# 35 Execution Time Analysis for Embedded Real-Time Systems

Andreas Ermedahl
*Mälardalen University*

Jakob Engblom
*Virtutech AB*

## 35.1 Introduction

This chapter deals with the problem of how to estimate and analyze the execution time of embedded real-time software, in particular the worst-case execution time (WCET).

A real-time system must react within precise time constraints, related to events in its environment and the system it controls. This means that the correct behavior of a real-time system depends not only on the *result* of the computation but also on the *time* at which the result is produced. Thus, knowing the execution time characteristics of a program is fundamental to the successful design and execution of a real-time system [22,57].

The size and the complexity of the software in the real-time system are constantly increasing. This makes it hard, or even impossible, to perform exhaustive testing of the execution time. Furthermore, the hardware used in real-time systems is steadily becoming more complex, including advanced computer architecture features such as caches, pipelines, branch prediction, and out-of-order execution. These features increase the speed of execution on average, but also make the timing behavior much harder to predict, since the variation in execution time between fortuitous and worst cases increase.

*Execution time analysis* is any structured method or tool applied to the problem of obtaining information about the execution time of a program or parts of a program. The fundamental problem that a timing analysis has to deal with is the following: the execution time of a typical program (or other relevant piece of code) is not a fixed constant, but rather varies with different probability of occurrence across a range of times. Variations in the execution time occur due to variations in input data, as well as the characteristics of the software, the processor, and the computer system in which the program is executed.

The WCET of a program is defined as the longest execution time that will ever be observed when the program is run on its target hardware. It is the most critical measure for most real-time work. The WCET is needed for many different types of system analysis for real-time systems. For example, it is a critical component of schedulability analysis techniques, it is used to ensure that interrupts are handled with sufficiently short reaction times, that periodic processing is performed quickly enough, and that operating system (OS) calls return to the user application within a specified time-bound. The simplest case is the question whether a particular small piece of code executes within an allocated time budget.

The *best-case execution time* (BCET) is defined as the shortest time ever observed. The BCET can, for example, be of interest in case some code should not finish too quickly, or to analyze the potential jitter in the execution time. The *average-case execution time* (ACET) lies somewhere in-between the WCET and the BCET, and depends on the execution time distribution of the program.

## 35.1.1 The Need for Timing Analysis

Reliable timing estimates are important when designing and verifying many types of embedded and real-time systems. This is especially true when the system is used to control safe critical products, such as vehicles, aircraft, military equipment, and industrial plants. Basically, only if each hard real-time component of such a system fulfills its timing requirements the whole system could be shown to meet its timing requirements.

However, whether timing analysis is needed for a program is not a black-and-white question. In reality, there is a continuum of criticality for real-time systems, as shown with some typical examples in Figure 35.1. Depending on the criticality of the system, an approximate or less accurate analysis might be acceptable. It is really a business issue, where the cost of a timing-related failure has to be weighed against the cost of various means of preventing or handling such a failure.

In many embedded systems, only some part of the system is actually time critical. In a mobile telephone, for example, the parts that deal with communication and signal coding have hard real-time requirements, while the user interface is less time-critical. The fact that only a part of the code is timing-critical helps make timing analysis feasible, since trying to analyze the total system would be virtually impossible owing to its size and complexity.

For other type of systems, the goal is to maintain a high throughput on average, and not necessarily that each task is completed within specified time limits. In a telecom system, for example, it is acceptable that calls are dropped occasionally, and that parts of the system crash and reboot. Glitches are annoying but not dangerous as long as the system as a whole keeps running.

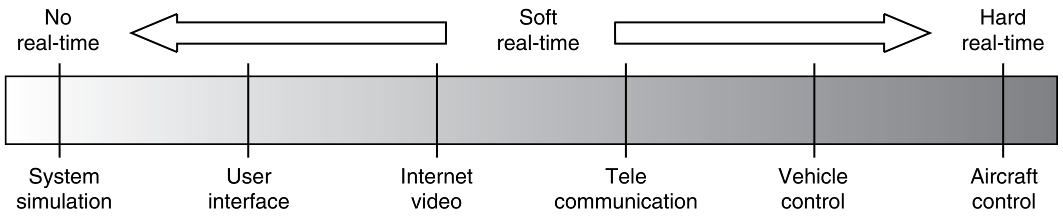


FIGURE 35.1 Continuum of timing criticality.

### 35.1.2 WCET Analysis

Since the WCET is a key component in the analysis of real-time systems, a lot of research and tool development have been devoted to the WCET determination problem. A timing analysis with a main focus on WCET determination is called a *WCET analysis*, even though most tools also produce other information such as the BCET or maximum stack depth.

Figure 35.2 shows how different timing estimates relate to the WCET and the BCET. The example program has a variable execution time, and the darker curve shows the probability distribution of its execution time. Its minimum and maximum are the *BCET* and the *WCET*, respectively. The lower gray curve shows the set of actually observed and measured execution times, which is a subset of all executions. Its minimum and maximum are the *minimal measured time* and the *maximal measured time*, respectively. In most cases, the program state space and the hardware complexity are too large to exhaustively explore all possible executions of the program. This means that the measured times will in many cases overestimate the BCET and underestimate the WCET.

A WCET analysis derives an *estimate* of the WCET for a program or a part of the program. To guarantee that no deadlines are missed, a WCET estimate must be *safe* (or conservative), i.e., a value greater than or equal to the WCET. To be useful, avoiding over allocation of system resources, the estimate must also be *tight*, i.e., provide little or no overestimation compared to the WCET. Similarly, a BCET estimation should not overestimate the BCET and provide acceptable underestimations. Some real-time literature does not maintain the crucial distinction between the WCET and its estimates derived by timing analysis tools. We will strive to avoid such confusion in this chapter.

It should be noted that all timing analysis techniques presented here are focused on the issue of timing a single program or code fragment. This means that timing analysis does not consider that several tasks or programs normally are run together on the same computer, that there might be an OS which schedules and interrupts the programs, etc. All such interactions are handled on a higher analysis level, e.g., using schedulability analysis, where it should be shown that the whole computer system works even in the most stressful situations. For such analyses, reliable WCET estimates are an important input.

The following two sections will explain in more detail the issues that any timing or WCET analysis must consider. In particular, both the *software behavior* (Section 35.2) and the *hardware timing* (Section 35.3) must be considered to derive reliable timing estimates.

Timing and WCET analysis can be performed in a number of ways using different tools. The two main methodologies employed are *measurements* (Section 35.4) and *static analysis* (Section 35.5). In general, measurements are suitable for less time-critical software, where the average case behavior is of interest. For time-critical software, where the WCET must be known, static analysis or some type of *hybrid method* (Section 35.6) is preferable. The research invested in static WCET analysis has resulted

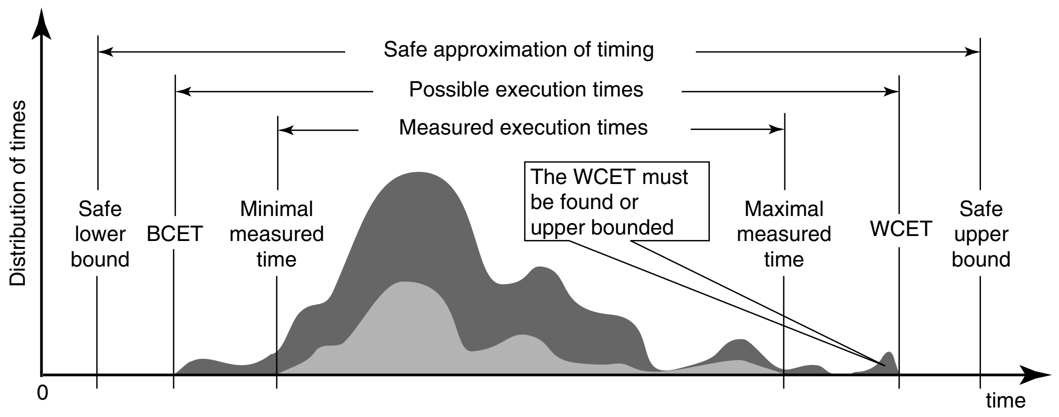


**FIGURE 35.2** Example distribution of execution time.

in sophisticated commercial tools (Section 35.7) able to produce estimates within a few percent of the WCET. Experiences from industrial usage of these tools (Section 35.8) and a chapter summary (Section 35.9) complete this chapter.

