# Obtaining Worst-Case Execution Time Bounds on Modern Microprocessors

Daniel Kästner, Markus Pister, Simon Wegener, Christian Ferdinand
AbsInt GmbH
D-66123 Saarbrücken, Germany
info@absint.com

*Abstract*—**Many embedded control applications have real-time requirements. If the application is safety-relevant, worst-case execution time bounds have to be determined in order to demonstrate deadline adherence. If the microprocessor is timing-predictable, worst-case execution time guarantees can be computed by static WCET analysis. For high-performance multi-core architectures with degraded timing predictability, WCET bounds can be computed by hybrid WCET analysis which combines static analysis with timing measurements. This article summarizes the relevant criteria for assessing timing predictability, gives a brief overview of static WCET analysis and focuses on a novel hybrid WCET analysis based on non-intrusive real-time instruction-level tracing.**

***Keywords— worst-case execution time, static analysis, real-time tracing, timing predictability, path analysis, functional safety***

## I. Introduction

In real-time systems the overall correctness depends on the correct timing behavior: each real-time tasks has to finish before its deadline. All current safety standards require reliable bounds of the worst-case execution time (WCET) of real-time tasks to be determined.

With end-to-end timing measurements timing information is only determined for one concrete input. Due to caches and pipelines the timing behavior of an instruction depends on the program path executed before. Therefore, usually no full test coverage can be achieved and there is no safe test end criterion. Techniques based on code instrumentation modify the code which can significantly change the cache and pipeline behavior (probe effect): the times measured for the instrumented software do not necessarily correspond to the timing behavior of the original software.

One safe method for timing analysis is static analysis by Abstract Interpretation which provides guaranteed upper bounds for WCET of tasks. Static WCET analyzers are available for complex processors with caches and complex pipelines, and, in general, support single-core processors and multi-core processors. A prerequisite is that good models of the processor/System on-Chip (SoC) architecture can be determined. However, there are modern high performance SoCs which contain unpredictable and/or undocumented components that influence the timing behavior. Analytical results for such processors are unrealistically pessimistic.

A hybrid WCET analysis combines static value and path analysis with measurements to capture the timing behavior of tasks. Compared to end-to-end measurements the advantage of hybrid approaches is that measurements of short code snippets can be taken which cover the complete program under analysis. Based on these measurements a worst-case path can be computed. The hybrid WCET analyzer TimeWeaver avoids the probe effect by leveraging the embedded trace unit (ETU) of modern processors, like Nexus 5001™ [16], which allows a fine-grained observation of a core's program flow. TimeWeaver reads the executable binary, reconstructs the control-flow graph and computes ranges for the values of registers and memory cells by static analysis. This information is used to derive loop bounds and prune infeasible paths. Then the trace files are processed and the path of longest execution time is computed. The computed time estimate provide valuable feedback for assessing system safety and for optimizing worst-case performance. TimeWeaver also provides feedback for optimizing the trace coverage: paths for which infeasibility has been proven need no measurements; loops for which the analyzed worst-case iteration count has not been measured are reported.

In this article we give an overview of timing predictability in general and provide criteria for selecting suitable WCET analysis methods. We will outline the methodology of hybrid WCET analysis and report on practical experience with the tool TimeWeaver.

## II. Timing Predictability

In general, a system is predictable if it is possible to predict its future behavior from the information about its current state. We consider predictability under the assumption that the hardware works without unexpected errors. Hardware faults like soft errors or transient faults have to be addressed by specific error handling mechanisms to ensure overall system safety.

In [4] the program input and the hardware state in which execution begins are identified as the primary sources of uncertainty in execution time. *Hardware-related timing predictability* can be expressed as the maximal variance in execution time due to different hardware states for an arbitrary but fixed input. Analogously, *software-related timing predictability* corresponds to the maximal variance in execution time due to different inputs for an arbitrary but fixed

hardware state. A basic assumption is uninterrupted program execution without interferences. In a concurrent system, interferences due to concurrent execution additionally have to be taken into account.

To ensure the correct timing behavior it is necessary to demonstrate the deadline adherence of each task. To this end, the worst-case execution time of each task has to be determined, i.e. the concept of software-related predictability as defined above can be reduced to the predictability of the worst-case execution path.

