By ALEX LIM, Principal Field Application Engineer and Multicore Lead, LDRA (a TASKING Company), www.ldra.com

Don’t Do It: Ignore Worst-Case Execution Time

WCET analysis is essential for proving multicore real-time systems meet safety-critical deadlines under all operating conditions.

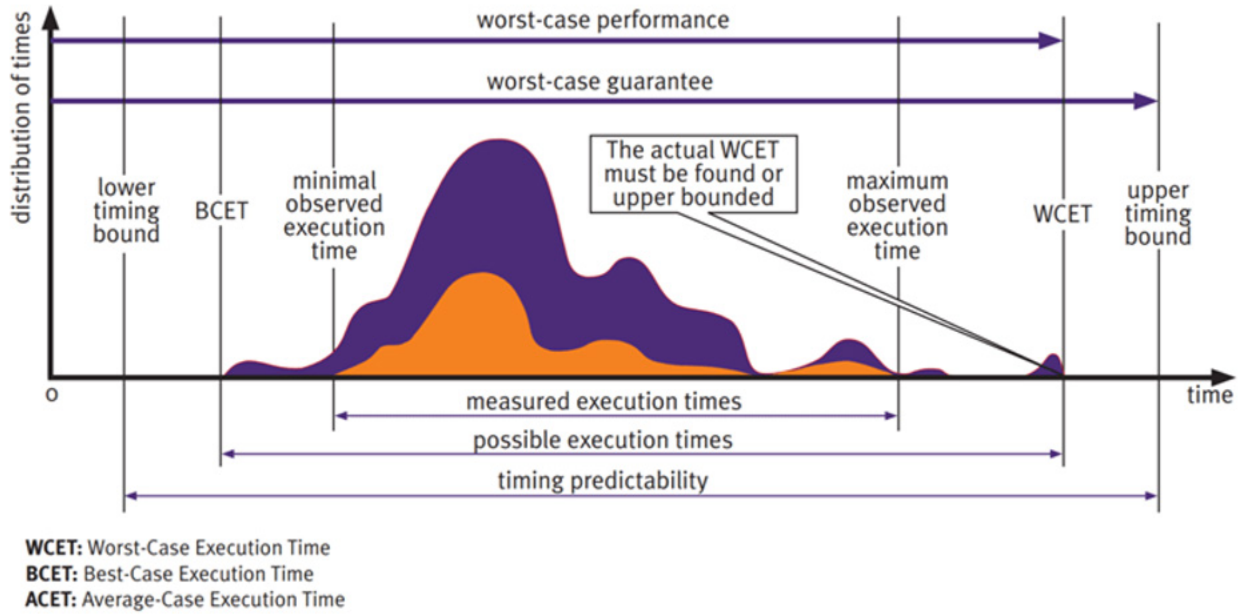
In real-time embedded systems, even a single missed deadline can compromise safety and reliability. From automotive and avionics to industrial automation and defense, developers must demonstrate that worst-case execution time (WCET) is well understood and consistently bounded, ensuring a reliable upper limit that no task will ever exceed during operation.

Estimating WCET is far from straightforward, especially in modern multicore systems, so it isn’t surprising that even the most seasoned development teams underestimate it and often implement it poorly. The worst-case execution time for mission-critical tasks can fluctuate depending on what else is running, which resources are shared, and how the processor behaves under load.

As systems grow more complex, and as artificial intelligence and machine learning (AI/ML) workloads increase memory and data traffic, developers must confront WCET head-on. Ignoring it isn’t an option. Underestimating WCET can compromise safety, reliability, and certification, even when average execution times appear acceptable *(Fig. 1)*.

Even tasks that meet timing requirements on average may still miss deadlines under real-world load. Certification standards force developers to demonstrate that:

- Every safety-critical task meets its deadline under all operating conditions.
- WCET stays within system constraints despite interfer-



1. Measured execution times (orange) are but a subset of the possible execution times (purple). To ensure that the worst-case execution time (WCET) can be met, developers use measured execution times to estimate the WCET. For safe and reliable execution, developers can choose an upper timing bound that guarantees the actual WCET will be met. (Source: LDRA, a TASKING Company)

ence.
- Timing remains deterministic even as other cores execute concurrently.

Measured execution times capture only the subset of possibilities observed during testing. On real multicore hardware, interference patterns can change every run depending on cache state, memory traffic, and other shared-resource contention. The worst-case measured is rarely the true WCET; it's merely the upper bound observed during testing. That gap is precisely where catastrophic failures can occur.

Other reasons WCET can't be overlooked in real-time, mission-critical systems include:

- **Hidden concurrency effects:** Even tasks that share no code or data can interfere via shared caches, memory buses, or peripherals, introducing unpredictable latency spikes.
- **Rare event vulnerability:** Timing violations often occur under unusual but critical circumstances, such as bursts of sensor data, AI/ML workload spikes, or simultaneous interrupts — exactly when reliability matters most.
- **Safety and certification implications:** Standards like ISO 26262, DO-178C, and IEC 61508 require evidence that timing guarantees are met. Ignoring WCET can jeopardize certification or introduce liability.
- **Dynamic system behavior:** Memory contention, cache eviction, and task overlap vary over time. Relying on average execution times may mask worst-case scenarios.
- **Mission-critical impact:** For hard real-time systems such as aircraft control, automotive safety, or medical devices, missing a deadline even once could be catastrophic.