## 35.2 Software Behavior

One of the main reasons for the success of computers in embedded applications is that they are *programmable*, making it possible to use a standard part (the processor) for a variety of applications, and enabling new functions that could never have been implemented using fixed-function logic. However, this also means that to correctly analyze and understand the behavior and timing of an embedded system we need to understand the software behavior.

Embedded software comes in many different flavors, using many different languages. In our experience, the most timing-critical real-time software is written in C or Ada, with some assembly language. It is also common to use various code-generating tools working on a high-level model of the software (e.g. MatLab/Simulink, LabView, UML, StateCharts, SDL, and Esterel) to make programming more efficient. This means that a wide variety of codes and coding styles have to be considered for WCET analysis tools.

There have been some studies on the properties of embedded software, and they indicate a huge range of programming practices, from very disciplined simple and predictable code, to code with deep loop nesting, complex if-statements, and significant use of pointers [6,9,49]. Dynamic memory allocation is rare due to resource constraints, but the logic can be very convoluted.

The software behavior contributes a large part of the execution time variability of a program, often dominating the effect of local hardware timing variability [32]. Note that this does not mean that the hardware timing analysis is unnecessary; we need the hardware timing to get a concrete execution time. But we need to have a good understanding of the software behavior to get a precise analysis result.

As illustrated in Figure 35.3, even small codes might exhibit variable and interesting behavior. The example code consists of two functions `task_N` and `convert`. The first function reads two values from two different sensors* and calls `convert` twice, with the values read in each case. The results of the calls to `convert` are used to set an actuator. The `convert` function contains a typical case of software variability, having a loop that iterates a variable number of times depending on input data, and conditional statements where one branch takes longer than the other branch to execute.

A good real-life example of input-dependent flow is the message-polling loop described in Ref. 15, where the number of CAN messages received over a network can vary in number, immediately impacting the execution time of an interrupt handler. On the contrary, some compute-oriented codes exhibit a single possible execution path, by virtue of having no input-dependent loops or decision statements [65]. Instructions in such codes will only exhibit execution time variation owing to hardware effects. This fact has triggered research for rewriting programs to only have single-path behavior [44].

```
// The main function
void task N(void) {
     // Read values from sensors
     int val1 = SENSOR1;
     int val2 = SENSOR2;
     // To hold calculated values
     int res1 = 0;
     int res2 = 0;
     // Call twice with different values
     res1 = convert(val1);
     res2 = convert(val2);
     // Set actuator to calculated sum
     ACTUATOR = res1 + res2;
}
```

```
// Convert read value
int convert(int val) {
     int i = 0;
     int j = 0;
     total = 0;
     while(i <= val) {
          if(j < 5)
               j++;
          if(j > val) break;
          total = total + j - 2;
          i++;
     }
return total;
}
```

**FIGURE 35.3** Code illustrating software variability.

*This is a simplification compared to a real system where the sensor values needs to be calibrated after reading.

Note that complex software behavior is equally problematic for static tools and measurement techniques. A program with complex structure and variable flow will be just as hard to measure correctly as to analyze statically. Real-time code has to be written for predictability and analyzability regardless of the techniques used to analyze their timing.

## 35.3 Hardware Timing

All timing analysis ultimately has to deal with the timing of the hardware on which a program is executing, and the precise sequence of machine instructions that actually make up a program. As hardware gets more complicated, analyzing the hardware timing behavior becomes progressively harder.

The main complexity in hardware timing analysis is the behavior of the processor itself, along with its memory system. Other components of a computer system, such as IO, networks, sensors, and actuators, have less impact on the program timing. They usually dictate when a program is executed in response to external stimuli, but do not affect the instruction-level execution time to any appreciable extent.

Processor instruction timing has been getting increasingly variable over time, as features improving average-case performance and overall throughput are invented and put in use. Typically, performance improvements are achieved using various speculation and caching techniques, with the effect that the span between best-case and worst-case times increases. The goal is for the common case or average case to be close to the best case, and that this best case is better than the best-case times for previous processors. In the process, the worst case typically gets worse, and the precise timing becomes more dependent on the execution history.

Traditional 8- and 16-bit processors typically feature simple architectures where instructions have fixed execution times, and each instruction has minimal effect on the timing of other instructions. There is little variability in execution time. Somewhat more complex 32-bit processors are designed for cost-sensitive applications. Typical examples are the ARM7, ARM Cortex-M3, and NEC V850E. With simple pipelines and cache structures, variability is present but fairly easy to analyze and limited in scope.

The high end of the embedded market requires processors with higher performance, and these are beginning to deploy most of the variability-inducing features, such as caches, branch prediction, and aggressive pipelining. These range from the ARM11 and MIPS24k designs where the pipeline is still in-order, to full out-of-order superscalar processors, such as the PowerPC 755 and 7448, or even Intel Xeons. Analyzing such processors require quite sophisticated tools and methods in the hardware timing analysis. Nevertheless, there have been impressive success stories for tools analyzing quite complicated hardware systems with good results [20,52].

The mainstream trend in computer architecture is still to add ever-more speculation to improve overall performance. However, it is clear that embedded computer architecture is becoming its own field, and that some designs actually have the issues of real-time systems as their primary design requirements. Such designs emphasize predictability, short interrupt latencies, and bounded worst cases over maximal average-case performance.

### 35.3.1 Memory Access Times

Even disregarding caches, the time to access main memory on a modern system is highly variable. Modern RAM technology uses a variety of techniques to take advantage of access locality to improve performance, but this also increases variability in execution time. For static analysis, this means that knowing the precise sequence of addresses of memory accesses becomes important. It is also common to mix different memory technologies and thus memory access times in a system, and at the very least the area of memory accessed by an instruction needs to be known for a tight analysis to be performed [15,52].

### 35.3.2 Long Timing Effects

Conventional wisdom in WCET analysis used to hold that the timing of instructions flowing through a simple pipeline (in-order issue, single instruction per cycle, in-order completion) could be accounted for

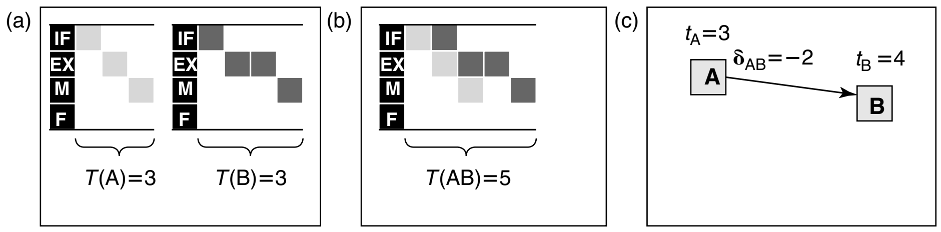


**FIGURE 35.4** Pipelining of instruction execution. (a) Separate execution, (b) sequence execution, and (c) modeling the timing using times and timing effects.

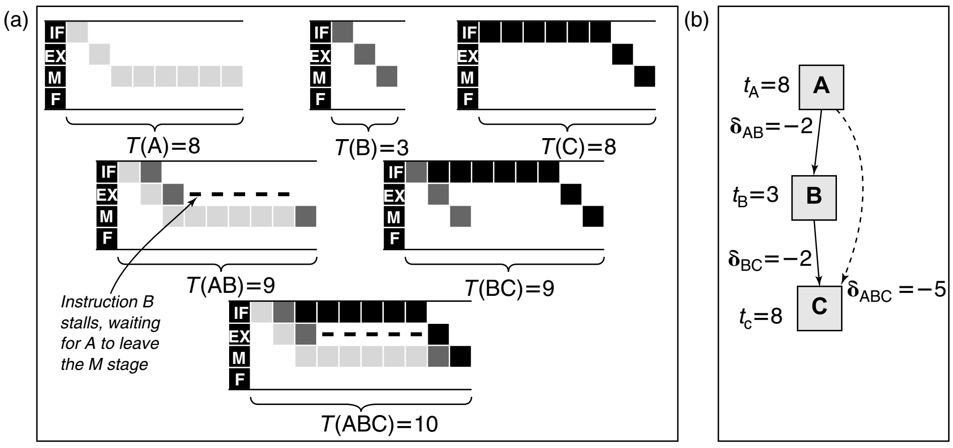


**FIGURE 35.5** Example long timing effect. (a) Execution times and (b) timing model.

by pair wise combination of instructions, as illustrated in Figure 35.4. However, this is not true. Even very simple three-stage pipelines exhibit a phenomenon known as *Long Timing Effects* (LTEs). The essence of an LTE is that interactions occur between instructions that are not adjacent to each other, as illustrated in Figure 35.5.