This leads to the following two main criteria for execution time predictability:

- It must be possible to determine an upper bound of the maximal execution time which is guaranteed to hold.
- To enable precise bounds on the maximal execution time to be determined the behavioral variance, i.e. the maximal variance in execution time due to different hardware states, has to be as low as possible. In general, the larger the behavioral variance is
  - the more the execution time depends on the execution history,
  - the less meaningful is one particular execution time measurement in a specific execution context, and
  - the larger can be the gap between the largest measured execution time and the true worst-case execution time.

Even in single-core processors timing predictability is compromised by performance-enhancing hardware mechanisms like caches, pipelines, out-of-order execution, branch prediction and other mechanisms for speculative execution, which can cause significant variations in timing depending on the hardware state. Interestingly hardware speculation has recently been discovered to constitute a critical security vulnerability [21, 19].

For multi-core processors all challenges to timing predictability are relevant that apply to single-core processors. In addition, there are new challenges imposed by the multi-core design. In the following we will first discuss timing predictability on single-core processors and then address specific challenges for multi-core processors.

### A. *Single-Core Processors*

For simple non-pipelined architectures adding up the execution times of individual instructions is enough to obtain a bound on the execution time of a basic block. However, modern embedded processors try to maximize the instruction-level parallelism by sophisticated performance-enhancing features, like caches, pipelines, or speculative execution. Pipelines increase performance by overlapping the executions of consecutive instructions. For timing measurements this means that there may be big variations between the execution times measured with different starting states of the hardware. Furthermore there may be a significant gap between the largest measured execution time and the true worst-case execution time. For a timing analyzer it means that it is not feasible to consider individual instructions in isolation. Instead, they have to be analyzed collectively—together with their mutual interactions—to obtain tight timing bounds. In the following we will give an overview of timing-relevant hardware features and discuss their effect on timing measurements and on static analysis methods.

In general, the challenges for timing analysis of single-core architectures originate from the complexity of the particular execution pipeline and the connected hardware devices. Commonly used performance-enhancing features are caches, pipelines, out-of-order execution, speculative execution mechanisms like static/dynamic branch prediction and branch history tables, or branch target instruction caches. Many of these hardware features can cause *timing anomalies* [29] which render WCET analysis more difficult. Intuitively, a timing anomaly is a situation where the local worst-case does not contribute to the global worst-case. For instance, a cache miss —the local worst-case—may result in a globally shorter execution time than a cache hit because of hardware scheduling effects. In consequence, it is not safe to assume that the memory access causes a cache miss; instead both machine states have to be taken into account. An especially difficult timing anomaly are domino effects [22]: A system exhibits a *domino effect* if there are two hardware states $s$, $t$ such that the difference in execution time (of the same program starting in $s$, $t$ respectively) may be arbitrarily high. E.g., given a program loop, the executions never converge to the same hardware state and the difference in execution time increases in each iteration. In consequence, loops have to be analyzed very precisely and the number of machine states to track can grow high. For timing measurements this means that the difference between measured and true worst-case execution time caused by an incomplete hardware state coverage can grow arbitrarily high.

The article [37] categorizes the timing compositionality of computing architectures according to the presence of timing anomalies. *Fully compositional* architectures, such as the ARM7, contain no timing anomalies; individual components, e.g., basic blocks, can be considered separately and their worst-case information can be combined. *Compositional* architectures only contain bounded timing effects, i.e., additional delays (e.g., due to an access to a shared resource or due to a preemption or interrupt) can be bounded by a constant and added to the local worst-case figures (e.g. TriCore 1797). *Non-compositional* architectures contain *domino effects*, i.e., unbounded anomalies (e.g. PowerPC 755). Depending on the state of the pipeline and the predictors, the occupancy of functional units, and the contents of the caches—i.e., the execution history—an instruction needs only a few or several hundred cycles to complete its execution [8]. A rigorous definition of compositionality is given in [14].

As the runtime of embedded control software often is dominated by load/store operations, memory subsystems nowadays introduce queues before the caches to buffer them and overcome early stall conditions like cache misses. Often this is complemented by fast data forwarding for consecutive accesses into cache lines that have already been requested by previous pending instructions, where the requested data might already be present in the core. This helps to reduce the number of transactions over the slow system bus. In the abstract model of the timing analysis, the representation of these hardware features has to be close to the concrete hardware to achieve

