tp event;
#endif
   struct rt_cs_tp *next; /* Pointer to next active critical section */
   struct rt_cs_tp *prev; /* Pointer to previous active critical section */
} rt_cs_tp;
```

Figure 5.4: Data type `rt_cs_tp`

```
#define RT_SCHED_GET_RELEASE_INTERVAL_LENGTH  _IOR(200, 0x01, RTIME)
#define RT_SCHED_SET_RELEASE_INTERVAL_LENGTH  _IOW(200, 0x02, RTIME)
```

Figure 5.5: The `ioctl` commands for the EDF scheduler

```
extern RT_TASK *rt_current;

#define rt_current_id ((rt_current)->id)
```

Figure 5.6: Declaration of `rt_current`

that it should compensate for scheduling overhead, so its length would depend on the computer system used. It means that a process will be released in the interval $[r_i^a - t, r_i^a]$, where t is the *scheduler release interval length* (SRIL). The disadvantage to this approach is that a process can be released before its actual release time, and thus can misread a certain value, if that value will only become available at the actual release time of the process.

The scheduler release interval length can be set using two *ioctl commands*[1] of which the declarations can be found in figure 5.5, and it defaults to 0. The device which should be used is `/dev/rt_sched`, which uses major device number 10 — the Linux *misc* device — and minor device number 200 in the current implementation.

A pointer to the task which is currently running is kept in the variable `rt_current`, and because the task identifier of the currently running task is needed frequently, a macro `rt_current_id` has been defined. These declarations can be found in figure 5.6.

The function `rt_schedule()` performs the following steps:

1. It searches the list of active tasks for those that can be released. Any task that can be released in the scheduler release interval is released, i.e. its state is changed to *ready*.
2. Check if the current task is finished, in which case it should be removed from the process stack.

[1] An *ioctl command* is a command which can be sent to a part of the kernel using the `ioctl` system call.

3. Now calculate the lowest pre-emption deadline Δ_r by simply looking at the Δ_i of the process on top of the stack.
4. Then the scheduler determines which task is to run next, by picking the task with the lowest absolute deadline.
5. Determine if the task just selected may pre-empt the process on top of the stack by checking the conditions according to SRP. If it may not, the task on top of the stack becomes the next task to run.
6. The scheduler now determines if the next task to run is going to be pre-empted within a certain time interval by looking at the absolute release times of all sleeping and released, but not pre-empted or running tasks.
7. If a pre-emptor has been found, set the timer right, otherwise don't set the timer — note that the last option cannot happen with at least one periodic task.
8. If the new task to run is not the same as the one already running, pre-empt the process which was running — i.e. make its state *pre-empted* — and put ourselves on top of the stack. Then perform a task switch. The state of the new task to run is set to *running*.

The current implementation provides no support for aperiodic tasks.

Task management

In order to manage the real-time tasks, several functions are provided. Their declarations can be found in figure 5.7.

The functions `rt_task_init()` and `rt_task_make_periodic()` are used to set up a task. The first function returns a task identifier, which is needed to call all other functions. All other functions return 0 on success and all functions return a negative error value on failure. This function is typically called from the `init_module()` function which is called whenever a module is loaded into the Linux kernel.

The `rt_task_delete()` function is used to withdraw a task from the system. This function is typically called from the `cleanup_module()` function, which is called whenever a Linux kernel module is removed from the kernel.

The function `rt_task_wait()` is called when a task is finished with its work for this period. It changes the task state to *delayed* and sets a new absolute release time and a new absolute deadline. In the current implementation it is not necessary for the real-time task to do this itself, as all periodic real-time tasks are running within a framework, which is shown in figure 5.8.

The function `rt_task_suspend()` changes a task's state from *delayed* to *dormant*. A task should not suspend itself, unless it is very sure it will be woken up by somebody else. The function `rt_task_wakeup` is its opposite, it changed a task's state from *dormant* back to *delayed*, and a task should **never** do this to itself. These last two functions are not used in the current implementation, but left over from the normal RT-Linux functions.

```
extern int rt_task_init(void (*fn)(int data), int data, int stack_size);
extern int rt_task_make_periodic(int id,
                                 RTIME start_time,
                                 RTIME period,
                                 RTIME deadline);

extern int rt_task_delete(int task_id);
extern int rt_task_wait(void);

extern int rt_task_suspend(int task_id);
extern int rt_task_wakeup(int task_id);

extern inline void rt_use_fp(int flag)
```

Figure 5.7: Real-time task operations

```
void rt_periodic_task_frame(int data)
{
    while (1) {
        /*
         * Calls to rt_cs_enter and rt_cs_leave are not strictly necessary,
         * used only for debugging
         */
        rt_cs_enter(rt_current->R);
        rt_current->fn(data);       /* fn() runs as a transaction */
        rt_cs_leave(rt_current->R);
        rt_task_wait();
    }
}
```

Figure 5.8: Real-time task framework

5.3 Critical sections

Because the SRP combined with transactions guarantees mutual exclusion, the real-time process programmer no longer has to worry about resource claims or releases. He only has to specify which task uses which resource, and as a task runs, its resources are automatically claimed and released. The function declarations pertaining to resource management are displayed in figure 5.9.

The function `rt_cs_init()` is called to initialize a critical section. It returns a critical section identifier on return when all went well, or a negative error value if it did not. This critical section identifier is needed to call all other functions, which return 0 on success or a negative error value if they executed unsuccessfully. This function is, like `rt_task_init()`, typically only called from `init_module()`.

The `rt_cs_delete()` function removes a critical section from the system. And, like `rt_task_delete()`, this function is also typically only called from `cleanup_module()`.

The function `rt_cs_used_by_task()` is used to indicate which resource is used by which task. Like `rt_cs_init()`, this is also typically called from `init_module()`. The function `rt_cs_no_longer_used_by_task()` does the exact reverse of `rt_cs_`

```
/*
 * NEVER call these from RT tasks, ONLY call these from non-RT
 * kernel functions (something like init_module and cleanup_module)
 */
extern int rt_cs_init(void);
extern int rt_cs_delete(int cs_id);
extern int rt_cs_used_by_task(int cs_id, int task);
extern int rt_cs_no_longer_used_by_task(int cs_id, int task_id);

extern int rt_cs_enter(set css);
extern int rt_cs_leave(set css);
```

Figure 5.9: Critical section operations

```
static RT_TASK *task_stack[RT_MAXTASKS];
```

Figure 5.10: Definition of the process stack

`used_by_task()`, and is, obviously, typically called from the `cleanup_module()` function.

The functions `rt_cs_enter()` and `rt_cs_leave()` are currently used for debugging only. Checking whether a resource is already in use — yelling if it is — and actually marking the resource as used is done by `rt_cs_enter()`. Setting a resource's state to unused is done by the function `rt_cs_leave()`. In the current implementation they are called from the periodic task framework.

The current implementation does not provide support for a transaction to release a resource "early", i.e. before the end of its execution for a certain period.

5.4 The process stack

When a process becomes *running* it is put onto the stack of processes, In order to handle this stack, a small kernel module was written.

The stack consists of a simple array of pointers to task structures. It is shown in figure 5.10. As the stack can never contain more than the maximum number of processes possible, the maximum size needed is known in advance. A simple array index pointer is used to keep track of the stack pointer.

The declarations of the functions it provides can be found in figure 5.11. The function `stack_push()` pushes a task onto the stack, `stack_pop()` removes the task on top from the stack. The function `stack_top()` returns the task structure which is on top of the stack. Checking if the stack is empty can be done with the function `stack_empty()` and `clear_stack()` makes the stack empty. This last function is only necessary on initialization.

In order to debug the stack, a device called `/dev/rt_stack` was added, together with a couple of *ioctl commands*. The device `/dev/rt_stack` currently uses major device number 10 and minor device number 201 in the current implementation. The *ioctl commands* available can be found in figure 5.12. The difference between the *dump* and *print* commands is the verbosity of the output they generate.

```
extern int stack_push(RT_TASK *t);
extern void stack_pop(void);
extern RT_TASK *stack_top(void);
extern int stack_empty(void);
extern void clear_stack(void);
```

Figure 5.11: Stack functions

```
#define STACK_DUMP_STACK   _IO(201, 0x01)
#define STACK_PRINT_STACK  _IO(201, 0x02)
#define STACK_CLEAR_STACK  _IO(201, 0x03)
```

Figure 5.12: The `ioctl` commands for the stack module

```
void dump_rt_tasks(void);
void dump_rt_critsects(void);

void partial_dump_rt_tasks(void);
void partial_dump_rt_critsects(void);
```

Figure 5.13: Informative functions

5.5 The information module

Wanting to know whether all these functions operate correctly, another module was written which is able to dump all active task structures and all active critical section structures. This module provides only four functions, two to dump the active task structures and two to dump the active critical section structures. Their declarations can be found in figure 5.13.

The functions that begin with `partial_` dump only part of the structures, notably the sets telling which task uses which resource, and the tasks' periods P_i, relative deadlines D_i, and pre-emption deadlines Δ_i. Their output is written to the kernel log-files.

Chapter 6
Testing

In this chapter the tools and methods used to debug and test are presented, followed by a discussion of the test setup and results.

6.1 Tools used for debugging and testing

In order to get some quantifiable data out of the scheduler and resource allocation policy implementation, a collection of tools was written. These are very useful for testing and debugging a scheduler and a resource allocation policy, and reporting a collection of performance related numbers from a test run.

These tools are all normal non-real-time Linux processes. In order to communicate between the real-time environment and the tools two real-time FIFOs are used. One to transfer information about the scheduling of processes and the time of resource claims and releases, this one will be called *sched-fifo*. The other one is to transfer debugging information about the resource claims and releases, notably the state of the mutually exclusive critical section, this one will be called *resource-fifo*.

Because slight errors can have a big impact when writing code that is to function in *kernel space* and because of the fact that a specialized kernel debugger is unavailable, debugging in the classical manner is impossible. This was the first reason such tools were wanted. Wanting to know what the scheduler and resource allocator were doing was the second. The idea came from the `crono` [Ripo97a] program by Ismael Ripoll.

He used a real-time FIFO to transfer the scheduling events to a user space program which displays it graphically on screen. The reason these new tools were written is that `crono` wouldn't function properly on the systems used and it would take too much time figuring out how to get `crono` to work. It looked pretty complex when it was examined, and it was concluded it would be easier to start building a new version. The only advantage `crono` has over the new one is speed. Some command line programs were also added to directly convert the list of schedule events into a graphical or textual representation.

The names of the tools that have been written are as follows:

- `diag2gif`: creates a GIF file containing a graphical representation of the schedule events listing it is given as input,
- `diag2png`: creates a PNG file containing a graphical representation of the schedule events listing it is given as input,
- `diag2txt`: creates a text file containing a verbose textual representation of the schedule events listing it is given as input,

- `diag2xpm`: creates an XPM file containing a graphical representation of the schedule events listing it is given as input,
- `diagview`: opens a window containing an interactive graphical representation of the schedule events listings it is given as input,
- `diag2num`: a modification to `diag2txt` to provide more compact input for
- `dat2avg`: which calculates average numbers for each task, useful to generate test reports.

These tools are very useful for debugging, because it can immediately be seen when the scheduler loses track with a given task set, and correct the situation. And more severe errors[1] are caught by the Linux exception handler, resulting in a nice "Oops" on the display and in the kernel log-file. Then it is possible to trace the place of the error, and to try to figure out which part of the source code matches with the assembly code where the error occurs.

To ease the task of debugging and testing, a script was implemented, which consists of these steps:

1. Load up the appropriate system modules:
   - `rt_fifo_new.o`: this module contains the real-time FIFO device driver,
   - `rt_stack.rkmo`: this module contains the code for the stack that is used to put pre-empted processes and the running process on,
   - `rt_sched.rkmo`: this module contains the code for the scheduler and resource allocation policy,
   - `rt_dump_info.rko`: this module contains the code for a few routines which will dump information about all active real-time processes and resources.
2. Optionally change the scheduler release interval length.
3. Start `getevents` in the background, this program will read a specified[2] amount of data from the *sched-fifo*.
4. Start `rt_cs_getevents` in the background, this program will read a specified[2] amount of data from the *resource-fifo*.
5. Insert the module or modules containing the real-time processes.
6. While the real-time processes are running and the real-time environment is generating lots of data about what is going on and `getevents` and `rt_cs_getevents` are collecting this data, wait until `getevents` and `rt_cs_getevents` are finished.
7. Remove the module or modules containing the real-time processes.
8. Optionally remove the following real-time system modules:
   - `rt_dump_info.rko`,
   - `rt_sched.rkmo`,
   - `rt_stack.rkmo`: with the current implementation removing real-time tasks is a process which happens bluntly, resulting in various errors when used with different sets of real-time tasks and resources after each other due to inconsistent pre-emption deadlines and an overpopulated stack; only `rt_stack.rkmo` cannot be removed before `rt_sched.rkmo`

[1] These more severe errors are typically problems with pointer handling: trying to follow a `NULL` pointer, or trying to use a pointer to some previously deallocated kernel memory.

[2] This amount is specified on the command line as an option to the program.

and `rt_dump_info.rko` have been removed, due to inter-modular dependencies,
- `rt_fifo_new.o`: it is not strictly necessary to remove this module, as it doesn't contain any known bugs, and there were no problems with leaving this module in during testing.

9. Use `bin2diag` to translate the binary output from `getevents` into a textual schedule events listing. The definition and a sample of the textual schedule events listing was already given in section 4.5.
10. Optionally use the program `rt_cs_bin2txt` to convert the binary output from `rt_cs_getevents` into a textual list of resource events.
11. Now there are several options:
    - use `diag2txt` to construct a more readable list of events from the schedule events listing, figure 6.1 contains a sample[3],
    - use any of `diag2gif`, `diag2png`, or `diag2xpm` to create a graphical representation of the schedule events listing, an example of such a graphic can be found in figure 6.2,
    - use `diagview` to interactively view the schedule events listing, figure 6.3 contains a screen-shot of the program,
    - use `diag2num` in association with `dat2avg`[4] to extract the average execution time per process per period, the average delay between process release and actual process execution, and the average resource usage time per resource per process, of which a sample is shown in figure 6.4.

---

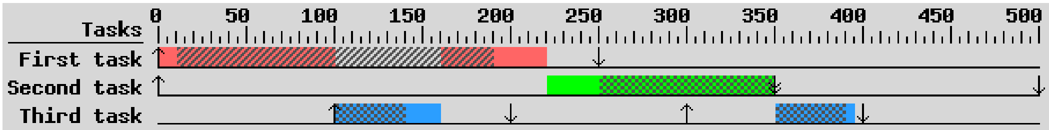
[3]Note that the samples shown here are completely fictional and do not adhere to any algorithm or protocol at all, they are intended to show the possibilities of the tools.

[4]Common usage of `diag2num` and `dat2avg` is something like this:

```
diag2num -d <input.diag> | dat2avg
```

```
Total number of tasks: 3
  Task  1 "First task"
      Action: release at 0
      Action: start of execution at 0
      Action: claim resource 1 at 10
      Action: end of execution at 100
      Action: start of execution at 160
      Action: free resource 1 at 190
      Action: end of execution at 220
      Action: deadline at 250
    Highest action time of this task: 250
    Total execution time of this task: 160
    Total number of active periods of this task: 1
    Total number of periods of this task: 1
    Average execution time per period of this task: 160.00
  Task  2 "Second task"
      Action: release at 0
      Action: start of execution at 220
      Action: claim resource 2 at 250
      Action: redefinition of deadline to 350 at 300
      Action: free resource 2 at 350
      Action: end of execution at 350
      Action: deadline at 500
    Highest action time of this task: 500
    Total execution time of this task: 130
    Total number of active periods of this task: 1
    Total number of periods of this task: 1
    Average execution time per period of this task: 130.00
  Task  3 "Third task"
      Action: release at 100
      Action: start of execution at 100
      Action: claim resource 2 at 100
      Action: free resource 2 at 140
      Action: end of execution at 160
      Action: deadline at 200
      Action: release at 300
      Action: start of execution at 350
      Action: claim resource 2 at 350
      Action: free resource 2 at 390
      Action: end of execution at 395
      Action: deadline at 400
    Highest action time of this task: 400
    Total execution time of this task: 105
    Total number of active periods of this task: 2
    Total number of periods of this task: 2
    Average execution time per period of this task: 52.50
  Resource  1 "First resource"
  Resource  2 "Second resource"
Highest action time of any task: 500
Grand total execution time of all tasks: 395
```

Figure 6.1: Sample output of `diag2txt`



Symbol	Meaning
Coloured area	Process execution
Striked through area	Resource allocated
↑	Process release
↓	Process deadline
↡	Process inherited deadline[5]

[5] A process inherited deadline means that a process inherits a priority, which is expressed as an absolute deadline.

Figure 6.2: Sample output of `diag2gif`

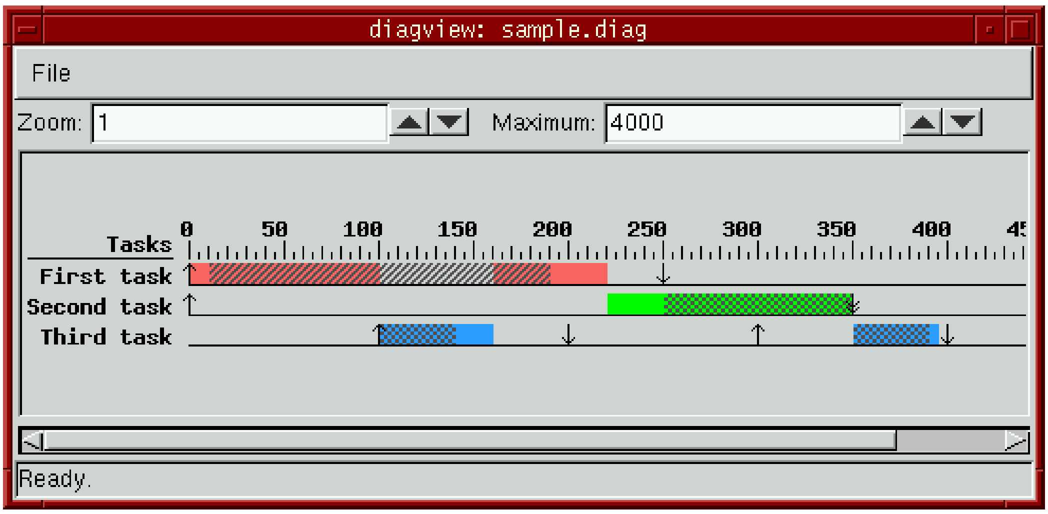


Figure 6.3: Screen-shot of `diagview`

```
3 tasks total
task 1
 avg. exec time 160.00
 avg. time between release and exec 0.00
 resource 1
  avg. usage time 180.00
task 2
 avg. exec time 130.00
 avg. time between release and exec 220.00
 resource 2
  avg. usage time 100.00
task 3
 avg. exec time 52.50
 avg. time between release and exec 25.00
 resource 2
  avg. usage time 40.00
```

Figure 6.4: Sample output of `dat2avg`

6.2 Test setup

The test setup consisted of two very different systems. The first system, which was used for debugging, is a 80486 class machine, running at 66 MHz, and is equipped with 16 MB of memory. The second system, which was used for development, is a PentiumPro class machine, running at 180 MHz, and is equipped with 32 MB of memory.

To test the scheduler and resource allocator a test scheme was implemented, which can be found in figure 6.5. All combinations of a scheduler release interval length of 0, 2, 5, 10, and 20 with 2, 4, 6, or 10 processes and 1, 2, 3, or 4 resources total were tested. Each process uses exactly one resource, which may be shared with other processes. Therefore when testing with 2 processes, at most 2 resources could be used. The combination of 2 processes and 3 or 4 resources does not occur. When enough resources are available for all processes, no resource is shared among processes, but each process uses its own resource. This occurs only when testing with 2 processes and 2 resources or 4 processes and 4 resources.

Because SRP combined with transactions claims all resources simultaneously at the beginning of each period and releases them all simultaneously when the work for that period is done, and because the lowest floor D_R of all resources used by a transaction is taken as that transaction's pre-emption deadline Δ_i, it is believed that testing with one resource per process is as good as testing with more than one resource per process.

The periods, relative deadlines, and programmed resource usage times used for the test processes can be found in two tables. Table 6.1 uses the unit μs for all numbers, table 6.2 uses the unit clock ticks[6]. When testing with n tasks, where $n < 10$, only the first n task parameters are used. The programmed resource usage is depicted as an algorithm in figure 6.6.

Because the tests were meant to measure the scheduling overhead and the influence of the scheduler release interval length the programmed resource usage times have been kept relatively small. The periods have been kept regular, because this ensures that many processes will be released at once in certain time intervals, which means the scheduler has to do more work and is thus tested under more load than when irregular periods would have been used. The deadlines have been chosen gently, to reduce the chance of an unfeasible task set.

Measurements have been taken by letting the scheduler and resource functions write timing information to the *sched-fifo* and *resource-fifo*. It has been measured that writing to a real-time FIFO takes approximately 10–20 clock ticks on the i486 and approximately 5 clock ticks on the i686 — the i686 is a lot more regular than the i486.

The following section contains the results generated by running the test scheme nine times on each computer system, and discusses those test results.

[6]Note: 1.19318 clock ticks equals 1 μs, so 1 clock tick equals approximately 0.8381 μs.

```
# Walk through several scheduler release interval lengths
for s in {0, 2, 5, 10, 20}
do
    # Walk through several numbers of processes
    for p in {2, 4, 6, 10}
    do
        # Walk through several numbers of resources
        for r in {1... min{p, 4}}
        do
            set scheduler release interval length to s
            in parallel do
                run p tasks using r resources total
                collect data
```

Figure 6.5: Scheduler and resource allocator test scheme