The example in Figure 35.5 shows an interaction between instructions A and C that is not visible when just analyzing the neighboring pairs of instructions AB and BC. It has been shown that such timing effects can occur across arbitrary distances [12,14]. In experiments, some processors have been shown to exhibit many very long LTEs [47]. The practical consequence of this is that a static WCET analysis tool has to be very careful about how interactions between instructions are analyzed, and has to ensure that all possible LTEs are found or a safe margin added to account for them. Note that since WCET tools are allowed to selectively *overestimate* execution times, this problem is tractable. There have also been attempts to make a processor not have any LTEs, thus facilitating analysis [47].

## 35.3.3 Caches

Caches add to the variability of execution time of a program, and make the difference between the worst case and average case quite large [62]. This effect is well known, and cache optimization is a concern for any performance-oriented programmer. On average, caches work well, but a poorly designed cache or poor use of a cache has the potential to cause disastrous worst cases. From a timing analysis and predictability perspective, caches with least-recently-used replacement policies are preferable, since their state converges over time, making the behavior more predictable [20].

The traditional assumption in analyzing cache behavior is that misses are always worse than cache hits, but as discussed below in Section 35.3.5 this is not necessarily true on all processors.

To overcome some of the unpredictability of caches for critical code, many embedded processors offer the ability to *lock* parts of the cache. This makes it possible to obtain predictable timing for selected code, even in the presence of interrupts and task switching [43,61]. An extreme variant of this is to use a software-controlled cache, where the programmer has complete control and responsibility for what is in the cache and what is not.

### 35.3.4 Branch Prediction

*Dynamic branch prediction* mechanisms try to predict which way a particular branch instruction will go (taken or not-taken), long before the branch has actually been resolved in the processor pipeline. The processor will then speculatively fetch instructions along the predicted path, and if the prediction was wrong, the speculatively fetched instructions will have to be *squashed* and instruction fetching redirected. Thus, branch prediction can affect the state of both instruction and data caches, as well as the processor pipeline, and it has a large impact on the variability in instruction execution time.

Effective branch prediction is very important in modern processors with deep pipelines and high clock frequencies. Very sophisticated predictor mechanisms have been developed that on average achieve very high prediction rates. Indeed, to get above 75% accuracy, dynamic prediction is necessary [13].

However, these branch predictors typically have complicated worst-case behavior. There are examples of cases where executing more iterations of an inner loop takes less time than iterating fewer iterations [13], due to branch prediction effects in a loop which is visited several times. Finally, just like the FIFO caches discussed in Section 35.3.5, dynamic branch predictors do not necessarily converge to a known state over time, thus complicating WCET analysis.

### 35.3.5 Timing Anomalies

When WCET researchers started working on complex, out-of-order processors with caches and branch prediction, a phenomenon known as *timing anomalies* was observed [39]. Timing anomalies are cases where a *local worst case does not entail the globally worst case*. Examples are that a cache hit for a particular instruction causes a longer execution time for a program than a cache miss.

Timing anomalies are usually associated with out-of-order dynamic scheduling of instructions. Most anomalies are similar to the known *scheduling anomalies* studied in scheduling theory, where making a certain task (instruction) faster can cause an entire schedule to take longer. As noted in Ref. 46, there are some cases where the behaviors of processors go beyond scheduling anomalies, since a dynamic decision in a branch predictor can cause the actual set of instructions executed to change.

A strict in-order pipeline does not suffer timing effects in and of itself [12,39]; however, the cache attached to such a pipeline might. For example, a miss in a cache using the FIFO replacement policy might create a better cache state for later code, causing the overall execution to speed up. Since the FIFO cache has a potentially infinite memory, this can cause problems at any later point in time [46].

A variant of a timing anomaly called an *acceleration effect* is that the global slow-down ensuing from a slow-down of some instruction is greater than the local penalty. For example, an instruction being delayed by 8 cycles causing the overall program to be delayed by 11 cycles, as seen in the examples in Ref. 39. Such acceleration effects can be unbounded [39].

### 35.3.6 Multicore and Multiprocessor Systems

The use of multiple processors and its cores is a clear trend in computer architecture. Depending on how systems are designed and programmed, using multiple processor cores can both benefit and hinder timing analysis.

A system using many specialized processors, each with its own defined task, is easier to analyze than a system combining all the tasks onto a single processor. Less interference between tasks and less competition for shared resources such as caches makes the analysis easier. Private memory for each processor is definitely the recommended design here, as that helps predictability and reduces variability, at the expense of some more work for the programmers. This design template is common in mobile phones, where you typically find an ARM main processor combined with one or more digital signal processor (DSP)s on a single chip. Outside the mobile phone space, the IBM–Sony–Toshiba Cell processor contains a PowerPC core along with eight DSP-style processors designed with timing predictability in mind [27].

On the contrary, using a classic shared-memory multiprocessor model like those found in PCs and servers makes it significantly harder to analyze the timing of programs. Programs might be interrupted and scheduled on a different processor in mid-execution, and shared data will cause cache misses due to the cache coherency activity on other processors. Even with scheduling tricks, such as binding a particular task to a particular processor, there are just too many complicating interacting factors to make any precise timing analysis impossible.

### 35.3.7 Custom Accelerator Hardware

Various forms of accelerator hardware is becoming more common in embedded systems, implemented as part of an ASIC, System-on-Chip, or FPGA. This means that real-time software will contain calls to activate the accelerator hardware, and that the WCET analysis will need to account for these calls. This does not have to make the WCET analysis more difficult, as the execution time of accelerator functions is typically fixed or easy to compute. Compared to a software implementation of the same function, hardware accelerators typically exhibit much simpler behavior and less variation. The support necessary in the WCET analysis tool is to be able to identify calls to hardware accelerators, and to provide information about the time the calls take.

## 35.4 Timing by Measurements

The classic method for obtaining information about the execution time of a program is to execute the program many times with different input data, and then measure the execution time for each test run. Finding the input that causes the WCET to occur is very difficult in the general case, and guaranteeing that it has been found is basically impossible without some form of static analysis. Nevertheless, measurements are often the only means immediately at the disposal of a programmer and are useful when the average case timing behavior of the program is of interest.

On the hardware side, the measurement method has the potential advantage of being performed on the actual hardware. This avoids the need to construct a model of the hardware as required by static analysis techniques (an advantage shared with hybrid analysis techniques as discussed below in Section 35.6). However, measurement requires that the target hardware is available, which might not be the case for systems where hardware is developed in parallel with the software [59].

Note that as hardware gets more complex and execution time variability increases (see Section 35.3), it becomes harder and harder to explore all possible timing with measurements. A static analysis tool has the advantage that it in principle can consider all possible executions and thus the entire possible execution time span.

Measurements can be performed in the lab prior to software deployment, or in the field after deployment. Measuring in the field has the advantage that only real executions are observed, but the clear disadvantage that the data is obtained only after the system has been fielded. If some mistake was made when dimensioning the system, timing-related failures could occur in the field. For systems that can tolerate occasional timing problems and which are continuously upgraded, online measurements of performance can be immensely useful.

### 35.4.1 Measurement Techniques

Over the years, many software- and hardware-based timing measurement techniques have been developed [56]. We ignore manual methods, such as using a stopwatch, as that is too low in accuracy for real-time code. There are some issues that all measurement methods need to address:

- *Probe effect*: Measuring a system might cause the timing to change. This is especially common when the program is instrumented or extended with measurement support.

