Worst-Case Execution Time Analysis for Real-Time Systems

Group d501a
Department of Computer Science
Aalborg University

December 18, 2008
Title:
Worst-Case Execution Time Analysis for Real-Time Systems

Project period:
DAT5,
2nd of September to 21st of December, 2008

Project group:
Computer Science, d501a

Members:
Andreas Engelbrecht Dalsgaard
Mads Christian Olesen
Martin Toft

Supervisors:
Kim Guldstrand Larsen
René Rydhof Hansen

Copies:
6

Pages in the report:
93

Pages including appendices:
99

Abstract:
WCET analysis is important, as the WCET of processes is required for scheduling of Real-Time Systems. Tight WCETs are desirable, since they allow for scheduling of more processes and thus entail more efficient systems. Traditionally, very pessimistic WCETs have been found, as optimisation techniques such as caching and pipelining in the underlying hardware architecture have been mostly ignored.

We present a method for WCET analysis based on model checking, where tight WCETs are achieved by modelling the process and the hardware architecture together with their interactions. In connection with designing the method, we have also devised a number of formalisations for caches with different replacement policies.
## Contents

1 Introduction 7

2 Hard Real-Time Systems 9
   2.1 Process-Based Scheduling .......................... 11
   2.2 Process Interactions .............................. 15
   2.3 Worst-Case Execution Time ......................... 18

3 Static Analysis 21
   3.1 Control Flow Graphs ............................. 22
   3.2 Lattice Theory ................................. 23
   3.3 Abstract Interpretation .......................... 24

4 Model Checking 27
   4.1 Timed Automata ................................ 29
   4.2 Timed Computation Tree Logic ..................... 33
   4.3 TCTL Model Checking ............................. 34
   4.4 Model Checking in UPPAAL ......................... 35

5 Optimisation Techniques of Modern Processors 39
   5.1 Caching ......................................... 39
   5.2 Pipelining ....................................... 48
   5.3 Branch Prediction ................................ 51
   5.4 Out-of-Order Execution ........................... 52
   5.5 Speculative Execution ............................ 54

6 Timing Anomalies 55
   6.1 Formal Definition ............................... 57
   6.2 Methods for Elimination of Anomalies .......... 59

7 Related Work on WCET Analysis 63
   7.1 WCET Analysis using Abstract Interpretation .... 63
   7.2 WCET Analysis using Model Checking .......... 68

8 WCET Analysis using UPPAAL 71
   8.1 Initial UPPAAL Model ............................ 71
   8.2 Separate Program and Abstract Hardware Model 77
   8.3 Program Model Generation ........................ 83
   8.4 pyuppaal ....................................... 84
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Conclusion</td>
<td>87</td>
</tr>
<tr>
<td>10</td>
<td>Future Work</td>
<td>89</td>
</tr>
<tr>
<td>A</td>
<td>Instruction Sequence Model</td>
<td>95</td>
</tr>
<tr>
<td>B</td>
<td>Program Model</td>
<td>97</td>
</tr>
<tr>
<td>C</td>
<td>List of Acronyms</td>
<td>99</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The modern world depends heavily on advanced technology: food production plants, trains, planes, hospital equipment, nuclear reactors and even weapons, all of these inventions can be real-time systems, which are controlled by software. They are real-time, because they contain components in which time plays a significant role. For example, the sampling of different sensors in a plane must happen with precise intervals, and an x-ray machine must guarantee that a patient is only exposed to x-rays in a short, fixed period of time. In such environments, it is clear that bad timing can cause loss of human lives. To be safe and efficient, real-time systems are scheduled according to different scheduling schemes. Common to most scheduling schemes is that the Worst-Case Execution Time (WCET) of each software process running in the real-time system must be known in advance.

Traditionally, the WCET of processes has been estimated by either measuring or using static analysis. Measurement-based methods are simple and need little adaption to work for new hardware architectures. They are, however, likely to underestimate the real WCET. Static analysis deduces different properties by analysing program code without executing it. In static analysis, some level of detail of the program is abstracted away in order to keep the analysis tractable. For example, some static analysis tools only consider the behaviour of individual statements and declarations.

Due to technological advances in the hardware industry, today even small embedded systems have processors with features such as caches, pipelines, branch prediction, out-of-order execution, speculative execution and other optimisation techniques. It is important that methods for estimating WCETs take these techniques into account, as they can produce effects known as timing anomalies. If the method does not account for these features, it may not be sound. Mathematical formalisations of the techniques are required for the them to be correctly reasoned about, but unfortunately the complexity and irregular behaviour of some of the techniques make it very hard to design such formalisations.

The contribution of this project is a model-checking-based method for estimating WCETs that take the underlying hardware architecture into account in an elegant, general way. So far, the method only includes caches, but it is ready to be extended in the future. The method works by modelling the software separate from the hardware, that the software is supposed to be executed on and then using the model checker UPPAAL to predict a WCET estimate for
the execution. In addition to describing the method and the associated examples, we have made a compiler that compiles program code written in a simple, imperative language into complete UPPÅAL models.

The remainder of the report is structured as follows: In Chapter 2 we introduce hard real-time systems, schemes for scheduling them and finally a brief insight into traditional WCET estimation. Chapter 3 describes static analysis, including control flow graphs, lattice theory and abstract interpretation. Chapter 4 studies model checking, both in general and how systems are modelled in UPPÅAL. Chapter 5 gives in-depth descriptions of the optimisation techniques found in modern processors and how some of the techniques can be formalised. Chapter 6 introduces the concept of timing anomalies, which poses a great challenge to WCET analysis. In Chapter 7 we describe related work with regards to WCET analysis — how other authors utilise abstract interpretation and model checking, respectively. Chapter 8 presents our WCET estimation method and the accompanying compiler. In Chapter 10 we discuss possibilities for future work.
Chapter 2

Hard Real-Time Systems

This chapter is based on parts of Chapter 1 and Chapter 13 in [9]. Other sources are referenced where they are used.

Real-Time Systems (RTSs) are systems that must produce output from a given input within a specified time interval. The following definition of a real-time system, borrowed from The Oxford Dictionary of Computing, formulates this property more precisely:

\[
\text{Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.}
\]

Due to e.g. “acceptable timeliness”, the definition is not quite rigid though, and it comes as no surprise that many definitions of real-time systems exists. Acceptable timeliness means that the maximum lag time is dictated by the environment in which the real-time system operates. For example, an embedded missile guidance system requires an answer within a few milliseconds, whereas a computer-controlled car assembly line might be satisfied with getting an answer within a second.

RTSs are classified as either soft or hard. Hard RTSs do not allow deadline misses and consider late process completions useless, whereas soft real-time systems tolerate missed deadlines and provides a reduced service quality accordingly. Only hard real-time systems are explored in this report.

Processes in RTSs can either be periodic or sporadic. A periodic task is a task that is run (“released”) over and over with a given time-period in between. After each release the process must have completed its execution within a certain time-period, its deadline. An example of a periodic task could be a process that must check a sensor reading every 10 milliseconds (the period) and trigger some action within 5 milliseconds (the deadline) if the sensor reading is below a certain threshold. A sporadic task is a task that is released sporadically by the environment. A sporadic task can be released at any time, and must be responded to within a certain deadline.

As the order in which a system’s processes execute is not specified in advance, and may depend on synchronisation primitives, the execution of a concurrent system may exhibit substantial non-determinism. For example, if preemption is
disallowed, six processes can be executed in \(6! = 720\) ways on a single processor. In addition, preemptive behaviour and multiprocessor systems make the number of different execution orders explode to even higher levels. This is usually not a problem, as long as all execution orders produce the same output. However, as the different orders potentially yield different timings, it is not possible to give any real-time guarantees in this setting. To give these guarantees, it is essential to restrict the non-determinism exhibited by a system. This act is known as scheduling, which is a major topic of this chapter.

A scheduling scheme consists of a scheduling algorithm and a method for predicting a system’s worst-case behaviour when it is scheduled according to the scheduling algorithm. The algorithm determines the order in which the system’s resources are used. In this chapter, system resources only include the processor or set of processors executing the processes. As mentioned before, real-time systems have temporal requirements, and the job of the prediction method is to tell whether these requirements are fulfilled when a system is scheduled according to the chosen algorithm.

Scheduling schemes are classified as either static or dynamic. Static schemes do predictions before execution, whereas dynamic schemes perform decisions both before execution and on run-time. In this chapter we only consider preemptive, priority-based schemes. The decisions are therefore distributing and redistributing priorities to processes. At all times the process with the highest priority that needs the processor, is executing.

As it is hard to analyse arbitrary complex, concurrent systems, it is necessary to restrict the considered process model. The following, simple, restricted process model is the foundation for this chapter:

- The system has a fixed number of processes.
- The system’s overhead, such as context-switching, is assumed to have zero cost.
- All processes are periodic and have known periods. The execution start of a process is called its “release”.
- All processes are completely independent of each other.
- Each process must complete before it is next released (i.e. processes have deadlines equal to their periods).
- All processes have fixed WCETs. Where the WCET of a process is the longest of all possible execution times of a process.

With this model it is possible that at some point in time all processes are released simultaneously, putting maximum load on the processor. This phenomenon is known as a critical instant.

For each process \(i\) we know the following:

- The period with which the process is released, denoted by \(T_i\).
- The WCET, denoted by \(C_i\). This concept is defined later on.

Given a set of processes, it is possible to plot an execution of the processes in a time-line diagram. For example, in Figure 2.1 on the facing page a process
set execution is demonstrated. Process \(a\) misses a deadline, as it is not done executing when its deadline sets in at 50 time units.

The simplistic process model and scheduling assumptions that we have presented in this chapter has a number of limitations that should be noted:

- The control flow of the processes is disregarded.
- The cost of context switching is set to 0. This is quite unrealistic, as the cost on some platforms can be quite high.

### 2.1 Process-Based Scheduling

Scheduling schemes that do not preserve the notion of processes in executing a RTS exist, such as the cyclic executive scheme. We will however restrict ourselves to schemes that do preserve the notion of processes, as the cyclic executive scheme is best suited for simple systems containing only periodic tasks.

Depending on the chosen scheme, one or more scheduling attributes decide which process should execute at any one time. If interprocess communication is not taken into account, a process must be in one of three states:

1. Runnable
2. Suspended waiting for a timing event (Mostly periodic processes)
3. Suspended waiting for a non-timing event (Mostly sporadic processes)

As mentioned before, real-time scheduling schemes are either static or dynamic. The classification refers to the way priorities are assigned to processes. In static schemes, priorities are assigned pre-runtime and do not change during execution. Dynamic schemes, on the other hand, assign priorities to processes...
whenever they are released. In this way, a process' priority can change during execution of the system, however, only at certain times. Some dynamic schemes are called adaptive. These schemes extend the traditional dynamic behaviour by not having their priority assignments limited to process releases, i.e. a process' priority may be changed at any time during execution. The adaptive approach is useful if the system at hand easily can become overloaded, as it can respond to overloading quicker than traditional dynamic schemes.

In the following two sections we present the static scheme Fixed-Priority Scheduling (FPS) and the dynamic scheme Earliest Deadline First (EDF).

2.1.1 Fixed-Priority Scheduling

In this scheduling scheme, a fixed, static priority is computed pre-runtime for each process. These priorities are then used at runtime, where the runnable process with the highest priority is always executing. Out of the two schemes that we consider in this chapter, FPS is the most widely used.

For the simple process model we consider, there exists an optimal priority assignment scheme: the rate monotonic priority assignment. The scheme is optimal, as any process set that can be scheduled with a preemptive, fixed-priority scheme can also be scheduled using the rate monotonic assignment scheme. Basically processes with shorter periods are given higher priorities. Recall that $T_i$ denotes the process period and $P_i$ denotes the priority for process $i$. For any two processes $i$ and $j$ the following holds: $T_i < T_j \Rightarrow P_i > P_j$.

Using a test devised by Liu and Layland [18], it is possible to determine if all processes in a system will meet their deadlines when scheduled with the FPS scheme. The test consists of checking the following condition, where $n$ is the number of processes in the system and $C_i$ is the WCET of process $i$:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \left(2^{\frac{1}{n}} - 1\right).$$

The test guarantees that a system satisfying the condition can be scheduled using FPS, however, it does not say anything about systems that do not satisfy the condition. Table 2.1 on the facing page shows the bound for small values of $n$. The bound asymptotically approaches 69.3% (from above), which means that in general any system with a combined utilisation of less than 69.3% will be schedulable by FPS.

System $A$ in Table 2.2 on the next page is an example of a system that cannot be scheduled by FPS. The units of the time values in the table are not defined, but as long as all the values are in the same unit the utilisation bound test is usable. With regard to priorities, a higher number means a higher priority. For example, process $c$ has the highest priority (three) in system $A$. The utilisation $U$ is calculated as a fraction between computation time $C$ and period $T$, i.e. $U = \frac{C}{T}$.

In Figure 2.1 on the preceding page, system $A$ has been scheduled using FPS. It is clear from the time-line that process $a$ misses its deadline, hence the system is not schedulable using this scheme. Consequently, we expect the test not to guarantee schedulability, which it does not, as the utilisation bound for system $A$, $0.24 + 0.25 + 0.33 = 0.82$, is above the threshold for three processes seen in Table 2.1 on the next page.
Table 2.1: Utilisation bounds for process sets of \( n \) processes, scheduled using FPS.

<table>
<thead>
<tr>
<th>( n )</th>
<th>Utilisation bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100.0%</td>
</tr>
<tr>
<td>2</td>
<td>82.8%</td>
</tr>
<tr>
<td>3</td>
<td>78.0%</td>
</tr>
<tr>
<td>4</td>
<td>75.7%</td>
</tr>
<tr>
<td>5</td>
<td>74.3%</td>
</tr>
<tr>
<td>10</td>
<td>71.8%</td>
</tr>
<tr>
<td>100</td>
<td>69.6%</td>
</tr>
<tr>
<td>300</td>
<td>69.3%</td>
</tr>
</tbody>
</table>

Table 2.2: System \( A \), where \( T \) is period, \( C \) is computation time, \( P \) is priority, and \( U \) is utilisation.

<table>
<thead>
<tr>
<th>Process</th>
<th>( T )</th>
<th>( C )</th>
<th>( P )</th>
<th>( U )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>50</td>
<td>12</td>
<td>1</td>
<td>0.24</td>
</tr>
<tr>
<td>( b )</td>
<td>40</td>
<td>10</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>( c )</td>
<td>30</td>
<td>10</td>
<td>3</td>
<td>0.33</td>
</tr>
</tbody>
</table>

System \( B \) in Table 2.3 shows that the test is sufficient but not necessary. The system’s combined utilisation is 100%, which is clearly above the bound for three processes. Nonetheless, the system is schedulable as is witnessed by the time-line in Figure 2.2 on the next page.

2.1.2 Earliest Deadline First Scheduling

EDF is an dynamic scheduling scheme, which means that it decides at run-time which process to schedule next. In general the relative deadline of a process is known: After process \( i \) is released a maximum of \( T_i \) time units may elapse before the process has responded. In EDF the scheduler, at runtime, continuously monitors the absolute deadlines of all running processes. If for example process \( i \) is released at time \( x \), the absolute deadline will be \( x + T_i \). When the currently running process completes, the scheduler selects the next process to run as the one with the smallest (nearest) absolute deadline.

Analogous to the test for whether a system could be scheduled with FPS

<table>
<thead>
<tr>
<th>Process</th>
<th>( T )</th>
<th>( C )</th>
<th>( P )</th>
<th>( U )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>80</td>
<td>40</td>
<td>1</td>
<td>0.50</td>
</tr>
<tr>
<td>( b )</td>
<td>40</td>
<td>10</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>( c )</td>
<td>20</td>
<td>5</td>
<td>3</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Table 2.3: System \( B \).
[18] also stated a test for scheduling with EDF.

\[ \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \]

This test, in addition to being simpler than the one for FPS, is actually necessary and sufficient. Directly comparing the two formulae one can deduce that EDF must be able to schedule a superset of the systems that FPS can.

This is the only advantage though – FPS is better than EDF in many other ways:

- As the priorities in FPS are static, it is easier to implement. The dynamic priorities in EDF calls for a bigger run-time system that will have a higher overhead.

- It is easier to modify FPS to support processes without deadlines than to modify EDF. In FPS these processes are merely assigned priorities, whereas in EDF they must be given an artificial deadline.

- FPS’ behaviour during overload situations is more predictable (the processes with low priorities will miss their deadlines first). An overload situation might be a fault condition and it is therefore important to know what the behaviour will be. EDF is prone to cause a domino effect during overload which entails that a large number of processes will miss their deadlines.

- A process’ deadline is not the only parameter that matters: Some processes might be inherently more important than others, e.g. because the failure of one process is fatal for the system, whereas the failure of another is merely inconvenient. Other factors are more easily incorporated into FPS’ notion of priorities than into EDF’s notion of deadlines. When considering the behaviour during overload situations this can mean the system can, under FPS, continue running but with degraded functionality.

- Since the utilisation-based test for FPS is sufficient but not necessary, it is misleading. FPS is able to schedule some systems that fail the test.
In general, most systems use the FPS scheduling scheme, because of these disadvantages.

2.2 Process Interactions

One of the requirements of our simplistic process model was that no interaction between processes was allowed. However, in almost any realistic system, processes will need to interact: most notably by requiring access to the same resource. This access needs to be controlled such that no two processes are accessing the resource at the same time, as this could lead to an undefined state. In order to ensure this mutual exclusion, processes need to lock resources before they use them. A lock on a resource can only be acquired by one process, and if a process attempts to acquire an already locked resource, it will become blocked and must wait for the resource to be unlocked before proceeding (with locking the resource). The time in which a process has locked a resource is termed the critical section.

The problem with this blocking of processes is illustrated in Figure 2.3. The two processes \( a \) and \( c \) both need access to some resource. The low priority process \( a \) is released first and acquires the lock. The high priority process \( c \) is then released but cannot complete because it cannot acquire the lock. The medium priority process \( b \) is then released and preempts process \( a \). In total the high priority process \( c \) is not only delayed due to the blocking from process \( a \), but it is also waiting for the lower priority process \( b \) to complete. In a sense the priority of processes has been inverted, as a lower priority process is allowed to complete, while the high priority process must wait.