```
# Resource id's run from 0 to number of resources minus 1
let r = 0
# P stands for the number of processes
for t in {1... P}
do
    task t uses resource r
    # R stands for the number of resources
    let r = (r + 1) mod R
```

Figure 6.6: Resource usage of test tasks

	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7	Task 8	Task 9	Task 10
Period	1000	1000	2000	2000	3000	3000	4000	4000	5000	5000
Relative deadline	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
Programmed resource usage time	1	1	1	1	2	2	2	2	3	3

Table 6.1: Test tasks' parameters in μs

	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7	Task 8	Task 9	Task 10
Period	1193	1193	2386	2386	3579	3579	4772	4772	5965	5965
Relative deadline	596	1193	1789	2386	2982	3579	4176	4772	5369	5965
Programmed resource usage time	1	1	1	1	2	2	2	2	3	3

Table 6.2: Test tasks' parameters in clock ticks

	Number of tasks							
	2		4		6		10	
SRIL	Table	Page	Table	Page	Table	Page	Table	Page
0	D.1	40	D.6	41	D.11	42	D.16	43
2	D.2	40	D.7	41	D.12	42	D.17	43
5	D.3	40	D.8	41	D.13	42	D.18	44
10	D.4	40	D.9	41	D.14	42	D.19	44
20	D.5	40	D.10	41	D.15	43	D.20	44

Table 6.3: Test results reference table

6.3 Test results

The test results come from nine test runs. For each run the *average execution time* (AET), the *average time between release and execution* (ATRE), and the *average resource usage time* (ARU) have been determined for each task. This was done by counting for each task the execution time per period, the time between release and actual execution for each period, and the resource usage time per period. Averaging these numbers for each task produced the AET, ATRE, and ARU per task.

Those average results of nine test runs are combined so the AET, ATRE, and ARU of each task can be calculated over nine runs. Then for each task the *average scheduling overhead* (ASO) has been determined by subtracting the ARU from the AET.

The test results are presented in a set of tables. Because there are twenty tables with results, these are placed in appendix D. Table 6.3 contains an overview of which table can be found where. The unit of the numbers in the tables is clock ticks. The rows with *i486* in front of them are the results of the tests with the 80486 class machine, the rows with *i686* in front of them are the results of the tests with the PentiumPro class machine.

Because the mean values of test runs don't say much by themselves, the sample standard deviations of all test results have also been computed, these can be found in appendix E. The sample standard deviations of the ASO have not been determined, as the ASO is the result of an operation on the ARU and the AET.

Pre-emption

Assuming all values are subject to the *Normal distribution* the sample standard deviations have been examined. And on a few occasions these are relatively high, given the assumption of the Normal distribution. One of them was investigated to see where that could be coming from. The test with 10 tasks, 4 resources and a SRIL of 0 clock ticks was selected, where the ARU of tasks 7, 8, and 10 is showing an abnormally large sample standard deviation. The average resource usage times are respectively 53, 48, and 77, with sample standard deviations of 20.80, 17.85, and 31.08 respectively. So the measurements of all nine runs for these tasks have been taken apart, and they are shown in table 6.4.

It is evident something happened. Because debugging info is written into a real-time FIFO twice during the time counted as resource usage, and writing to

Test #	1	2	3	4	5	6	7	8	9
Task 7	37.8	82.2	80.0	38.9	38.9	43.3	81.1	38.9	40.0
Task 8	38.9	78.9	36.7	40.0	40.0	40.0	40.0	80.0	37.8
Task 10	40.0	74.3	40.0	72.9	111.4	104.3	40.0	108.6	102.9

Table 6.4: Individual test results of the ARU in clock ticks of tasks 7, 8, and 10, using 10 tasks, 4 resources and a SRIL of 0, run on the i486

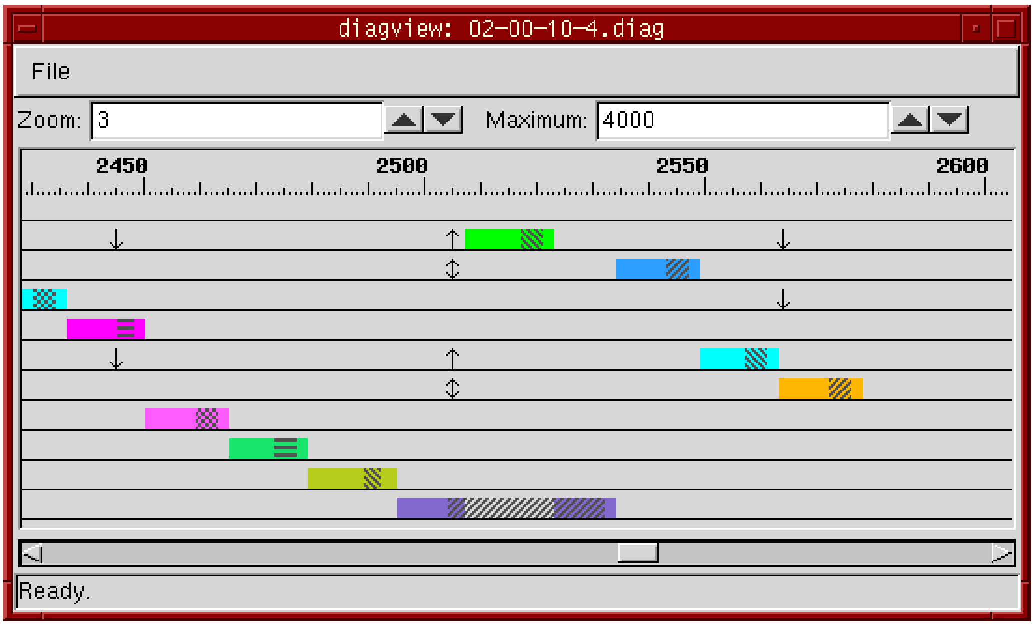


Figure 6.7: Screen-shot of a part of test run 2

a real-time FIFO takes approximately 10–20 clock ticks on the i486, the expected resource usage lies somewhere between 20 and 45 clock ticks for tasks 7, 8, and 10. Several of the numbers from table 6.4 are a lot above the numbers just mentioned.

Let's take a closer look at test run 2, because here all values are above expectations. Figure 6.7 shows a part of test run 2, in units of 10 clock ticks. Task 10 has been pre-empted by task 1, as can clearly be seen. And the tools used simply count the total resource usage time, whether the process is actually running or not. After examining a few others of these cases it showed all these high sample standard deviations were caused by pre-emption and tools not counting correctly. Therefore it was concluded all uncommonly high sample standard deviations are the result of pre-emption, and, therefore, nothing to worry about.

On a few occasions the ARU of the lower priority tasks[7] is much higher than the rest. This is also caused by faulty measurements due to pre-emption. These tasks happened to be pre-empted more often than others and therefore show higher average resource usage times.

[7]The fact that the ARU of task 1 is much higher is dealt with later in this report.

6.4 Interpretation of test results

The scheduler release interval length

What can be said about all these numbers? First it may be noticed that setting the scheduler release interval length has virtually no effect. The SRIL can only affect the average time between release and execution, but the ATRE does not change significantly when the SRIL changes. So the idea about using the SRIL in order to compensate for scheduling overhead doesn't work at all. Therefore it is advised to keep the scheduler release interval length on 0 clock ticks, because then the system simply uses a release time.

The average scheduling overhead

The second number of interest was the average scheduling overhead. The test results show that the ASO increases slightly with the number of tasks. The base ASO is approximately 90 clock ticks for the i486 and approximately 42 clock ticks for the i686. Adding real-time tasks adds about 3 clock ticks per task for the i486 and about 1 clock tick per task for the i686 to the base ASO.

The ASO consists of two real-time FIFO writes, the EDF scheduling and SRP algorithm, and the actual task switch. Subtracting the time for the real-time FIFO writes gives approximately 50–70 clock ticks of ASO for the i486 and approximately 32 clock ticks of ASO for the i686. Due to time constraints — these tests were run in the final phase of this assignment — it was not possible to determine the time needed for an actual task switch. But as performing a task switch between two real-time tasks consists only of saving and restoring some CPU registers and changing the stack, it is believed that a real-time task switch doesn't cost much. The rest of the time is spent performing the EDF and SRP algorithms.

The average time between release and execution

The average time between release and execution increases as the priority of a process decreases. This is to be expected: higher priority processes get to the CPU sooner than lower priority processes, so their ATRE should be lower. Thus the ATRE shows no surprises.

Other interesting results

The test results showed a few peculiarities. The scheduling of the first invocation takes much longer than all other invocation scheduling times. It is believed this is due to caching, but further research is needed to be conclusive about this.

Another peculiarity is the fact that the highest priority task, task 1, always seems to have a higher average resource usage time than the other tasks, while they are expected to have about the same ARU. It is not clear what is the cause of this. A possible cause may be the fact that the current test setup always defines task 1 first, and the current implementation always places tasks defined earlier later in the list of active tasks. As this list is searched from head to tail, task 1 is always seen last during the scanning of this list by the scheduler. But then it would be expected that the average resource usage time is inversely proportional to the process priority, and the other tasks do not show this behaviour.

In order to perform more accurate measurements or to measure the capabilities of the system in a just-feasible situation, where using real-time FIFOs to produce scheduling timing information makes the system unfeasible, other types of measurement are needed. In this context it is needed to measure using external hardware — an oscilloscope, logic analyzer or another computer — in order to get better results.

The tests were tried with 20 tasks, but those failed miserably on the i486, as it just didn't have enough processing power to handle that many tasks with such small periods. The i686 could handle that many tasks, but these weren't included because there would be no reference material with the i486. Of course the periods could have been enlarged, but it was decided — also due to time constraints — that there were enough test results.

Expectations

It is expected that adding more tasks increases the average scheduling overhead in a linear fashion, with a relatively high base ASO and a low increase per task. It is far more difficult to give an estimate of the influence on the ASO when adding resources to the system. Because this depends heavily on the priority of the processes using that new resource. It is believed the influence on the ASO is minimal, but the influence on the feasibility of the system can be enormous, depending on the priorities of the processes using the added resource.

Chapter 7
Conclusions

The first conclusion is that an open source non-real-time system is a very good basis to build a real-time system on. Linux already has two different real-time extensions that can be worked with. Real-Time Linux was chosen as a basis to enhance with EDF and SRP, because it already contained a dynamic scheduler and appeared to have less complex code than KU Real-Time Linux, making it easier to enhance RT-Linux.