- *Resolution*: Executing code can take a very short time, and the resolution of the timing system has to be fine enough to accurately capture the variations that occur. Typically, you want microseconds or better resolution.
- *Interruptions*: The program under measurement might be interrupted by hardware or OS scheduling intervals. This will lead to intermittent large increases in the end-to-end execution time of a program, and this effect needs to be identified and compensated for [67].
- *Visibility*: It is typically hard to deduce the detailed execution path that a program under measure took in a particular measurement run. This leads to problems interpreting the results and attributing execution time appropriately [15,67].

Most of these problems can be resolved by designing a hardware platform that supports debugging and timing measurements from the outset. Today, built-in debug support in embedded processors is improving thanks to Moore's law, allowing more functionality to be put on a single chip. Still, far from all, embedded computer platforms have useful debug and analysis features.

Other issues that need to be considered when selecting a suitable measurement technique are *Cost*, e.g., special purpose hardware solutions, such as an emulator, are more often costly than general purpose ones, such as an oscilloscope. *Availability*, not all type of target systems are using an OS with suitable timing facilities support. *Retargetability*, a solution suitable for a particular processor and a hardware platform might not be directly applicable on another one [56,67].

Some of the more important measurement techniques used in practice today are:

- *Oscilloscopes and logic analyzers*: Both these methods look at the externally visible behavior of a system while it is running, without affecting its execution. Using an oscilloscope (Figure 35.6c) typically involves adding a bit-flip on an externally accessible pin of the processor to the program segments of interest, and then observing the resulting waveform to divine the periodicity and thus the execution time [56,67]. A logic analyzer (Figure 35.6b) looks at the data or address bus of the system and can see when certain instructions are being fetched, greatly improving the visibility. However, it requires that relevant memory transactions do reach the bus, which is not necessarily the case on a system with a cache.
- *Hardware traces*: Hardware traces and debug ports are extra features added on-board processor chips to help debugging and inspection. The most well-known example is the ARM embedded trace macrocell (ETM), and JTAG and Nexus debug interfaces. Note that such interfaces while powerful often have particular limitations that can make their use more complicated than one might think [3].
- *High-resolution timers*: Adding code to a program to get timing information from timers in the system and then collect start and stop times. Sometimes, special clock hardware is added to a system for this purpose. This will slightly modify the program code, under programmer control.
- *Performance counters*: Most high-end processors from families such as x86, PowerPC, and MIPS offer built-in performance counters that can be used to determine details about the performance of programs. However, such mechanisms are typically oriented toward spotting hot-spots and performance problems in a program and less toward reliable and precise execution time measurements.
- *Profilers*: Profilers are typically provided with compilers. They are dependent on good timing sources in hardware to obtain time measurements, but can provide very good insight into where a program

**FIGURE 35.6** Tools for dynamic timing analysis. (a) Emulator, (b) logic analyzer, and (c) oscilloscope.

is spending its time. Profilers can use periodic interrupts to sample program execution, which does not provide for very precise measurements, or instrument the code, which creates a probe effect.

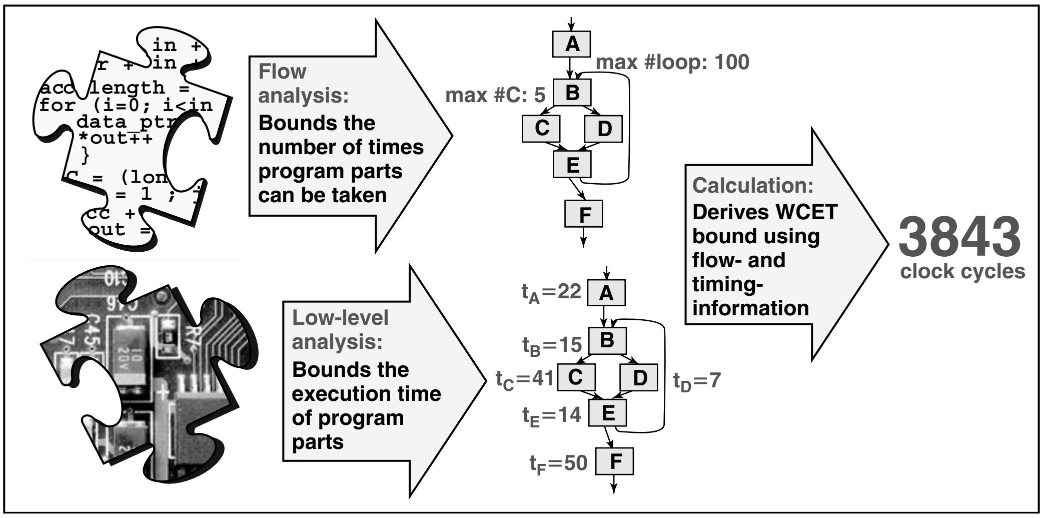
- *Operating system facilities*: OS support for timing measurement can take many forms. *High-water marking* is a common feature of real-time OSs, where the longest execution time observed for a particular task is continuously recorded. There can also be command-line tools for timing programs available. Note that all OS-based solutions depend on the availability of suitably precise timing facilities in the hardware.
- *Emulator*: An *in-circuit emulator* (Figure 35.6a) is a special-purpose hardware that behaves like a particular processor but with better debug and inspection capabilities. Provided that they do match the target processor, they can provide very detailed data. Today, emulators are being replaced with hardware trace facilities, since they are too hard to construct for current processors. There is also the risk that they do not actually perfectly match the behavior of the real processor [67].
- *Simulators*: Processor simulators are sometimes used as a substitute for the real hardware for the purpose of timing analysis. Developing and validating correct simulators is very hard [11,12], and in practice there are very few simulators that are guaranteed to be totally accurate.

## 35.5 Timing by Static Analysis

As mentioned in Section 35.1.2, measurements are suitable for soft real-time applications where the average timing is of interest. However, for hard real-time applications, where the WCET must be known, static analysis techniques are preferable since they provide stronger evidence about the worst possible execution time of a program.

A *static timing analysis tool* works by statically analyzing the properties of the program that affect its timing behavior. It can be compared to determining the stability of a bridge by investigating its design, instead of building the bridge and testing it by running heavy trucks across it. Most static timing analysis has been focused on the WCET determination problem. Given that the inputs and analyses used are all correct, such a *static WCET analysis* will derive a WCET estimate larger than or equal to the actual WCET.

In general, both the software behavior and the hardware timing must somehow be bounded to derive a safe and tight WCET estimate. Consequently, static WCET analysis is usually divided into three (usually independent) phases, as depicted in Figure 35.7: (1) a *flow analysis* phase, where information on the possible execution paths through the program is derived; (2) a *low-level analysis* phase, where information



Flow analysis:
Bounds the number of times program parts can be taken
max #loop: 100
max #C: 5
A
B
C
D
E
F
Low-level analysis:
Bounds the execution time of program parts
$t_A$=22
$t_B$=15
$t_C$=41
$t_D$=7
$t_E$=14
$t_F$=50
Calculation:
Derives WCET bound using flow- and timing-information
3843
clock cycles

**FIGURE 35.7** Phases in static WCET analysis.

about the execution time of program instructions is obtained; and (3) a *calculation* phase, where the flow and the timing information derived in the first two phases are combined to derive a WCET estimate.

Some tools integrate two or more of these phases. For example, the approach in Ref. 39 performs all three functions at the same time, approximately executing the program. Nevertheless, all three phases are needed. As discussed in Section 35.7, most tools also include a *decoding phase*, for reading the program to be analyzed, and a *visualization phase*, for presenting the result of the WCET analysis.

### 35.5.1 Flow Analysis—Bounding the Software Behavior

The purpose of the *flow analysis* phase is to derive bounds on the possible execution paths of the analyzed program, i.e., to find constraints on the dynamic behavior of the software. Such *flow information* can be provided manually by the system programmer, or by an automatic flow analysis. The result is information on which functions that get called, bounds on loop iterations, dependencies between conditionals, etc.

To find exact flow information is in general undecidable*: thus, any flow analysis must be approximate. To ensure a safe WCET estimate, the flow information must be a safe (over)approximation including (at least) *all* possible program executions.