In short, WCET is the single most important factor in determining whether a real-time system is able to operate safely and reliably under all conditions. Ignoring it risks failures that testing averages alone can't reveal.

Why is WCET Analysis So Difficult?

Multicore timing coupling

Even tasks that share no data, no functions, and run on separate cores can interfere with each other due to shared infrastructure that includes L2/L3 caches, memory buses and interconnects, shared peripherals, and RAM access patterns. These hidden interference channels create timing spikes that are impossible to predict from code inspection alone.

Concurrent execution is non-deterministic

Execution overlap between tasks changes from run to run. A task may overlap with another for milliseconds in one test and barely at all in another. This variability directly affects WCET and makes analysis an iterative, measurement-driven process.

Data size changes cache behavior

When a task's working set exceeds L1 cache capacity, it depends on shared L2 or RAM access. Multiple tasks crossing this threshold can spike cache contention and unpredictably inflate timing, sometimes by 40% or more in real systems.

Synthetic interference isn't realistic

Generated interference patterns can't replicate real hardware contention. WCET must be assessed on the target, under realistic load, using the actual cache and memory configuration.

Analyzing Multicore WCET in Three Complementary Stages

As recognized by industry guidance documents and standards, developers can employ various techniques to measure execution timing in complex, multicore processor-based systems. Three proven methods can provide valuable timing data throughout the development cycle:

1. Early-Stage Development: Halstead's Metrics and Static Analysis

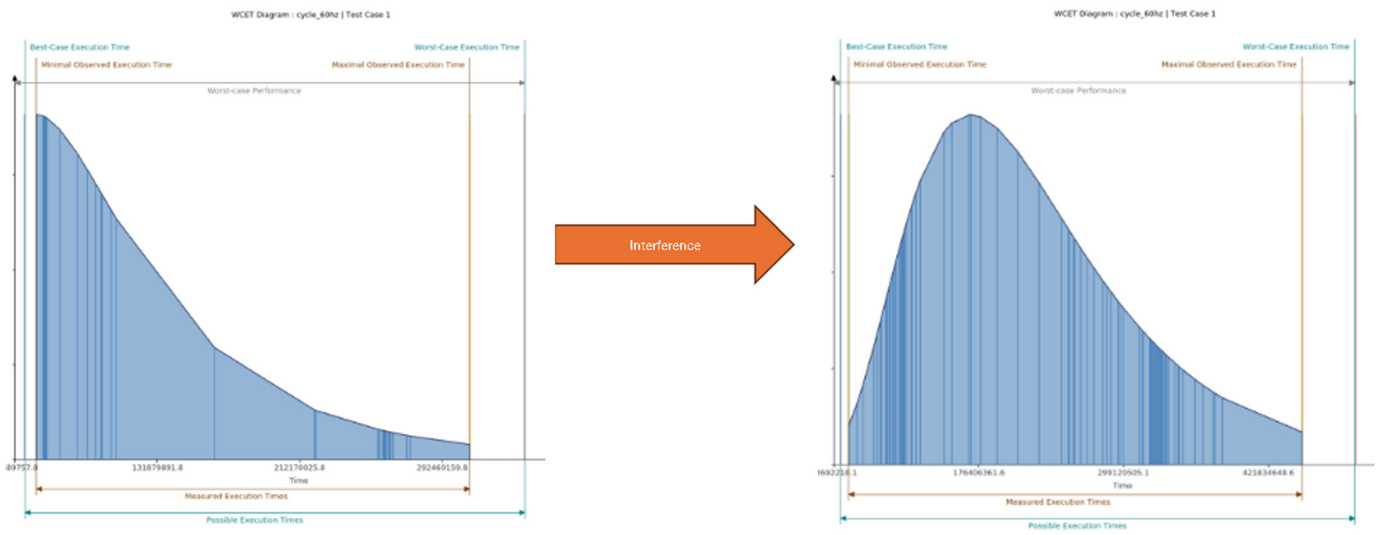
Halstead's complexity metrics can act as an early warning system for developers. This approach provides insights into the complexity and resource demands of specific sections of code. By employing static analysis, developers can use Halstead data with real-time measurements from the target system. This helps developers ensure a more efficient path to lower the WCET.

Such metrics and others shed light on timing-related aspects of code, including module size, control flow structures, and data flow. By identifying code sections of larger size, higher complexity, and more intricate data flow patterns, developers can prioritize their efforts and fine-tune code that puts the highest demands on processing time. Optimizing these resource-intensive areas early in the lifecycle is an effective way to reduce risk of timing violations and simply later analysis processes.

2. Mid-Stage Development: Empirical Dynamic Analysis of Execution Times

When modules fail to meet timing requirements, developers can measure, analyze, and track individual task execution times to help identify and mitigate timing issues (empirical analysis). It's critical to eliminate the influence of configuration differences between development and production, such as compiler options, linker options, and hardware features. Therefore, analysis must occur in the actual environment where the application will run. As a result, empirical analysis can't be fully employed until a test system is in place.