It is not too hard to modify RT-Linux, because it has a clear structure. Using a different scheduler is easy, because the scheduler is a single kernel module. It cannot be changed while there are a number of real-time tasks running. But the machine doesn't have to be rebooted to change the real-time scheduler, which is possible when no real-time tasks but the Linux kernel task are running. It therefore is very easy to experiment with different schedulers.

Extending RT-Linux with the Stack Resource protocol is also not very hard. The SRP does not require much extra work, especially SRP in combination with transactions. As SRP guarantees mutual exclusion already — it is only necessary to specify which transaction uses which resource — it is not needed to explicitly use a mutual exclusion semaphore to claim a resource.

Test results show that the current implementation uses quite a lot of scheduling overhead, which consists of a relatively high base scheduling overhead, and only a small addition for each transaction in the system. The current implementation is not optimal, as a single list is used containing all transactions currently in the system, and the scheduler scans this list for suitable transactions several times. Multiple process lists or queues may diminish this overhead. It is believed, but not certain, that optimization should be able to at least halve the scheduling overhead.

The test results also show that the scheduler release interval length has no influence, and can therefore easily be omitted. The average time between release and execution behaves as expected, it is inversely proportional to the priority of a transaction, or, as one may also put it, proportional to the relative deadline of a transaction.

Test results also showed a peculiarity in the very first invocation of every test run: scheduling takes considerably longer the first time. This is believed to be due to caching mechanisms. They also showed that the highest priority transaction always has a higher average resource usage time than the others, the cause to this is suspected to be due to the ordering of the transactions in the list of active transactions, but this is not sure.

It is expected that adding transactions is of linear influence on the average scheduling overhead, and that adding resources is of no influence on the ASO. Adding resources, however, is expected to be of great influence on the schedulability of the system.

The tools developed are very useful to see what actually is going on underneath. These tools can also easily be used to debug and test other real-time operating systems, they are not bound to RT-Linux. All that is needed is a program to convert the scheduling events to a schedule events listing, which is described in appendix C.

The results are believed to be suitable for allowing students to experiment with real-time systems, and they could very well be usable for a real-world problem, provided the implementation is optimized in order to gain more performance.

Chapter 8

Recommendations for further research

Further research is needed to improve performance. Measurements could be made how the system performs without using real-time FIFOs. Another measuring system, perhaps using an oscilloscope, logic analyzer, or another computer even, connected to the test computer, possibly with some debugging code to send signals to the ports the other, external, device is listening on, could be a solution. Because preemption appeared to disrupt the method of measurement used, It is suggested a better measuring method is looked into anyway, be it if that method uses real-time FIFOs or not.

Transactions using multiple resources were not tested. Such tests could be carried out to see how scheduling overhead is affected by tasks using multiple resources. Tests using a mix of tasks using either no, one or several resources could be run as well.

Also it could be tested how the system operates under an overload situation, i.e. the system is presented with more transactions than it can handle. Furthermore several refinements can be introduced to the system, like allowing a transaction to release certain resources before its finish for the period, or implementing the classical SRP protocol, without transactions. The effect of those refinements can then be measured, so it is possible to build an overview of different resource allocation policies and their performance.

It is also suggested to do more investigations to the overhead of performing a schedulability analysis, such as the one presented in [Laan97]. The current implementation provides no support to detect a non-schedulable task set being presented to it. The only problem with a schedulability analysis that can be seen already, is that it is difficult to establish the worst-case run-time C_i of a given task. This depends upon the total number of real-time tasks, which means an accurate estimate is needed of the average scheduling overhead associated with a set of transactions. At least a more accurate estimate than the one mentioned in this report.

Finally it is suggested that the cause of the peculiarities in the test results — first invocation takes longer and highest priority or first defined task uses more resource time — is determined, and possibly, eliminated.

Appendix A
Abbreviations

This appendix lists all abbreviations used in this report and what they stand for.

Abbreviation	**Meaning**
AET	Average Execution Time
ARU	Average Resource Usage time
ASO	Average Scheduling Overhead
ATRE	Average Time between Release and Execution
CPU	Central Processing Unit
EDF	Earliest Deadline First
FIFO	First-In First-Out
GIF	Graphics Interchange Format
Kurt	KU Real-Time Linux
PNG	Portable Network Graphics
RR	Round-Robin
RT-Linux	Real-Time Linux
SRIL	Scheduler Release Interval Length
SRP	Stack Resource protocol
XPM	X PixMap

Appendix B
Bibliography

[Bake91] T.P. Baker, Stackbased Scheduling of real-time processes, *The journal of real-time systems*, Vol. 2, pp. 67-99, 1991

[Bara97a] M. Barabanov and V. Yodaiken, Introducing Real-Time Linux, *Linux Journal* **34**, pp. 19–23, 1997

[Bara97b] M. Barabanov, *A Linux-based Real-Time Operating System*, Master's thesis, New Mexico Institute of Mining and Technology, 1997

[Bara98] M. Barabanov et al., *Real-Time Linux*, `http://rtlinux.cs.nmt.edu/~rtlinux/`, 1998

[IEEE96] Portable Operating System Interface (POSIX), [IEEE/ANSI Std. 1003.1, 1996 Edition], The Institute of Electrical and Electronics Engineers, New York, 1996

[Jans98] P.G. Jansen, H. Scholten, and R. Laan, Flexible Scheduling in Multimedia Kernels: an Overview, to be published in *Proceedings of International Conference on Multimedia & Telecommunications management*, 1998

[Laan97] R. Laan, *Schedulability tests for EDF and DM with Blocking*, Master's thesis, University of Twente, 1997

[Liu73] C.L. Liu and J.W. Layland, Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, *Journal of the ACM*, **20**, 1, pp. 40–61, 1973

[Ripo97a] I. Ripoll, *Crono*, `http://bernia.disca.upv.es/~iripoll/rt-linux/graphic/crono/`, 1997

[Ripo97b] I. Ripoll, *EDF scheduler for RT-Linux*, `http://bernia.disca.upv.es/~iripoll/rt-linux/graphic/`, 1997

[Srin98] B. Srinivasan et al., *KURT: The KU Real-Time Linux*, `http://hegel.ittc.ukans.edu/projects/kurt/`, 1998

[Yoda95] V. Yodaiken and M. Barabanov, A Real-Time Linux, *Technical report*, New Mexico Institute of Technology, 1995

Appendix C

Scheduler events listing grammar

skip

{TAB, LF, SPACE}

comment

 # until LF

keywords

`:BODY`	=	BODY_t
`ACTI`	=	ACTI_t
`DEADLINE`	=	DEADLINE_t
`END`	=	END_t
`EXEC`	=	EXEC_t
`REDEF`	=	REDEF_t
`RESOURCE`	=	RESOURCE_t
`RSCL`	=	RSCL_t
`RSFR`	=	RSFR_t
`TASK`	=	TASK_t

grammar

start	:	*Chronogram*
Chronogram	:	*Header* BODY_t *Body*
Header	:	*Header Header_line*
	|	ϵ
Header_line	:	TASK_t *Decnum Decnum*
	|	RESOURCE_t *Decnum Decnum*
Body	:	*Body Body_line*
	|	ϵ
Body_line	:	ACTI_t *Decnum Decnum Decnum*
	|	END_t *Decnum Decnum*
	|	EXEC_t *Decnum Decnum Decnum*
	|	RSCL_t *Decnum Decnum Decnum*
	|	RSFR_t *Decnum Decnum Decnum*
	|	REDEF_t DEADLINE_t *Decnum Decnum Decnum*
Decnum	:	*Decnum1 Digit*
Decnum1	:	*Decnum1 Digit*
	|	ϵ
Digit	:	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Appendix D

Test results

The unit of the numbers in the tables is clock ticks, 1.19318 clock ticks equals 1 μs, so 1 clock tick equals approximately 0.8381 μs.

CPU &	task #	1 resource				2 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	148	22	35	113	158	22	39	119
	2	114	175	25	89	122	185	28	93
i686	1	57	14	13	44	58	14	14	44
	2	54	73	13	42	56	74	13	44

Table D.1: Results with a SRIL of 0 clock ticks and 2 real-time tasks

CPU &	task #	1 resource				2 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	147	21	35	112	159	22	41	118
	2	116	174	26	90	121	186	28	93
i686	1	57	14	13	44	58	14	14	44
	2	55	73	13	42	56	74	13	44

Table D.2: Results with a SRIL of 2 clock ticks and 2 real-time tasks

CPU &	task #	1 resource				2 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	150	21	35	116	156	21	39	117
	2	116	177	26	90	121	182	29	93
i686	1	57	14	13	44	58	14	13	45
	2	55	73	12	43	56	74	13	43

Table D.3: Results with a SRIL of 5 clock ticks and 2 real-time tasks

CPU &	task #	1 resource				2 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	147	22	35	112	157	21	38	119
	2	113	174	26	87	122	184	29	93
i686	1	57	14	13	44	59	14	14	45
	2	55	73	12	43	56	75	13	44

Table D.4: Results with a SRIL of 10 clock ticks and 2 real-time tasks

CPU &	task #	1 resource				2 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	149	22	37	113	153	21	38	115
	2	115	176	26	89	122	180	29	93
i686	1	57	14	12	45	59	14	14	45
	2	54	73	12	42	57	75	12	44

Table D.5: Results with a SRIL of 20 clock ticks and 2 real-time tasks

CPU &	task #	1 resource				2 resources				3 resources				4 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	152	22	35	117	158	21	39	119	161	21	41	120	167	21	44	123
	2	119	179	26	93	126	185	28	98	132	187	32	100	140	194	35	105
	3	110	298	25	85	118	312	28	89	123	322	31	91	135	335	36	98
	4	115	434	25	90	124	456	28	95	129	472	31	98	140	499	35	105
i686	1	58	14	13	45	60	14	14	45	61	14	15	46	63	14	15	48
	2	56	74	12	44	57	76	12	45	59	78	13	45	60	79	14	46
	3	54	127	12	42	54	131	13	40	57	133	14	43	58	135	14	44
	4	57	192	13	44	59	196	12	47	60	202	13	47	62	205	14	48

Table D.6: Results with a SRIL of 0 clock ticks and 4 real-time tasks

CPU &	task #	1 resource				2 resources				3 resources				4 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	147	21	34	113	155	22	37	118	160	21	41	119	165	21	43	122