Upper bounds on the number of loop iterations are needed to derive a WCET estimate at all. Without such *loop bounds*, any instruction occurring in a loop might be taken an infinite number of times, leading to an unbounded WCET estimate. Similarly, recursion depth must also be bounded. For the loop in Figure 35.7, the flow analysis has derived a loop bound of 100. This bound has been added to the control-flow graph as a `max #loop: 100` annotation.

Flow analysis can also identify *infeasible paths*, i.e., paths that are executable according to the control-flow graph structure, but not feasible when considering the semantics of the program and possible input data values. In contrast to loop bounds, infeasible path information is not required to find a WCET estimate, but may tighten the result. An extreme case of an infeasible path is dead code. In Figure 35.7, the infeasible path annotation `max #C: 5` is specifying that node `C` can be executed at most five times.

There are a number of approaches to automatic loop bound and infeasible path analyses, using techniques such as abstract interpretation, symbolic execution, Presburger analysis, specialized data flow analyses, and syntactical analysis on parse trees [1,5,23,25,26,30,33,38,39,58]. Some of the methods are general, while others are specialized for certain types of code constructs. The methods also differ in the type of codes they analyze, i.e., source, intermediate (inside the compiler), or machine codes. Most WCET analysis tools allow the user to provide additional flow information as manual annotations [17,21,30,34].

Once again, consider Figure 35.3 as an illustration of the work a flow analysis must perform. After a careful examination of the system it has been found that the two sensors, `SENSOR1` and `SENSOR2`, can give readings within certain boundaries, $0 \leq$ `val1` $\leq 100$ and $0 \leq$ `val2` $\leq 200$.

After studying the code in `convert` we see that its loop will, for each call, iterate as many times as the value of the input argument `val`. A safe upper loop bound for the two calls to `convert` in this program would therefore be to assume that the loop iterates 200 times each time `convert` is called, i.e., a bound corresponding to the largest possible value of `val2`.

However, the loop bound could be improved by noting that the first time that `convert` is called, its input argument cannot be larger than 100. This gives that in total, for the whole execution of the program, the loop in `convert` could iterate at most $100 + 200 = 300$ times. Furthermore, we note that each time `convert` is called, the `j++;` statements on row 22 could be executed at most five times. The last two observations are not required to derive a WCET estimate, but could result in a tighter estimate.

### 35.5.2 Low-Level Analysis—Bounding the Hardware Timing

The purpose of low-level analysis is to determine and bound the possible timing of instructions (or larger code parts) given the architectural features of the target hardware. For example, in Figure 35.7 each basic block in the control-flow graph has been annotated with an upper execution time derived by a low-level

*A perfect flow analysis would solve the halting problem.

analysis. The analysis requires access to the actual binary code of a program, since it has to analyze the processor timing for each instruction.

For modern processors, it is especially important to analyze the effects of complex features such as pipelines, caches, branch predictors, speculative, and out-of-order execution. As explained in Section 35.3, all these features increase the variability in instruction timing and make the prediction more complex.

Most static low-level analyses work by creating a *timing model* of the processor (and other hardware that affect the instruction timing). The model does not need to include all hardware details, as long as it can provide bounds on the timing of instructions. It is common to use safe approximations where precise timing is impossible or very difficult to provide. Typical examples include assuming that an instruction always misses the cache, or that an instruction whose precise execution time depends on data always executes for the longest possible time. Note that certain intuitive assumptions might not always be correct, as discussed in Section 35.3.5.

One should note that standard cycle-accurate processor simulators differ in functionality from timing models. When determining a timing bound for an instruction not just one single concrete execution of the instruction, but rather all possible executions must be accounted for by the timing model. This can be a very large set of possible states for a complex processor.

The complexity of the low-level analysis and the timing model depends on the complexity of the processor used. For simple 8- and 16-bit processors, the timing model construction is fairly straightforward but still time consuming [42]. For somewhat more advanced 16- and 32-bit processors, using simple (scalar) pipelines and maybe caches, the timing effects of different hardware features can be analyzed separately, making the models fairly efficient and simple [12]. For more advanced processors, the performance enhancing features such as branch prediction, out-of-order execution, and caches typically influence each other, and models that integrate all aspects of the processor timing behavior are needed [20,35,52]. Obviously, such timing models can get very complex, with large state spaces and correspondingly long analysis times.

The low-level analyses in today's WCET analysis tools are usually fully automatic. Only information on the hardware characteristics, such as CPU model, processor frequency, cache and memory layouts, need to be provided as parameters to the analysis. The tools usually allow the user to assist and improve the low-level analysis results, e.g., by specifying what memory area certain memory accesses go to and known bounds on register values [21].

### 35.5.3 Calculation—Deriving the WCET Estimate

The purpose of the calculation phase is to combine the flow and timing information derived in the preceding phases to derive a WCET estimate and the corresponding worst-case execution path(s). For example, in Figure 35.7 the calculation derives a WCET estimate of 3843 clock cycles. This corresponds to an execution where each of the nodes `A`, `B`, `E`, and `F` are taken 100 times, while `C` and `D` are taken 5 and 95 times, respectively.

There are three main categories of calculation methods proposed in the WCET literature: *tree-based*, *path-based*, and *IPET (implicit path enumeration technique)*. The most suitable calculation method depends on the underlying program representation as well as the characteristics of the derived flow and timing information.

In a *tree-based calculation*, the WCET estimate is generated by a bottom-up traversal of a tree corresponding to a syntactical parse tree of the program [7,8,37,41]. The syntax-tree is a representation of the program whose nodes describe the structure of the program (e.g., sequences, loops, or conditionals) and whose leaves represent basic blocks. Rules are given for traversing the tree, translating each node in the tree into an equation that expresses the node timing based on the timing of its child nodes.

Figure 35.8a shows an example control-flow graph with timing on the nodes and loop bound flow information. Figure 35.8d illustrates how a tree-based calculation method would proceed over the graph according to the program syntax-tree and given transformation rules. Collections of nodes are collapsed into single nodes, simultaneously deriving a timing for the new node.