To account for environmental and application variability between runs, sufficient tests must be executed repeatedly to ensure accurate, reliable, and consistent results (empirical dynamic analysis). To execute sufficient tests within a rea-



2. Timing coupling interference can significantly impact worst-case execution time.

sonable timeframe, automation is essential. Automation also reduces developer workloads as well as eliminates human error that can occur during manual processes.

3. Late-Stage Development: Application Control and Data-Coupling Analysis

Evaluating contention involves identifying task dependencies within applications, both from a control and data perspective. Using control and data-coupling analysis, developers can explore how execution and data dependencies between tasks affect each another. The standards insist on such analyses not only to ensure that all couples have been exercised, but also because of their capacity to reveal potential problems.

How an End-to-End Solution Improves the WCET Analysis

Because WCET depends on both software and hardware behavior, developers need tools that instrument code running on the target, capture timing, and trade data in real-time; reveal data, control, and timing coupling; and support reproducible, standards-aligned verification.

Accurate WCET analysis depends on understanding how code behaves once it runs on the target hardware. Together, these tools give developers a complete view of execution behavior, enabling more accurate and defensible WCET analysis:

- **Compilers** influence WCET by shaping the determinism of machine code. Predictable optimizations and stable code generation help ensure execution timing remains consistent.
- **Debuggers** give developers real-time insight into system execution under load. Trace-enabled tools reveal instruction flow, task overlap, memory access patterns, and timing spikes caused by multicore interference.
- **Software quality and verification tools** instrument code to collect detailed runtime measurements. They help developers identify data, functional, and timing coupling; validate execution paths; and correlate results with system requirements. This provides the traceable evidence needed to support safety and certification objectives.

For WCET, hardware analysis is equally essential. It involves identifying all potential interference channels in the target's specific hardware configuration and then designing tests or configuring the system to trigger the worst-case conditions under which WCET should be measured.

Mitigating Timing Coupling to Keep WCET Under Control

Timing coupling occurs when tasks affect each other's execution time, even if they don't share code or data. On multicore systems, hidden dependencies arise through shared caches, memory buses, peripherals, and other hardware resources. These interactions can create unexpected delays, inflating WCET and causing mission-critical tasks to miss deadlines *(Fig. 2)*.

Once interference sources are identified, developers can apply mitigation techniques such as:

- **Minimizing datasets to fit within L1 cache:** Reduces cache contention and prevents tasks from evicting each other's working data.
- **Adjusting multicore configuration:** Assigning core affinity, setting cache modes, or prioritizing tasks helps isolate critical workloads from interference.
- **Using L2 cache partitioning or RTOS-level partitioning:** Ensures each task or core has reserved resources, preventing cross-task contention from inflating WCET.
- **Disabling L2 when determinism outweighs raw performance:** In some systems, predictable timing is more important than throughput.
- **Re-running WCET analysis under full load:** Validates that mitigations work under realistic system conditions.

Other techniques for managing timing coupling include **task scheduling and prioritization; limiting concurrent access to shared peripherals to reduce** unpredictable blocking times when multiple cores attempt to access the same device; and analyzing **control and data dependencies to** identify indirect interactions between tasks that could propagate delays.

Using tracing and runtime instrumentation, developers can verify not only task execution time, but also *why* timing changes occur, pinpoint the source of interference, and confirm that mitigation strategies are effective. This visibility is critical for ensuring that WCET bounds are reliable and that mission-critical tasks meet deadlines under all operating conditions.

LDRA recently conducted a study in conjunction with the U.S. Army, which showed that latent timing problems can cause more than 40% of performance issues for safety-critical and/or timing-critical applications. Such a large increase in timing from interference isn't acceptable. *(See the full paper: Optimizing Data Coupling and Control Coupling in Multi-Core Avionic Software: A Case Study with Xilinx Ultrascale+ Processors)*

The Path Forward

Modern guidance such as A(M)C 20-193 reinforces the need for developers in avionics to provide evidence that multicore interference is understood, WCET is bounded, and deterministic behavior can be guaranteed and not just assumed.

In the automotive and industrial domains, functional-safety (FuSa) standards don't provide such detailed guidance as A(M)C 20-193. However, multicore devices are frequently deployed and missing WCET requirements may also cause safety hazards.

As real-time systems incorporate larger datasets, especially for AI and ML inference, memory-access patterns, cache pressure, and multicore interference will only grow more significant. Developers who rely solely on average execution time or isolated task measurements risk missing critical timing bugs.

With deterministic compilers, trace-capable debuggers, and runtime verification tools, combined with static, dynamic, and coupling analyses, developers gain the system-level visibility needed to identify hidden timing couplings, measure WCET accurately on real hardware, validate deterministic behavior, and build safe, reliable and certifiable systems that meet real-time deadlines.

Ignoring WCET is no longer an option. But with the right tools, measurement techniques, and approach, developers can finally gain control over worst-case execution times and ensure real-time systems remain safe, reliable, and certifiable under all conditions.