	2	118	174	25	93	124	182	28	96	132	187	32	100	138	191	35	103
	3	110	293	25	84	116	308	28	88	123	323	31	92	132	330	35	97
	4	116	427	25	91	123	450	28	95	129	474	31	99	140	491	35	105
i686	1	59	14	13	46	60	14	14	46	62	14	14	47	64	14	15	49
	2	56	75	13	43	57	76	13	45	59	78	13	45	60	80	15	45
	3	55	128	12	43	54	130	13	42	58	133	14	43	57	137	14	43
	4	58	194	12	46	58	196	13	45	60	202	13	47	62	206	13	48

Table D.7: Results with a SRIL of 2 clock ticks and 4 real-time tasks

CPU &	task #	1 resource				2 resources				3 resources				4 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	153	21	35	118	160	21	40	120	162	21	40	122	166	21	44	123
	2	120	180	27	93	127	187	29	97	132	189	32	100	138	193	35	103
	3	110	301	25	85	117	317	29	89	123	324	32	91	131	331	36	95
	4	117	437	25	92	123	461	27	96	129	474	31	98	139	491	35	104
i686	1	58	14	14	44	60	14	13	47	62	14	15	47	63	14	15	48
	2	56	74	12	44	57	77	13	44	58	78	14	44	60	80	14	46
	3	54	127	13	41	54	131	12	42	57	133	14	43	58	136	15	43
	4	58	191	13	45	58	197	12	45	61	201	13	48	62	206	14	48

Table D.8: Results with a SRIL of 5 clock ticks and 4 real-time tasks

CPU &	task #	1 resource				2 resources				3 resources				4 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	151	21	35	116	154	21	37	117	163	21	42	121	168	21	43	125
	2	118	177	25	92	123	181	29	95	133	189	32	101	138	194	35	103
	3	110	296	25	85	115	305	28	87	123	324	31	92	132	334	35	97
	4	116	431	26	91	124	446	28	97	129	475	31	98	139	495	36	103
i686	1	57	14	13	44	61	14	14	47	63	14	15	48	63	14	15	48
	2	56	74	12	44	57	77	13	44	59	79	13	46	59	80	14	45
	3	53	126	12	41	55	131	12	43	57	134	15	42	57	136	14	43
	4	58	190	13	46	58	198	12	46	61	203	13	47	62	205	14	49

Table D.9: Results with a SRIL of 10 clock ticks and 4 real-time tasks

CPU &	task #	1 resource				2 resources				3 resources				4 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	152	21	35	117	156	21	37	119	161	21	41	120	171	21	47	124
	2	118	178	26	92	125	182	28	97	132	188	32	100	138	198	35	103
	3	109	297	26	83	117	306	28	88	124	323	31	93	132	337	36	96
	4	116	431	24	91	124	450	28	96	130	475	32	98	139	499	35	103
i686	1	58	14	13	45	59	14	14	45	62	14	15	47	63	14	15	48
	2	56	74	12	44	58	76	13	46	59	79	14	45	60	80	14	46
	3	54	128	13	41	55	131	12	42	57	135	14	43	57	136	14	44
	4	57	193	12	45	59	197	12	46	60	204	13	47	62	206	14	48

Table D.10: Results with a SRIL of 20 clock ticks and 4 real-time tasks

CPU &		1 resource				2 resources				3 resources				4 resources			
task #		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	150	21	36	115	156	21	37	119	161	21	40	121	168	20	43	125
	2	121	177	26	95	129	182	29	100	135	187	32	102	143	195	35	108
	3	112	299	25	87	120	314	28	91	126	323	31	94	134	339	35	99
	4	118	437	24	93	127	461	28	99	131	476	31	100	142	503	35	107
	5	123	450	26	98	132	476	30	102	140	494	34	106	147	520	36	111
	6	123	574	27	96	134	608	29	104	137	634	33	104	149	667	36	114
i686	1	59	14	12	46	60	14	14	46	63	14	15	48	64	14	15	49
	2	56	75	12	44	58	76	13	45	60	79	13	47	60	80	13	47
	3	54	128	12	42	56	131	13	43	58	135	14	44	59	137	13	45
	4	57	193	12	45	59	198	12	47	60	205	13	47	63	208	14	49
	5	60	199	14	46	62	205	13	50	65	211	14	52	66	216	16	50
	6	61	259	12	49	63	267	14	49	65	277	15	50	65	282	15	50

Table D.11: Results with a SRIL of 0 clock ticks and 6 real-time tasks

CPU &		1 resource				2 resources				3 resources				4 resources			
task #		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	149	21	33	115	158	20	38	119	160	20	40	120	165	20	43	122
	2	121	175	26	95	129	183	29	101	134	185	31	102	141	191	34	107
	3	113	299	26	87	120	315	29	91	126	322	31	95	133	334	35	99
	4	119	437	25	94	127	462	28	99	131	477	31	100	141	496	35	106
	5	124	454	26	98	133	476	30	103	140	492	33	106	147	514	36	111
	6	122	578	26	96	133	609	30	103	139	631	34	105	147	661	37	111
i686	1	59	14	13	46	59	14	13	46	62	14	15	47	64	14	15	49
	2	56	75	12	44	58	76	13	45	60	79	13	46	61	80	13	48
	3	54	128	12	42	55	130	12	43	58	135	14	44	59	137	13	46
	4	58	194	12	46	59	197	12	47	61	205	13	47	63	208	14	49
	5	60	200	13	47	62	204	14	48	65	212	15	51	66	216	15	51
	6	60	261	13	47	62	266	15	47	65	277	15	50	65	282	14	51

Table D.12: Results with a SRIL of 2 clock ticks and 6 real-time tasks

CPU &		1 resource				2 resources				3 resources				4 resources			
task #		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	153	20	35	118	157	21	36	120	161	21	42	119	169	21	44	125
	2	122	179	26	95	129	183	29	100	134	187	32	102	141	196	35	106
	3	113	303	25	88	120	313	28	91	126	323	32	94	134	340	35	99
	4	119	442	25	93	126	460	28	98	131	477	31	100	142	503	35	106
	5	124	455	26	97	131	477	29	102	139	495	33	106	147	518	36	111
	6	123	579	26	97	133	608	30	103	138	634	33	104	147	664	36	111
i686	1	59	14	13	46	60	14	13	46	63	14	14	49	64	14	15	49
	2	56	75	12	44	58	76	13	44	60	79	13	47	61	80	14	47
	3	54	128	12	42	55	130	13	42	58	136	13	45	58	136	13	45
	4	58	193	12	46	60	196	12	48	61	206	13	48	63	207	14	49
	5	61	200	13	47	62	204	14	48	65	211	14	51	67	216	15	51
	6	60	261	14	46	61	266	15	47	65	276	14	50	65	283	14	51

Table D.13: Results with a SRIL of 5 clock ticks and 6 real-time tasks

CPU &		1 resource				2 resources				3 resources				4 resources			
task #		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	152	21	35	117	158	21	38	121	163	21	41	122	167	20	44	124
	2	123	178	26	97	129	184	29	101	135	190	32	103	142	193	36	106
	3	112	303	24	88	119	317	28	91	126	326	31	94	135	337	36	99
	4	118	441	25	93	128	463	28	100	132	481	31	101	141	502	36	106
	5	124	458	26	98	132	479	30	102	140	500	33	107	146	520	36	109
	6	122	582	26	96	133	610	30	104	138	640	33	105	146	666	36	109
i686	1	59	14	13	46	60	14	13	47	63	14	14	49	63	14	15	48
	2	55	75	12	44	58	77	13	45	60	80	13	46	61	80	13	48
	3	55	127	12	43	56	131	12	44	59	136	14	45	59	137	13	46
	4	58	193	12	46	60	198	12	47	60	207	14	46	63	208	14	49
	5	60	199	13	47	62	206	14	49	65	213	15	50	66	216	15	51
	6	61	259	13	47	62	268	15	48	64	277	15	50	65	282	15	50

Table D.14: Results with a SRIL of 10 clock ticks and 6 real-time tasks

CPU &	task #	1 resource				2 resources				3 resources				4 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	151	21	35	117	158	21	37	121	163	20	40	123	167	20	44	124
	2	120	177	26	94	129	184	28	100	136	189	32	104	142	193	35	107
	3	113	299	24	89	118	317	28	90	126	327	31	95	134	340	35	100
	4	118	438	25	94	126	463	28	98	132	481	31	101	141	504	35	106
	5	123	452	26	96	131	476	30	101	140	498	32	107	148	516	37	111
	6	123	575	27	96	133	607	29	104	138	633	33	105	146	664	36	110
i686	1	59	14	13	46	60	14	13	47	63	14	15	48	64	14	15	48
	2	56	75	12	44	57	76	13	45	59	79	13	46	61	80	14	47
	3	54	128	12	42	55	130	12	43	58	135	14	44	59	137	14	45
	4	58	193	12	46	60	197	13	47	60	206	13	47	64	208	14	50
	5	60	200	13	46	62	204	13	48	65	212	15	49	67	217	16	51
	6	61	260	13	48	62	266	14	48	64	277	15	49	65	284	16	50

Table D.15: Results with a SRIL of 20 clock ticks and 6 real-time tasks

CPU &	task #	1 resource				2 resources				3 resources				4 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	153	22	33	120	156	21	36	120	164	23	40	125	172	22	43	129
	2	130	180	27	103	139	183	31	108	149	194	35	114	153	207	38	115
	3	124	318	27	97	131	328	31	100	141	348	33	107	146	360	37	109
	4	132	470	27	105	140	488	30	110	146	519	33	112	155	538	37	118
	5	136	478	27	109	143	501	32	111	156	528	35	120	161	559	39	122
	6	135	614	28	107	146	644	32	114	154	683	36	118	162	721	39	123
	7	141	742	28	113	150	782	32	118	157	823	34	123	176	864	53	122
	8	138	883	28	110	150	960	32	119	164	1015	38	126	185	1076	48	137
	9	152	789	29	123	156	841	32	124	167	887	36	131	174	946	39	135
	10	151	941	29	122	158	998	33	125	168	1054	36	131	204	1120	77	127
i686	1	60	14	13	46	61	14	14	47	64	14	15	49	66	14	15	50
	2	57	76	12	45	59	77	13	47	60	80	13	47	62	82	14	48
	3	55	130	12	43	57	133	12	44	59	137	14	45	59	141	14	44
	4	58	197	12	46	59	202	13	46	61	209	14	47	64	212	14	50
	5	61	202	14	47	62	208	15	47	65	215	15	51	67	220	15	52
	6	61	263	13	47	63	270	13	49	65	280	15	51	66	286	15	51
	7	64	312	13	51	66	320	13	53	67	332	15	52	68	340	15	53
	8	63	376	14	49	65	386	13	52	67	399	14	52	70	408	16	54
	9	67	319	14	53	69	327	15	55	72	338	16	56	72	346	16	56
	10	68	387	14	55	69	397	16	53	71	411	16	55	72	419	16	56

Table D.16: Results with a SRIL of 0 clock ticks and 10 real-time tasks

CPU &	task #	1 resource				2 resources				3 resources				4 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	151	22	34	117	157	21	37	120	170	23	39	131	174	22	43	130
	2	132	179	27	105	140	185	31	108	149	202	34	115	154	207	38	116
	3	122	315	27	95	130	327	31	100	142	356	35	107	148	361	37	110
	4	131	465	26	105	139	486	31	109	147	529	34	114	155	540	38	117
	5	136	477	27	108	144	500	32	112	156	536	35	122	162	563	38	123
	6	135	613	28	108	145	644	32	113	154	693	36	118	162	725	39	123
	7	142	740	29	113	149	778	32	117	160	838	35	125	180	867	49	131
	8	140	882	28	112	154	941	32	122	167	1033	47	119	185	1084	54	131
	9	147	794	29	118	156	840	33	124	169	900	37	132	173	948	39	134
	10	149	941	29	120	161	996	35	126	167	1069	37	131	201	1122	57	145
i686	1	59	14	13	46	62	14	14	48	64	14	15	49	65	14	15	50
	2	57	76	12	45	59	78	13	46	61	80	13	48	61	82	14	48
	3	56	129	12	44	56	134	12	43	59	138	14	45	59	139	14	45
	4	58	197	12	46	60	201	13	47	61	209	14	47	64	211	14	50
	5	61	203	14	47	63	208	13	50	65	215	16	50	67	219	16	51
	6	61	263	14	47	63	271	14	50	65	281	15	50	66	286	15	51
	7	63	312	13	51	65	322	13	53	68	333	15	53	67	339	15	52
	8	63	375	13	51	65	388	13	51	67	401	15	51	69	406	15	54
	9	66	321	15	51	69	327	16	53	72	339	17	55	74	345	16	58
	10	66	387	16	50	69	396	16	54	70	411	16	54	71	419	15	56

Table D.17: Results with a SRIL of 2 clock ticks and 10 real-time tasks

CPU & task #		1 resource				2 resources				3 resources				4 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	153	22	34	119	158	21	37	121	162	22	39	123	174	22	44	130
	2	132	180	27	105	139	185	31	108	146	190	34	112	154	208	38	116
	3	125	317	26	99	132	331	31	101	139	339	34	105	147	360	38	109
	4	131	470	26	105	140	492	31	109	145	508	33	112	155	539	38	117
	5	136	480	28	108	143	501	31	112	153	521	35	118	158	564	39	120
	6	135	616	27	108	145	645	32	113	152	674	36	116	158	712	40	118
	7	140	746	27	113	149	784	32	117	157	811	34	123	176	864	49	127
	8	140	890	27	112	153	951	32	121	157	1002	36	121	183	1076	58	125
	9	149	796	28	121	156	842	33	123	166	868	37	129	174	947	40	134
	10	148	945	30	118	159	997	35	124	164	1033	36	128	202	1120	56	145