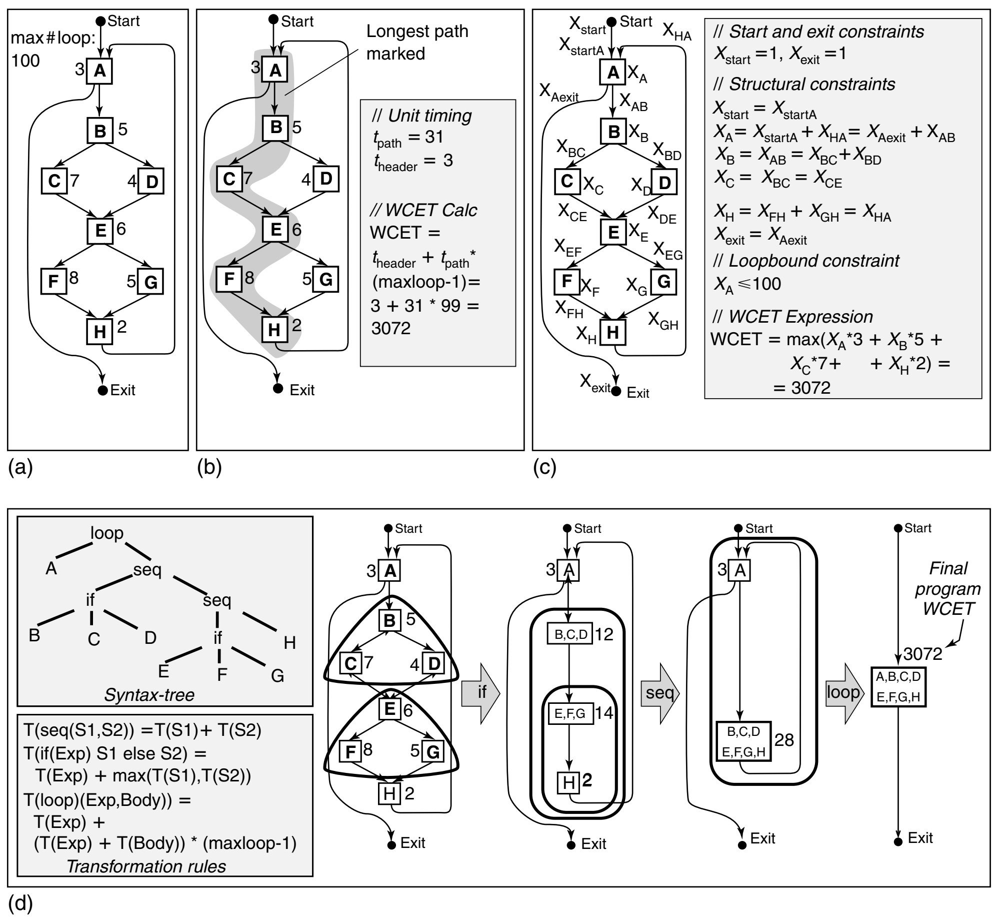

Start
max#loop: 100
A 3
B 5
C 7
4 D
E 6
F 8
5 G
H 2
Exit
(a)
Longest path marked
// Unit timing
t_path = 31
t_header = 3
// WCET Calc
WCET = t_header + t_path * (maxloop-1)= 3 + 31 * 99 = 3072
(b)
X_start X_startA X_HA X_Aexit X_A X_AB X_B X_BC X_BD X_C X_D X_CE X_DE X_E X_EF X_EG X_F X_G X_FH X_GH X_H X_exit
// Start and exit constraints
X_start =1, X_exit =1
// Structural constraints
X_start = X_startA
X_A= X_startA + X_HA= X_Aexit + X_AB
X_B = X_AB = X_BC+X_BD
X_C = X_BC = X_CE
X_H = X_FH + X_GH = X_HA
X_exit = X_Aexit
// Loopbound constraint
X_A ≤100
// WCET Expression
WCET = max(X_A*3 + X_B*5 + X_C*7+ + X_H*2) = = 3072
(c)
loop
A
seq
if
B
C
D
seq
if
H
E
F
G
Syntax-tree
T(seq(S1,S2)) =T(S1)+ T(S2)
T(if(Exp) S1 else S2) = T(Exp) + max(T(S1),T(S2))
T(loop)(Exp,Body)) = T(Exp) + (T(Exp) + T(Body)) * (maxloop-1)
Transformation rules
if
B,C,D 12
E,F,G 14
H 2
seq
B,C,D E,F,G,H 28
loop
Final program WCET
3072
A,B,C,D E,F,G,H
(d)

**FIGURE 35.8** Different calculation methods. (a) CFG with timing and flow information, (b) path-based calculation, (c) IPET calculation, and (d) tree-based calculation.

In a *path-based calculation*, the WCET estimate is generated by calculating times for different paths in parts of a program, to form an overall path with the worst execution time [24,53,54]. The defining feature is that possible execution paths are *explicitly* represented.

Figure 35.8b illustrates how a path-based calculation method would proceed over the graph in Figure 35.8a. The loop in the graph is first identified and the longest path within the loop is found. The time for the longest path is combined with the loop bound flow information to extract a WCET estimate for the whole program.

A frequently used calculation method is *IPET* [17,28,36,45,58]. Here, the longest path no longer is explicit, but instead *implicitly* defined. IPET represents the program flow and execution times using algebraic and logical constraints. Each basic block or edge in the basic block graph is given a time ($t_{\text{entity}}$) and a count variable ($x_{\text{entity}}$), the latter denoting the number of times that block or edge is executed. The WCET is found by maximizing the sum $\sum_{i \in \text{entities}} x_i * t_i$, subject to constraints reflecting the structure of the program and possible flows. The WCET estimate is then derived using integer linear programming (ILP) or by constraint solving. The result is a worst-case count for each node and edge, and not an explicit path like in path-based calculation. IPET is usually applied on a whole program basis, but can also be applied on smaller program parts in a bottom-up fashion [16,28].

Figure 35.8c shows the constraints and WCET formula generated by an IPET-based calculation method for the program illustrated in Figure 35.8a. The *start* and *exit constraints* states that the program must be started and exited once. The *structural constraints* reflect the fundamentals of program flow, where a basic block has to be entered the same number of times as it is exited. The *loop bound* is specified as a constraint on the number of times node A can be executed. Additional constraints on the execution counts of nodes can be used to tighten the WCET bound.

### 35.5.4 Relation between Source and Binary

Static WCET analysis has to be performed on the compiled object code of a program, since that is the only level that actually reveals the actual sequence of machine instructions being executed. However, performing flow analysis on the object-code level is harder than on source code, since the identity of variables and the program structure is obscured by the process of compilation.

Ideally, flow analysis should be performed on the source code level, where program flow is easier to divine than on the compiled code. Analyzing the source code also makes it simpler for the programmer to help the analysis with annotations. Combining source-code level flow analysis with object-code level timing analysis requires flow information to be *mapped* from source code to object code.

This mapping process is complicated by the fact that the compiler performs many transformations on the code during code generation. For example, transformations such as unrolling loops, inlining function calls, or removing unused code all make the relationship between source code and compiled code non obvious [10,34].

The pragmatic solution today is to use only low levels of optimization when compiling programs to be analyzed, relying on debug information in compiler provided symbol tables, and using manual intervention to resolve tricky cases [30,58]. Some research tools are attempting to integrate the WCET analysis with the compiler [19,23,24], which gets around the problem and should allow for a more precise analysis. However, it binds the analysis tightly to a particular compiler.

Note that tools operating on object code are to some extent dependent on the compilers used, since they need to understand the peculiarities of their code generation to build a model of the code structure in the decoding phase, as discussed in Section 35.7.

### 35.5.5 Analysis Correctness

To guarantee that a static WCET analysis tool derives safe WCET estimates, all parts of a tool must be proven safe. For the low-level analysis, the correctness of the timing model is crucial. Validation of a timing model is a complex and burdensome work, especially for complex CPUs [12,40]. Usually, a large number of test cases are used to compare the model with the actual target hardware, forcing the model to follow the same path as the execution on the hardware.

The correctness of flow analysis methods are much easier to prove, since the semantics of programming languages are well defined and the analysis methods used are often well studies in the compiler world. Even for machine code, the semantics of instructions are relatively simple and well defined, but unfortunately this does not hold for the timing behavior. The calculation part of a WCET tool is usually small and simple, and thus easy to validate, while functions such as visualization of results (see Section 35.7) are non critical and do not influence the correctness of the tool as such.

### 35.5.6 Supporting a New Target Architecture

Compared to other types of programming tools, such as compilers and functional simulators, a WCET analysis tool requires much more work to support a new processor, something which is reflected in the number of processors supported by different tools and the price of the tools.

The main bottleneck is the creation and validation of the CPU timing model [42]. This has led to the development of hybrid WCET analysis methods (see Section 35.6), where measurements are used instead of a static hardware model. Other approaches intended to simplify the porting of a WCET tools is to reuse existing cycle-accurate simulation models [2,12,66] or derive models from VHDL code [60].

## 35.6 Hybrid Analysis Techniques

Hybrid analysis techniques use measurements and static analyses in *complement* to each other. The basic idea is the following: first, a model of the program is constructed by static analysis of the code. The model is then annotated with measurement points at suitable places, partitioning the program into a number of smaller parts for measurement. The program, or smaller parts of the program, is then executed a number of times and time measurements are performed. For each measured program part, a worst measured execution time is noted. The measured times are then brought back into the static model and used to deduce a WCET estimate [2,63,66].

Similar to static WCET analysis, hybrid method requires that all loops are bounded. These loop bounds can either be derived using measurements (by keeping track on the largest number of iterations for a loop observed), through static analysis or by manual annotation.

The main advantage of the hybrid method is that no timing model of the processor needs to be created. In a sense, the low-level analysis is replaced with structured measurements. Furthermore, traditional test coverage criteria can be used to show that the program has been thoroughly tested.

The main drawback to the method is that there are no guarantees that the obtained WCET estimates are safe. Timing for program parts are derived using measurements on a subset of all program executions. Thus, it is hard to guarantee that each program part has encountered its worst case timing. Moreover, it is possible to add worst case timings for program parts that can never occur together, potentially ruining the precision.