![Figure 2.3: Example of priority inversion. The higher priority process \( c \) is delayed by lower priority processes \( a \) and \( b \) because it is waiting for the lock held by process \( a \).](image)

A way to overcome the problem of priority inversion is to use the principle of priority inheritance. With priority inheritance each process has, in addition
to its static priority, a dynamic priority. Processes are scheduled using their
dynamic priority. The dynamic priority for process $i$ is calculated by

$$P^D_i = \max\{\{P_j\} \cup \text{waiting\_for}(i)\}$$

where waiting\_for($i$) = \{ $P^D_j$ | process $j$ is blocked by process $i$ \}.

Using priority inheritance the example from before would have another out-
come than detailed in Figure 2.3 on the previous page. Using priority inheritance
the high priority process is only blocked for as long as the resource needed is
locked.

![Figure 2.4: Example of priority inheritance. The lower priority process $a$ is
allowed to preempt process $b$, because the higher priority process $c$ is waiting
for $a$ to complete.](image)

The maximum time a process can be blocked under the priority inheritance
scheme can be calculated. First, one must realise that there is an obvious bound
on the number of times a process can be blocked by lower priority processes:
the number of critical sections in the process. In the worst-case each time the
process tries to lock a resource, another process has it locked. In addition, a
process can only be blocked by a lower priority process once during each release,
potentially lowering the bound to the number of lower priority processes. The
maximum blocking time of process $i$ can be calculated as:

$$B_i = \sum_{k=1}^{K} usage(k, i) \times C_k,$$

where

$$usage(k, i) = \begin{cases} 
1 & \text{if there exists processes } a \text{ and } b \text{ that both use resource } k \text{ and } P_a < P_i \leq P_b \\
0 & \text{otherwise,} 
\end{cases}$$

and $C_k$ is the WCET of the longest critical section involving resource $k$. 
2.2.1 Priority Ceiling Protocols

The general priority inheritance protocol has the disadvantage that a high priority process can still be blocked once per lower priority process. This may lead to a high worst-case blocking time. In addition, transitive blocking is also possible: process $a$ is being blocked by process $b$, which again is being blocked by process $c$.

It is possible to mitigate this such that a process can be blocked at most once during its execution by lower priority processes. This is done using the priority ceiling protocols, of which two exist: The Original Ceiling Priority Protocol (OCPP) and the Immediate Ceiling Priority Protocol (ICPP). We will only describe the ICPP as this is the most commonly used, it is simpler, and the benefits of the protocols are the same.

Both these protocols can, on a uniprocessor system, ensure the following properties:

- A process is blocked at most once from lower priority processes.
- Deadlocks are prevented through ordering of resources.
- Transitive blocking is not possible.
- Mutual exclusion is inherent in the protocol itself.

2.2.2 Immediate Ceiling Priority Protocol (ICPP)

The ICPP works as follows:

- Each process has a static default priority
- Each resource $k$ has a static ceiling priority, which is found by $P_k = \max\{P_i | \text{process } i \text{ may use resource } k\}$
- Each process $i$ has a dynamic priority, calculated by $P^D_i = \max\{P_i\} \cup \{P_k | \text{process } i \text{ has locked resource } k \}$

We will now prove some of the properties of ICPP.

**Theorem 1** (ICPP ensures mutual exclusion). If a process $i$ has access to a resource $k$ under the ICPP then all other processes with higher (static) priority, i.e. the processes that are able to preempt process $i$, will not use that resource.

**Proof.** The proof is by contradiction. Assume that a process $j$ exists that uses resource $k$, and that $P_j > P^D_i$.

By substituting in the formula we get that:

$$P^D_i = \max\{P_i\} \cup \{\ldots, P_k, \ldots\},$$

as process $i$ has acquired resource $k$.

We also get that

$$P_k = \max\{P_i, P_j, \ldots\} \Rightarrow P_k \geq P_j,$$

as both process $i$ and $j$ use resource $k$. 


This leads to:

\[ P_i^D = \max\{P_i\} \cup \{\ldots, P_k, \ldots\} \Rightarrow P_i^D \geq P_k. \]

Putting it all together we get that:

\[ P_i^D \geq P_k \geq P_j, \]

which clearly contradicts our assumption that \( P_j > P_i^D \). \( \square \)

**Theorem 2** (Processes are blocked at most once). *When a process \( i \) starts executing, all the resources it needs must be free.*

**Proof.** The proof is by contradiction. Assume that when process \( i \) starts another process \( j \) holds the lock on a resource \( k \) that \( i \) needs.

As process \( i \) is allowed to preempt process \( j \), it must hold that

\[ P_i > P_j^D. \]

Since process \( j \) has locked resource \( k \) we have that:

\[ P_j^D = \max\{P_j\} \cup \{\ldots, P_k, \ldots\} \Rightarrow P_j^D \geq P_k. \]

As both processes \( i \) and \( j \) use resource \( k \) it must also hold that

\[ P_k = \max\{P_i, P_j, \ldots\} \Rightarrow P_k \geq P_i. \]

It then follows that:

\[ P_j^D \geq P_k \geq P_i, \]

which is clearly a contradiction of \( P_i > P_j^D \). \( \square \)

Recall that deadlocks can be prevented by avoiding a circular waiting condition. One method of doing so is imposing a partial ordering on all resources, and require that all processes acquire resources in non-decreasing order [13].

**Theorem 3** (ICPP is deadlock-free). *ICPP imposes a partial ordering on the resources, and as such prevents deadlocks.*

**Proof.** The proof follows directly by Theorem 2 and 1. If all the resources are free when the process starts, and all processes able to preempt it will not use the same resources, then no circular waiting can occur. \( \square \)

### 2.3 Worst-Case Execution Time

This section is based on [28]. The scheduling of a set of processes in a hard real-time system needs the WCET of each process in the system. This section will give an introduction to what WCET is and what is required of the process to be able to find a WCET. Furthermore, we will discuss two classes of methods for finding the WCET of a process. In Figure 2.5 on the facing page the distribution of execution times of a process can be seen. The WCET is, as mentioned earlier, the longest execution time of all execution times of the process. To be able to
find the WCET of a process, the process will naturally need to terminate, and the input needs to be bounded, which means we do not allow infinite input. One way to approximate the WCET is by running the process a number of times and observing the execution times and select the maximal observed execution time. This will usually not be the true WCET, however, as the true WCET might not be observed. Nevertheless, this is the basis of measurement-based methods for finding WCETs. It is possible to find safe upper bounds of the WCET by using static methods.

![Figure 2.5: Distributions of execution times a process.](image)

### 2.3.1 Measurement-Based Methods

Measurement-based methods find the WCET by executing the process and measuring the time required for the process to complete. The advantage of this approach is that it is relatively simple to implement, one just needs to instrument the code and measure its execution time. Furthermore, it does not need any considerable adaption to be applied to newer hardware architectures.

The main disadvantage is that these methods typically only find a maximal observed execution time which can be less than the actual WCET, as can be seen in Figure 2.5. Another disadvantage is that the process needs to be executed several times which can be time consuming if the process itself is time consuming. To find more precise estimates, more advanced methods have been developed. One method is to measure the time of all so-called “basic blocks”, which is sequential blocks of code that do not contain branching, i.e. there are be no if-statements or while-loops in a “basic block”. The WCET of each “basic block” together with the time required for branches is summed up to find the WCET of the process.

### 2.3.2 Static Methods

Static methods are typically used to find a timing upper bound of a process which is strictly larger than the WCET of the process. Furthermore, static methods do not execute the process but is based on analysis of the process’ code and some form of model of the system on which the process should be
executed. Static methods are often based on data flow analysis as is known from compiler techniques.
Chapter 3

Static Analysis

This chapter is based on [24] and [21], unless otherwise noted. Static analysis is the technique of arguing about a program's behaviour without executing it. Examples of behaviours can be:

- Will the value of variable \( x \) ever be negative?
- From which program statements could \( x \) have been assigned its current value?
- Will the value of variable \( x \) ever be read?
- What expressions have previously been calculated that are still valid?
- On a given processor, which data items are in the cache at a certain program point?
- Will the value of variable \( x \) always be the same at a certain program point?

In general, deciding these properties is impossible. This general result is due to Rice’s Theorem, as described in Theorem 4 [16].

**Theorem 4** (Rice). If \( S \) is a non-trivial \(^1\) property of Turing-recognisable languages, then the problem of deciding whether the language of a Turing machine \( M, L(M) \), satisfies the property \( S \), that is \( L(M) \in S \), is undecidable.

**Proof Sketch.** The proof is by reduction from the Halting Problem of Turing machines, that is known to be undecidable. If one was able to determine whether \( L(M) \in S \), then one would be able to solve the Halting Problem. Given a Turing machine \( M' \) and input \( w \), one is able to construct a Turing machine \( M \) such that \( L(M) \in S \) if and only if \( M' \) halts on \( w \), and thus if one can decide \( L(M) \in S \) one can also decide whether \( M' \) halts on \( w \).

---

\(^1\) A property \( S \) is defined to be non-trivial, if there exists Turing-recognisable languages \( L_1, L_2 : L_1 \in S \land L_2 \notin S \). The only two trivial properties are the property \( S_{all} \) that contains all Turing-recognisable languages, and the property \( S_{none} \) that contains no languages.
Since the problems to be decided are proven to be undecidable, static analysis deals with approximations instead. If a property has an answer within \{yes, no\} then an approximation would be allowed to answer within \{yes, no, maybe\}. Static analyses can be classified according to how they treat a “maybe” answer: A must analysis will treat a “maybe” as a “no”, whereas a may analysis will treat a “maybe” as a “yes”.

As an example, consider an analysis that determines whether a variable is positive at a certain program point. Assume that there are two code paths leading to that program point. Now, if the variable is positive in one branch, and negative in the other branch, the must analysis should answer “no”, because it cannot guarantee the variable is positive. The may analysis should answer “yes”, because the variable may be positive. That the analysis answers as we expect is known as “soundness”: if the analysis answers that the property holds, then the property must/may hold in the actual execution of the program.

This also means that not all properties that hold in the actual execution can be proven by the analysis (“completeness”). If the analysis was sound and complete at the same time, then it would not be an approximation, and it would, using Rice’s theorem, allow the solution of the Halting Problem.

The situation is depicted in Figure 3.1.

![Illustration of a sound and a complete approximation, as opposed to the actual real set, for a must analysis.](image)

### 3.1 Control Flow Graphs

A Control Flow Graph (CFG) is a graph detailing how control can flow through a program; this can both be on statements in a high level language description of the program or on a low level assembly description. The examples we will give in this introduction will be on statements in a high level language of programs. A CFG consists of nodes (statements) and edges. The control flow of a program closely follows the sequence in which statements will be executed at runtime.

A very small, sequential and simple program written in a tiny imperative language, \(x := 5; y := 6\), would result in the CFG shown in Figure 3.2. A simple if-construction like \(\text{if } (x == 7) \{ y := 6 \} \text{ else } \{ y := 7 \} \text{ end if}\) becomes the CFG shown in Figure 3.3.
In similar fashion a loop is represented as a cycle in the CFG. It is worth noting that the paths in the CFG is a superset (and therefore an over-approximation) of the actual execution paths of the program, as the CFG totally disregards the values of variables. A basic block graph is similar to a CFG, however, in a basic block graph the nodes represent a sequence of statements that do not contain branches.

3.2 Lattice Theory

Lattice theory is very used in static analysis methods, such as abstract interpretation. In this section we give a quick introduction to lattice theory.

**Definition 1 (Partial Order).** A partial order over a set \( L \) is a relation \( \sqsubseteq : L \times L \rightarrow \{\text{true, false}\} \) that is:

- **Reflexive:** \( \forall l \in L : l \sqsubseteq l \)
- **Transitive:** \( \forall l_1, l_2, l_3 \in L : l_1 \sqsubseteq l_2 \land l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3 \)
- **Anti-symmetric:** \( \forall l_1, l_2 \in L : l_1 \sqsubseteq l_2 \land l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2 \)

A partial order is sometimes denoted \((L, \sqsubseteq)\).

Given a partial order \((L, \sqsubseteq)\), a subset \( Y \subseteq L \) has an upper bound \( l \), if \( \forall l' \in Y : l' \sqsubseteq l \). A lower bound is defined accordingly: \( l \) is a lower bound of \( Y \), if \( \forall l' \in Y : l \sqsubseteq l' \).
We sometimes write \( l \sqsubseteq Y \) to denote a lower bound, and \( Y \sqsubseteq l \) to denote an upper bound.

A least upper bound of a set \( Y \subseteq L \) is an element \( l \in L \) such that

1. \( l \) is an upper bound of \( Y \), \( Y \sqsubseteq l \)
2. \( \forall l' \in L : Y \sqsubseteq l' \Rightarrow l \sqsubseteq l' \)

A least upper bound is denoted \( \sqcup Y \). A least lower bound, denoted \( \sqcap Y \), is defined symmetrically.

**Definition 2** (Complete Lattice). A complete lattice is a partial order \( (L, \sqsubseteq) \) with the additional requirement that \( \forall Y \subseteq L : \exists \sqcup Y \land \exists \sqcap Y \).

This requirement leads to the existence of the least element, \( \bot = \sqcup \emptyset = \sqcap L \) and of the greatest element, \( \top = \sqcap \emptyset = \sqcup L \).

### 3.3 Abstract Interpretation

Abstract interpretation is a systematic way to do static analysis. By describing the traditional semantics of a programming language, all details about the execution are described. Abstract interpretation seeks to find a semantic that abstracts away details that are unnecessary for the analysis at hand, but preserves enough details such that the analysis provides a meaningful approximation.

The classical example of abstract interpretation, from among others [10], is a signedness analysis. Consider a very simple language of integer arithmetic:

\[
S ::= Z \mid S + S \mid S - S \mid S \cdot S
\]

The “normal” semantic for this language would be in the domain of \( Z \), and an example of usage would then be:

\[
\begin{array}{c}
\text{[plus]} \quad 17 \rightarrow 17 \\
\text{[times]} \quad -5 \cdot 2 \rightarrow -10 \\
17 + (-5 \cdot 2) \rightarrow 7
\end{array}
\]

A more abstract version of the semantics is to only deal in the domain of \( D' = \{+,-,\pm\} \). The same statement in this domain would give rise to the following inference:

\[
\begin{array}{c}
\text{[plus]} \quad 17 \rightarrow + \\
\text{[times]} \quad -5 \cdot 2 \rightarrow - \\
17 + (-5 \cdot 2) \rightarrow \pm
\end{array}
\]

The final usage of the plus rule can only give the answer \( \pm \), since, given the arguments \(+\) and \( -\), the result can either be positive or negative — one would need the actual values to determine which.

This domain \( D' \) is definitely a more abstract version of the domain \( Z \), as there exists a homomorphism between the two domains. As such we can illustrate the relationship between \( D' \) and \( Z \) as done in Figure 3.4 on the next page.
In abstract interpretation we define an abstraction function $\alpha$ and a concretisation function $\gamma$, such that they together form a Galois connection. In our case, the two functions are defined as:

$$
\alpha : 2^\mathbb{Z} \rightarrow 2^{D'} \quad \gamma : 2^{D'} \rightarrow 2^\mathbb{Z}
$$

\[ 
\alpha(S) = \begin{cases} 
\{+\} & \text{if } \forall n \in S : n > 0 \\
\{-\} & \text{if } \forall n \in S : n < 0 \\
\{\pm\} & \text{otherwise}
\end{cases} 
\]

\[ 
\gamma(d) = \begin{cases} 
\{1, 2, \ldots\} & \text{if } d = \{+\} \\
\{-1, -2, \ldots\} & \text{if } d = \{-\} \\
\mathbb{Z} & \text{otherwise}
\end{cases} 
\]

The functions are illustrated in Figure 3.5 on the following page.

For $\alpha$ and $\gamma$ to form a Galois connection, the following is required to hold [21, p. 15]:

$$
\alpha(X) \subseteq Y \iff X \subseteq \gamma(Y).
$$

This condition ensures that the abstraction and concretisation are sound, but it does not state anything about the precision of the functions.

To check that this condition is maintained the domains must have an ordering. A commonly used ordering is lattices. The advantage of using lattices is that it makes it easy to find least upper and lower bounds.
Figure 3.5: The abstraction $\alpha$ and concretisation $\gamma$ functions, for this specific instance.

Figure 3.6: The abstraction $\alpha$ and concretisation $\gamma$ functions, for a general Galois connection.
Chapter 4

Model Checking

Model checking is an automated method for verifying that a model satisfies a property [5]. The model and the property describe a possible system and a desired behaviour, respectively, in an unambiguous, mathematical precise way. Systems such as electrical circuits in processors, guidance software in spaceships and control software in production plants can benefit by model checking. The properties are often of qualitative nature and related to safety:

- The system never deadlocks.
- It is always possible to return to the initial state.
- Variable $a$ never becomes greater than 10.

If a given model takes time into account, it is also possible for the properties to do so. For example, a property involving time is “the system does not deadlock within two days after reset”.

We include the theory of model checking in this project, as we utilise model checking to determine the WCET for programs. Chapter 8, starting on page 71, explains the approach in detail.

In model checking, systems are mostly modelled using finite state automata, and properties are expressed using logics. Both the automata and the logics can be tailored to the specific domain that one wishes to do model checking in. A transition system is one of the simplest models for modelling systems:

**Definition 3 (Transition System [5]).** A transition system $TS$ is a tuple $(S, Act, \rightarrow, I, AP, L)$, where $S$ is a set of states, $Act$ is a set of actions, $\rightarrow \subseteq S \times Act \times S$ is a transition relation, $I \subseteq S$ is a set of initial states, $AP$ is a set of atomic propositions, and $L : S \rightarrow 2^{AP}$ is a labelling function. $TS$ is called finite, if $S$, $Act$ and $AP$ are finite.

Figure 4.1 on the next page shows a transition system for a beverage vending machine that sells coffee and tea. It has the states \{pay, select_coffee, select_tea\}, the actions

\{insert_coin, select_coffee, select_tea, pour_coffee, pour_tea\},

27
the transition relation

$$\{(\text{pay}, \text{insert\_coin}, \text{select}), (\text{select}, \text{select\_coffee}, \text{coffee}),$$
$$\quad (\text{select}, \text{select\_tea}, \text{tea}), (\text{coffee}, \text{pour\_coffee}, \text{pay}),$$
$$\quad (\text{tea}, \text{pour\_tea}, \text{pay})\},$$

the initial states $$\{\text{pay}\}$$, the atomic propositions $$\{\text{paid}, \text{coffee\_selected}, \text{tea\_selected}\}$$, and the labelling function

$$L(s) = \begin{cases} 
\emptyset & \text{if } s = \text{pay} \\
\{\text{paid}\} & \text{if } s = \text{select} \\
\{\text{paid}, \text{coffee\_selected}\} & \text{if } s = \text{coffee} \\
\{\text{paid}, \text{tea\_selected}\} & \text{if } s = \text{tea} 
\end{cases}$$

The behaviour of the transition system in Figure 4.1 is intuitively clear, but the formal semantics of transition systems are defined through executions [5], also known as runs. We omit these definitions here, as the vending machine only serves as an introductory example. In Section 4.1 we present timed automata which cover the timing aspects of systems.