i686	1	60	14	13	47	62	14	14	48	63	14	15	48	65	14	15	50
	2	58	76	12	45	59	78	12	47	60	80	14	47	62	81	14	47
	3	55	131	12	43	56	134	12	44	59	137	13	46	59	139	13	46
	4	58	198	12	45	61	202	13	48	62	208	13	50	64	211	14	51
	5	62	203	12	50	63	209	14	49	64	215	16	49	67	220	16	51
	6	60	265	14	46	63	272	14	49	65	280	16	49	66	287	16	50
	7	63	314	14	48	65	322	13	52	67	333	14	52	68	341	15	53
	8	64	377	12	51	65	388	14	51	67	400	14	53	69	409	15	55
	9	66	318	15	51	69	329	16	53	71	338	16	55	74	346	17	57
	10	67	385	15	53	68	397	17	51	71	409	16	55	72	420	17	55

Table D.18: Results with a SRIL of 5 clock ticks and 10 real-time tasks

CPU & task #		1 resource				2 resources				3 resources				4 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	153	22	33	120	156	21	37	119	165	22	40	125	173	22	42	131
	2	131	180	27	104	139	185	32	108	147	194	34	113	154	207	37	116
	3	124	318	26	97	131	328	31	100	141	347	35	106	147	358	38	109
	4	129	469	26	103	140	488	31	108	145	518	33	112	156	536	38	118
	5	135	477	27	108	142	499	32	110	154	527	35	119	159	562	39	120
	6	134	612	27	107	145	641	32	113	153	681	37	117	162	720	38	124
	7	141	740	27	114	148	778	31	116	159	823	34	124	176	863	59	117
	8	138	881	27	112	155	937	33	123	162	1016	36	126	184	1077	63	122
	9	149	796	29	120	157	844	33	124	168	884	36	132	172	949	40	133
	10	151	945	30	121	159	1001	34	125	167	1052	36	131	205	1122	48	156
i686	1	61	14	13	48	61	14	14	48	63	14	15	49	66	14	15	51
	2	57	77	12	44	59	78	13	46	60	80	14	47	62	82	14	48
	3	55	130	12	43	57	134	13	44	59	137	14	45	59	141	14	45
	4	57	197	11	46	60	203	13	47	62	208	13	49	64	213	14	50
	5	61	202	14	48	62	209	13	49	65	216	15	50	67	220	14	52
	6	61	264	13	48	64	272	14	50	65	281	16	49	66	286	14	52
	7	64	313	12	52	65	323	13	52	67	332	16	52	69	340	16	54
	8	63	377	13	49	65	389	13	52	67	400	14	53	68	410	16	53
	9	67	319	15	52	69	329	17	52	72	338	16	57	72	345	16	57
	10	67	386	14	54	69	398	16	54	70	410	16	54	72	417	18	55

Table D.19: Results with a SRIL of 10 clock ticks and 10 real-time tasks

CPU & task #		1 resource				2 resources				3 resources				4 resources			
		AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO	AET	ATRE	ARU	ASO
i486	1	150	21	33	117	157	20	36	121	167	21	41	127	170	22	43	127
	2	131	177	27	104	139	182	31	108	149	195	34	115	153	203	38	115
	3	124	313	26	97	132	330	31	101	141	352	35	107	146	355	38	108
	4	131	464	26	105	140	491	30	109	146	524	34	112	154	532	37	117
	5	136	475	27	109	146	501	32	114	155	530	35	120	159	556	39	120
	6	135	611	28	108	145	646	32	113	156	685	36	120	161	715	39	122
	7	140	737	27	113	149	784	31	118	159	830	34	124	175	854	58	117
	8	141	877	27	113	148	966	31	117	164	1024	41	123	187	1066	71	116
	9	149	791	29	120	158	839	33	125	169	884	37	132	173	942	39	134
	10	148	940	29	119	159	997	33	126	166	1053	38	128	201	1115	49	152
i686	1	60	14	13	47	61	14	14	48	63	14	15	49	65	14	16	50
	2	57	76	12	45	59	78	12	47	61	80	14	47	62	82	14	48
	3	55	130	12	44	56	134	13	43	59	138	13	46	60	140	14	46
	4	58	197	12	45	60	201	13	48	61	210	14	48	64	212	14	50
	5	62	202	13	49	63	209	14	49	66	215	15	51	68	220	16	52
	6	61	264	13	48	63	272	14	50	65	281	15	50	67	288	15	52
	7	63	313	13	51	64	323	14	50	67	333	15	52	68	341	15	52
	8	64	376	13	50	64	387	14	50	67	401	16	52	70	409	14	55
	9	68	319	15	53	69	328	16	52	72	340	17	55	74	347	17	57
	10	68	387	14	53	69	397	16	53	71	412	17	54	72	420	17	56

Table D.20: Results with a SRIL of 20 clock ticks and 10 real-time tasks

Appendix E

Test results with sample standard deviations

The following tables contain the test results without the ASO value, but with the sample standard deviations for all test runs.

Sample standard deviations with (SRIL, # of RT tasks) = (0, 2)

		1 resource						2 resources					
CPU &		AET		ATRE		ARU		AET		ATRE		ARU	
task #		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	148	5.01	22	1.31	35	2.84	158	11.03	22	0.54	39	4.85
	2	114	0.90	175	5.92	25	0.79	122	2.78	185	11.27	28	1.00
i686	1	57	1.20	14	0.28	13	0.70	58	2.02	14	0.49	14	1.14
	2	54	0.72	73	1.26	13	0.88	56	0.83	74	2.39	13	1.03

Sample standard deviations with (SRIL, # of RT tasks) = (2, 2)

		1 resource						2 resources					
CPU &		AET		ATRE		ARU		AET		ATRE		ARU	
task #		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	147	6.52	21	0.88	35	2.29	159	7.33	22	0.71	41	2.73
	2	116	4.71	174	7.34	26	1.53	121	1.65	186	7.69	28	0.57
i686	1	57	1.18	14	0.52	13	0.88	58	1.88	14	0.52	14	1.24
	2	55	1.53	73	1.42	13	0.67	56	1.39	74	1.97	13	1.03

Sample standard deviations with (SRIL, # of RT tasks) = (5, 2)

		1 resource						2 resources					
CPU &		AET		ATRE		ARU		AET		ATRE		ARU	
task #		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	150	9.47	21	0.70	35	1.52	156	7.84	21	0.52	39	3.46
	2	116	3.81	177	9.92	26	0.88	121	1.65	182	8.22	29	0.85
i686	1	57	1.48	14	0.44	13	0.55	58	0.65	14	0.45	13	0.71
	2	55	0.95	73	1.60	12	0.71	56	0.81	74	1.05	13	0.40

Sample standard deviations with (SRIL, # of RT tasks) = (10, 2)

		1 resource						2 resources					
CPU &		AET		ATRE		ARU		AET		ATRE		ARU	
task #		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	147	5.47	22	1.03	35	2.11	157	7.49	21	0.89	38	1.99
	2	113	0.41	174	6.03	26	0.92	122	2.26	184	8.46	29	0.52
i686	1	57	0.64	14	0.42	13	0.80	59	1.09	14	0.32	14	0.75
	2	55	0.59	73	0.79	12	0.48	56	0.92	75	1.27	13	1.13

Sample standard deviations with (SRIL, # of RT tasks) = (20, 2)

		1 resource						2 resources					
CPU &		AET		ATRE		ARU		AET		ATRE		ARU	
task #		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	149	4.09	22	0.99	37	2.69	153	4.88	21	1.02	38	1.00
	2	115	1.90	176	4.42	26	1.09	122	2.27	180	5.79	29	0.94
i686	1	57	0.95	14	0.25	12	0.76	59	2.00	14	0.33	14	1.28
	2	54	0.50	73	1.04	12	0.95	57	0.60	75	1.96	12	0.43

Sample standard deviations with (SRIL, # of RT tasks) = (0, 4)

CPU &	task #	1 resource						2 resources						3 resources						4 resources					
		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU	
		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	152	8.63	22	0.91	35	3.38	158	5.23	21	1.10	39	2.07	161	5.64	21	0.85	41	1.16	167	5.62	21	1.15	44	1.20
	2	119	3.29	179	8.75	26	0.75	126	3.28	185	5.94	28	0.87	132	1.96	187	6.26	32	0.47	140	4.27	194	5.86	35	1.31
	3	110	1.78	298	14.43	25	1.38	118	1.91	312	7.68	28	0.72	123	1.12	322	7.12	31	0.79	135	5.73	335	9.92	36	2.66
	4	115	1.53	434	15.93	25	0.53	124	1.69	456	7.73	28	1.01	129	1.31	472	8.49	31	1.04	140	3.85	499	15.33	35	0.85
i686	1	58	0.66	14	0.37	13	0.77	60	1.84	14	0.55	14	0.66	61	1.43	14	0.53	15	0.83	63	0.88	14	0.52	15	0.90
	2	56	1.55	74	0.86	12	0.49	57	0.89	76	2.24	12	0.57	59	0.78	78	1.57	13	1.05	60	0.75	79	1.05	14	0.98
	3	54	1.44	127	1.14	12	1.19	54	0.85	131	2.58	13	1.30	57	1.75	133	2.71	14	1.48	58	1.82	135	1.62	14	1.21
	4	57	1.40	192	1.33	13	1.90	59	1.12	196	3.00	12	0.95	60	1.86	202	3.31	13	1.70	62	1.56	205	2.90	14	1.39

Sample standard deviations with (SRIL, # of RT tasks) = (2, 4)

CPU &	task #	1 resource						2 resources						3 resources						4 resources					
		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU	
		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	147	6.13	21	0.70	34	2.39	155	3.63	22	0.92	37	1.74	160	3.50	21	0.65	41	1.39	165	1.50	21	0.68	43	0.65
	2	118	2.51	174	6.26	25	0.43	124	1.71	182	4.07	28	0.78	132	2.61	187	3.92	32	0.61	138	1.66	191	1.45	35	0.56
	3	110	2.37	293	8.54	25	1.10	116	1.74	308	7.04	28	1.11	123	1.46	323	6.68	31	0.78	132	2.58	330	2.60	35	1.01
	4	116	3.50	427	11.12	25	0.90	123	1.54	450	7.35	28	1.37	129	2.01	474	8.38	31	0.40	140	1.66	491	3.75	35	0.72
i686	1	59	1.54	14	0.51	13	0.60	60	1.28	14	0.34	14	0.82	62	1.38	14	0.51	15	0.54	64	2.02	14	0.33	15	1.06
	2	56	2.00	75	1.77	13	0.47	57	0.78	76	1.31	13	0.92	59	1.27	78	1.43	13	0.62	60	1.03	80	2.11	15	1.32
	3	55	1.72	128	2.38	12	1.01	54	1.65	130	0.97	13	1.08	58	2.54	133	2.10	14	1.01	57	0.96	137	3.18	14	1.67
	4	58	2.43	194	3.89	12	0.77	58	2.33	196	1.67	13	1.03	60	2.01	202	4.26	13	1.23	62	1.04	206	3.48	13	1.45

Sample standard deviations with (SRIL, # of RT tasks) = (5, 4)

CPU &	task #	1 resource						2 resources						3 resources						4 resources					
		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU	
		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	153	5.76	21	1.29	35	1.73	160	8.92	21	0.71	40	3.94	162	4.81	21	0.60	40	1.35	166	6.59	21	0.79	44	1.14
	2	120	3.20	180	7.04	27	1.35	127	4.01	187	9.07	29	1.34	132	4.40	189	4.77	32	1.15	138	1.02	193	7.24	35	0.79
	3	110	2.44	301	10.86	25	1.42	117	2.12	317	12.22	29	0.70	123	0.86	324	6.70	32	0.72	131	0.92	331	6.67	36	1.05
	4	117	3.44	437	13.76	25	1.13	123	1.24	461	13.84	27	0.97	129	1.24	474	7.01	31	0.53	139	2.41	491	7.70	35	1.43
i686	1	58	0.29	14	0.45	14	0.64	60	2.10	14	0.40	13	0.56	62	0.99	14	0.49	15	1.27	63	1.20	14	0.60	15	0.97
	2	56	1.20	74	0.59	12	0.82	57	0.65	77	1.98	13	0.66	58	0.77	78	1.11	14	1.07	60	1.19	80	1.63	14	0.90
	3	54	1.90	127	1.72	13	1.13	54	0.79	131	3.03	12	1.36	57	1.40	133	1.40	14	1.46	58	1.66	136	2.21	15	1.41
	4	58	1.96	191	1.50	13	1.60	58	1.06	197	2.85	12	0.95	61	1.67	201	1.71	13	1.54	62	1.89	206	1.98	14	1.65

Sample standard deviations with (SRIL, # of RT tasks) = (10, 4)

CPU &	task #	1 resource						2 resources						3 resources						4 resources					
		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU	
		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	151	7.58	21	0.84	35	2.73	154	3.85	21	0.64	37	1.84	163	9.32	21	0.52	42	5.07	168	7.30	21	0.64	43	0.72
	2	118	2.87	177	8.05	25	0.26	123	0.90	181	4.23	29	0.42	133	4.37	189	9.56	32	0.41	138	2.34	194	7.59	35	1.29
	3	110	2.13	296	12.51	25	0.82	115	2.07	305	3.08	28	1.01	123	1.80	324	11.30	31	0.97	132	2.82	334	10.61	35	1.18
	4	116	3.23	431	15.17	26	1.35	124	2.18	446	4.82	28	0.80	129	4.07	475	12.26	31	0.60	139	3.88	495	13.21	36	3.01
i686	1	57	1.22	14	0.52	13	0.58	61	1.28	14	0.35	14	0.96	63	2.12	14	0.47	15	1.23	63	1.10	14	0.48	15	0.92
	2	56	0.60	74	1.17	12	0.63	57	1.01	77	1.18	13	0.47	59	0.99	79	2.01	13	0.86	59	0.68	80	0.89	14	1.09
	3	53	0.80	126	1.57	12	0.91	55	1.00	131	2.08	12	0.95	57	2.59	134	2.35	15	1.31	57	0.72	136	1.44	14	1.11
	4	58	0.90	190	1.90	13	1.00	58	1.01	198	2.77	12	1.14	60	1.72	203	4.34	13	1.31	62	0.58	205	1.11	14	0.74

Sample standard deviations with (SRIL, # of RT tasks) = (20, 4)

CPU &	task #	1 resource						2 resources						3 resources						4 resources					
		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU	
		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	152	6.82	21	0.46	35	2.04	156	7.06	21	0.74	37	1.94	161	4.51	21	0.96	41	1.21	171	12.85	21	0.56	47	6.35