satisfactory analysis precision. Due to their size, especially the dynamic branch prediction and the branch history tables consume a significant number of bits in the abstract state representation which increases the memory consumption of the analysis. Unknown or not precisely known effective addresses of memory requests further increase the timing analysis search space due to the number of possible scenarios (cache hit/miss, fast data forward or not, …). Concerning processor caches, both precision and efficiency depend on the predictability of the employed *replacement policy* [28, 8]. The Least-Recently-Used (LRU) replacement policy has the best predictability properties. Employing other policies, like Pseudo-LRU (PLRU), or First-In-First-Out (FIFO), or Random, yield less precise WCET bounds because fewer memory accesses can be precisely classified. Furthermore, the efficiency degrades because the analysis has to explore more possibilities. Another deciding factor is the write policy. Typically, there are two main options: *write-through* where a store is directly written to the next level in the memory hierarchy, and *write-back* where the data is written into the next hierarchy level if the concrete memory cell is evicted from the cache. The write-back policy induces timing uncertainty because the precise point in time when the write-back occurs is hard to predict; for example, it might happen after a task switch and slow down a different (and possibly higher-priority) task than the one that issued the store operation in the first place. Another timing analysis challenge is to model processor external devices which are typically connected with the caches over the system bus. Such devices are memory controllers for static (SRAM, Flash) or dynamic memory (DRAM, DDR or QDR) or controllers for system communication (CAN, FlexRay, AFDX). The corresponding bus protocol and memory chip timing have to be modeled precisely.

Individually, each of the above features can be modeled without complexity problems. Only their combination can actually result in a large number of possible system states during the abstract simulation of a basic block. Smart system configurations as described in [18] can decrease both the execution time variability and the analysis complexity. In consequence, the complexity of timing analysis decreases such that highly complex processors like the Freescale PowerPC 7448 can be handled. At the same time the accuracy of timing measurements will be improved.

Some events in modern architectures are either asynchronous to program execution (e.g., interrupts, DMA) or not predictable in the model (e.g., ECC errors in RAM or some hardware exceptions). Their effect on the execution time has to be incorporated externally, i.e., by adding penalties based on the worst-case occurrence of the events to the computed WCET, or by statistical means.

### B. Multi-Core Processors

Whereas timing analysis of single-core architectures already is quite challenging, the timing behavior of multi-core architectures is even more complex. A multi-core processor is a single computing component with two or more independent cores; it is called homogeneous if it includes only identical cores, otherwise it is called heterogeneous. Thus, all characteristic challenges from single-cores are still present in the multi-core design, but the multiple cores can independently run multiple instructions at the same time. Some multi-core processors can be run in lockstep mode where all cores execute the same instruction stream in parallel. This typically eliminates interferences between the cores, so from a timing perspective the processor behaves like a single-core.

When the processor is not run in lockstep mode, the inter-core parallelism becomes relevant. To interconnect the several cores, buses, meshes, crossbars, and also dynamically routed communication structures are used. In that case, the interference delays due to conflicting, simultaneous accesses to shared resources (e.g. main memory) are the main cause of imprecision. On a single-core system, the latency of a memory access mostly depends on the accessed memory region (e.g. slow flash memory vs. fast static RAM) and whether the accessed memory cell has been cached or not. On a multicore system, the latency also depends on the memory accesses of the other cores, because multiple simultaneous accesses might lead to a resource conflict, where only one of the accesses can be served directly, and the other accesses have to wait. The shared physical address space requires additional effort in order to guarantee a *coherent* system state: Data resident in the private cache of one core may be invalid, since modified data may already exist in the private cache of another core, or data might have already been changed in the main memory. Thus, additional communication between different cores is required and the execution time needed for this has to be taken into account. Multi-core processors which can be configured in a timing-predictable way to avoid or bound inter-core interferences are amenable to static WCET analysis [18, 36]. Examples are the Infineon AURIX TC275 [17], or the Freescale MPC 5777.

The Freescale P4080 [13] is one example of a multicore platform where the interference delays have a huge impact on the memory access latencies and cannot be satisfactorily predicted by purely static techniques. It consists of eight PowerPC e500mc cores which communicate with each other and the main memory over a shared interconnect, the CoreNet Coherency Fabric. The main problem for static analysis approaches is that the publically available documentation about the CoreNet is not enough to statically predict its behavior. Nowotsch et al. [24] measured maximal write latencies of 39 cycles when only one core was active, and maximal write latencies of 1007 cycles when all eight cores were running. This is more than 25 times longer than the observed best case. A sound WCET analysis must take the interference delays into account that are caused by resource conflicts. Unless interference is avoided by means of the overall software architecture, ignoring these delays might result in underestimation of the real WCET whereas assuming full interferences at all times might result in huge overestimation.