A logic for expressing timed properties is Timed Computation Tree Logic (TCTL). The syntax and semantics of TCTL, together with examples, are presented in Section 4.2.

Model checking is done in a brute-force manner, due to the fact that potentially all possible states in a system must be explored. Methods that optimise the exploration by e.g. not traversing into property violating branches have been devised, but model checking remains a computationally heavy task. Only a decade ago, this fact was a significant drawback of the method, since the slow computers of the time only allowed small systems to be checked. Fortunately, the drawback has almost disappeared today, because of more clever exploration optimisations, and because the speed of modern computers allows for verification of larger, realistic systems. In Section 4.3 we study how model checking is conducted with regards to timed automata and TCTL expressions.

Some model checkers, have the feature that they try to produce a trace when a property is violated somewhere in a model. The trace is a path through the model that shows how the model checker ended up in the faulty situation. This demonstrates another strength of model checking, as the trace can be used to track down the problem that caused the property violation. For example, a
trace through a model of a program can be projected back on the original code to assist programmers in fixing the problem.

A disadvantage of model checking is that the method works with models rather than directly with the actual applications. Models must be constructed from the applications before model checking can take place, and this is sometimes done using an automated tool, that abstracts away some level of detail. It is also possible that the tool has bugs, as it, just like the applications that we want to check, is built by humans. The morale is, that model checking, to the extent it is presented here, is only as good as the quality of the models and the properties.

4.1 Timed Automata

The behaviour of time-critical systems can be modelled using timed automata. A timed automaton is a transition system extended with real-valued clock variables [5]. The extension makes it easier for a model designer to model the time aspects of systems, but timed automata are not more expressive than transition systems. Compared to well-known variable types like booleans and integers, clock variables are different in two ways. Firstly, access to them is limited to read and reset, i.e. it is possible to read their value and to reset their value to zero. Secondly, their value increases automatically as time passes. This happens with the same rate for all clock variables, and the value of a clock variable can thus be interpreted as the amount of time that has passed since the variable was reset. To ease communication, clock variables are simply called clocks.

The actions in a timed automaton can be guarded by conditions on clocks. For example, for a light switch it might be the case that the light is switched on and then dimmed down, if the switch is touched two times within two seconds, whereas the light is switched on and then switched off, if the second touch happens after two seconds have passed. In model checking terminology, a possible action is said to be enabled, and an impossible action is said to be disabled. A condition on a clock is called a clock constraint:

**Definition 4 (Clock Constraint [5]).** A clock constraint over a set $C$ of clocks is formed according to the grammar

$$g ::= x < c \mid x \leq c \mid x > c \mid x \geq c \mid g \land g,$$

where $c \in \mathbb{N}$ and $x \in C$. Let $CC(C)$ denote the set of clock constraints over $C$.

Clock constraints that do not contain any conjunctions are atomic. Let $ACC(C)$ denote the set of all atomic clock constraints over $C$.

Besides controlling whether actions are enabled or not, clock constraints are also used to limit the time it is possible to stay in a state. This is made possible by invariants, which are clock constraints on states. Invariants can also prevent one from reaching a state, as a state can only be entered if its invariant is satisfied. As is clear in the following definition, states in timed automata are called locations:

**Definition 5 (Timed Automaton [5]).** A timed automaton $TA$ is a tuple $(Loc, Act, C, \rightarrow, Loc_0, Inv, AP, L)$, where
• **Loc** is a finite set of locations,

• **Act** is a finite set of actions,

• **C** is a finite set of clocks,

• $\rightarrow \subseteq \text{Loc} \times \text{CC}(C) \times \text{Act} \times 2^C \times \text{Loc}$ is a transition relation,

• **Loc** is a set of initial locations,

• **Inv : Loc → CC(C)** is an invariant-assignment function,

• **AP** is a finite set of atomic propositions, and

• **L : Loc → 2^AP** is a labelling function for the locations.

**ACC(TA)** denotes the set of atomic clock constraints that occur in either a guard or a location invariant of **TA**.

The transition relation $\rightarrow$ defines transitions between locations. The transitions are labelled with tuples $(g, \alpha, D)$, where $g \in \text{CC}(C)$, $\alpha$ is an action, and $D \subseteq C$ is a set of clocks. The notation $l \xrightarrow{g,\alpha,D} l'$ means that it is possible to do a transition from location $l$ to location $l'$ when the clock constraint $g$ holds. Moreover, if the transition is done, the clocks in $D$ are reset and the action $\alpha$ is performed. The precise interpretation of timed automata is studied in Section 4.1.1.

Figure 4.2 on the next page shows a timed automaton that models a computer scientist (**TA_CS**). Minutes are used as time unit in the automaton. The computer scientist is asleep for eight hours per day and awake for the remaining 16 hours. The circadian rhythm is forced to be stable, as the scientist only sleeps when a full 16-hour workday has passed and always sleeps for exactly eight hours. While awake, the scientist can choose to work, get coffee or dine. A period of work takes two hours, a trip to the coffee machine takes ten minutes, and a dinner in the cafeteria takes half an hour. The three tasks are voluntary and can be carried out during the day. To keep the figure brief, atomic propositions are not shown. The locations are labelled with atomic propositions equal to their names. For example, the location “awake” is labelled with the set \{awake\}.

Using **Definition 5** on the preceding page, the timed automaton **TA_CS** = (Loc, Act, C, $\rightarrow$, Loc, Inv, AP, L) can be formally defined:

• **Loc** = \{awake, asleep, working, getting_coffee, dining\},

• **Loc_0** = \{awake\},

• **Act** = \{fall_asleep, wake_up, start_working, stop_working, get_coffee, got_coffee, start_dining, stop_dining\},

• **C** = \{v, w, x, y, z\},

• $\rightarrow$ = \{(awake, v ≥ 960, fall_asleep, \{w\}, asleep),
     (asleep, w ≥ 480, wake_up, \{v\}, awake),
     (awake, v < 840, start_working, \{x\}, working),
     (working, x ≥ 120, stop_working, \emptyset, awake),
     ...\}
Figure 4.2: A computer scientist modelled using a timed automaton, $TA_{CS}$.

\begin{itemize}
  \item $Inv(l) = \begin{cases} 
    v \leq 960 & \text{if } l = \text{awake} \\
    w \leq 480 & \text{if } l = \text{asleep} \\
    x \leq 120 & \text{if } l = \text{working} \\
    y \leq 10 & \text{if } l = \text{getting\_coffee} \\
    z \leq 30 & \text{if } l = \text{dining} 
  \end{cases}$
\end{itemize}

Since the locations are labelled with atomic propositions equal to their names, the set $AP$ and the function $L$ are very simple and therefore omitted.

### 4.1.1 Semantics

Any timed automaton can be interpreted as a transition system [5]. Because timed automata use continuous time, the corresponding transition systems have infinitely many states and are infinitely branching. The transition system that corresponds to a given timed automaton is found by unfolding it. The states of the transition system consist of a control component, which indicates the location in the timed automaton, and a valuation of the clocks. Consequently, the states are of the form $(l, \eta)$, where $l$ is a location and $\eta$ is a valuation of the clocks.

**Definition 6** (Clock Valuation [5]). A clock valuation $\eta$ for a set $C$ of clocks is a function $\eta : C \rightarrow \mathbb{R}_{\geq 0}$, assigning to each clock $x \in C$ its current value $\eta(x)$. $Eval(C)$ denotes the set of all clock valuations over $C$.

The following definition introduces a satisfaction relation that formally defines what it means for a clock constraint to hold for a clock valuation:

**Definition 7** (Satisfaction Relation for Clock Constraints [5]). For a set $C$ of clocks, $x \in C$, $\eta \in Eval(C)$, $c \in \mathbb{N}$, and $y, y' \in CC(C)$, let $\models \subseteq Eval(C) \times CC(C)$
be defined by

\[ \begin{align*}
\eta \models \text{true} \\
\eta \models x < c & \iff \eta(x) < c \\
\eta \models x \leq c & \iff \eta(x) \leq c \\
\eta \models \neg g & \iff \eta \not\models g \\
\eta \models g \land g' & \iff \eta \models g \land \eta \models g'.
\end{align*} \]

A timed automaton can proceed in two ways: by letting time progress while staying in a location, or by taking a transition. In the corresponding transition system, the former is called a delay transition, and the latter is called a discrete transition. Delay transitions are labelled with the action of the transition in the timed automaton, whereas discrete transitions are labelled with a positive real number which indicates the amount of time to delay. The following definition specifies in detail how the corresponding transition system works:

**Definition 8** (Transition System Semantics of a Timed Automaton [5]). Let \( TA = (\text{Loc}, \text{Act}, C, \rightarrow, \text{Loc}_0, \text{Inv}, \text{AP}, L) \) be a timed automaton. The transition system \( TS(TA) = (S, \text{Act}', \rightarrow, I, \text{AP}', L') \) with

- \( S = \text{Loc} \times \text{Eval}(C) \),
- \( \text{Act}' = \text{Act} \cup \mathbb{R}_{\geq 0} \),
- \( I = \{ \langle l_0, \eta \rangle \mid l_0 \in \text{Loc}_0, \eta \in \text{Eval}(C) \land \eta(x) = 0 \text{ for all } x \in C \} \),
- \( \text{AP}' = \text{AP} \cup \text{ACC}(C) \),
- \( L'(l, \eta) = L(l) \cup \{ g \in \text{ACC}(C) \mid \eta \models g \} \), and
- the transition relation \( \rightarrow \) is defined by the following two rules:

  - **discrete transition**: \( \langle l, \eta \rangle \xrightarrow{\alpha} \langle l', \eta' \rangle \) if the following conditions hold:
    
    (a) there is a transition \( l \xrightarrow{g, \alpha, D} l' \) in \( TA \)
    
    (b) \( \eta \models g \)
    
    (c) \( \eta' = \text{reset } D \text{ in } \eta \)
    
    (d) \( \eta' \models \text{Inv}(l') \)

  - **delay transition**: \( \langle l, \eta \rangle \xrightarrow{d} \langle l, \eta + d \rangle \) for \( d \in \mathbb{R}_{\geq 0} \)
    
    (e) if \( \eta + d \models \text{Inv}(l) \)

where

\[(\text{reset } x \text{ in } \eta)(y) = \begin{cases} 
\eta(y) & \text{if } y \notin x \\
0 & \text{if } y \in x
\end{cases}\]

and

\((\eta + d)(x) = \eta(x) + d \text{ for all clocks } x \in C.\)
4.2 Timed Computation Tree Logic

TCTL is a variant of Computation Tree Logic (CTL) aimed at specifying properties for timed automata. Compared to CTL, TCTL covers the time aspects of systems. In the following definition, the formula $\Phi \cup^J \Psi$, where $\cup$ is the “until” operator, means that a state satisfying the property $\Phi$ is reached within $t \in J$ time units while only visiting states satisfying the property $\Psi$:

**Definition 9** (Syntax of TCTL [5]). Formulas in TCTL are either state or path formulae. TCTL state formulae over the set $AP$ of atomic propositions and the set $C$ of clocks are formed according to the following grammar:

$$\Phi ::= true \mid a \mid g \mid \Phi_1 \land \Phi_2 \mid \neg \varphi \mid \exists \varphi \mid \forall \varphi,$$

where $a \in AP$, $g \in ACC(C)$ and $\varphi$ is a path formula defined by:

$$\varphi ::= \Phi_1 \cup^J \Phi_2,$$

where $\Phi_1$ and $\Phi_2$ are state formulae, and $J \subseteq \mathbb{R}_{\geq 0}$ is an interval whose bounds are natural numbers.

For example, a path formula could be “legal $\cup^{[0,30]}$ deadlock”, which states that a deadlock is reached within 30 time units via legal states only, where the atomic propositions legal and deadlock mark states that are legal to reach and states that have no successors, respectively.

The modal operators $\Box$ (globally, $G$) and $\Diamond$ (finally, $F$) have timed variants that can be obtained in the following way:

$$\Diamond^J \Phi = true \cup^J \Phi,$$

$$\exists \Box^J \Phi = \neg \forall \Diamond^J \neg \Phi,$$

and

$$\forall \Box^J \Phi = \neg \exists \Diamond^J \neg \Phi.$$

The first formula, $\Diamond^J \Phi$, means that a state satisfying $\Phi$ is reached within the interval $J$. The second formula, $\exists \Box^J \Phi$, means that a path exists for which, during the interval $J$, $\Phi$ is satisfied. The third formula, $\forall \Box^J \Phi$, means the same as the second formula with the difference that $\Phi$ must hold for all paths.

Like the semantics for timed automata, the semantics for TCTL considers states of the form $(l, \eta)$. The interpretation of the state formulae $\forall \varphi$ and $\exists \varphi$ is limited to time-divergent paths, which are paths that have infinite execution time. Definitions 9.15 and 9.16 in [5] explain the concept in detail — for brevity the definitions are omitted here.

**Definition 10** (Satisfaction Relation for TCTL [5]). Let $TA = (Loc, Act, C, \rightarrow, Loc_0, Inv, AP, L)$ be a timed automaton, $a \in AP$, $g \in ACC(C)$, and $J \subseteq \mathbb{R}_{\geq 0}$. For state $s = (l, \eta)$ in $TS(TA)$ and TCTL state formulae $\Phi$ and $\Psi$, and TCTL path formula $\varphi$, the satisfaction relation $|=\ $ is defined for state formulae by

$$s |= true$$

$$s |= a \quad \text{iff} \quad a \in L(l)$$

$$s |= g \quad \text{iff} \quad \eta \models g$$

$$s |= \neg \Phi \quad \text{iff} \quad s \not\models \Phi$$

$$s |= \Phi \land \Psi \quad \text{iff} \quad (s |= \Phi) \text{ and } (s |= \Psi)$$

$$s |= \exists \varphi \quad \text{iff} \quad \pi \models \varphi \text{ for some } \pi \in \text{Paths}_{div}(s)$$

$$s |= \forall \varphi \quad \text{iff} \quad \pi \models \varphi \text{ for all } \pi \in \text{Paths}_{div}(s),$$
where \( \text{Paths}_{s_0}(s) \) is the set of time-divergent paths starting in state \( s \). For time-divergent path \( \pi \in s_0 \xrightarrow{d_0} s_1 \xrightarrow{d_1} \ldots \), the satisfaction relation \( \models \) for path formulae is defined by

\[
\pi \models \Phi \Uparrow \Psi \iff \exists i \geq 0. s_i + d \models \Psi \text{ for some } d \in [0, d_i] \text{ with } \\
\sum_{k=0}^{i-1} d_k + d \in J \text{ and } \\
\forall j \leq i. s_j + d' \models \Phi \lor \Psi \text{ for any } d' \in [0, d_j] \text{ with } \\
\sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{i-1} d_k + d,
\]

where for \( s_i = \langle l_i, \eta_i \rangle \) and \( d \geq 0 \) we have \( s_i + d = \langle l_i, \eta_i + d \rangle \).

In Definition 10, the time-divergent path \( \pi \in s_0 \xrightarrow{d_0} s_1 \xrightarrow{d_1} \ldots \) satisfies \( \Phi \Uparrow \Psi \) whenever at some time point in \( J \), a state is reached satisfying \( \Psi \) and at all previous time instants \( \Phi \lor \Psi \) is satisfied. In CTL, just \( \Phi \) is required to be satisfied in all previous states, and Example 9.31 in [5] shows why this is different for TCTL.

Whenever the initial states of a timed automaton \( TA \) satisfies a TCTL state formula \( \Phi \), \( TA \) satisfies \( \Phi \):

**Definition 11 (TCTL Semantics for Timed Automata [5]).** Let \( TA \) be a timed automaton with clocks \( C \) and locations \( \text{Loc} \). For TCTL state formula \( \Phi \), the satisfaction set \( \text{Sat}(\Phi) \) is defined by:

\[
\text{Sat}(\Phi) = \{ s \in \text{Loc} \times \text{Eval}(C) \mid s \models \Phi \}.
\]

The timed automaton \( TA \) satisfies TCTL state formula \( \Phi \) if and only if \( \Phi \) holds in all initial states of \( TA \):

\[
TA \models \Phi \iff \forall l_0 \in \text{Loc}_0. (l_0, \eta_0) \models \Phi,
\]

where \( \eta_0(x) = 0 \) for all \( x \in C \).

### 4.3 TCTL Model Checking

Given a timed automaton \( TA \) and a TCTL state formula \( \Phi \), we want to check whether \( TA \) satisfies \( \Phi \). From the semantics of TCTL it is clear that the transition system \( TS(TA) \) corresponding to \( TA \) needs to be analysed [5]. This poses a problem, as \( TS(TA) \) has uncountably many states. The problem is overcome by analysing the region transition system \( RTS(TA, \Phi) \), which is a finite quotient of \( TS(TA) \) with respect to an equivalence relation.

States in \( TS(TA) \) that satisfy the same atomic clock constraints, and which are the start of TCTL equivalent, time-divergent paths, are grouped together as states in \( RTS(TA, \Phi) \). Model checking of \( RTS(TA, \Phi) \) is feasible, as the number of equivalence classes in \( TS(TA) \) is finite. Instead of checking the original formula \( \Phi \) for \( RTS(TA, \Phi) \), a new formula \( \hat{\Phi} \) is created from \( \Phi \) by eliminating timing parameters. The formula \( \hat{\Phi} \) is a CTL formula, and CTL
model checking is used to check whether \( RTS(TA, \Phi) \) satisfies \( \Phi \). CTL model checking is explained in Section 6.4 in [5].