	2	118	2.49	178	7.21	26	1.16	125	3.78	182	7.44	28	0.62	132	3.21	188	5.03	32	1.06	138	2.93	198	12.98	35	1.73
	3	109	2.65	297	11.32	26	1.46	117	2.38	306	12.61	28	1.06	124	2.16	323	8.78	31	0.98	132	2.29	337	15.73	36	1.28
	4	116	2.91	431	13.75	24	1.36	124	1.85	450	13.00	28	0.60	130	2.24	475	11.23	32	1.18	139	0.71	499	17.21	35	1.19
i686	1	58	1.33	14	0.37	13	0.91	59	0.83	14	0.35	14	0.54	62	1.10	14	0.62	15	1.16	63	0.94	14	0.34	15	0.95
	2	56	0.94	74	1.17	12	0.71	58	1.42	76	0.89	13	0.83	59	1.01	79	1.25	14	1.32	60	0.99	80	0.92	14	1.07
	3	54	1.76	128	2.61	13	1.32	55	2.04	131	2.20	12	0.84	57	0.86	135	2.50	14	0.83	57	1.26	136	0.96	14	0.93
	4	57	1.49	193	1.75	12	1.24	59	1.19	198	3.21	12	1.02	60	0.63	204	2.89	13	1.23	62	1.86	206	1.14	14	1.53

Sample standard deviations with (SRIL, # of RT tasks) = (0, 6)

CPU &	task #	1 resource						2 resources						3 resources						4 resources					
		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU	
		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	150	3.72	21	0.67	36	3.21	156	3.74	21	0.90	37	2.35	161	5.00	21	1.21	40	2.36	168	6.20	20	1.15	43	1.50
	2	121	1.77	177	3.84	26	0.92	129	2.91	182	4.29	29	1.55	135	1.25	187	6.14	32	0.86	143	3.82	195	7.34	35	0.67
	3	112	0.89	299	4.51	25	1.31	120	2.55	314	5.27	28	1.32	126	1.16	323	6.72	31	0.63	134	2.03	339	10.12	35	1.26
	4	118	0.70	437	5.12	24	1.18	127	1.76	461	6.84	28	1.24	131	1.24	476	7.01	31	1.01	142	3.29	503	12.43	35	1.31
	5	123	2.05	450	5.29	26	1.01	132	3.00	476	8.39	30	0.48	140	2.31	494	8.09	34	1.39	147	3.94	520	13.93	36	1.44
	6	123	3.77	574	4.84	27	1.95	134	3.60	608	10.21	29	1.25	137	1.52	634	8.97	33	0.58	149	3.08	667	17.17	36	1.02
i686	1	59	1.58	14	0.65	12	0.61	60	1.04	14	0.34	14	0.72	63	1.51	14	0.51	15	1.58	64	1.11	14	0.44	15	0.95
	2	56	0.69	75	1.92	12	0.88	58	1.82	76	1.37	13	0.49	60	0.98	79	1.79	13	0.70	60	1.18	80	1.16	13	0.70
	3	54	1.26	128	2.81	12	1.00	56	1.95	131	2.51	13	1.15	58	1.22	135	2.33	14	1.40	59	0.81	137	0.81	13	1.46
	4	57	1.61	193	2.58	12	0.98	59	1.37	198	4.17	12	1.17	60	2.20	205	3.56	13	0.63	63	0.54	208	1.38	14	1.69
	5	60	1.19	199	2.38	14	1.25	62	1.20	205	4.33	13	0.82	65	1.45	211	4.01	14	1.25	66	0.79	216	1.37	16	1.17
	6	61	1.41	259	1.68	12	1.32	63	2.58	267	4.20	14	1.19	65	1.61	277	4.80	15	2.43	65	0.82	282	1.48	15	1.55

Sample standard deviations with (SRIL, # of RT tasks) = (2, 6)

CPU &		1 resource						2 resources						3 resources						4 resources					
		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU	
task #		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	149	3.93	21	1.25	33	0.73	158	4.31	20	0.75	38	2.49	160	4.05	20	0.81	40	1.46	165	2.23	20	0.51	43	0.96
	2	121	1.76	175	5.25	26	0.85	129	4.00	183	4.29	29	1.83	134	0.54	185	4.57	31	0.65	141	1.94	191	2.38	34	0.72
	3	113	2.24	299	6.17	26	1.90	120	2.99	315	8.34	29	1.74	126	1.40	322	5.17	31	0.70	133	1.58	334	5.45	35	1.20
	4	119	2.29	437	8.36	25	1.64	127	2.41	462	10.87	28	0.79	131	2.38	477	4.62	31	0.79	141	1.16	496	5.21	35	1.59
	5	124	2.04	454	7.61	26	1.43	133	3.58	476	11.16	30	1.93	140	2.80	492	6.24	33	1.30	147	1.51	514	2.96	36	1.18
	6	122	1.01	578	7.41	26	1.37	133	3.06	609	14.47	30	0.77	139	1.73	631	8.06	34	1.43	147	2.37	661	2.66	37	1.18
i686	1	59	1.34	14	0.54	13	0.88	59	1.46	14	0.30	13	1.02	62	1.71	14	0.37	15	0.98	64	1.30	14	0.62	15	1.14
	2	56	0.89	75	1.15	12	0.40	58	0.85	76	1.56	13	0.67	60	0.95	79	1.70	13	0.99	61	0.80	80	1.43	13	0.52
	3	54	1.17	128	2.15	12	1.20	55	2.17	130	1.63	12	1.32	58	1.41	135	2.94	14	1.95	59	1.47	137	1.78	13	0.65
	4	58	1.98	194	2.22	12	1.69	59	1.05	197	2.88	12	0.89	61	1.22	205	4.01	13	2.01	63	1.79	208	2.13	14	1.59
	5	60	1.96	200	2.11	13	1.81	62	1.94	204	2.83	14	1.60	65	1.23	212	2.97	15	1.10	66	1.22	216	1.95	15	1.59
	6	60	1.77	261	2.94	13	1.67	62	2.04	266	2.89	15	2.30	65	1.14	277	2.96	15	1.27	65	0.53	282	2.50	14	1.44

Sample standard deviations with (SRIL, # of RT tasks) = (5, 6)

CPU &		1 resource						2 resources						3 resources						4 resources					
		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU	
task #		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	153	10.08	20	0.60	35	4.60	157	4.55	21	0.92	36	0.80	161	3.94	21	0.65	42	2.46	169	3.20	21	0.84	44	1.89
	2	122	2.03	179	10.70	26	1.44	129	4.59	183	5.14	29	0.88	134	0.97	187	4.17	32	0.60	141	1.95	196	3.79	35	0.67
	3	113	3.05	303	11.70	25	1.21	120	2.15	313	10.52	28	1.24	126	2.30	323	6.36	32	0.91	134	2.24	340	4.82	35	0.97
	4	119	1.67	442	14.57	25	1.00	126	1.51	460	10.70	28	1.08	131	1.08	477	6.41	31	0.68	142	1.63	503	7.00	35	1.43
	5	124	2.38	455	13.78	26	1.87	131	1.35	477	14.21	29	0.88	139	2.25	495	6.28	33	1.18	147	0.90	518	9.11	36	0.94
	6	123	2.20	579	13.86	26	2.76	133	1.48	608	14.72	30	0.61	138	2.23	634	6.51	33	1.76	147	1.62	664	8.91	36	1.27
i686	1	59	1.49	14	0.40	13	0.46	60	2.19	14	0.34	13	0.89	63	1.41	14	0.38	14	0.84	64	1.04	14	0.42	15	0.41
	2	56	0.92	75	1.57	12	0.79	58	1.47	76	2.47	13	1.01	60	1.05	79	1.70	13	1.46	61	1.20	80	1.18	14	0.82
	3	54	1.47	128	2.41	12	1.02	55	1.42	130	1.97	13	0.62	58	1.81	136	2.56	13	0.99	58	1.05	136	1.72	13	0.81
	4	58	1.85	193	2.97	12	0.82	60	1.08	196	2.36	12	0.70	61	1.24	206	2.71	13	1.16	63	1.04	207	2.23	14	1.48
	5	61	1.74	200	2.74	13	1.06	62	1.31	204	3.48	14	1.56	65	1.06	211	1.96	14	1.11	67	1.01	216	2.15	15	1.93
	6	60	1.88	261	3.14	14	1.25	61	1.57	266	3.89	15	0.73	65	1.60	276	1.78	14	0.94	65	1.86	283	2.36	14	1.49

Sample standard deviations with (SRIL, # of RT tasks) = (10, 6)

CPU &		1 resource						2 resources						3 resources						4 resources					
		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU	
task #		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	152	6.81	21	0.82	35	3.63	158	5.03	21	0.73	38	1.82	163	7.94	21	0.79	41	4.01	167	4.33	20	0.95	44	1.74
	2	123	4.98	178	7.42	26	1.52	129	3.66	184	5.48	29	0.98	135	1.97	190	7.81	32	1.60	142	3.14	193	4.48	36	2.19
	3	112	1.66	303	13.50	24	1.35	119	1.18	317	9.58	28	0.82	126	3.02	326	9.19	31	0.93	135	3.54	337	8.59	36	1.34
	4	118	1.57	441	14.89	25	1.24	128	3.23	463	10.46	28	1.15	132	2.10	481	9.33	31	1.24	141	2.44	502	11.34	36	2.10
	5	124	1.84	458	14.00	26	1.62	132	2.13	479	11.04	30	0.42	140	1.68	500	11.49	33	1.18	146	2.34	520	8.51	36	1.08
	6	122	1.27	582	14.95	26	1.25	133	2.99	610	11.06	30	0.61	138	2.18	640	12.83	33	1.09	146	1.14	666	10.36	36	1.50
i686	1	59	0.76	14	0.35	13	1.48	60	0.91	14	0.39	13	0.54	63	1.39	14	0.52	14	1.38	63	0.68	14	0.58	15	1.24
	2	55	0.99	75	1.02	12	0.62	58	1.63	77	0.95	13	0.50	60	0.69	80	1.54	13	0.79	61	0.73	80	0.98	13	0.65
	3	55	1.90	127	1.09	12	1.05	56	2.06	131	2.70	12	1.06	59	0.67	136	1.73	14	1.26	59	0.63	137	1.30	13	1.19
	4	58	2.07	193	1.70	12	1.64	60	1.40	198	4.03	12	0.80	60	1.64	207	1.80	14	1.38	63	1.12	208	1.31	14	1.51
	5	60	2.01	199	1.48	13	0.92	62	1.27	206	4.17	14	2.07	65	1.38	213	2.34	15	1.58	66	1.31	216	1.44	15	1.78
	6	60	1.36	259	1.95	13	1.31	62	2.18	268	4.56	15	1.25	64	1.10	277	2.67	15	1.45	65	0.44	282	1.82	15	1.44

Sample standard deviations with (SRIL, # of RT tasks) = (20, 6)

CPU &	task #	1 resource AET $\overline{x}$	AET σ	ATRE $\overline{x}$	ATRE σ	ARU $\overline{x}$	ARU σ	2 resources AET $\overline{x}$	AET σ	ATRE $\overline{x}$	ATRE σ	ARU $\overline{x}$	ARU σ	3 resources AET $\overline{x}$	AET σ	ATRE $\overline{x}$	ATRE σ	ARU $\overline{x}$	ARU σ	4 resources AET $\overline{x}$	AET σ	ATRE $\overline{x}$	ATRE σ	ARU $\overline{x}$	ARU σ
i486	1	151	6.22	21	0.36	35	2.78	158	6.56	21	0.72	37	1.09	163	7.47	20	0.96	40	2.08	167	3.17	20	0.33	44	1.29
	2	120	1.73	177	6.30	26	1.27	129	3.48	184	6.36	28	0.45	136	4.11	189	7.59	32	0.68	142	3.39	193	3.43	35	1.20
	3	113	2.55	299	7.32	24	2.12	118	1.04	317	9.74	28	0.93	126	2.05	327	11.38	31	0.88	134	1.29	340	4.52	35	0.87
	4	118	3.27	438	8.17	25	1.62	126	1.57	463	10.39	28	0.95	132	4.20	481	13.32	31	0.82	141	2.17	504	5.36	35	1.51
	5	122	0.87	452	9.59	26	1.37	131	2.17	476	13.86	30	0.73	140	4.50	498	17.84	32	1.08	148	2.93	516	6.25	37	1.49
	6	123	3.69	575	9.76	27	1.27	133	3.25	607	14.05	29	1.15	138	3.85	633	12.76	33	0.85	146	2.09	664	6.77	36	1.57
i686	1	59	0.82	14	0.44	13	0.40	60	1.06	14	0.53	13	0.65	63	1.46	14	0.44	15	0.77	64	1.71	14	0.45	15	1.13
	2	56	1.03	75	1.12	12	0.42	57	0.72	76	1.29	13	0.77	59	0.74	79	1.72	13	0.85	61	0.95	80	2.08	14	0.97
	3	54	1.10	128	1.71	12	1.34	55	0.84	130	2.14	12	1.03	58	1.00	135	3.43	14	1.45	59	1.55	137	2.94	14	0.79
	4	58	1.82	193	2.06	12	1.35	60	1.96	197	2.41	13	1.06	60	1.36	206	2.85	13	1.56	64	1.99	208	4.45	14	1.10
	5	60	0.91	200	1.56	13	1.13	62	1.05	204	2.38	13	1.13	65	0.94	212	2.94	15	0.94	67	1.63	217	3.61	16	1.51
	6	61	1.01	260	1.81	13	1.45	62	0.83	266	2.65	14	0.89	64	1.46	277	2.99	15	2.04	65	1.19	284	5.05	16	1.02

Sample standard deviations with (SRIL, # of RT tasks) = (0, 10)

CPU &	task #	1 resource AET $\overline{x}$	AET σ	ATRE $\overline{x}$	ATRE σ	ARU $\overline{x}$	ARU σ	2 resources AET $\overline{x}$	AET σ	ATRE $\overline{x}$	ATRE σ	ARU $\overline{x}$	ARU σ	3 resources AET $\overline{x}$	AET σ	ATRE $\overline{x}$	ATRE σ	ARU $\overline{x}$	ARU σ	4 resources AET $\overline{x}$	AET σ	ATRE $\overline{x}$	ATRE σ	ARU $\overline{x}$	ARU σ
i486	1	153	4.00	22	0.67	33	1.33	156	3.38	21	0.91	36	1.38	164	4.64	23	1.36	40	2.35	172	4.52	22	0.90	43	2.26
	2	130	1.80	180	4.22	27	1.07	139	1.67	183	3.89	31	0.75	149	6.34	194	7.05	35	0.55	153	2.30	207	5.17	38	0.95
	3	124	3.29	318	6.29	27	2.05	131	2.30	328	3.81	31	0.63	141	4.24	348	13.49	33	0.75	146	2.35	360	7.83	37	1.00
	4	132	4.09	470	6.24	27	0.86	140	2.53	488	3.96	30	0.42	146	2.73	519	14.25	33	0.92	155	3.63	538	10.30	37	0.95
	5	136	3.24	478	5.32	27	1.78	143	0.54	501	4.86	32	0.85	156	4.30	528	11.43	35	1.46	161	3.33	559	11.29	39	1.63
	6	135	3.02	614	7.79	28	1.32	146	1.72	644	4.83	32	1.97	154	3.97	683	13.44	36	1.53	162	2.45	721	13.10	39	1.28
	7	141	1.98	742	7.71	28	0.73	150	2.43	782	5.24	32	1.56	157	5.26	823	18.64	34	2.03	176	2.35	864	16.70	53	20.80
	8	138	2.18	883	7.43	28	1.46	150	3.90	960	15.04	32	1.36	164	11.22	1015	19.14	38	10.13	185	4.61	1076	18.66	48	17.85
	9	152	5.92	789	8.12	29	1.02	156	2.98	841	8.91	32	2.04	167	4.54	887	24.17	36	1.88	174	3.65	946	17.32	39	1.04
	10	151	4.09	941	11.63	29	1.20	158	3.16	998	9.17	33	2.19	168	7.25	1054	23.91	36	1.45	204	3.54	1120	18.46	77	31.08
i686	1	60	0.96	14	0.34	13	0.94	61	0.97	14	0.47	14	1.21	64	1.09	14	0.52	15	1.26	66	1.81	14	0.44	15	0.91
	2	57	0.97	76	1.09	12	0.56	59	1.65	77	1.17	13	0.68	60	0.61	80	1.02	13	0.71	62	1.20	82	1.80	14	1.01