To improve predictability of avionics systems the Certification Authorities Software Team (CAST) [5] advocates to either deactivate or control existing interference channels. If deactivation is not possible the software architecture has to be able to prevent or bound the interferences. One hardware element where such mechanisms are required is the interconnect, i.e., the Network-on-Chip (NoC) or shared bus connecting main memory to the individual cores. Several

approaches to address interference on shared memory accesses have been discussed in literature, most of them in the context of Integrated Modular Avionics (IMA). They typically rely on a time-triggered static scheduling scheme, e.g., corresponding to the avionics standard ARINC 653. As an example, with the approaches of [30] or [24] precise static WCET bounds can be computed, albeit at the cost of high computational complexity. For systems which do not implement such rigorous software architectures or where the information needed to develop a static timing model is not available, hybrid WCET approaches are the only solution.

## III. WCET Guarantees on Predictable Processors

The most successful formal method for WCET computation is Abstract Interpretation-based static program analysis. Static program analyzers compute information about the software under analysis without actually executing it. Semantics-based static analyzers use an explicit (or implicit) program semantics that is a formal (or informal) model of the program executions in all possible or a set of possible execution environments. Most interesting program properties—including the WCET—are undecidable in the concrete semantics. The theory of abstract interpretation [6] provides a formal methodology for semantics-based static analysis of dynamic program properties where the concrete semantics is mapped to a simpler abstract model, the so-called abstract semantics. The static analysis is computed with respect to that abstract semantics, enabling a trade-off between efficiency and precision. A static analyzer is called *sound* if the computed results hold for any possible program execution. Applied to WCET analysis, soundness means that the WCET bounds will never be exceeded by any possible program execution. Abstract interpretation supports formal soundness proofs for the specified program analysis. Like model checking and theorem proving, it is recognized as a formal method by the DO-178C and other safety standards (cf. Formal Methods Supplement [26] to DO-178C [27]). It is based on a mathematically rigorous concept and provides the highest possible confidence in the correctness of the results (cf. IEC-61508, Ed. 2.0 [15], Table C.18).

In addition to soundness, further essential requirements for static WCET analyzers are *efficiency* and *precision*. The analysis time has to be acceptable for industrial practice, and the overestimation must be small enough to be able to prove the timing requirements to be met.

Over the last few years, a more or less standard architecture for timing analysis tools has emerged [9, 11]. It neither requires code instrumentation nor debug information and is composed of three major building blocks:

- control-flow reconstruction and static analyses for control and data flow,
- micro-architectural analysis, computing upper bounds on execution times of basic blocks,
- path analysis, computing the longest execution paths through the whole program.

The data flow analysis of the first block also detects infeasible paths, i.e., program points that cannot occur in any real execution. This reduces the complexity of the following micro-architectural analysis. Basic block timings are determined using an abstract processor model (*timing model*) to analyze how instructions pass through the pipeline taking cache-hit/ cache-miss information into account. This model defines a cycle-level abstract semantics for each instruction's execution yielding in a certain set of final system states. After the analysis of one instruction has been finished, these states are used as start states in the analysis of the successor instruction(s). Here, the timing model introduces non-determinism that leads to multiple possible execution paths in the analyzed program. The pipeline analysis has to examine all of these paths.

In the following sections we will focus on the commercially available tool aiT [1] which implements the architecture described above. It is centered around a precise model of the microarchitecture of the target processor and is available for various 16-bit and 32-bit single-core and multi-core microcontrollers. aiT determines the WCET of a program task in several phases corresponding to the reference architecture described above, which makes it possible to use different methods tailored to each subtask [34]. In the following we will give an overview of each analysis stage.