The recipe in Algorithm 1, where \( \cong \) denotes the equivalence used to obtain \( RTS(TA, \Phi) \), is used to do TCTL model checking:

**Algorithm 1** (Basic recipe of TCTL model checking [5]).

*Input:* Timed automaton \( TA \) and TCTL state formula \( \Phi \) (both over \( AP \) and \( C \))

*Output:* Whether \( TA \models \Phi \)

1. \( \hat{\Phi} := \) eliminate the timing parameters from \( \Phi \);
2. determine the equivalence classes under \( \cong \);
3. construct the region transition system \( TS = RTS(TA, \Phi) \);
4. apply a CTL model checking algorithm to check whether \( TS \models \hat{\Phi} \);
5. \( TA \models \Phi \) if and only if \( TS \models \hat{\Phi} \);

Timing parameters are eliminated from TCTL formulae by replacing intervals different from \([0, \infty)\) with equivalent atomic clock constraints [5]. The resulting formulae are TCTL\(_0\) formulae, where TCTL\(_0\) denotes the set of TCTL formulae in which all intervals are equal to \([0, \infty)\). Because atomic clock constraints are the only timing aspects that occur in TCTL\(_0\), and because the constraints can be expressed as atomic propositions, TCTL\(_0\) is a subset of CTL. Consequently, the resulting formula can be checked using CTL model checking.

Theorem 9.37 in [5] provides a recipe for transforming a TCTL formula into a TCTL\(_0\) formula.

The construction of the region transition system depends on the relation \( \cong \), as it decides which states are equivalent in \( TS(TA) \) with regard to satisfying atomic clock constraints. In \( TS(TA) \), this means that \( (l, \eta) \cong (l, \eta') \) if \( \eta \cong \eta' \).

To motivate a proper relation, the book [5] puts forward three conditions that the relation must satisfy:

1. Equivalent clock valuations should satisfy the same clock constraints that occur in \( TA \) and \( \Phi \):

   \[ \eta \cong \eta' \implies (\eta \models g \iff \eta' \models g) \text{ for all } g \in ACC(TA) \cup ACC(\Phi), \]

   where \( ACC(TA) \) and \( ACC(\Phi) \) denote the set of atomic clock constraints that occur in \( TA \) and \( \Phi \), respectively. These constraints are either of the form \( x \leq c \) or \( x < c \).

2. Time-divergent paths starting in equivalent states should be equivalent. This guarantees that equivalent states satisfy the same path formulae.

3. The number of equivalence classes under \( \cong \) is finite.

Definition 9.42 in [5] presents a relation that satisfies the three conditions. We do not study TCTL model checking in greater detail — the interested reader is encouraged to consult Section 9.3 in [5].

### 4.4 Model Checking in UPPAAL

UPPAAL is a model checker for real-time systems [6]. It has been developed since 1995 by researchers at Aalborg University, Denmark, and Uppsala University, Sweden. In this section we introduce the features in UPPAAL that are relevant...
to the modelling done in Chapter 8 starting on page 71. The primary source for this section is [6].

To model interactions in the real world accurately, UPPAAL uses networks of timed automata. The automata interact by synchronizing with each other through synchronization channels. Compared to the timed automata in Section 4.1, the timed automata in UPPAAL have only one initial state and are extended with the following features:

- **Automata templates**, which make it possible to have several instances of the same automaton defined with different parameters.

- **Constants**, which are declared as `const name value`. Constants must have integer values.

- **Bounded integer variables**, which are declared as `int [min, max] name`. If `[^min,^max]` is omitted, the default range is [-32768,32768]. Expressions involving integers may be used in guards (constraints on constants, integers and clocks), invariants and assignments. The bounds are checked on verification time, where states with bound violations are discarded.

- **Binary synchronization channels**, which are declared as `chan name`. An edge labelled with `name!` synchronizes with an edge labelled with `name?`. If several synchronization pairs are possible at a time instant during verification, UPPAAL chooses a pair non-deterministically.

- **Broadcast channels**, which are declared as `broadcast chan name`. This enables a single sender, which has `name!` on an edge, to synchronize with an arbitrary number of receivers, which have `name?` on an edge. Broadcast sending does not block the sender, as the sender can take the `name!` action even though there are no receivers.

- **Urgent synchronization channels**, which are declared by prefixing a channel declaration with `urgent`. If a synchronization transition on an urgent channel is enabled, delays must not occur. No clock guards are allowed on edges that use urgent synchronization channels.

- **Urgent locations**, which are semantically equivalent to locations with an added clock `x` and an added invariant `x<=0`, where `x` is reset on all incoming edges. Consequently, delays are not allowed in an urgent location.

- **Committed locations**, which are even stricter than urgent locations. A state in the verification process of a network of timed automata is committed if any of the locations in the state is committed. A committed state cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations.

- **Arrays** of clocks, channels, constants and integers. Arrays are declared by appending a size to the variable name, e.g. `clock a[2]; chan c[3];` or `int[0,100] i[5];`.

- **Initialisers**. It is possible to initialise integer variables and arrays of integers, e.g. `int i := 10;` or `int ia[3] := {0, 42, 60};`. 

36
Expressions in UPPAAL can involve clocks and integer variables. Four types of expression labels exist: guards, synchronisations, assignments and invariants. Guards, synchronisations and assignments are used on edges, whereas invariants are used on locations. The precise behaviour of a network of timed automata in UPPAAL is defined by the semantics in Definition 3 in [6].

Figure 4.3 shows some of the automata elements in UPPAAL. Locations $A$ and $B$ form a timed automaton, where $A$ is the initial location, $B$ is the terminal location, $A$ has an outgoing edge to $B$, $A$ has the invariant $x \leq 2$, and the edge connecting $A$ and $B$ has the guard $x \geq 2$. The invariant and the guard both involve the clock $x$, and the semantics is that one is forced to stay in location $A$ for two time units and then move to $B$. Locations $C$ and $D$ form a timed automaton, in which the edge is able to synchronise through a channel $c$ with the edge in the automaton formed by locations $E$ and $F$. Location $G$ is an urgent location, and location $H$ is a committed location.

In UPPAAL, the properties that make up system specifications are called queries. UPPAAL’s query language is a simplified version of TCTL, the logic that we presented in Section 4.2 starting on page 33. Compared to TCTL, the query language does not allow nesting of path formulae. In addition, the version of TCTL that we presented did not include the modal operators $\Box$ (globally, $G$) and $\Diamond$ (finally, $F$) directly in its syntax, whereas these operators are directly available in UPPAAL as $\lbrack \rbrack$ and $\lbrack \rbrack$, respectively. We end this chapter with two examples of queries in UPPAAL:

- $E<>Train1.Crossing$ and $Train2.Crossing$: A path exists in which we eventually end up in a state, where the automata $Train1$ and $Train2$ are in the location $Crossing$ at the same time. Without further information about the system, this is probably not a desirable property to satisfy.

- $A[\lbrack \rbrack$ not deadlock$: The system never deadlocks (“for all paths it holds on the entire subsequent path that the system does not deadlock”).
Chapter 5

Optimisation Techniques of Modern Processors

Modern processors use a number of optimisation techniques to speed up average performance. Many of these techniques will however make the worst-case performance much harder to predict, because they introduce an element of non-determinism or a very high dependence on the previous state (which might be unknown or hard to predict). In this chapter some of these techniques will be described. The descriptions are based on [25] and [11]. Other sources are referenced when used.

5.1 Caching

Caching is an optimisation technique used to bridge the large speed difference between the processor and the main memory. The speed up is achieved by placing a smaller, but much faster memory chip, called a cache, between the main memory and the processor. Caching takes advantage of the principle of temporal locality, which states that the same part of main memory is often accessed several times in succession. More explicitly there are two types of locality: temporal locality which states that if a data item has just been used it will probably be used again soon after, and spatial locality which states that if a data item has just been used data items nearby will probably be needed soon after. The cache holds a copy of recently used parts from main memory, or parts of main memory that are waiting to be written out to main memory. If a memory block is not in the cache and needs to be fetched from main memory into the cache, it is called a cache miss, the opposite is called a cache hit.

A cache has a number of important parameters, affecting its performance and predictability:

**Capacity** of the cache is the number of bytes that can be stored in the cache. The capacity is of course very important, since this limits the amount of main memory that can be cached, and thus given quick access to. The smaller the cache, the slower the execution, in general.

**Cache line size** is the number of bytes that are transferred from or to main memory in one transfer. The cache consist of $k = \frac{\text{capacity}}{\text{cache line size}}$ cache
Main memory $M$ is split into memory blocks $m_1, m_2, \ldots, m_n \in M$ for the purpose of caching. Given a memory block $m_i$, the address of the memory block can be found by the function $adr(m_i) = i - 1$. For instance, the address of $adr(m_1) = 0$. For simplicity, we assume that the size of memory blocks and cache lines is the same.

The associativity of the cache determines in which cache lines a memory block can reside. “Fully associative” means that a memory block can reside in any cache line. The opposite, “direct mapped”, means that a memory block can reside in precisely one cache line. A middle road is $A$-way set associative caches, which partition the cache into $k/A$ disjoint sets, called cache sets, and maps each memory block to exactly one set. Cache sets is denoted $s_1, s_2, \ldots, s_{k/A}$. The $n$’th cache line in a cache set is called “way $n$”. Common $A$-way set associative caches are for $A \in \{2, 4, 8\}$.

The associativity is a trade-off between quick lookup (direct mapped being the fastest) and effectiveness of the cache in achieving more hits. Fully associative and direct mapped are special cases of an $A$-way set associative cache with $A = k$ or $A = 1$, respectively.

An example of a 2-way set associative cache can be seen in Figure 5.1. The cache lines, in which a memory block can reside, can generally be calculated by $\text{cacheSet}(m) = \{l_i \mid i = (adr(m) \mod \frac{k}{A}) \cdot A + j\}$, where $m$ is a memory block and $j \in \{1, \ldots, A\}$.

![Figure 5.1: Illustration of which memory blocks can reside in which cache lines in a 2-way set associative cache. The arrows represent which cache lines a memory block can be cached in.](image)

The replacement policy determines what memory block to evict, when a new memory block needs to enter the cache, and the cache is already full. The policy has a very large impact on the predictability of the cache. Some common policies are: Least-Recently Used (LRU), Pseudo Round-Robin (PRR), First-In First-Out (FIFO) and Pseudo Least-Recently Used (PLRU).

Handling writes can be an important aspect of designing a cache and can have a huge impact on the overall performance [11, p. 483]. The problems can be split into two cases: What to do on write hits and what to do
on write misses. Write hits occur when a data item in a memory block has to be modified and the memory block is in the cache. Write misses happen when the memory block is not in the cache. For write hits the simplest approach is to modify the memory block in the cache and write the change directly to main memory; this approach is called write through. The drawback is that this will affect performance as every write will cause a write to main memory. Another more complicated approach is to only modify the memory block in the cache and only write the memory block to main memory when it is evicted from the cache. This means we now need to keep track of which memory blocks have to be written back to memory when they are evicted from the cache. This is usually done by using a dirty bit for each cache line to indicate if the memory block has been modified or not. For write misses, the simplest approach is to only write the change to main memory. This method is called “no write allocate”. A more complex approach is “write allocate” where the memory block is fetched into the cache on writes since, by the temporal locality principle, the memory block will probably soon be used again. Often hardware using write back will use write allocate and hardware using write through will use no write allocate to avoid complicating the hardware design. The different options are summarised in Table 5.1.

<table>
<thead>
<tr>
<th>Write Allocate</th>
<th>Write Back</th>
<th>Write Through</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Allocate</td>
<td>Write to cache now, write to memory later</td>
<td>Write to cache now, write to memory now</td>
</tr>
<tr>
<td>No Write Allocate</td>
<td>N/A</td>
<td>Only write to memory, update cache if needed</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of the different ways of handling writes. Write allocate/write back and no write allocate/write through are the commonly used.

The notation of caches and cache replacement policies have been inspired by [4, 15, 7, 14]. There are generally two types of cache designs: A unified cache and separate instruction and data caches. A unified cache is simpler and the space used for data versus the space used for instructions will be balanced without additional work. Separate instruction and data caches can provide faster access times since they can be accessed simultaneously. Furthermore, it is possible to adjust the parameters of the caches for different needs. For instance, it is uncommon to modify instructions and therefore they need not be written back to main memory. Due to the different advantages, it is common to have both separate instruction and data caches and a unified cache, where the latter stores both instructions and data memory blocks. This is done by having multiple levels of caches. For instance, processors often have separate level 1 (L1) caches and then have one or more unified larger and slower L2/L3/... caches. If more than one level of caches are used, another property becomes important: Are the caches exclusive or inclusive? An inclusive cache means that e.g. the L2 cache contains everything the L1 cache contains, plus some more. An exclusive cache means that the content of the caches are totally disjunct.
5.1.1 Cache Replacement Policies

Cache replacement policies are used for choosing which memory block in the cache to evict. Making an optimal decision for this can be hard as this requires knowledge of which memory blocks will be used in the future execution. Based on the temporal locality principle, a good policy keeps recently used memory blocks in the cache and evicts the least recently used memory block. This is exactly what the LRU replacement policy does. A nice property of this policy is that unused memory blocks will eventually be removed from the cache. This property also holds for the FIFO policy. It does, however, not hold for PRR and PLRU.

An example demonstrating the replacement policies will be given along with a formal definition of the different replacement policies. In the definitions of the replacement policies, a cache is considered a set \( L = \{l_1, l_2, \ldots, l_k\} \) of cache lines, and \( M = \{m_1, m_2, \ldots, m_n\} \) is the set of all memory blocks. We extend \( M \) with the empty element \( I \) which represent a cache line which does not contain any memory block: \( M' = M \cup \{I\} \). The element is also called an invalid cache line. The capacity \( k \) and associativity \( A \) is usually a power of two and will in the following be assumed to be so. Furthermore, a concrete cache state is defined:

**Definition 12 (Concrete Cache State).** A concrete cache state is a mapping \( c : L \to M' \), and \( C_c \) is the set of all concrete cache states.

For each of the replacement policies a concrete update function is defined which describes the effect of a memory access to the memory block \( m \) with the concrete cache state \( c \).

The order of the cache lines is used to model the age of memory blocks in the concrete cache state for LRU and FIFO cache replacement policies. The least recently used memory block is placed in cache line \( l_1 \). For all policies but PRR the policy is described for a fully associative cache. For PRR, the definition is given for an \( A \)-way set associative cache. The interested reader can find a formal definition of an \( A \)-way set associative LRU policy in [4].

LRU is often considered the ideal policy, and other replacement policies are usually compared against LRU. The LRU replacement policy can be split into two cases: cache hits and cache misses. In case of a cache hit the memory block accessed should be marked as the most recently used memory block. In the second case the memory block that is accessed is not in the cache and the least recently used memory block should be evicted. An example of the first case that demonstrates the concrete LRU update function can be seen in Figure 5.2 on the facing page. The figure shows a concrete cache state \( c \), which is updated to a new concrete cache state \( c' \) when the memory block \( m_2 \) is accessed. It should be noticed that \( m_2 \) is the second oldest memory block in \( c \). As \( m_2 \) is accessed again it should be the youngest memory block in \( c' \). Another result is that the memory blocks that are younger than \( m_2 \) in \( c \) should have their age increased by one in \( c' \). The age of memory blocks that are older than \( m_2 \) in \( c \) keep their age in \( c' \).

The second case can be seen in Figure 5.3 on the next page. The figure shows how a concrete cache state \( c \) is updated when the memory block \( m_5 \), that is not in \( c \), is accessed: a cache miss. In \( c \), \( m_3 \) is the youngest
Concrete cache state $c$

\[
\begin{array}{c|c}
 l_1 & m_3 \\
 l_2 & m_1 \\
 l_3 & m_2 \\
 l_4 & m_4 \\
\end{array}
\]

$U_{LRU}(c, m_2)$

\[
\begin{array}{c|c}
 l_1 & m_2 \\
 l_2 & m_3 \\
 l_3 & m_1 \\
 l_4 & m_4 \\
\end{array}
\]

Concrete cache state $c'$

\[
\begin{array}{c|c}
 l_1 & m_3 \\
 l_2 & m_1 \\
 l_3 & m_2 \\
 l_4 & m_4 \\
\end{array}
\]

Figure 5.2: Update of cache set state $c$ to $c'$ in the case of a cache hit. LRU replacement policy is used to update the ages of memory blocks in the cache.

Concrete cache state $c$

\[
\begin{array}{c|c}
 l_1 & m_3 \\
 l_2 & m_1 \\
 l_3 & m_2 \\
 l_4 & m_4 \\
\end{array}
\]

$U_{LRU}(c, m_3)$

\[
\begin{array}{c|c}
 l_1 & m_5 \\
 l_2 & m_3 \\
 l_3 & m_1 \\
 l_4 & m_2 \\
\end{array}
\]

Concrete cache state $c'$

\[
\begin{array}{c|c}
 l_1 & m_3 \\
 l_2 & m_1 \\
 l_3 & m_2 \\
 l_4 & m_4 \\
\end{array}
\]

Figure 5.3: Update of cache set state $c$ to $c'$ in the case of a cache miss. LRU replacement policy is used to choose which memory to evict.

memory block and $m_4$ is the oldest. Since $m_5$ is not already in the cache, and $m_4$ is the oldest, $m_4$ will be evicted from the cache and $m_5$ will be the youngest memory block in $c'$.

**Definition 13 (Concrete LRU Cache Update Function).** The update function $U_{LRU} : C_c \times M \to C_c$ takes a concrete cache state and a memory block as input and produces a concrete cache state as output. $U_{LRU}(c, m) = c'$, where $c'$ is defined as:

\[
c' = \begin{cases} 
[l_1 \mapsto m, \\
 l_i \mapsto c(l_{i-1}) \mid i \in \{2, \ldots, h\}], & \text{if } \exists h : c(l_h) = m \\
[l_1 \mapsto m, \\
 l_i \mapsto c(l_{i-1}) \mid i \in \{h + 1, \ldots, A\}], & \text{otherwise.}
\end{cases}
\]

FIFO is a simpler replacement policy than LRU. Therefore, the main advantage of FIFO is that the update logic is cheaper to implement. The update logic can be implemented with a round-robin counter for each cache set that points to oldest memory block. That said, FIFO is quite similar to LRU. It can also be split in two similar cases, one for cache hits and one for cache misses. An example of the first case, where a memory block in the cache is accessed, can be seen in Figure 5.4 on the following page. Since FIFO only uses a counter to represent the oldest memory block, $c$ and $c'$ are equal on cache hits.

The second case for cache misses of a memory block $m$ is similar to that of LRU. The oldest memory block will be evicted from the cache, and $m$ will become the youngest memory block.