The user or the tool also has to somehow provide suitable input data to ensure that all program parts are executed. The method also requires that there exists suitable measuring mechanisms to obtain the timing for program parts, see Section 35.4.1.

## 35.7 Tools for WCET Analysis

During the last couple of years a number of WCET analysis tools have been developed, both research prototypes and commercial tools. The latter are all research projects that have evolved into commercial products. Commercial tools include *aiT* (`www.absint.com`) and *Bound-T* (`www.tidorum.fi`), both static WCET analysis tools, and *Rapitime* (`www.rapitasystems.com`), which is a hybrid analysis tool. For a more detailed presentation of commercial and research tools, we refer to Ref. 64.

The tools differ in the processors supported and the types of codes they can analyze. In general, all tools contain the three analysis phase described in Section 35.5 (see Figure 35.9). Normally, there is also a *decoding* phase, where the executable program is read and converted into an internal representation for the analysis. The decoding phase can be quite complicated depending on the instruction set and the compiler used to generate the code (see Section 35.5.4). Most tools also contain some form of *visualization* of the results, e.g., a program flow graph where different program parts have been annotated with the number of times they are taken in the worst-case execution.

Most tools allow the user to improve the results and guide the analyses by various forms of *manual annotations*. Certain annotations are mandatory, e.g., the specific hardware configuration used, the start address of the code that should be analyzed, possible values of input data, and bounds for loops that cannot be determined by the flow analysis. Some annotations are optional, e.g. information on infeasible paths or the memory area certain memory accesses go to, but may give a tighter WCET estimate.

Some tools also provide other information that can be obtained with a small extra effort, based on the analysis already made while doing the WCET. A typical example is the maximum stack depth a program can use, which is useful for configuring certain OSs.

## 35.8 Industrial Experience with WCET Analysis Tools

As mentioned above, there are several commercially available WCET analysis tools today. The tools have been used to analyze software timing in avionics, space, and car industries, e.g., aiT has been used during

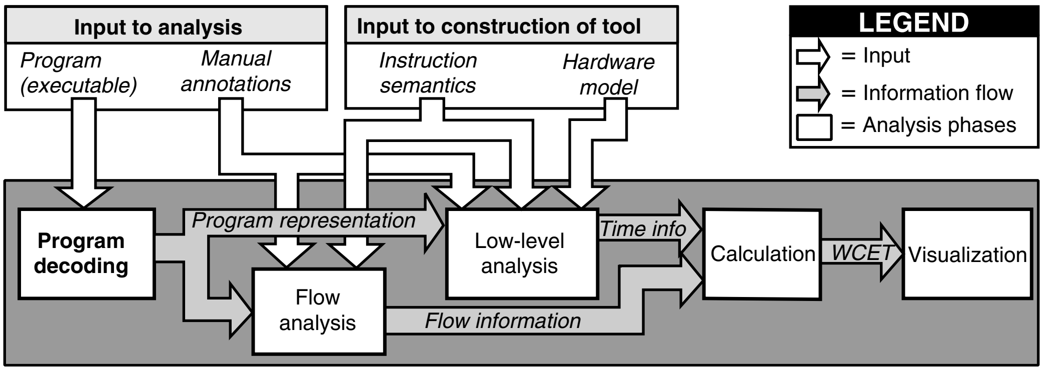


**FIGURE 35.9** WCET analysis tool layout.

the development of the software for the fly-by-wire system in Airbus A380 [64]. There are published reports about industrial [4,15,18,28,29,50–52,59] and educational use [42] of these tools. The following list summarizes some of the most important lessons reported:

- Fundamentally, it is possible to apply today's WCET analysis tools to a variety of industrial systems and codes. The tools used derived safe WCET estimates in comparison to measured times and received positive feedback regarding analysis time, precision, and usefulness.
- WCET analysis is not yet a fully automated "one-click" analysis. Manual user interaction, typically annotations to describe the program flow and bounds on input data, is required to obtain useful results. This, in turn, requires a detailed understanding of the analyzed code and its operating environment [4,15,18,51].
- A higher degree of automation and support from the tools, e.g., automatic loop bound calculation, would in most cases have been desirable. The tools used today are only able to handle a subset of the loops encountered [4,29,48,50,52].
- The structure of the code of a system has significant impact on the analyzability of a system. Many small well-defined tasks scheduled by a strict priority Real-time OS or a time-triggered schedule is easier to analyze than monolithic interrupt-driven programs based on an infinite main loop [15,51,52].
- Manual annotations can sometimes be burdensome to generate, even when the code is well structured and easy to understand, owing to the volume of annotations needed. For example, if a C macro construct requires an annotation to be analyzed, that macro requires an annotation for each use [52].
- Once a static analysis tool is setup, reanalyzing a system after changes is typically faster than measuring it. This could potentially save large amounts of time in development [29].
- Code should be written to facilitate timing analysis, by adhering to strict coding rules. Typically, you want all loops to be statically bounded, input-data dependence minimized, pointer use minimized, and code structure as simple and straightforward as possible. Such coding style is commonly recommended for safety critical and real-time systems programming [22,31,55].
- Presenting the results in a graphical user interface is very valuable for obtaining an overview of the analyzed code and to interpret and understand the result of the analysis [4,15,18].
- A single WCET bound, covering all possible scenarios, is not always what you want. Sometimes it is more interesting to have different WCET bounds for different execution modes or system configurations rather than a single WCET bound. The latter would usually be an overapproximation. In some cases, it was possible to manually derive a parametrical formula [4,15,50] showing how the WCET estimate depends on some specific system parameters.
- Automatically generated code can be very amenable to the WCET analysis, provided the code generator is designed with predictability and analyzability in mind [52]. Generated code can also be

hard to analyze, if predictability and analyzability were not designed into the code generator since code generators can generate very convoluted and complex code that is hard to understand for both human programmers and automatic program analyses.

- Not only application-level code is interesting to analyze, but also other code such as OS services and interrupt handlers [6,15,50].
- The large diversity in processors used in embedded systems hinders the deployment of static WCET analysis. A tool often has to be ported to the particular processor(s) used in a project, which costs both time and money [4,42] see also Section 35.5.6.
- Measurements and static timing analysis should not be seen as isolated tasks. Instead, they could complement each other, together giving a better understanding of the system timing and increase the trust in the resulting timing estimates [15,67].

## 35.9 Summary

This chapter has dealt with the problem of how to estimate and analyze the execution time of programs. Knowing the execution time characteristics of a program is fundamental to the successful design and execution of a real-time system. For hard real-time systems, reliable WCET estimates are especially important.

A variety of methods are available to performing timing and WCET analysis, ranging from manual measurements to automated static analyses. The method(s) suitable for a particular application depends in high degree on its real-time criticality and the availability of tools suited to the particular system.

For the static WCET estimation problem, a lot of research investment has been made, and today several commercial WCET analysis tools are available. Industrial usage of these and other WCET analysis tools show very promising results in published case studies.

## References

1. H. Aljifri, A. Pons, and M. Tapia. Tighten the computation of worst-case execution-time by detecting feasible paths. In *Proc. 19th IEEE International Performance, Computing, and Communications Conference (IPCCC2000)*. IEEE, February 2000.
2. G. Bernat, A. Colin, and S. Petters. pWCET: A tool for probabilistic worst-case execution time analysis. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'03)*, 2003.
3. A. Betts and G. Bernat. Issues using the nexus interface for measurement-based wcet analysis. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET'2005)*, July 2005.
4. S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. Applying static WCET analysis to automotive communication software. In *Proc. 17th Euromicro Conference of Real-Time Systems (ECRTS'05)*, pp. 249–258, July 2005.
5. T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra. Exploiting branch constraints without exhaustive path enumeration. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET'2005)*, pp. 40–43, July 2005.
6. A. Colin and I. Puaut. Worst-case execution time analysis for the RTEMS real-time operating system. In *Proc. 13th Euromicro Conference of Real-Time Systems (ECRTS'01)*, June 2001.
7. A. Colin and G. Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *Proc. 14th Euromicro Conference of Real-Time Systems (ECRTS'02)*, pp. 50–59, 2002.
8. A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Journal of Real-Time Systems*, 18(2/3): 249–274, 2000.
9. J. Engblom. Why SpecInt95 should not be used to benchmark embedded systems tools. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, May 1999.