- In the *decoding phase* the instruction decoder reads and disassembles the input executable(s) into its individual instructions. Architecture specific patterns decide whether an instruction is a call, branch, return or just an ordinary instruction. This knowledge is used to reconstruct the basic blocks of the control flow graph (CFG) [33]. Then, the control flow between the basic blocks is reconstructed. In most cases, this is done completely automatically. However, if a target of a call or branch cannot be statically resolved, then the user can write some annotations to guide the control flow reconstruction.
- The combined loop and value analysis determines safe approximations of the values of processor registers and memory cells for every program point and execution context. These approximations are used to determine bounds on the iteration number of loops and information about the addresses of memory accesses. Contents of registers or memory cells, loop bounds, and address ranges for memory accesses may also be provided by annotations if they cannot be determined automatically. Value analysis information is also used to identify conditions that are always true or always false. Such knowledge is used to infer that certain program parts are never executed and therefore do not contribute to the worst-case execution time or the stack usage.
- In the micro-architectural analysis phase cache and pipeline analysis has to be combined because the pipeline analysis models the flow of instructions through the processor pipeline and therefore computes the precise instant of time when the cache is queried and its state is updated. The combined cache and pipeline analysis represents an abstract interpretation of the program's execution on the underlying system architecture. The execution of a program is simulated by feeding instruction sequences from a control-flow graph to the timing model which computes the system state changes at cycle granularity and

keeps track of the elapsing clock cycles. The correctness proofs according to the theory of abstract interpretation have been conducted by Thesing [35]. The cache analysis presented by [10] is incorporated into the pipeline analysis. At each point where the actual hardware would query and update the contents of the cache(s), the abstract cache analysis is called, simulating a safe approximation of the cache effects. The result of the cache/pipeline analysis either is a worst-case execution time for every basic block, or a *prediction graph* that represents the evolution of the abstract system states at processor core clock granularity [7].

- The path analysis phase uses the results of the combined cache/pipeline analysis to compute the worst-case path of the analyzed code with respect to the execution timing. The execution time of the computed worst-case path is the worst-case execution time for the program. Within the aiT framework, different methods for computing this worst-case path are available.

aiT has been successfully employed in the avionics [12, 11, 31] and automotive [23] industries to determine precise bounds on execution times of safety-critical software. It is available for a variety of microprocessors ranging from simple processors like ARM7 to complex superscalar processors with timing anomalies and domino effects like Freescale MPC755, or MPC7448, and multicore processors like Infineon AURIX TC27x.

## IV. Hybrid WCET Analysis

Techniques to compute worst-case execution time information from measurements are either based on end-to-end measurements of tasks, or they construct a worst-case path from timing information obtained for a set of smaller code snippets in which the executable code of the task has been partitioned. With end-to-end timing measurements, timing information is only determined for one concrete input. As described above, due to caches and pipelines the timing behavior of an instruction depends on the program path executed before. Therefore, usually no full test coverage can be achieved and there is no safe test end criterion. Approaches that instrument the code to obtain timing information about the code snippets of a task modify the code which can significantly change the cache and pipeline behavior (probe effect): the times measured for the instrumented software do not necessarily correspond to the timing behavior of the original software.

The solution which is implemented in the hybrid WCET analysis tool TimeWeaver [2] combines static context-sensitive path analysis with non-intrusive real-time instruction-level tracing to provide worst-case execution time estimates. By its nature, an analysis using measurements to derive timing information is aware of timing interference due to concurrent execution and multicore resource conflicts, because the effects of asynchronous events (e.g. activity of other running cores or DRAM refreshes) are directly visible in the measurements. The probe effect is completely avoided since no code instrumentation is needed. The computed estimates are safe upper bounds with respect to the given input traces, i.e., TimeWeaver derives an overall upper timing bound from the execution time observed in the given traces. Thus, the coverage of the input traces on the analyzed code is an important metric that influences the quality of the computed WCET estimates.

The trace information needed for running TimeWeaver is provided out-of-the-box by embedded trace units of modern processors, like NEXUS IEEE-ISTO 5001™ [16] or ARM CoreSight™ [3]. They allow the fine-grained observation of a program execution on single-core and multicore systems. Examples for processors supporting the NEXUS trace interface are the NXP QorIQ P- and T-series processors (using either an e500mc or an e5500/e6500 core).

### A. *NEXUS Traces*

On the PowerPC architecture TimeWeaver relies on NEXUS program flow trace messages. Such traces consist of trace segments separated by trace events. TimeWeaver maps the events to points in the control-flow graph (trace points) and the segments to program paths between these points. This is done for those parts of the trace that reach from the call of the routine used as analysis entry till the end of that routine or any other feasible end of execution. Such parts are called trace snippets. A single trace may contain several trace snippets. TimeWeaver can operate on one or more traces given as trace files, each containing one or more trace snippets.

A NEXUS trace event encodes its type, a time stamp containing the elapsed CPU cycles since the last trace event and the contents of the branch history buffer, which can be used to reconstruct execution path decisions and allows to map trace segments to the control-flow graph of the corresponding executable.