43
Definition 14 (Concrete FIFO Cache Update Function). The update function \( U_{\text{FIFO}} : C_c \times M \to C_c \) takes a concrete cache state and a memory block as input and produces a concrete cache state as output. \( U_{\text{FIFO}}(c, m) = c' \), where \( c' \) is defined as:

\[
c' = \begin{cases} 
  c, & \text{if } \exists l_h : c(l_h) = m \\
  [l_1 \mapsto m, \ l_i \mapsto c(l_{i-1}) \ | \ i \in \{2, \ldots, A\}], & \text{otherwise.}
\end{cases}
\]

PRR uses a round-robin counter like FIFO, however, in PRR, there is only a single counter for the entire cache. This counter therefore points to the way that will have memory blocks evicted next. PRR can be split in three cases: cache hits, cache misses (where there is an invalid way in the cache set), and other cache misses.

The case of a cache hit is, like FIFO, trivial, as it will not change the counter nor the concrete cache state. The second case is demonstrated in part (a) of Figure 5.5 on the next page. The figure shows a concrete cache state \( c \) of a 2-way set associative cache. The cache state \( c \) is obtained by doing a sequential read of \( m_1, m_2, \ldots, m_8 \), where \( c \) only contained invalid cache line before the sequential read. It is clear that \( m_1 \) is inserted in way 1 of cache set \( s_1 \). To understand this please note that \( \text{cacheSet}(m_1) = \{l_1, l_2\} \) and, since both \( l_1 \) and \( l_2 \) are invalid, \( l_1 \mapsto m_1 \). The counter is not modified in this case either. The third case is a cache miss of a memory block \( m \). This will evict the memory blocks in the first cache line in \( \text{cacheSet}(m) \) pointed to by the counter.

Part (b) of Figure 5.5 on the facing page shows what happens if a sequential read of \( m_9, m_{10}, m_{11}, m_{12} \) is performed. Notice that continually doing sequential reads would never evict the non-bold memory blocks, resulting in half of the cache containing data that is not used.

To represent the value of the counter we introduce a string of bits \( b \), where \( |b| \) represents the length of the bit string \( b \) and \( B_x \) is the set of all bit strings with length \( x \). To represent individual bits in the string we will use the notation \( b_y \) to represent the \( y \)th bit of \( b \). The bit string \( b \) can thus be written as \( b = b_0 b_1 \ldots b_n \). For PRR caches, we have that \( 2^{|b|} = A \). We will use \( \text{dec}(b) \) to denote the decimal value of the bit string \( b \) and \( \text{bin}(d, q) \) to get the bit string of length \( q \) to representing the decimal value \( d \). We assume decimals are stored in Most Significant Bit First (MSB) first order.
We define the function:

$$\text{minCL}(m, c) = \{l_i \mid l_i \in \text{cacheSet}(m), c(l_i) = I, \forall l_j \in \text{cacheSet}(m) : j > i \Rightarrow c(l_j) \neq I\},$$

which gives the first empty cache line in the cache set of the memory block $m$ in the concrete cache state $c$.

**Definition 15 (Concrete PRR Cache Update Function).** The update function $U_{\text{PRR}} : C_c \times B_{\log(A)} \times M \rightarrow C_c \times B_{\log(A)}$ takes a concrete cache state, a memory block and a bit string as input and produces a concrete cache state and bit string as output. $U_{\text{PRR}}(c, b, m) = (c', b')$, where $c'$ is defined as:

$$c' = \begin{cases} 
    c, & \text{if } \exists l_h : c(l_h) = m \\
    [l_g \mapsto m, \quad] & \\
    l_i \mapsto c(l_i) \mid i \in \{1, \ldots, k\} \setminus \{g\}, & \text{if } \forall l_h : c(l_h) \neq m \wedge \\
    l_g \in \text{minCL}(m, c) & \\
    [l_j \mapsto m, \quad] & \\
    l_i \mapsto c(l_i) \mid i \in \{1, \ldots, k\} \setminus \{j\} \}, & \text{otherwise, with} \\
\end{cases}$$

and $b'$ is defined as:

$$b' = \begin{cases} 
    b' = b, & \text{if } \exists l_h : c(l_h) = m \\
    b' = b, & \text{if } \forall l_h : c(l_h) \neq m \wedge \\
    l_g \in \text{minCL}(m, c) & \\
    b' = \text{bin}((\text{dec}(b) + 1) \mod 2^{|b|}, |b|) & \text{otherwise.}
\end{cases}$$

**PLRU** is an approximation of LRU using fewer bits but giving results close to those of LRU. PLRU can be split in three cases like PRR: cache hits, cache misses (where there is an invalid way in the cache set), and other cache misses.

---

(a) Concrete cache state $c$

<table>
<thead>
<tr>
<th></th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ way 1</td>
<td>$m_1$</td>
<td>$m_2$</td>
<td>$m_3$</td>
<td>$m_4$</td>
</tr>
<tr>
<td>way 2</td>
<td>$m_5$</td>
<td>$m_6$</td>
<td>$m_7$</td>
<td>$m_8$</td>
</tr>
</tbody>
</table>

(b) Concrete cache state $c'$

<table>
<thead>
<tr>
<th></th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ way 1</td>
<td>$m_9$</td>
<td>$m_2$</td>
<td>$m_1$</td>
<td>$m_4$</td>
</tr>
<tr>
<td>way 2</td>
<td>$m_5$</td>
<td>$m_{10}$</td>
<td>$m_7$</td>
<td>$m_{12}$</td>
</tr>
</tbody>
</table>

Figure 5.5: 2-way set associative cache with PRR as replacement policy. The arrow indicates the way the pointer is currently pointing at. The example is inspired by [15].
For PLRU a binary tree is used to keep track of the ages of memory blocks. An example of this can be seen in Figure 5.6.

Like for PRR we use a string of bits $b$, however, for PLRU this bit string represents the binary tree. We use the notation $b = b_1b_2b_3b_4b_5b_6b_7$ to specify the value of a bit string of length seven. For PLRU the length of this bit string is $|b| = A - 1$.

The two first cases of PLRU are similar to those of PRR, except the binary tree is updated. The last case is a bit different, since a memory block needs to be evicted. To choose which, the binary tree is used to find one of the oldest memory blocks in the cache. As mentioned, the binary tree is used to keep track of the ages of the memory blocks in the cache; how this is done will be demonstrated by an example and later by a formal definition.

As can be seen in the figure, there is a bit associated to each tree node. The value of the bit associated to a node decides which of the node’s children to look at. For instance, the bit string $b = 0100100$, where $b_1 = 0$, means that we need to look at $b_2$, and as $b_2 = 1$, we need to look at $b_5$. Finally, the value of $b_5$ is 1, so we need to look at $l_4$. After a memory block in $l_4$ has been used, we flip the bits $b_1$, $b_2$ and $b_5$, thereby pointing to another cache line. To flip a bit, we define the function $flip(b_x) = \overline{b_x}$.

We overload the function $flip$ to be able to selectively flip a set of bits, such as $flip(b, S)$, where $b$ is a bit string and $S$ is the set of bits in $b$ to be flipped. To get the level in the hierarchy a bit is at, we define the function $level(i) = \lceil \log_2(i) + 1 \rceil$. Furthermore, we define a “points-to” function $ptr(b) = ptr_r(b, 1)$ that, given $b$ as defined before, gives the cache line index 4, as explained above. The function uses the recursive function $ptr_r$ defined by:

$$ptr_r(b, i) = \begin{cases} 1, & \text{if } i > |b| \\ \frac{A}{2^{level(i)}} \cdot dec(b_i) + ptr_r(b, i \cdot 2 + dec(b_i)) & \text{otherwise.} \end{cases}$$

The intuition behind the $ptr_r$ function is to traverse the bit-tree, and calculate how much the current node “contributes” to the summation of the final line-index. If the bit currently being looked at is 0, then the contribution is also 0, as we then go left in the tree. If the bit currently being looked at is 1, then the contribution depends on which level in
the tree the bit is situated: The top-most bit contributes $\frac{1}{2}$, because it effectively constrains the search to the rightmost half of the cache lines. The second level bits contributes $2^{-2}A$ as this level decides which quarter of the cache lines to search, and so on.

In the example it is not necessary to look at all bits in the bit string, only the three bits $b_1$, $b_2$ and $b_5$. These bits are referred to as the significant bits of $b$. We define a helper function $\text{signifBits}(b, j) = \text{signifBits}_r(b, 1, j)$ that gives the significant bits of a cache line index and a bit string, for instance $\text{signifBits}(b, 4)$ would give the set $\{b_1, b_2, b_5\}$. The function uses the recursive function $\text{signifBits}_r$ defined by:

$$\text{signifBits}_r(b, i, n) = \begin{cases} \{b_i\}, & \text{if level}(i) = \text{level}(b) \\ \{b_i\} \cup \text{signifBits}_r(b, i \cdot 2^+ \text{bin}(n - 1, \log_2 |b|), \text{level}(i)), & \text{otherwise.} \end{cases}$$

The intuition is that if we are at the final level, the current bit is the last significant bit, so the recursion stops. Otherwise we inspect the bit string defined by the cache line we are searching for and follow the path it designates through the tree. For example, $\text{bin}(4 - 1, \log_2 8) = \text{bin}(3, 3) = 011$, and following that path in Figure 5.6 on the facing page leads one through the bits: $\{b_1, b_2, b_5\}$, which corresponds to the walk: go left, go right, go right.

**Definition 16 (Concrete PLRU Cache Update Function).** The update function $U_{\text{PLRU}} : C_c \times B_{A-1} \times M \to C_c \times B_{A-1}$ takes a memory block, bit string, and a concrete cache state as input and produces a concrete cache states and bit string as output. $U_{\text{PLRU}}(c, b, m) = (c', b')$, where $c'$ is defined as:

$$c' = \begin{cases} c, & \text{if } \exists l_h : c(l_h) = m \\ [l_j \mapsto m, \ \ l_i \mapsto c(l_i) \mid i \in \{1, \ldots, A\} \setminus \{g\}], & \text{if } \forall l_h : c(l_h) \neq m \land l_g \in \text{minCL}(m, c) \\ [l_j \mapsto m, \ \ l_i \mapsto c(l_i) \mid i \in \{1, \ldots, A\} \setminus \{j\}], & \text{otherwise, with } j = \text{ptr}(b), \end{cases}$$

and $b'$ is defined as:

$$b' = \begin{cases} \text{flip}(b, \text{signifBits}(b, h)), & \text{if } \exists l_h : c(l_h) = m \\ \text{flip}(b, \text{signifBits}(b, g)), & \text{if } \forall l_h : c(l_h) \neq m \land l_g \in \text{minCL}(m, c) \\ \text{flip}(b, \text{signifBits}(b, j)), & \text{otherwise, with } j = \text{ptr}(b). \end{cases}$$

In Table 5.2 on the next page an example for PLRU, inspired by [15], is given. The example shows that unused memory blocks might stay forever in a cache. A fully associative cache with eight cache lines is shown, where cache line $l_1$ is be repeatedly used, and cache lines $l_5, \ldots, l_8$ are being replaced with new memory blocks as a consequence of cache misses.
Table 5.2: Example of PLRU showing that unused memory blocks in \( l_2 \), \( l_3 \) and \( l_4 \) might stay in the cache forever. This is done by showing a sequence of memory accesses that start and end in the same state without evicting the lines in question.

### 5.2 Pipelining

Pipelining is a well-known concept for instance in the car industry where cars are assembled on a pipeline with specialised workers or robots carrying out the same action repeatedly. To understand how this can be applied to a simple processor it should be realised that executing a single instruction can easily take more than one cycle. For instance executing an instruction with two operands, on the simple processor modelled in Figure 5.7 on the facing page, is done by setting the instruction type on the ALU operation line and waiting for the operands to be made available in Data register1 and Data register2. When the operands are made available, the instruction is be executed. After the instruction has been executed, and as soon as any previous result in the write register has been written to cache or main memory, the result is stored in the write register. Let us assume that fetching an operand from the cache takes one cycle, executing the instruction takes one or two cycles depending on the instruction type, and writing the result takes one cycle. In total this is four to five cycles for executing a single instruction.

Applying the concept of pipelining to processors means that the tasks of the processor are split into stages which are then carried out consecutively. An example of stages could be “Instruction Fetch”, “Instruction Decode”, “Operand Fetch”, “Instruction Execution”, and “Write Back”. All stages are performed in parallel and are designed to only take one cycle each. In Figure 5.8 on page 50 a five stage pipeline is shown.

The advantage of using pipelining is an overall increase of the number of instructions executed per cycle. For instance, if each stage takes one cycle and pipelining is not used, it takes five cycles to execute one instruction. Using a five stage pipeline an instruction can be completed each cycle. This can be seen in Table 5.3 on the facing page.

At this point it might seem tempting to increase the number of pipelines, however, this will usually not yield a much better performance. The reason is that in practice instruction prefetching might not work as planned and instead introduce pipeline stalls. A pipeline stall is a situation where the pipeline cannot
Table 5.3: The first six cycles of a five stage pipeline.
perform useful work but instead performs \textbf{NOPs} (No OPerations).

```c
1  ...  
2  \textbf{while} (i != 42) { 
3      \textbf{if} (j == 1) { 
4          j = 42;  
5      } 
6  i++;  
7  } 
8  ...  
(a)
```

The code in Figure 5.9 introduces a pipeline stall. Table 5.4 on the facing page shows what happens in the pipeline for each cycle. In the two first cycles everything is fine, but in cycle 3 a \textbf{NOP} is fetched in the Instruction Fetch stage. The reason to have \textbf{NOPs} in the assembly code is to keep the program correct. The problem is that the pipeline fetches the \textbf{NOP} before it decodes the \textbf{BNE} (Branch if Not Equal) instruction, and then realises that it might actually need to fetch the instruction at line 13 of part b of Figure 5.9 instead. Rather than always performing \textbf{NOPs} after branches, a better idea is, of course, to perform useful work. This technique is called a delay slot. A delay slot is an instruction that is placed after branches, and the instruction is executed regardless of the branch. It is then the task of the compiler to try to place a useful instruction in

Figure 5.8: Five stage pipeline.

Figure 5.9: Prefetching example.
Table 5.4: Example of pipeline stall as a result of branching.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CMP i, 42</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BNE Next</td>
<td>CMP i, 42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>NOP</td>
<td>BNE Next</td>
<td>CMP i, 42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>NOP</td>
<td>NOP</td>
<td>BNE Next</td>
<td>CMP i, 42</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 5.4: Example of pipeline stall as a result of branching.

the delay slot or a NOP, in case no useful instruction can be placed there. In case of conditional branches such as BNE, delay slots are not the only problem, since the pipeline has to stall and wait for the result of executing CMP in the Instruction Execution stage. Before CMP can be executed, the operands (in this case i) need to be fetched from the cache or main memory. In Table 5.4 i is assumed to be in the cache. Even though there is a cache hit, the Instruction Fetch stage in cycle 4 still does not know the result of CMP, which means it stalls and has to insert a NOP instruction. If the pipeline had even more stages this would result in stalling for even more cycles. One technique to decrease the number of pipeline stalls is branch prediction.

5.3 Branch Prediction

Branch prediction is an optimisation technique which can be used to avoid pipeline stalls. We introduce it through a small example. In Figure 5.10 a code snippet and the corresponding assembly code can be seen. If this code is executed on a pipelined processor, and the variable i is not cached, the pipeline has to stall until i is fetched from memory, since the processor does not know whether to prefetch the instruction in either line 4 or 6.

1 ... 
2 if (i == 0) {
3   j = 42;
4 } else {
5   k = 42;
6 }
7 ...

(a)

(b)

Figure 5.10: Code snippet and corresponding assembly code.

A small detail about line 5 in part b of Figure 5.10 is that BR (BBranch) is an unconditional branch. Unconditional branches in a pipelined processor are, like other instructions, fetched by the “Instruction Fetch” stage in one cycle and decoded in the “Instruction Decode” stage in the next cycle. As the destination address of the branch is not known before the branch instruction has been decoded, the “Instruction Fetch” stage fetches the instruction immediately after the branch instruction (line 6). In this case the pipeline stalls, as the fetched instruction is useless for the subsequent execution and is thus not decoded.
If the value of $i$ in line 2 could somehow be predicted, it would be possible to prefetch the instructions of line 4 or 6. This is, however, not possible in general. Instead of giving up this has given rise to a number of different branch prediction techniques.

One idea is to statically predict that branches are never taken. This only gives rise to problems when branches are indeed taken, and in this case the computation would have to be “undone”. Two different techniques exist for this, where both are complex. A common rule is: if the target of a conditional branch is backward then the branch is taken. The reason is that this indicates a loop and loops are generally iterated several times. On the other hand, if the target of a conditional branch is forward the branch is not taken. The reason for this is that forward branches are used in error checking and errors rarely happen. This prediction is, however, not as good as the one for backward branches, as many forward conditional branches are not related to error checking.

Since conditional branches often are inside loops, a noticeable performance decrease can result if branches are repeatedly mispredicted. To avoid this, dynamic branch prediction has been designed to learn from past experiences. A common technique for learning is to use a history table as seen in Table 5.5.

<table>
<thead>
<tr>
<th>Valid</th>
<th>Address</th>
<th>Prediction bit(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 5.5: History table.

5.4 Out-of-Order Execution

Another technique to avoid pipeline stalls is to execute instructions out of order. In effect, the constraint that instructions should be executed in sequential order is no longer valid, although the sequential semantic must be maintained. For instance, in the code snippet in Figure 5.11 on the next page, line 2 might lead to a pipeline stall if the memory block containing $i$ is not in the cache. If the memory block containing $j$ is in the cache, it would be more efficient to execute line 3 before line 2.

Out-of-order execution require the pipeline to be redesigned. An overview of new design can be seen in Figure 5.12 on the facing page, where the Instruction Fetch and Instruction Decode stages have been collapsed. Another change is that the design usually has two or more Instruction Execution units that perform parallel out-of-order execution. The collapsed stage works in-order and makes a “pool” of instructions available to the Instruction Execution units. After instructions have been executed, the results are committed back to the cache or main memory. This is done in the commit stage which is in-order.
Figure 5.11: Assembly code snippet demonstrating out-of-order execution where \( r_x \) are registers.

The operand stage has been renamed to Reservation Station. The Reservation Station is responsible for making operands available and resolving conflicts.

Figure 5.12: Overview of out-of-order execution processor design.

Executing instructions out-of-order and in parallel gives raise to three types of data dependencies. The data dependencies are demonstrated in the code in Figure 5.11:

- **Read After Write (RAW)** can be seen in line 3 and 4, where \( r_3 \) will be written to in line 3 and the result is read in line 4. The RAW dependency blocks out-of-order execution of line 4 before line 3.