10. J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution times analysis for optimized code. In *Proc. of the 10th Euromicro Workshop of Real-Time Systems*, pp. 146–153, June 1998.
11. J. Engblom. On hardware and hardware models for embedded real-time systems. In *Proc. IEEE Real-Time Embedded Systems Workshop, Held in Conjunction with RTSS2001*, December 2001.
12. J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden, April 2002. ISBN 91-554-5228-0.
13. J. Engblom. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *Proc. 8th IEEE Real-Time/Embedded Technology and Applications Symposium (RTAS'03)*, May 2003.
14. J. Engblom and B. Jonsson. Processor pipelines and their properties for static WCET analysis. In *Proc. 2nd International Workshop on Embedded Systems (EMSOFT2001)*, 2002.
15. O. Eriksson. *Evaluation of Static Time Analysis for CC Systems*. Master's thesis, Mälardalen University, Västerås, Sweden, August 2005.
16. A. Ermedahl, F. Stappert, and J. Engblom. Clustered worst-case execution-time calculation. *IEEE Transaction on Computers*, 54(9): 1104–1122, 2005.
17. A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden, June 2003.
18. A. Ermedahl, J. Gustafsson, and B. Lisper. Experiences from industrial WCET analysis case studies. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET'2005)*, pp. 19–22, July 2005.
19. H. Falk, P. Lokuciejewski, and H. Theiling. Design of a WCET-aware C compiler. In *Proc. 6th International Workshop on Worst-Case Execution Time Analysis (WCET'2006)*, July 2006.
20. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. 1st International Workshop on Embedded Systems (EMSOFT2000), LNCS 2211*, October 2001.
21. C. Ferdinand, R. Heckmann, and H. Theiling. Convenient user annotations for a WCET tool. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis (WCET'2003)*, 2003.
22. J. Ganssle. Really real-time systems. In *Proc. of the Embedded Systems Conference, Silicon Valley 2006 (ESCSV 2006)*, April 2006.
23. J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system C programs. In *Proc. 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005)*, February 2005.
24. C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1): 53–70, 1999.
25. C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, 18(2–3): 129–156, 2000.
26. C. Healy and D. Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, June 1999.
27. P. Hofstee. Power efficient processor architecture and the cell processor. In *11th Symposium on High-Performance Computing Architecture (HPCA-11)*, February 2005.
28. N. Holsti, T. Långbacka, and S. Saarinen. Worst-case execution-time analysis for digital signal processors. In *Proc. EUSIPCO 2000 Conference (X European Signal Processing Conference)*, 2000.
29. N. Holsti, T. Långbacka, and S. Saarinen. Using a worst-case execution-time tool for real-time verification of the DEBIE software. In *Proc. DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457)*, September 2000.
30. N. Holsti and S. Saarinen. Status of the Bound-T WCET tool. In *Proc. 2nd International Workshop on Worst-Case Execution Time Analysis (WCET'2002)*, 2002.

31. G. Holzmann. The power of 10: Rules for developing safety-critical code. *IEEE Computer*, 39(6): 95–99, June 2006.
32. C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architecture. In *Proc. 28th International Symposium on Computer Architecture (ISCA 2001)*.
33. D. Kebbal and P. Sainrat. Combining symbolic execution and path enumeration in worst-case execution time analysis. In *Proc. 6th International Workshop on Worst-Case Execution Time Analysis (WCET'2006)*, July 2006.
34. R. Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2003.
35. X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *Proc. 25th IEEE Real-Time Systems Symposium (RTSS'04)*, pp. 92–103, December 2004.
36. Y-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. 32nd Design Automation Conference*, pp. 456–461, 1995.
37. S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An accurate worst-case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7): 593–604, 1995.
38. Y. A. Liu and G. Gomez. Automatic accurate time-bound analysis for high-level languages. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, pp. 31–40, June 1998.
39. T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, June 2002.
40. S. Montán. *Validation of Cycle-Accurate CPU Simulator against Actual Hardware*. Master's thesis, Dept. of Information Technology, Uppsala University, 2000. Technical Report 2001–2007.
41. C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on a source-level timing schema. In *Proc. 11th IEEE Real-Time Systems Symposium (RTSS'90)*, pp. 72–81, December 1990.
42. S. Petersson, A. Ermedahl, A. Pettersson, D. Sundmark, and N. Holsti. Using a WCET analysis tool in real-time systems education. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET'2005)*, Palma de Mallorca, Spain, July 2005.
43. I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proc. 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, 2002.
44. P. Puschner. The single-path approach towards WCET-analysable software. In *Proc. IEEE International Conference on Industrial Technology*, pp. 699–704, December 2003.
45. P. Puschner and A. Schedl. Computing maximum task execution times—A graph-based approach. *Journal of Real-Time Systems*, 13(1): 67–91, 1997.
46. J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *Proc. 6th International Workshop on Worst-Case Execution Time Analysis (WCET'2006)*, July 2006.
47. C. Rochange and P. Sainrat. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In *Proc. 2nd conference on Computing frontiers*, May 2005.
48. M. Rodriguez, N. Silva, J. Esteves, L. Henriques, D. Costa, N. Holsti, and K. Hjortnaes. Challenges in calculating the WCET of a complex on-board satellite application. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis (WCET'2003)*, 2003.
49. C. Sandberg. Inspection of industrial code for syntactical loop analysis. In *Proc. 4th International Workshop on Worst-Case Execution Time Analysis (WCET'2004)*, June 2004.
50. D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static timing analysis of real-time operating system code. In *Proc. 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, October 2004.
51. D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz. Static WCET analysis of real-time task-oriented code in vehicle control systems. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, November 2006.

52. J. Souyris, E. L. Pavec, G. Himbert, V. Jegu, G. Borios, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis (WCET'2005)*, July 2005.
53. F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4): 339–355, 2000.
54. F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *Proc. 4th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'01)*, November 2001.
55. D. B. Stewart. Twenty-five most common mistakes with real-time software development. In *Proc. of the Embedded Systems Conference, San Francisco 2001 (ESCSF 2001)*, April 2001.
56. D. B. Stewart. Measuring execution time and real-time performance. In *Proc. of the Embedded Systems Conference, San Francisco 2004 (ESCSF 2004)*, March 2004.
57. D. B. Stewart. Twenty-five most common mistakes with real-time software development. In *Proc. of the Embedded Systems Conference, Silicon Valley 2006 (ESCSV 2006)*, April 2006.
58. S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.
59. S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *Proc. of the IEEE International Conference on Dependable Systems and Networks (DSN-2003)*, June 2003.
60. S. Thesing. Modeling a system controller for timing analysis. In *Proc. 6th International Conference on Embedded Software (EMSOFT2006)*, October 2006.
61. X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *Proc. International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03)*, pp. 272–282, San Diego, CA, June 2003.
62. L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proc. Design, Automation and Test in European Conference and Exhibition (DATE'05)*, March 2005.
63. I. Wenzel, B. Rieder, R. Kirner, and P. P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In *Proc. Design, Automation and Test in European Conference and Exhibition (DATE'05)*, Volume 1, pp. 606–611, 2005.
64. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem—Overview of methods and survey of tools. *ACM Transactions in Embedded Computing Systems (TECS)*, 2007 (Accepted for publication).
65. F. Wolf and R. Ernst. Execution cost interval refinement in static software analysis. *Journal of Systems Architecture, The EUROMICRO Journal, Special Issue on Modern Methods and Tools in Digital System Design*, 47(3–4): 339–356, 2001.
66. F. Wolf, J. Kruse, and R. Ernst. Timing and power measurement in static software analysis. *Microelectronics Journal, Special Issue on Design, Modeling and Simulation in Microelectronics and MEMS*, 6: 91–100, 2002.
67. Y. Zhang. *Evaluation of Methods for Dynamic Time Analysis for CC Systems AB*. Master's thesis, Mälardalen University, Västerås, Sweden, August 2005.