Microprocessor debugging solutions like the Lauterbach PowerDebug Pro [20] allow to record NEXUS trace events as they are emitted during program execution and to export them in various formats. TimeWeaver can process those exports for its timing analysis as described below.

Here is a sample NEXUS trace excerpt (with some information removed) in ASCII format:

```
+056 TCODE=1D PT-IBHSM F-ADDR=F1F4 HIST=2 TS=8847
+064 TCODE=21 PT-PTCM EVCODE=A TS=88F1
+072 TCODE=1C PT-IBHM U-ADDR=03DC HIST=1 TS=8D62
+080 TCODE=21 PT-PTCM EVCODE=A TS=8E2F
+088 TCODE=21 PT-PTCM EVCODE=A TS=8FBA
+096 TCODE=21 PT-PTCM EVCODE=A TS=9105
+104 TCODE=1C PT-IBHM U-ADDR=02CC HIST=1 TS=9275
+112 TCODE=1C PT-IBHM U-ADDR=01F0 HIST=1 TS=93BF
+120 TCODE=21 PT-PTCM EVCODE=A TS=997B
+128 TCODE=1C PT-IBHM U-ADDR=0044 HIST=1 TS=9B02
+136 TCODE=21 PT-PTCM EVCODE=A TS=9F21
```

This output has been generated using the following command in the Lauterbach Trace32 tool:

```
Trace.export.ascii <file> nexus /showRecord
```

Each line corresponds to a trace event. The number at the beginning of the line is the trace record number. The second and third column represent the particular trace event type followed by type-specific information like branch history and program address information associated with the event. The `TS` number at the end is a time stamp.

Debugging solutions differ in the format in which they export trace data. Some debuggers allow the user to configure the output. TimeWeaver can currently import traces which have been exported by Lauterbach, PLS or iSYSTEM debuggers. Whenever the format is configurable, we have identified a minimal set of information needed to perform the TimeWeaver analysis. Additionally, TimeWeaver can be easily extended to support other trace formats.

### B. *TimeWeaver Toolchain*

The main inputs for TimeWeaver are the fully linked executable(s), timed traces and the location of the analyzed code in the memory (entry point, which usually is the name of a task or function). Optionally, users can specify further semantical information to the analysis, like targets of computed calls, loop bounds, values of registers and memory cells. This information is used to fine-tune the analysis. The analysis proceeds in several stages: decoding, loop/value analysis, trace analysis, and path analysis. Most steps in this tool chain are shared with aiT, leveraging its powerful static analysis framework.

The decoding phase of TimeWeaver is mostly identical to the decoding phase of aiT. One important difference is that when encountering call targets which cannot be statically resolved, TimeWeaver can be instructed to extract the targets of unresolved branches or calls from the input traces. To this end there is a feedback loop between the CFG reconstruction and the trace analysis step (cf. Fig. 1). As an alternative, the same user annotations can be used as in the aiT tool chain.

In the next phase, several microarchitectural analyses are performed on the reconstructed CFG starting with the combined *loop and value analysis*, again equal to the aiT tool chain. It determines possible values of registers and memory cells, addresses of memory accesses, as well as loop and recursion bounds. Based on this, statically infeasible paths are computed, i.e., parts of the program that cannot be reached by any execution under the given configuration. This is important because each detected infeasible path increases the trace coverage. Such paths are pruned from further analysis. If the value analysis cannot compute a loop bound or if the computed bound is not precise enough, users can specify custom bounds by means of annotations which are used by the analysis. The loop transformation allows loops in the CFG to be handled as self-recursive routines to improves analysis precision [32].

After value analysis, the analyzer has annotated each instruction in the control-flow graph with context-sensitive analysis results. This context-sensitivity is important because the precision of an analysis can be improved significantly if the execution environment is considered [32]. For example, if a routine is called with different register values from two different program points, the execution time in both situations might be different. Depending on the context settings, this is taken into account leading to higher precision in the analysis result.