- **Write After Read (WAR)** can be seen in line 4 and 5, where \( r_2 \) needs to be read in line 4 before it can be written in line 5.

- **Write After Write (WAW)** can be seen in line 4 and 6, as both line 4 and 6 write their result to register \( r_1 \), but the result of line 6 must be the last to be written to \( r_1 \).

The reader might have noticed that both WAR and WAW could be overcome by having additional so-called “secret” registers to store the result in. For instance, the WAR example could be solved by first copying the value of \( r_2 \) to...
another register, that would be used by the instruction in line 4. This would allow both instructions to be executed out of order. This technique is generally known as register renaming and is performed by the Instruction Fetch and Decode stage, Reservation Stations, and the Commit stage. The technique is able to eliminate many WAR and WAW dependencies [25, p. 280].

5.5 Speculative Execution

Branch prediction and out-of-order execution might not be enough to avoid pipeline stalls. For instance, out-of-order execution does not work well if there are many branches and the basic blocks are small. Branch prediction does not always help enough, if, for instance, a memory block needs to be fetched from main memory or a long floating point operation has to complete before the result of a conditional branch is known.

Speculative execution can improve the average performance of execution. This is done by executing instructions without knowing if they are actually going to be executed. This might, however, also result in slowdowns. For instance, if speculation results in execution of instructions that require memory blocks that are not in main memory, thereby causing a page fault, as the memory blocks have been swapped out to disk. To prevent this scenario, modern processors have special speculative instructions which only use cached memory blocks.

Another problem with speculative execution is that executing instructions without strictly obeying checks in conditional branches might cause overflows or exceptions that would not otherwise happen. This is an undesirable situation, and it needs to be resolved in hardware. A way to do this is to add a poison bit to each register and set the bit when a register stores speculative results and only raise exceptions when it is confirmed that an exception truly will happen.
Chapter 6

Timing Anomalies

A processor exhibits timing anomalies when the local execution time of a single instruction has a counter-intuitive influence on the global execution time of an entire program [12]. For instance, a cache miss, rather than a cache hit, at a particular point of execution might yield a shorter program execution time. Timing anomalies are of crucial relevance to WCET analysis, as this task in general assumes that local worst-case decisions produce the global WCET. This chapter introduces timing anomalies in detail, discusses their impact on WCET analysis and presents methods for dealing with processors that exhibit timing anomalies.

We will introduce timing anomalies through a concrete example [19], where a cache hit triggers an overall longer execution time than a cache miss. The execution takes place on a simplified processor model with three resources: a Load-Store Unit (LSU), an Integer Unit (IU) and a Multi-Cycle Integer Unit (MCIU). Table 6.1 contains a sequence of instructions that will be executed on the model. The second column shows in which cycles the instructions are dispatched. The LOAD instruction uses the LSU, the ADD uses the IU, and the MUL uses the MCIU. The IU permits out-of-order execution, whereas the LSU and the MCIU do not. The left-most register is the destination register, whereas the other registers are source registers.

The instructions’ use of registers makes them dependent. For example, the LOAD instruction must have loaded a value into register r4, before the subsequent ADD is able to use the register for addition. Using labels, it is clear that B depends on A, D depends on C, and E depends on D. All the dependencies are RAW dependencies. The LOAD instruction executes for 2 cycles, if there is a cache hit,

<table>
<thead>
<tr>
<th>Label</th>
<th>Disp. cycle</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>LOAD r4,0(r3)</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>ADD r5,r4,r4</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>ADD r11,r10,r10</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>MUL r12,r11,r11</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
<td>MUL r13,r12,r12</td>
</tr>
</tbody>
</table>

Table 6.1: Instructions for timing anomaly example [19].
and 10 cycles otherwise. On the other hand, cache contents are ignored for ADD and MUL, which execute for 1 and 4 cycles, respectively.

Figure 6.1 shows the execution of the instruction sequence, with and without a cache hit for the LOAD instruction. Consider the upper half of the figure. B is dispatched at cycle 2, but due to dependency execution of B has to wait until cycle 3, where A is done executing. At cycle 3, C is dispatched, but B is still executed first, as it has been waiting and therefore is older. In the figure’s lower half, A has a cache miss and delays the execution of B until cycle 11. As the IU permits out-of-order execution, C is executed in cycle 3, immediately after having been dispatched. The instruction sequence exposes a timing anomaly for the processor model, as the execution with a cache miss has a shorter overall execution time than the execution with a cache hit.

The example illustrated in Table 6.1 on the preceding page and Figure 6.1 is called a scheduling anomaly. Another example is a speculation anomaly, which occurs with branch prediction [22]. Recall that a mispredicted branch causes unnecessary instruction fetching that pollutes the cache. If the first instruction in the mispredicted branch is a cache miss, then the branch condition might be evaluated before the mispredicted branch can cause more harmful fetches. This anomaly is illustrated in Figure 6.2.

A third type of timing anomalies, mentioned in [22], is cache timing anomalies. Contrary to the other two types, these are anomalies caused entirely by strange cache behaviour, i.e. no interaction with out-of-order execution, branch prediction or other optimisation techniques is required. It is notable that in-
order architectures, such as the Motorola ColdFire 5307, can exhibit speculation and cache timing anomalies, hence to avoid anomalies it is not enough to avoid out-of-order units [22, 26].

6.1 Formal Definition

Timing anomalies have not been well-defined until 2006, where a formal definition was given by Becker et al. in [22]. Their goal was to create a definition that is also able to cover the timing anomalies exhibited by future architectures. They begin with the following three observations regarding hardware model, abstraction and locality:

Hardware Model It is important that the definition take the hardware model into account, as it is the cause of timing anomalies, both with regard to the number of anomalies and the type. For example, processors with one or more out-of-order units probably suffer from more anomalies than processors without such units.

Abstraction It is necessary to abstract away some degree of detail to do feasible timing analysis. Abstraction makes is possible to deal with unknown input data and huge state-spaces, though the drawback unfortunately is that we have to settle with less precise answers. The idea is that the chosen level of abstraction dictates which types of timing anomalies can be covered by a definition.

Locality Identifying the local worst-case is a task that traditionally has not been formally defined and, for lack of anything better, was done intuitively. A notion of locality is needed for a formal definition of this task. In [22], Becker et al. argue that micro-operations (instruction fetch, execute, etc.) are the appropriate locality level, as this is the level, where timing differences first are exposed.

The observations inspire Definition 17-22 borrowed from [22]. In the context of timing analysis, a transition system is suitable to model the cycle-level behaviour of a computer architecture, as input and output are abstracted away.

The first definition enables us to consider the locality level of micro-operations when identifying the local worst-case.

Definition 17 (Locality Constraint). A locality constraint \( l \) for a transition system \( T = (S, R) \) is a convex predicate on \( S \), i.e. \( l \) only holds on consecutive states in any path \( \pi \) through \( T \). We assume that locality constraints model the sequence of states that is executing a micro-operation. We denote the restriction of \( \pi \) to \( l \) by \( \pi_l \), i.e. the restriction of the path \( \pi \) to the subpath of \( \pi \) in which \( l \) holds. Note, that this is still a path.

In the following, \( \circ \) denotes concatenation.

Definition 18 (Local Worst-Case Path). Given a set of locality constraints \( \mathcal{L} \) and a set of paths \( \Pi \), a path \( \pi \in \Pi \) is a local worst-case path, if and only
if for every locality constraint \( l \in \mathcal{L} \) and every path \( \pi' \in \Pi \) it holds that if 
\[
\pi = \pi_{\text{pre}} \circ \pi_l \circ \pi_{\text{post}}, \quad |\pi_l| > 0 \quad \text{and} \quad \pi' = \pi_{\text{pre}} \circ \pi_l' \circ \pi'_{\text{post}} \quad \text{then} \quad |\pi_l| \leq |\pi'_l|.
\]

A path is called a non-local worst-case path if it is not a local worst-case path.

**Definition 19** (Program). A program (or a control flow graph) \( P \) is a directed graph \( P = (V, E) \), \( E \subseteq V \times V \), in which the nodes \( V \) represent instructions, and an edge \((u, v)\) \( \in E \) represents flow of control from \( u \) to \( v \).

A sequence \( \sigma \) through a program \( P = (V, E) \) is a finite sequence of instructions, such that \((\sigma_i, \sigma_{i+1}) \in E \) for all \( i \in \{0, \ldots, |\sigma| - 1\} \).

Not all paths in the transition system correspond to a possible path through the modelled program, and we only want to compare paths that do. This is accomplished by a retirement function that maps paths in the transition system to paths through the program.

**Definition 20** (Retirement Function). A transition system \( T = (S, R) \) and a program \( P = (V, E) \) can be related by a retirement function \( \rho : S \rightarrow 2^V \) that maps each state of the transition system to the set of instructions of the program that is retired right after this state:

\[
\rho(s) = \{ \iota \mid \text{instruction } \iota \text{ is retired right after } s \}.
\]

We extend \( \rho \) to paths in the following way:

\[
\rho^* : S^* \rightarrow 2^V^* \\
\text{perm} : 2^V^* \rightarrow 2^V^*
\]

\[
\rho^*(s_1s_2\ldots s_n) = \text{perm}(\rho(s_1)\rho(s_2)\ldots \rho(s_n)) \quad \text{where} \quad \rho(s_i)_I = I_i \quad \text{or} \quad \rho(s_i)_I = I_i \setminus \{ \iota \} , \omega \in \text{perm}(I_1^I I_2^I \ldots I_n^I) \}
\]

A set of paths \( \Omega \) through a transition system \( T \) restricted to sequence \( \sigma \) (w.r.t. \( \rho \)) is defined as \( \Omega_\sigma^\sigma = \{ \pi \in \Omega \mid \sigma \in \rho^*(\pi), \rho(\pi_{|\pi|-1}) \neq \emptyset \} \).

**Definition 21** (Hardware Model). A (possibly abstracted) hardware model \( C \) maps a program \( P \) to a transition system \( T \), a set of locality constraints \( \mathcal{L} \) on \( T \), and a retirement function \( \rho \) that relates the states of the transition system with the instructions of the given program \( P \).

As mentioned before, a concrete hardware model is deterministic, and nondeterminism is necessary for timing anomalies. In [22] it is claimed that it is possible to formally define the relation between concrete and abstract hardware models, but the authors omit such a definition for reasons of brevity.

With the previous definitions in mind, timing anomalies can now be defined.

**Definition 22** (Timing Anomaly). A hardware model \( C \) exhibits timing anomalies, if there exists a program \( P \) with \( C(P) = (T, \mathcal{L}, \rho) \), a finite sequence \( \sigma \) through \( P \), and a non-local worst-case path \( \pi \in \Pi(T)_C^\sigma \), such that \(|\pi| > |\pi'| \) for all local worst-case paths \( \pi' \in \Pi(T)_C^\sigma \).

The definitions give rise to the situation illustrated in Figure 6.3 on the next page. In the analysis of a transition system, the path \( \pi_{\text{pre}} \) has been executed,
and future execution is non-deterministic. If a hardware model exhibits timing anomalies, it is necessary to follow the non-local worst-case path $\pi_l$ to find the globally longest path. As it is non-local, the path $\pi_l$ is not the longest path on locality constraint $l$ (see Definition 18 on page 57), and it symbolises e.g. a cache hit.

6.2 Methods for Elimination of Anomalies

In [19], the primary source for this section, Lundqvist and Stenström present two methods for eliminating timing anomalies; the pessimistic serial-execution method and the program modification method. The methods enable us to estimate the WCET of programs running on dynamically scheduled processors, where timing anomalies might arise.

6.2.1 The Pessimistic Serial-Execution Method

This method is straight-forward compared to the program modification method presented in Section 6.2.2, as it does not try to model the execution more accurately, it just models all instructions as being executed in-order in the processor’s units. For instance, in the example presented in Table 6.1 on page 55 and Figure 6.1 on page 56, the IU would be forced to execute instructions in-order. WCET is then determined by summing all instruction latencies in the units and then adding the miss penalties for all instruction and data cache misses.

Lundqvist and Stenström put forward a claim, which makes the method usable. We present the claim as a theorem, as Lundqvist and Stenström provide a proof for its correctness:

**Theorem 5.** The WCET corresponding to a serial execution of an instruction sequence, assuming the instructions’ worst-case latencies, is always higher than the WCET corresponding to any pipelined execution of the same instruction sequence.

**Proof.** Instructions cannot execute slower than in-order since this would mean that some functional units in the processor are idle sometime. This cannot be
true since instructions are always available for execution. The only possibility for an instruction to stall is cache misses which we add separately.

Clearly, the estimate produced by this method is safe but it might also be very pessimistic. This does not necessarily pose a problem, as safety is typically more important than very fast execution in the domain of hard real-time systems [9].

6.2.2 The Program Modification Method

The serial-execution method presented in the previous section produces very pessimistic WCET estimates, and if tighter estimates are desired, it is necessary to model pipelined execution accurately and deal with timing anomalies. This is what the program modification method accomplishes by modifying the program such that it is safe for a subsequent analysis to rely on local worst-case assumptions. According to Lundqvist and Stenström, the modification must ensure that the following two conditions are true during a simulation of the resulting program:

1. All instructions with variable latency must still lead to a predictable pipeline state. Moreover, the worst-case latency must be used for all instructions, and unknown events such as accesses to the instruction cache must also lead to a predictable pipeline state.

2. If a reduction of the number of paths in a small section of the program is done by choosing the longest path or removing the shortest paths, then the state of the pipeline and the cache before and after the paths must be the same.

The first condition can be satisfied by forcing the processor’s units to do in-order execution when executing the variable-latency instructions. Unfortunately, this is not supported directly in processors today. Processors might, however, have special instructions that are able to force equivalent conditions. For example, the sync instruction in the PowerPC architecture can be inserted on both sides of a variable-latency instruction to ensure

- in-order execution,
- that the maximum latency will be the worst-case latency, and
- that the resulting pipeline state is predictable.

The second condition is more difficult to satisfy, as not only the pipeline state but also the cache state must be handled. The pipeline state is again forced to be predictable using sync in the PowerPC or equivalent functionality in other architectures. To be able to compare two paths, it is necessary to manipulate the corresponding caches, such that the paths execute under the same circumstances. This is architecture dependent, and there are three options available:

1. Invalidate all blocks in the caches. This is possible on almost all processors.

2. Invalidate only the blocks that differ in the two caches. Support for invalidation on the block level is required for this.
3. Replace blocks that differ with blocks that will be needed in the future. Support for explicitly loading blocks into a cache is required for this.

The first two options are not attractive solutions, as it is likely that the performance will suffer too much. The third option, on the other hand, is appealing, as the preloaded instructions are likely to cancel out the performance drop caused by overwriting the differing blocks. Again, the choice of architecture is important, as the third option requires special instructions to preload the cache. Lundqvist and Stenström do not mention how common it is for architectures to include such instructions, but they mention that instruction and data cache block touch instructions (icbt and dcbt) exist in the PowerPC.

6.2.3 Method Limitation

Unfortunately, the methods presented in Sections 6.2.2 and 6.2.1 for eliminating timing anomalies introduce performance drops. Even small performance drops could entail that more expensive processors must be used for a particular task. Real-time systems are often used in embedded systems, which are produced in large numbers, and even a two dollar more expensive processor can result in large expenses.
Chapter 7

Related Work on WCET Analysis

In this chapter we review how other authors have conducted WCET analysis. Specifically, we look at the challenges of using abstract interpretation and model checking in Sections 7.1 and 7.2, respectively.

7.1 WCET Analysis using Abstract Interpretation

In [4, 14, 15] the authors present a number of complementary approaches to find the WCET using abstract interpretation as the underlying technique. The task of finding the WCET of a process is done by using several individual analyses. These analyses are typically performed on the machine code of the process, as this is the only level that contains sufficient information about memory locations. The information is needed for e.g. the cache analysis to predict which memory locations that might be in the same memory block. They have also investigated how the analyses can be integrated in different ways and thereby produce more precise answers.

The different abstraction levels are illustrated in Figure 7.11. In the high-level program neither the instructions nor the values of their operands are known. At the assembly level the machine instructions are known, but their operands are typically only referred to symbolically (using labels such as “L2”), thus their actual addresses are not known. At the machine code (sometimes known as “object”) level the actual bits that are fed into the processor can be seen, and therefore the addresses are readable (with some effort).

The analyses are:

Value Analysis is used to predict memory addresses of memory blocks in the data cache, since these often are not known statically but only at runtime. The most basic example is dereferencing a pointer: to know which memory address is accessed, one must know the value of the pointer. Value analysis approximates this information by giving a potential superset of the values

---

1The machine code is correct x86 instructions for the factorial function’s assembly.
Figure 7.1: The different abstraction levels in the program compilation process.

the pointer can have. Value analysis is also used to determine bounds on
loops. In some cases loop bounds cannot be detected and must then be
annotated.

Path Analysis is used to find the worst execution path (in terms of time usage)
through the control flow of the process. This requires the control flow of
the process to be reconstructed from the machine code.

Processor Behaviour Analysis models the behaviour of the processor and
related subsystems. This includes modelling processor features such as
 caches, pipelines, branch prediction, out-of-order execution, and specula-
tive execution.

An important part of the processor behaviour analysis is the cache analysis.
The cache analysis is used to predict if a memory block is in the cache, might
be in the cache, or if it is definitely not in the cache. It is typically split into
two separate analyses: a must and a may analysis. The analyses are important
because they can give a large improvement to the WCET estimates [23, p. 8].

7.1.1 Cache Analysis

The cache analysis works on the CFG of the program. Each node in the CFG
is assigned an abstract cache state, which is a representation of the various
concrete cache states that the processor might be in, at that execution point.
In this section we present a definition of an abstract cache along with may and
must analyses for the LRU and FIFO cache replacement policies. The definition
of LRU and examples are based on [4, 14, 15]. We start by defining joint functions
and abstract update functions.

Definition 23 (Abstract Cache state). An abstract cache state is a function
\( c : L \to 2^M \), where \( L \) is the set of cache lines, and \( M \) is the set of memory
blocks. The mapping indicates the maximal age of an item in the cache. \( C_c \)
denotes the set of all abstract cache states. We use \( A \) to denote the number of
cache lines, \( |L| \).

For example, assuming two cache lines, 1 and 2, and two store elements, \( m_1 \)
and \( m_2 \), the abstract cache \( \{1 \to \emptyset, 2 \to \{m_1, m_2\}\} \) is a representation of the
concrete caches \( \{1 \to m_1, 2 \to m_2\} \) and \( \{1 \to m_2, 2 \to m_1\} \).
When a CFG node has more than one incoming transition, a join function is used to determine a sound approximation of the cache contents at this CFG node.