	3	55	1.40	130	1.85	12	0.78	57	1.22	133	2.61	12	0.91	59	0.61	137	1.92	14	0.82	59	1.06	141	2.96	14	1.56
	4	58	1.09	197	2.10	12	1.35	59	1.28	202	3.47	13	1.43	61	1.54	209	2.43	14	1.49	64	0.55	212	2.28	14	1.32
	5	61	1.71	202	2.40	14	1.31	62	1.90	208	2.31	15	1.65	65	0.94	215	1.72	15	1.32	67	1.11	220	1.01	15	1.81
	6	61	1.93	263	1.45	13	1.74	63	1.86	270	2.39	13	0.88	65	1.23	280	1.91	15	0.85	66	1.25	286	1.40	15	1.73
	7	64	1.16	312	3.25	13	1.59	66	1.43	320	3.53	13	1.68	67	1.52	332	2.49	15	1.29	68	0.90	340	3.02	15	1.58
	8	63	1.63	376	3.15	14	1.49	65	1.85	386	2.85	13	1.70	67	0.87	399	2.80	14	1.47	70	2.23	408	3.43	16	1.83
	9	67	1.42	319	3.01	14	1.65	69	1.45	327	2.75	15	2.54	72	2.38	338	3.32	16	2.44	72	1.86	346	1.75	16	1.60
	10	68	3.12	387	3.43	14	2.28	69	1.45	397	2.21	16	1.70	71	2.88	411	3.45	16	2.26	72	1.88	419	2.72	16	1.57

Sample standard deviations with (SRIL, # of RT tasks) = (2, 10)

CPU &	task #	1 resource						2 resources						3 resources						4 resources					
		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU	
		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	151	3.85	22	1.05	34	2.23	157	3.49	21	0.78	37	1.36	170	7.63	23	1.65	39	1.20	174	4.31	22	0.69	43	1.91
	2	132	2.72	179	4.93	27	0.84	140	1.85	185	2.35	31	0.57	149	2.85	202	12.01	34	0.84	154	2.90	207	5.98	38	1.09
	3	122	2.22	315	4.64	27	1.22	130	0.73	327	5.35	31	0.82	142	2.93	356	13.79	35	2.45	148	3.14	361	6.56	37	1.23
	4	131	4.47	465	7.00	26	1.41	139	0.95	486	5.69	31	0.47	147	3.80	529	17.17	34	0.71	155	3.67	540	10.02	38	1.64
	5	136	3.50	477	9.91	27	1.77	144	2.37	500	7.10	32	1.16	156	4.03	536	14.24	35	1.86	162	3.99	563	14.84	38	1.33
	6	135	3.03	613	11.66	28	0.87	145	1.38	644	7.09	32	1.26	154	3.62	693	17.80	36	0.63	162	4.09	725	17.08	39	1.65
	7	142	3.24	740	13.72	29	1.23	149	2.06	778	7.31	32	0.80	160	3.88	838	23.07	35	2.11	180	4.78	867	17.63	49	18.00
	8	140	2.73	882	15.38	28	1.20	154	5.82	941	23.42	32	1.23	167	14.43	1033	26.20	47	18.83	185	9.58	1084	20.95	54	20.71
	9	147	3.41	794	14.04	29	1.61	156	3.44	840	3.25	33	1.87	169	5.14	900	27.71	37	1.87	173	3.43	948	21.02	39	1.26
	10	149	3.73	941	15.83	29	1.04	161	3.03	996	5.12	35	1.23	167	4.57	1069	32.29	37	1.23	201	6.62	1121	23.74	57	26.69
i686	1	59	1.00	14	0.62	13	0.91	62	1.35	14	0.64	14	1.07	64	1.00	14	0.34	15	1.00	65	1.38	14	0.40	15	1.06
	2	57	1.11	76	1.40	12	1.25	59	0.79	78	1.57	13	0.81	61	0.61	80	1.19	13	1.19	61	0.63	82	1.59	14	0.82
	3	56	1.47	129	2.24	12	1.63	56	1.12	134	2.17	12	1.54	59	0.97	138	2.18	14	0.98	59	0.85	139	1.34	14	0.65
	4	58	1.49	197	2.87	12	0.83	60	1.17	201	2.55	13	0.95	61	1.30	209	2.05	14	2.02	64	0.50	211	1.93	14	0.90
	5	61	1.42	203	2.02	14	1.08	63	0.95	208	2.09	13	1.51	65	0.80	215	1.90	16	1.55	67	1.59	219	1.35	16	1.28
	6	61	1.38	263	2.95	14	1.25	63	0.75	271	2.42	14	1.00	65	0.64	281	2.05	15	1.24	66	1.29	286	2.23	15	1.51
	7	63	3.53	312	4.39	13	2.50	65	2.51	322	3.82	13	1.47	68	0.84	333	2.25	15	1.96	67	0.75	339	2.67	15	1.88
	8	63	3.13	375	3.36	13	1.85	65	0.74	388	3.17	13	0.86	67	1.22	401	2.40	15	1.24	69	1.22	406	2.19	15	1.15
	9	66	1.87	321	6.24	15	1.71	69	1.75	327	1.60	16	1.74	72	1.81	339	3.77	17	1.57	74	2.17	345	2.43	16	1.50
	10	66	3.64	387	6.71	16	1.93	69	1.20	396	1.45	16	1.66	70	2.49	411	4.25	16	1.26	71	1.46	419	2.65	15	1.88

Sample standard deviations with (SRIL, # of RT tasks) = (5, 10)

CPU &	task #	1 resource						2 resources						3 resources						4 resources					
		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU	
		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	153	4.37	22	0.91	34	3.93	158	4.19	21	0.95	37	2.86	162	2.00	22	1.34	39	0.75	174	6.82	22	1.22	44	4.29
	2	132	3.75	180	4.59	27	0.61	139	2.07	185	6.12	31	0.65	146	1.15	190	3.03	34	0.94	154	3.72	208	9.40	38	2.16
	3	125	4.08	317	6.15	26	0.98	132	1.49	331	8.40	31	0.70	139	1.97	339	5.39	34	0.91	147	3.01	360	12.54	38	2.41
	4	131	3.87	470	8.06	26	1.21	140	1.24	492	9.97	31	0.66	145	3.38	508	6.89	33	0.97	155	3.35	539	15.12	38	2.49
	5	136	3.02	480	6.70	28	0.87	143	1.65	501	7.99	31	0.85	153	1.70	521	5.89	35	2.09	158	2.25	564	17.86	39	2.74
	6	135	3.30	616	9.29	27	1.15	145	1.73	645	8.80	32	1.26	152	2.38	674	6.10	36	1.02	158	4.14	712	12.94	40	1.11
	7	140	3.96	746	16.90	27	1.49	149	1.30	784	11.04	32	1.12	157	1.89	811	9.68	34	2.18	176	3.54	864	21.06	49	17.77
	8	140	2.43	890	30.26	27	1.39	153	5.04	951	24.17	32	2.48	157	1.66	1002	8.62	36	1.36	183	4.52	1076	24.53	58	21.90
	9	149	4.08	796	16.48	28	1.81	156	2.22	842	9.93	33	1.98	166	3.65	868	4.49	37	1.77	174	5.76	947	21.84	40	0.93
	10	148	3.11	945	19.80	30	0.70	159	2.66	997	10.95	35	2.37	164	3.55	1033	6.21	36	2.25	202	10.41	1120	26.46	56	25.65
i686	1	60	1.63	14	0.40	13	0.92	62	0.65	14	0.49	14	0.93	63	1.22	14	0.51	15	0.62	65	1.13	14	0.41	15	0.82
	2	58	0.49	76	1.80	12	0.90	59	0.78	78	0.76	12	0.66	60	1.57	80	1.55	14	0.90	62	1.06	81	1.31	14	1.03
	3	55	1.13	131	2.68	12	0.68	56	1.56	134	1.52	12	1.35	59	0.95	137	2.32	13	1.87	59	0.95	139	1.78	13	1.17
	4	58	1.18	198	3.09	12	1.12	61	1.62	202	2.09	13	1.20	62	1.22	208	2.84	13	1.08	64	1.22	211	2.50	14	1.57
	5	62	0.95	203	2.98	12	1.47	63	0.95	209	1.27	14	0.84	64	0.56	215	2.72	16	1.39	67	0.98	220	2.36	16	1.34
	6	60	1.10	265	2.70	14	2.11	63	1.71	272	1.41	14	1.11	65	2.08	280	2.83	16	1.44	66	1.30	287	1.70	16	1.09
	7	63	2.07	314	3.31	14	1.69	65	1.59	322	1.97	13	1.62	67	1.04	333	3.25	14	1.27	68	1.10	341	3.31	15	1.70
	8	64	1.77	377	2.95	12	1.07	65	1.95	388	2.52	14	1.66	67	1.08	400	3.59	14	1.77	69	1.42	409	3.60	15	1.73
	9	66	1.62	318	1.66	15	1.23	69	1.11	329	3.09	16	0.93	71	1.82	338	2.76	16	1.61	74	1.40	346	7.20	17	1.51
	10	67	1.45	385	2.86	15	1.71	68	2.11	397	2.91	17	1.39	71	1.21	409	3.12	16	1.59	72	0.80	420	7.82	17	1.51

Sample standard deviations with (SRIL, # of RT tasks) = (10, 10)

CPU &	task #	1 resource						2 resources						3 resources						4 resources					
		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU	
		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	153	2.37	22	0.56	33	2.66	156	3.16	21	0.74	37	1.46	165	6.21	22	1.66	40	1.91	173	5.50	22	0.77	42	0.86
	2	131	1.43	180	2.50	27	0.71	139	1.98	185	2.99	32	0.84	147	2.80	194	9.88	34	1.06	154	2.59	207	7.01	37	1.24
	3	124	2.34	318	4.44	26	0.83	131	2.57	328	7.49	31	1.35	141	4.48	347	10.08	35	1.08	147	4.84	358	8.67	38	1.26
	4	129	1.39	469	6.88	26	0.83	140	0.90	488	7.93	31	0.85	145	2.84	518	13.89	33	1.42	156	3.45	536	13.89	38	2.03
	5	135	1.71	477	4.48	27	1.29	142	1.43	499	7.94	32	0.90	154	5.56	527	16.55	35	1.39	158	2.66	562	13.24	39	1.03
	6	134	1.31	612	5.39	27	1.24	145	1.24	641	8.83	32	1.32	153	3.29	681	21.33	37	1.33	162	1.62	720	15.26	38	0.93
	7	141	1.59	740	8.37	27	1.77	148	0.87	778	7.39	31	0.92	159	6.91	823	23.16	34	1.79	176	4.38	863	19.85	59	22.87
	8	138	2.37	881	7.13	27	2.12	155	3.93	937	21.32	33	1.85	162	10.05	1016	29.95	36	1.51	184	4.51	1077	22.50	63	29.70
	9	149	4.08	796	8.44	29	1.13	157	3.64	844	9.54	33	1.40	168	6.11	884	36.59	36	2.60	172	3.54	949	23.88	40	0.70
	10	151	4.99	945	8.76	30	1.75	159	3.97	1001	9.53	34	2.45	167	5.14	1052	38.50	36	2.34	205	3.81	1122	25.64	48	15.66
i686	1	61	1.30	14	0.67	13	0.45	61	1.65	14	0.33	14	0.87	63	0.70	14	0.35	15	0.69	66	1.48	14	0.30	15	1.17
	2	57	0.54	77	0.83	12	0.77	59	0.53	78	1.82	13	0.61	60	0.73	80	0.68	14	0.54	62	1.00	82	1.43	14	1.44
	3	55	1.54	130	1.31	12	0.80	57	2.56	134	2.74	13	1.63	59	1.04	137	1.41	14	1.34	59	0.53	141	2.55	14	1.47
	4	57	1.14	197	1.97	11	0.88	60	1.69	203	4.29	13	1.40	62	1.27	208	1.40	13	1.73	64	0.23	213	2.68	14	1.43
	5	61	0.91	202	2.00	14	0.83	62	1.75	209	2.74	13	1.46	65	1.25	216	1.31	15	1.58	67	1.08	220	1.67	14	1.92
	6	61	1.19	264	1.84	13	1.18	64	2.06	272	2.20	14	1.56	65	1.39	281	1.85	16	1.33	66	0.98	286	1.99	14	1.80
	7	64	1.75	313	2.75	12	1.46	65	1.74	323	3.35	13	2.21	67	0.97	332	2.18	16	1.95	69	2.43	340	3.75	16	1.46
	8	63	1.92	377	2.64	13	1.78	65	0.98	389	3.26	13	1.72	67	0.83	400	2.46	14	2.24	68	1.21	410	4.04	16	1.39
	9	67	0.84	319	2.44	15	1.61	69	3.55	329	2.40	17	2.51	72	3.11	338	3.58	16	0.99	72	1.68	345	1.20	16	1.50
	10	67	1.88	386	2.51	14	1.42	69	0.85	398	5.48	16	1.30	70	1.63	410	4.69	16	2.26	72	2.05	417	1.67	18	2.26

Sample standard deviations with (SRIL, # of RT tasks) = (20, 10)

CPU &	task #	1 resource						2 resources						3 resources						4 resources					
		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU		AET		ATRE		ARU	
		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
i486	1	150	2.44	21	0.64	33	1.37	157	3.43	20	0.89	36	1.39	167	5.99	21	1.60	41	2.38	170	4.03	22	0.63	43	1.24
	2	131	2.01	177	2.77	27	1.18	139	1.93	182	3.50	31	0.52	149	3.81	195	9.77	34	1.55	153	1.86	203	4.67	38	1.15
	3	124	1.99	313	4.66	26	0.83	132	4.29	330	8.39	31	1.41	141	3.78	352	12.79	35	1.63	146	2.34	355	5.23	38	1.02
	4	131	3.30	464	6.36	26	1.10	140	0.92	491	7.54	30	0.58	146	3.05	524	17.12	34	1.49	154	1.34	532	6.96	37	1.49
	5	136	2.67	475	5.59	27	1.72	146	4.42	501	6.53	32	0.84	155	2.89	530	15.51	35	1.44	159	2.10	556	9.63	39	1.03
	6	135	2.21	611	6.99	28	1.88	145	1.95	646	8.14	32	1.19	156	3.01	685	18.27	36	1.51	161	2.03	715	10.85	39	0.84
	7	140	4.38	737	9.24	27	1.32	149	2.09	784	7.59	31	1.43	159	2.98	830	25.76	34	1.77	175	2.43	854	9.34	58	22.11
	8	141	3.64	878	11.26	27	2.00	148	0.90	966	7.39	31	1.16	164	9.32	1024	29.52	41	16.74	187	5.44	1066	9.86	71	35.05
	9	149	5.01	791	9.53	29	1.44	158	4.08	839	10.34	33	1.33	169	4.81	884	25.37	37	1.87	173	2.89	942	11.62	39	1.20
	10	148	3.44	940	11.41	29	1.04	159	3.54	997	11.73	33	1.87	166	2.75	1053	28.59	38	1.86	201	6.20	1115	12.40	49	24.48
i686	1	60	1.76	14	0.47	13	0.92	61	1.12	14	0.29	14	0.61	63	0.93	14	0.43	15	1.12	65	1.34	14	0.52	16	1.07
	2	57	0.91	76	1.60	12	0.89	59	0.95	78	1.06	12	0.81	61	1.02	80	0.83	14	1.34	62	1.55	82	1.57	14	1.25
	3	55	1.41	130	2.43	12	0.77	56	1.67	134	1.43	13	1.04	59	0.72	138	2.62	13	0.92	60	1.38	140	2.79	14	1.40
	4	58	2.66	197	2.68	12	1.13	60	1.83	201	2.40	13	0.48	61	0.96	210	3.35	14	1.04	64	1.79	212	4.01	14	1.30
	5	62	2.09	202	2.14	13	0.93	63	1.17	209	1.97	14	1.19	66	0.91	215	2.03	15	1.63	68	2.21	220	4.23	16	1.18
	6	61	1.48	264	3.00	13	0.89	63	1.43	272	0.98	14	1.32	65	0.99	281	2.34	15	1.93	67	1.01	288	5.67	15	0.92
	7	63	2.07	313	4.06	13	0.80	64	1.10	323	2.60	14	0.76	67	1.45	333	3.64	15	2.04	68	0.83	341	5.78	15	1.74
	8	64	2.58	376	3.92	13	1.77	64	1.42	387	3.09	14	1.44	67	2.23	401	3.44	16	2.90	70	1.97	409	6.54	14	1.27
	9	68	4.24	319	2.80	15	1.89	69	1.75	328	1.66	16	2.10	72	1.53	340	7.83	17	1.34	74	2.41	347	5.90	17	1.39
	10	68	2.04	387	5.74	14	1.50	69	1.18	397	1.94	16	2.24	71	1.21	412	7.40	17	2.03	72	1.45	420	7.70	17	1.20