Definition 24 (Join Function). A join function is a mapping from two abstract cache states to single abstract cache state:

\[ \text{JOIN} : \hat{C}_c \times \hat{C}_c \rightarrow \hat{C}_c. \]

As mentioned before, a cache analysis is usually split into two separate analyses: a must analysis that is used to predict which memory blocks are definitely in the cache and a may analysis used to predict which memory blocks are definitely not in the cache. Join functions will be defined for both must and may analyses.

**Must Analysis**

The must analysis is fundamental to cache analysis as it provides information on which memory block are in the cache at a given execution point. This allows sharpening of the results of a WCET analysis by predicting definite cache hits.

Before presenting abstract update functions, we present a join function that is used for both LRU and FIFO replacement policies.

Definition 25 (LRU/FIFO Must Join Function). For LRU and FIFO replacement policies the join function is defined as

\[ \text{JOIN}(\hat{c}_1, \hat{c}_2) = \{ l_i \mapsto \{ m \} \mid y \in X \iff \exists j, k : y \in \hat{c}_1(l_j) \land y \in \hat{c}_2(l_k) \land i = \text{max}(j, k) \} \]

That is, the age a memory block in the cache can have at a node, is the maximal age the item has in its predecessors.

In the following we present abstract update functions for the replacement policies.

LRU’s abstract must update function is quite similar to the concrete update function for LRU. The difference is that cache lines can contain sets of memory blocks. Therefore it needs to be e.g. determined how cache hits affect other memory blocks in the same cache line containing the memory block being accessed — for the must analysis the ages are upper bounds, meaning that it is safe to let the other memory blocks stay. In Figure 7.2 an example demonstrating the abstract must update function for LRU is shown.

A formal definition of the abstract must LRU cache update function is given below.

Definition 26 (Abstract Must LRU Cache Update Function). The update function, \( \hat{U}_{LRU} : \hat{c}_c \times M \rightarrow \hat{c}_c \), takes an abstract cache state and a memory block as input and produces an abstract cache state as output.

\[ \hat{U}_{LRU}(\hat{c}, m) = \hat{c}' \text{ where } \hat{c}' \text{ is defined as:} \]

\[
\hat{c}' = \begin{cases} 
[l_1 \mapsto \{ m \},] & i \in \{ 1 \}, \\
[l_i \mapsto \hat{c}(l_{i-1}) \mid i \in \{ 2, \ldots, h-1 \}], & \text{if } \exists h : m \in \hat{c}(l_h) \\
[l_h \mapsto \hat{c}(l_{h-1}) \cup (\hat{c}(l_h) \setminus \{ m \})], & i \in \{ h+1, \ldots, A \}, \\
[l_i \mapsto \hat{c}(l_i) \mid i \in \{ h \}], & \text{otherwise.} 
\end{cases}
\]
The update function has two cases: a cache hit and a cache miss. On a cache miss the memory block accessed is moved to the first cache line, and the rest of the cache lines are moved down one line (with the last cache line dropping out). On a cache hit, the memory block accessed is moved to the first cache line, and the cache lines from the first to the one were the hit was, are moved one down.

FIFO’s update logic is a bit more simple than that of LRU and the abstract update function is therefore even more similar to the concrete update function. The difference between the concrete and the abstract update function is that the latter uses abstract cache states. A definition of the abstract must FIFO cache update function is given below.

**Definition 27 (Abstract Must FIFO Cache Update Function).** The update function \( \hat{U}_{\text{FIFO}} : \hat{C}_c \times M \rightarrow \hat{C}_c \) takes a memory block and an abstract cache state as input and produces an abstract cache states as output. \( \hat{U}_{\text{FIFO}}(\hat{c}, m) = \hat{c}' \) where \( \hat{c}' \) is defined as:

\[
\hat{c}' = \begin{cases} \\
\hat{c}
| \{ l_1 \mapsto \{ m \} \}, & \text{if } \exists h : m \in \hat{c}(l_h) \\
\hat{c}(l_i{-}1) | i \in \{2, \ldots, A\}, & \text{otherwise.}
\end{cases}
\]

Again the update function has two cases: On a cache hit the cache is not altered. On a cache miss the memory block is moved to the first cache line, and the rest of the cache lines are moved one down, with the last falling out.

PRR is more complex to describe than LRU and FIFO. The problem it that the effect of its update function can be hard to predict. According to [15], an important processor property, which enable us to make precise statements about timing behaviour, is that the cache replacement policy should be immune to “chaos” (their wording). For instance, if the value analysis is not able to give precise information about which memory block will be accessed, this can cause the replacement policy to lose information. If the replacement policy can recover information, it is immune to “chaos”.

PRR is not immune to “chaos”. In Figure 7.3 on the next page an example is given, where PRR is not able to recover knowledge. The figure shows two abstract cache states in the state \( \hat{c} \). Four memory blocks are known to be in the cache and the last memory access was \( m_4 \). The abstract cache state \( \hat{c}' \) shows the state after a dynamic memory access which the value
analysis could not predict the memory address of. This results in not knowing if the memory access will result in a cache hit or a cache miss on the abstract cache state $\hat{c}$. What can be seen in part (b) of Figure 7.3 is that we have to assume the worst, which means a cache miss and since we do not know the memory address of the memory block we cannot safely predict which of the four cache sets the memory block will be placed in only that it will be in way two. After three of such updates all counter information will be lost [15, p. 6]. Based on this knowledge, it was chosen not to try and find an abstract PRR cache update function.

**PLRU** We defer an abstraction for the PLRU replacement policy to future work.

**May Analysis**

The may analysis is used to find all memory blocks that could possibly be in the cache at a given point. This can be used to predict if a memory block is definitely not in the cache, and is useful to reduce the number of possible states in the remainder of the processor behaviour analysis.

For the may analysis the following join function is used:

**Definition 28** (LRU/FIFO May Join Function). For LRU and FIFO replacement policies the join function is defined as

$$JOIN(\hat{c}_1, \hat{c}_2) = \{l_i \rightarrow X | y \in X \iff \exists j, k : y \in \hat{c}_1(l_j) \land y \in \hat{c}_2(l_k) \land i = \min(j, k)\}$$

That is, the age a memory block in the cache can have at a node, is the minimal age the item has in its predecessors.

The replacement policies are presented below.

**LRU**’s abstract may update function is a bit different from the abstract must update analysis. The difference is that for the may analysis it is safe to increase the ages, as the ages are lower bounds. In Figure 7.4 an example is presented which demonstrates the abstract may update function.
Abstract cache state $\hat{c}$

\[
\begin{array}{c|c}
 l_1 & \{m_1\} \\
 l_2 & \{m_2, m_3\} \\
 l_3 & \{\} \\
 l_4 & \{m_4\}
\end{array}
\]

Abstract cache state $\hat{c}'$

\[
\begin{array}{c|c}
 l_1 & \{m_3\} \\
 l_2 & \{m_1\} \\
 l_3 & \{m_2\} \\
 l_4 & \{m_4\}
\end{array}
\]

$U_{LRU}(\hat{c}, m_3)$

Figure 7.4: Illustrating abstract may update function for LRU.

A formal definition of the abstract may LRU cache update function is given below.

**Definition 29 (Abstract May LRU Cache Update Function).** The update function $U_{LRU} : \hat{C}_c \times M \rightarrow \hat{C}_c$ takes an abstract cache state and a memory block as input and produces an abstract cache state as output. $U_{LRU}(\hat{c}, m) = \hat{c}'$ where $\hat{c}'$ is defined as:

\[
\hat{c}' = \begin{cases}
  l_i \mapsto \{m\}, & \text{if } \exists l_h : m \in \hat{c}(l_h) \\
  l_i \mapsto \hat{c}(l_{i-1}) & i \in \{2, \ldots, h\}, \\
  l_{h+1} \mapsto \hat{c}(l_{h+1}) \cup (\hat{c}(l_h) \setminus \{m\}), \\
  l_i \mapsto \hat{c}(l_i) & i \in \{h + 2, \ldots, A\}, \\
  l_1 \mapsto \{m\}, \\
  l_i \mapsto \hat{c}(l_{i-1}) & i \in \{2, \ldots, A\},
\end{cases}
\]

FIFO’s abstract may cache update function is the same as for the abstract must cache update function. In case of cache hits both may and must FIFO update function do not change the abstract cache state. In case of cache misses the accessed memory block is inserted as the first and the age of all other memory blocks are increased by one. A formal definition of $U_{FIFO}$ is therefore omitted.

**PRR and PLRU** do not give useful results in a may analysis. The reason is, as mentioned in Section 5.1.1, that PRR and PLRU do not have the property that memory blocks eventually will be evicted. This means that one does not know for sure when a memory block leaves the cache. This makes the may analysis useless as it cannot be determined if a memory block ever leaves the cache [15]. Therefore, the may analysis will just collect the history of memory accesses made by the process.

### 7.2 WCET Analysis using Model Checking

In [27] the author argues that model checking is inadequate for performing WCET analysis. The author claims that model checking “seems to encounter an exponential state-space explosion”. In particular the author notes that the subproblem of cache-behaviour prediction has a good abstraction (see Section 7.1.1), to which model checking has no equivalent and therefore model checking must be used only on concrete caches, of which there are prohibitively many. In response to [27] the article [20] was published, in which the author argues that
Table 7.1: The results presented in [20]. WCET(A) is the WCET when using the first (abstract cache) approach, and WCET(C) is the second (concrete cache) approach. T(A) and T(C) is the analysis times in minutes.

<table>
<thead>
<tr>
<th>Case study</th>
<th>Code size</th>
<th>WCET(A)</th>
<th>WCET(C)</th>
<th>T(A)</th>
<th>T(C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>robot</td>
<td>250 instr.</td>
<td>387</td>
<td>376</td>
<td>0:46</td>
<td>0:36</td>
</tr>
<tr>
<td>collision</td>
<td>1100 instr.</td>
<td>502</td>
<td>478</td>
<td>2:00</td>
<td>4:56</td>
</tr>
<tr>
<td>flight</td>
<td>2500 instr.</td>
<td>1782</td>
<td>1749</td>
<td>2:16</td>
<td>15:24</td>
</tr>
</tbody>
</table>

model checking not only can be used for WCET estimation but even improves it.

7.2.1 Why Model Checking Can Improve WCET Analysis

The article [20] assumes a simple RISC-processor with an external pipeline analysis and focuses only on instruction-level caching. The approach of [20] is two-fold:

- Use abstract interpretation for cache analysis and model checking for path analysis. This approach is very similar to the approach described in Section 7.1.1, except that model checking is used in place of integer linear programming for the path analysis.

- Use model checking for both cache analysis and path analysis. This approach operates on the concrete cache states, and the concrete cache contents is part of the current state during model checking.

The model being used is a Basic Block Automaton (BBA), which is an overlay of the CFG with extra information such as loop bounds. A finite set of variables is part of the state, and the transitions between basic blocks are guarded by boolean expressions, and transitions can have effects on the variables.

For each node in the control flow graph, a location exists in the BBA. The loop bounds are translated into guards such that they are obeyed. A variable, cycles, is used to keep track of how many cycles which have been used in the current execution. In addition, if using the second approach, the current contents of the cache is also registered in variables. For each transition, cycles is updated with the cost of executing the transition (including the cost of retrieving memory blocks, if the cache analysis cannot guarantee a hit). The model is then checked for compliance with the property that a terminating state has been reached and cycles > N, for some WCET estimate N. If the model does not satisfy the property, then N is a WCET over-approximation. A binary search is then done to find the smallest N that does not satisfy the property, which must be the tightest bound on the WCET using this approach.

The results are presented in Table 7.1. The conclusions drawn from them are:

Using concrete cache states gives a small improvement on the order of 1.5% - 5%. This is in accordance with [27] that states that the abstraction done loses little precision in practice.
Using concrete cache states has acceptable analysis times. The author argues that RTSs larger than the “flight” case study are quite unusual. An analysis time of 15 minutes is deemed quite acceptable.

Model checking can not only be used for WCET analysis, it can even improve it. Using concrete cache analysis, tighter bounds on the WCET can be given.

### 7.2.2 Model-Based Schedulability Analysis

In the article [8] the authors outline an approach for doing schedulability analysis using model checking, instead of only doing WCET analysis, and then using the schedulability formulae given in Section 2.1.

The main problem with the “classic” schedulability formulae is that they do not take the control flow of the processes into account. This is primarily a problem with sporadic processes. Consider the following code example:

```java
if (b) 
  RealtimeSystem.fire(1); //Fire sporadic task 1 
else
  RealtimeSystem.fire(2); //Fire sporadic task 2
```

If this is the only place in the system where the sporadic tasks are fired, and the sporadic tasks have equal WCET and minimum inter-arrival times, the model checking approach can use this information to “deduce” that the two sporadic events will never be fired at the same time. This can deem systems schedulable, that are not deemed so using the “classic” formulae approach, because the formulae ignore all control flow.

The authors of [8] have implemented a tool, SARTS, that is able to convert a program written in Safety Critical Java (SCJ) for the Java Optimized Processor (JOP) processor into a timed automata model that can be checked using UPPAAL. If UPPAAL can verify that the model is deadlock-free, then the system is schedulable.

The authors have tested a single case study, using two different compilers and a number of different settings in UPPAAL. The conclusion is that the compiler can have a big influence on the analysis times, however, using the most aggressive optimisations settings, UPPAAL is able to check the models within one minute. The authors also note that their method is capable of taking the cost of context-switching into account, something not possible with the “classic” method.
Chapter 8

WCET Analysis using UPPAAL

We will now present the different models we have designed for modelling a cache analysis in UPPAAL. After presenting the models we present some tools we have developed to automate the processes involved.

8.1 Initial UPPAAL model

Our initial idea is based heavily on emulating the approach described in Section 7.1.1 on page 64. We will show that it is possible to model concrete cache states and a concrete update function using clocks, stopwatches, and networks of timed automata. We will then in the next section show how it is possible to approximate abstract cache states and the results of the abstract cache update function and abstract join function again using a networked timed automaton, clocks, stopwatches, and searching regions using convex hull. All the techniques are already implemented in UPPAAL.

We will focus on a worst analysis of a fully associative cache with a LRU replacement strategy. We will only model reads in the initial model and we will assume a processor where all instructions take one cycle after the operands have been fetched. This assumption only holds for very simple processors. Furthermore we will assume the results of a value analysis is available together with control flow information including loop bounds.

To model the execution of a program we need to model instructions, instruction sequences, if-statements and loops. When modelling instructions we need to model operand fetches and the execution of the instruction. The difficult part in our model is to model operand fetches as this includes modelling a cache and a cache update function.

To model operand fetches we model the fetch of the memory block with the operand. We represent the age of each memory block with a clock in UPPAAL. The value of the clock indicates when the memory block was last accessed, and thus all clocks will have distinct values. The constant CACHESIZE is used to refer to the number of memory blocks the cache can contain. A memory block is said to be in the cache if the age of the memory block is less than or equal to CACHESIZE. In the presented models the value of this constant is two to keep the models relatively small. By having a clock for every memory block we can represent a concrete cache state. In Figure 8.1 on the next page a model of
the concrete cache update function $U_{LRU}(m_1)$, updating the memory block $m_1$ can be seen (note that memory blocks are denoted $m_x$ in the figure instead of $m_x$). The clocks for memory blocks are organised in an array named cacheage. `CACHEFETCHTIME` and `MEMFETCHTIME` represent the cycles required to fetch a memory block from the cache or from main memory, respectively. The clock counter is used as a temporary counter of cycles.

In the system we use the clock `walltime` to track the number of cycles needed for the execution to complete. The first location, A, in Figure 8.1 is a committed location and has two outgoing edges. The path from the right outgoing edge represents a cache miss and the left represents a cache miss.

The outgoing edge of A guarded by the guard `cacheage[m1] > CACHESIZE`, means the path can only be taken if the memory block $m_1$ is not in the cache, hence in case of a cache miss. The following location, B, has an invariant that allows us to stay in the location until `counter <= MEMFETCHTIME-1` is no longer true, the invariant also stops all clocks in the cacheage array while staying in the location. The effect of this, together with the guard `counter == MEMFETCHTIME-1` on Location B’s only outgoing edge, is that the global `walltime` clock will be increased by `MEMFETCHTIME-1`. This will, however, not affect the concrete cache state as all clocks in cacheage is stopped. When taking Location B’s outgoing edge, the clock `counter` will be updated to zero. Location C has an invariant that allows us to stay in the location until `counter <= 1`. Together with the guard `counter == 1` on C’s only outgoing edge, this force all clocks to be increased by one. The effect of this is that memory blocks in the cache will have their age increased. Taking the outgoing edge on Location C will further update the clock `counter` to zero and set the age of memory block $m_1$ to one. This represents that $m_1$ is the youngest memory block in the cache.

The path following the second outgoing edge of Location A, that is the case of a cache hit, will not be explained as thoroughly since there are many similarities. However an important detail is that in case of a cache hit only the $h$ first clocks should be increased by one, where $h$ is the age of the memory block.
being accessed, in this case \( m_1 \). This is done by making a bit vector with a bit for each memory block, where the bit is one if the memory block’s age should be increased, that is if it is less or equal to \( h \) and zero otherwise. The value of the bit in the bit vector is then set. This bit vector is made by synchronising on the `makeBitvector` channel. A variable \( h \) is used to transfer the index of \( m_1 \) in the `cacheage` array to another timed automaton. This timed automaton can be seen in Figure 8.2. It compares the value of all clocks in the `cacheage` array with the value of the clock for \( m_1 \).

![Figure 8.2: UPPAAL model to make a bit vector.](image)

Before executing an instruction, the operands need to be fetched and then the instruction can be executed. In Figure 8.3, the model for executing an instruction is illustrated. In the model it is assumed that the clock `counter` has the value zero before arriving in Location A. Furthermore, `INSTRUCTIONTIME` is the time it takes to execute an instruction, which is one in this model.

![Figure 8.3: UPPAAL model of instruction execution.](image)

Modeling the execution of the instruction is done by having an invariant which allows us to stay in Location A until the clock `counter` is larger than `INSTRUCTIONTIME`, while the guard `counter == INSTRUCTIONTIME` prevents us from leaving until `INSTRUCTIONTIME` time units have elapsed. While staying in Location A all clocks in the array `cacheage` are stopped. The effect of this is that `walltime` is increased by the value of `INSTRUCTIONTIME`.

Instruction sequences are denoted as \( i_1^{m_1}, i_2^{m_2}, m_3, \ldots \), where \( i_n^{x} \) is the \( n \)’th instruction using the finite operand sequence \( x \) of memory blocks. They can be modelled by modelling each instruction \( i \) and making a transition between \( i_1 \)
and $i_2$ and so forth. Modeling an instruction $i$ is done by first modeling the operand fetches of the instruction and then the execution of the instruction. For instance a very simple divide instruction $\text{DIV}^{m_1,m_2}$ uses two operands. To model this each operand should be fetched and then the instruction should be executed. An example of this can be seen in Figure 8.4 on the next page, where the operands are $m_1$ and $m_2$.

Figure 8.4 on the facing page shows an instruction sequence with one instruction. An example demonstrating an instruction sequence of three instructions can be seen in Appendix A on page 95, excluding the last location and transition.

Modeling an if-statement can be done by using the pattern in Figure 8.5 on page 76. An instruction sequence should be inserted between Location A and B, between Location C and D, and between Location F and G. The instruction sequences should be inserted by making a transition from, for instance, Location A to the first location of the instruction sequence and by making a transition from the last location in the instruction sequence to Location B.

To model a loop the pattern in Figure 8.6 on page 76 is used. Again, instruction sequences should be inserted. In this case an instruction sequence for the loop condition should be inserted between Location B and C, and furthermore an instruction sequence for the loop body should be inserted between Location D and E.

A location named done is inserted in the model and a transition from the last location to done is inserted. An example can be seen in Appendix A on page 95 and it is used in the verification of the model.

Having modelled an entire program we verify that a program can terminate in at most $q$ time units, by checking a logic query of the form: At the start, and for all possible paths, either the program has terminated, or otherwise $\text{walltime}$ is less than or equal to $q$. The query for this is:

$$A || \text{Program.done} || \text{walltime} \leq q$$

We then find the smallest $q$ that satisfies the formula. This is the tightest bound for the WCET we can give. This is currently done manually, in a somewhat brute-force manner of doubling $q$, until the formulae is satisfied, and then doing a binary search for the smallest value of $q$.

### 8.1.1 Abstract Cache Analysis

The representation of abstract cache states, as opposed to completely accurate concrete cache states, happens almost automatically using the convex hull approximation in UPPAAL. The convex hull approximation will, given two states that differ only in their clock values, compute a region that encompasses both these states. The approximation is illustrated in Figure 8.7 on page 77.

When applied to our model, the over-approximation always encompasses the state corresponding to the abstract state that would be calculated by the abstract interpretation. To see this, note that the function from Definition 25 on page 65 selects the highest age for an cache item. In our representation, that is the highest value for the clocks representing the data item. As such, the “upper-right corner” of the over-approximated region is the state we are the most interested in, as that is the worst-case. UPPAAL will, however, continue the search in the entire zone, which is still sound, but is a possible loss of precision.
Figure 8.4: Model of execution of instruction $\text{DIV}^{m_1,m_2}$. First $m_1$ is fetched, then $m_2$ is fetched, and finally the division instruction is executed.
Figure 8.5: Pattern of if-statement model.

Figure 8.6: Pattern of loop model.
To see that the over-approximation is still sound, note that the two concrete cache states will still be checked, as both states are contained within the over-approximated region. The possible loss of precision comes from the fact that some states that are impossible to reach in the concrete cache might be checked, and their WCET might be worse than the WCET for the concrete traces.

### 8.1.2 UPPAAL CORA Models

We have also experimented with another approach: using the Cost Optimal Reachability branch of UPPAAL. UPPAAL CORA works on priced timed automata, which are timed automata with a cost-rate associated with each location. UPPAAL CORA can then be asked to satisfy reachability properties, while minimising the cost.

We exploit this, to have the cost be the WCET in cycles, and then using negative costs for the cache and memory access times. In this way UPPAAL CORA will actually maximise the execution time. Finding the WCET bound is then simply a matter of asking for the "best" trace that satisfies the reachability of the done location.

It is however a bit of an exploitation to use negative weights, and not quite supported. The approach has the advantage that only a single run of the model checker is needed. We chose to work further in the normal branch of UPPAAL because it was more updated, while still keeping in mind that UPPAAL CORA might provide a sensible solution to the problem.

### 8.2 Separate Program and Abstract Hardware Model

In the initial model the model of the program and the model of the cache/processor behaviour is mixed in one model along with the bit vector model that models part of the cache behaviour. This would make it a bit hard to get an overview of what the model does. One way to make this clearer would be to split the mixed model in two parts a separate model of the program and a separate abstract hardware model. This could then make it easier to understand larger
programs and larger abstract hardware models. At the same time this would provide separation of concerns for each model.

The program model should model the control flow of the program and which memory blocks should be read and written. A simple sequential example can be seen in Figure 8.8. The example models reading memory block $m_3$, then $m_2$, and after that an instruction is executed. After the instruction has been executed, the result is stored in memory block $m_2$. A larger example can be seen in Appendix B on page 97.

![Figure 8.8: Example of separate program model.](image)

The abstract hardware model should model the cache and processor behaviour. This includes modelling the cache update function and the instruction execution but also other aspects of the cache, for instance, handling writes and the bit vector model in the initial model. The initial model is not designed to handle writes, and without writes only very simple programs can be modelled. As described in Section 5.1.1 on page 42, there are two cases of handling writes, write hits and write misses. We use write back for write hits and write allocate for write misses. This is a common combination, which is complicated compared to “write through/no write allocate”. For instance, this requires that we keep track of which memory blocks, in the cache, have been altered. This is usually done by setting a dirty bit and when a memory block leaves the cache it is checked if the memory block should be written to main memory or can simply be thrown away.

Another limitation of the initial model is that there is a clock for every memory block. In real world programs this is a problem, as too many clocks are needed. A way around this problem is to only have clocks for memory blocks in the cache. However this will further complicate the model of the update function as we then need to keep track of which memory blocks are in the cache. This is done by having an array storing which memory blocks are in the cache. The size of the array is the number of memory blocks the cache can store. To represent
the ages of cache items, an array of clocks, of the same size is made. This array is like
in the initial model named cacheAge. To find the index storing a memory block, the function
getIndex is used. If the memory block is not in the cache getIndex will return −1.

A queue is used to store free indices of the two arrays. For instance, when
the abstract hardware model is initialised, all indices are pushed on this queue
using the function initCache. When a memory block is evicted from the cache
a single index is pushed on the queue by the function pushQueue. The indices
are later popped from the queue by the function insertInCache which inserts
a memory block into a free index and makes it the youngest memory block in
the cache.

![Diagram](image1)

Figure 8.9: Abstract hardware model: initialisation of abstract hardware.

The abstract hardware model is relatively large, and therefore it has been
split in five figures, each describing logical units. The five figures can be seen in
Figure 8.9, 8.10, 8.12, 8.11, and 8.13. Each figure is described separately below.
The Location Init is a central location in the abstract hardware model and
occurs in all figures. The Location ReadCache occurs in both Figure 8.12 and
8.11.

![Diagram](image2)

Figure 8.10: Abstract hardware model: instruction execution.
In Figure 8.9 the model for how the abstract hardware model is initialised is shown. It does this by calling the function initCache as explained earlier. Figure 8.10 is used for execution of instructions and is very similar to Figure 8.3 in the initial model. The last three figures are related to handling of hits and misses when reading or writing to the cache. This is the most complex part of the abstract hardware model and each model is described below.

**Read hits** are handled by the part of the abstract hardware model in Figure 8.11. The model starts in the Location Init and in case the program model tries to synchronise on the readCache channel we will end up in the ReadCache Location. Taking this transition sets the counter clock to zero and the tmpIndex variable to the value of getIndex(tmp), which is −1, if the memory block tmp is not in the cache or the index otherwise. In case of a cache hit tmpIndex cannot be −1 and we end up in Location A. If tmpIndex is −1 we will take a transition (not shown) and end in Location A of Figure 8.12. Location A to C of Figure 8.11 is equivalent to the bit vector model in the initial model. Location C to E is equivalent to Location F and D of Figure 8.1 except the counter clock is not reset as this is not needed since it is reset in the synchronisation transition. The transition from Location E to Init synchronises back to the program model.

**Read misses** are handled by the part of the abstract hardware model shown in Figure 8.12. The two first locations of the figure are the same as for the Figure 8.11. However as mentioned before in the case where tmpIndex is −1 we would end up in Location A in Figure 8.12. From Location A to C is similar to Location B to D in Figure 8.1 except the last part cacheage[m1] = 1 on the last transition. The effect of this part is to set the clock of the accessed memory block to one to make it the youngest memory block in the cache. The reason this is not done in the abstract hardware model is that a memory block that is not in the cache does not yet have a clock. Instead we have to choose a member of the cache to evict. Since we are modelling LRU this should be the memory block in the cache with a clock value larger than CACHESIZE. From Location C to G we check all memory blocks in the cache to find the memory block to evict. In Location C we have three possibilities either the memory block with index zero will have an age less than or equal to CACHESIZE, in which case we should simply look at the next index, or it will be larger, in which case we should evict this memory block. If the dirtyBits[0] is not set to one we take the transition from Location C to E. This will push index zero on the queue of free indices. If the dirtyBits[0] is one we must first wait for the time MEMWRITETIME while we stop all clocks in the cacheage array. Afterwards we push index zero on the queue of free indices. We continue like this for all indices until we get to Location G. Here we are forced to take the transition to Location H which result in running the insertInCache(tmp) function that will pop a free index from the queue of free indices, insert the memory block, set the age to one, and set the corresponding dirty bit to zero. Finally it will synchronise back to the program model.

**Write hits** are handled by the part of the abstract hardware model shown in
Figure 8.13. The first Location is Init and when the program model tries to synchronise on the writeCache channel we will end up in the WriteCache Location. Like the previous two models we will set tmpIndex to the value of getIndex(tmp). If tmpIndex is equal to −1 it means that the memory block tmp refers to is not in the cache. How this is handled will be described under write misses. If tmpIndex is not −1 the memory block is in the cache, which means we have a write hit and we will end up in Location A. Here we will wait the time CACHEWRITETIME, that is the time it takes to write a memory block to the cache. While waiting, all clocks in the cacheage array are stopped, having the effect of increasing the walltime clock with the value of CACHEWRITETIME. Taking the transition from Location A to B we set the dirty bit of the memory block and finally we synchronise back to the program model.

Write misses are handled in roughly the same way as cache read misses. That is, the memory block is read from main memory and stored in the cache. If there are no free indices in the cache, a memory block will be evicted. Finally, the dirty bit of the memory block is set to one. All this except the last step is the same as is modelled in Figure 8.12.
Figure 8.12: Abstract hardware model: read miss.

Figure 8.13: Abstract hardware model: write hit.
8.3 Program Model Generation

To create interesting program models, without having to model these by hand, we would like to generate these from some description of the program we want to model. To model programs precisely the model should be based on the machine code of a program since this is the only level that contains enough information [15]. This is however a time consuming task, instead as a first step we have made a compiler that can translate a simple imperative language into program models. The advantage of doing this is that we do not need to make a control flow analysis on machine code nor value analysis since we does not allow array’s. It should even be possible to reuse most of the code generator for future work.

The BNF of the language can be seen in Figure 8.14. As can been seen the language supports identifiers, expressions, branches, and loops. Keywords are written as var and the strings IDENTIFIER and NUMBER represent regular expressions recognising respectively identifiers and numbers.

```
program  →  statement
statement →  var IDENTIFIER
statement →  output expression
statement →  if ( boolean ) { statement } else { statement }
statement →  while ( boolean ) NUMBER { statement }
statement →  output ; statement
statement →  IDENTIFIER = expression
expression →  expression + expression
expression →  ( expression )
expression →  NUMBER
expression →  IDENTIFIER
boolean →  expression = expression
boolean →  expression < expression
boolean →  expression > expression
boolean →  ! boolean
```

Figure 8.14: BNF of the syntax of the language.

8.3.1 Compiler

The compiler was written using Python [3] and PLY. “PLY is an implementation of lex and yacc parsing tools for Python” [2]. Implementing the compiler using these tools gave us the following advantages:

- It allowed us to write the compiler very quickly (a bit more than a single work day)
- Very modular design
- Easy to change design
- Easy to reuse code
Most of these nice features comes from the fact that Python uses dynamic types and that PLY is very well integrated with Python.

8.4 pyuppaal

In order to generate UPPAAL models easily we have created a library (using Python [3]), that is capable of reading and writing UPPAAL XML-files, and has a Python representation of the UPPAAL objects we need: templates, locations and transitions. In this way we can concentrate on creating Python objects in our compiler-code, and pyuppaal will handle the generation of XML-files.

One additional feature we have developed in pyuppaal is the ability to auto-layout templates. We quickly realised that auto-generating models lead to quite big and cumbersome models. We had to inspect these models for debugging purposes, to find out why they were not working as expected — this is hard if no layout is given in the model. UPPAAL's standard layout algorithm is simply to layout all locations in a grid with total disregard to how the transitions in, which is not very helpful.

We have therefore implemented auto-layouting of templates, by calling out to the Graphviz [1] graph visualisation software. More specifically, the dot tool allows for the automatic layout of directed graphs in a hierarchal structure. We exploit this by converting our pyuppaal templates to graphs in dot, letting dot do the layout, and converting the coordinates from the dot graph to our pyuppaal template. This gives us somewhat nicely laid out graphs: transitions are mostly non-crossing, and labels are not overlapping. The layout is not perfect, as dot has a preference for curved edges, and UPPAAL's edges do not completely follow the same curve. We do, however, translate the edge (and edge waypoints) as this gives a nicer, though sometimes a little strange, result. An example of an UPPAAL template, before and after layout, can be seen in Figure 8.15 on the next page.
Figure 8.15: Example of the layout done by pyuppaal (before and after) — much better, but not perfect. The example is a triple-nested while loop program.
Chapter 9

Conclusion

We have seen that to be able to schedule a hard RTS the WCETs of the processes of the system are needed. Two classes of methods for finding the WCET exist: measurement-based methods and static methods. Measurement-based methods will not in general give safe upper bounds of the WCET.

Modern processors have a number of features (caches, pipelines, branch prediction, out-of-order execution) that make them prone to exhibit timing anomalies. To produce safe WCET bounds, static methods need to incorporate these features. To do this, formalisations of the features are needed.

A central part of modern hardware that provide performance improvements is caches. Work has been done to formalise caches and cache replacement policies. We have continued this work and present formalisations of the PRR and PLRU replacement policies.

Many of the static methods that exist today for WCET estimation are based on abstract interpretation. Abstract interpretation seems to give good results for cache analysis. However, we have found no good general abstraction for pipelining and other features of modern hardware.

We have presented an approach to use model checking to find the WCET of a program on a simple processor model with caching, using UPPAAL. Our approach has the advantage that the model of the processor and the model of the program are separated. We have developed a compiler for a simple toy language to produce models of programs compatible with our processor model.
Chapter 10

Future Work

In this work we examined the various cache policies in use, and tried to develop abstract versions of some of these. We found that the LRU and FIFO policies were quite easy to model, whereas the PLRU and PRR policies were quite hard to model. This is an obvious area for further research into good abstractions for these policies, as they are in actual use in RTSs.

Another area of further research is into modelling pipelines. For pipelines no general good abstraction has been found. We believe that simple pipelines will be relatively easy to model within a model checking framework. We also believe that other features in modern processors (like branch prediction, out-of-order execution, speculative execution, etc.) can be modelled, but not necessarily in an efficient way.

It is an obviously promising combination, like done in [20], to combine abstract interpretation and model checking. The idea is to use abstract interpretation for the cache analysis and model checking for the subsequent pipeline analysis. We believe this combination could be able to yield significant benefits and thus should be investigated.

In this work we used the UPPAAL model checker. UPPAAL does, however, have a number of features that we are somewhat abusing: it only does symbolic representation of clock values while being explicit about the rest of the state. It might be beneficial to be symbolic about the entire state representation (a little like abstract cache states versus concrete cache states). This symbolic representation might be done using Binary Decision Diagrams (BDDs), an efficient, compact representation of sets. We should at least investigate what other model checkers are available, and whether another suits our purposes better.

We researched whether it was possible to separate the program and hardware model. We believe this is worthwhile to continue doing in our future models. Having the models separated also makes it possible to test a program against a number of similar processors by exchanging the hardware model, and finding the cheapest processor that is able to meet the deadlines needed.

We have done initial research into the ARM [17] processor line and believe it is worthwhile to try to model one or more of these processors. The ARM processors are relatively cheap, but not as powerful, compared to other processors. Therefore a tighter bound on the WCET is needed, compared to more powerful processors, as the system resources are scarcer.

Extra work will be needed to make the analysis work for a real-life processor
and programming language, compared to the simple, unrealistic processor we have currently modelled. In particular, the executables for the chosen processor need to be analysed, and a value analysis needs to be performed to sort out the values of pointers (for use in addresses) in the machine-level code. In addition the control-flow of the executable should be partly re-constructed, as much information can have been lost in the compiler optimisation process.

The programs we have analysed are all small examples. To prove whether our approach works in practice we of course need to try it out on real-life size programs. In this process we will no doubt discover performance bottle-necks that need to be sorted out.

A final area of future work is to combine our approach for finding the WCET time of processes with the work done in [8], for doing control-flow sensitive scheduling analyses. For some programs, the inclusion of the control-flow in the schedulability analysis will without a doubt allow for scheduling of more systems.
Bibliography


Appendix A

Instruction Sequence Model
Appendix B

Program Model
Appendix C

List of Acronyms

CFG  Control Flow Graph
WCET  Worst-Case Execution Time
LRU  Least-Recently Used
FIFO  First-In First-Out
PLRU  Pseudo Least-Recently Used
PRR  Pseudo Round-Robin
LSU  Load-Store Unit
IU  Integer Unit
MCIU  Multi-Cycle Integer Unit
RTS  Real-Time System
FPS  Fixed-Priority Scheduling
EDF  Earliest Deadline First
OCPP  Original Ceiling Priority Protocol
ICPP  Immediate Ceiling Priority Protocol
RAW  Read After Write
WAR  Write After Read
WAW  Write After Write
CTL  Computation Tree Logic
TCTL  Timed Computation Tree Logic
MSB  Most Significant Bit First
BBA  Basic Block Automaton
SCJ  Safety Critical Java
JOP  Java Optimized Processor
BDD  Binary Decision Diagram