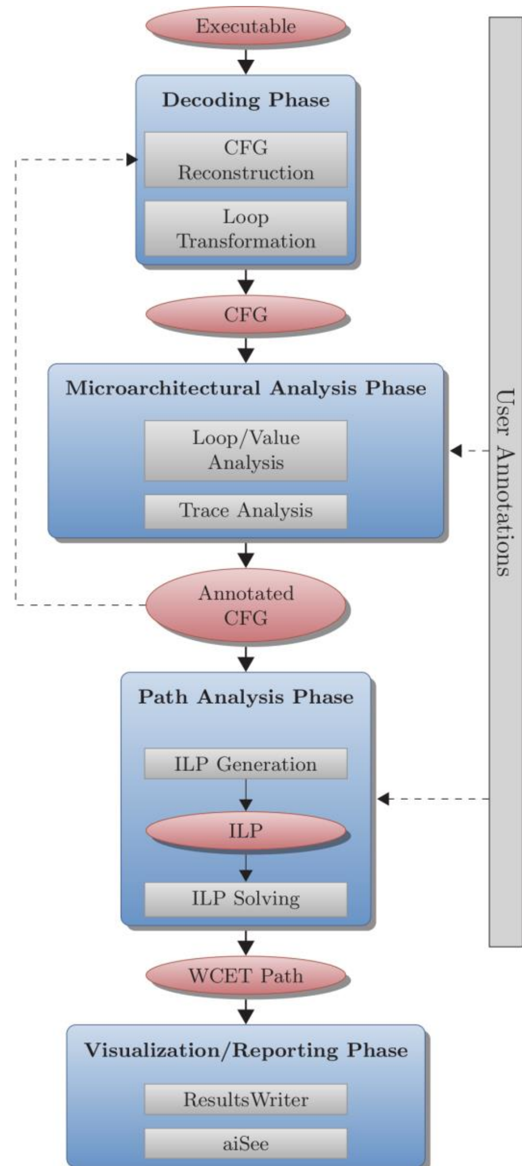


Fig. 1. TimeWeaver tool chain structure

In the *trace analysis* step the given traces are analyzed such that each trace event is mapped to a program point in the control-flow graph. This mapping defines the trace points and segments mentioned above and is not only necessary for the whole analysis but also ensures that the input trace matches the analyzed binary. In case a preemptive system has been traced, interrupts are detected and reported. The extracted timing information, i.e., the clock cycles which have been elapsed between two consecutive trace points are annotated to the CFG in a context-sensitive manner.

After the trace conversion, a CFG which combines the results of value analysis and traced execution timings (both context-sensitive) is available. This graph is the input for the next step, the path analysis phase. Here, the trace segment times alongside the control-flow graph are used to generate an integer linear program (ILP) formulation to compute the worst-case execution path with respect to the traced timings. At this point, the recorded times for each pair of trace segment and

analysis context, get maximized. The ILP formulation is structurally the same as in the path analysis of aiT [33] with the exception that the involved execution times are not computed by a micro-architectural pipeline analysis but are extracted from the input traces. The generated ILP is fed to a solver whose solution is the worst-case execution path alongside its costs, i.e., the WCET estimate of the analyzed task. This solution is annotated to the CFG for the final step, namely reporting and visualization.

As mentioned above, the input traces might contain asynchronous events like DRAM refreshes which can lead to exceptionally high trace segment times. TimeWeaver allows to address these with a filter for trace segment times based on their cumulative frequency (CF), i.e. their occurrence percentage. The threshold refers to a percentage of occurrences ordered by execution times (as in the survival graph, see below). A threshold of 0% is passed by all occurrences. A threshold of 5% is passed by all but the 4 most expensive ones (in terms of execution time) if there are 100 occurrences, by all but the 9 most expensive ones if there are 200 occurrences, etc. Trace segment times that do not pass the specified threshold are ignored in the ILP generation. The filter function is applied for each trace segment separately. TimeWeaver also allows to simulate the effect of the CF filter in its statistics view to experiment with different filter values.

### C. TimeWeaver Result Reporting and Visualization

Besides the global WCET estimate and the execution path triggering it, TimeWeaver offers a variety of reporting facilities:

- WCET estimate per routine (including cumulative information of called sub routines),
- Context-specific WCET estimate per routine (including cumulative information of called sub routines),
- Determined loop bounds (distinguishes between traced, analyzed, and effective bounds) including loop scaling conflicts,
- Variance of trace segment times (context-sensitive),
- Trace coverage with respect to the number of basic blocks and instructions in the analyzed code, and
- Memory access information along the computed worst-case path.

In addition to the above described statistics, TimeWeaver provides the following visualizations:

- Analysis result graph to interactively explore the results,
- Per trace segment distribution graph for the recorded segment times (cf. Fig. 2), and a
- Per trace segment survival graph to show the cumulative frequency of the recorded segment times (cf. Fig. 3).

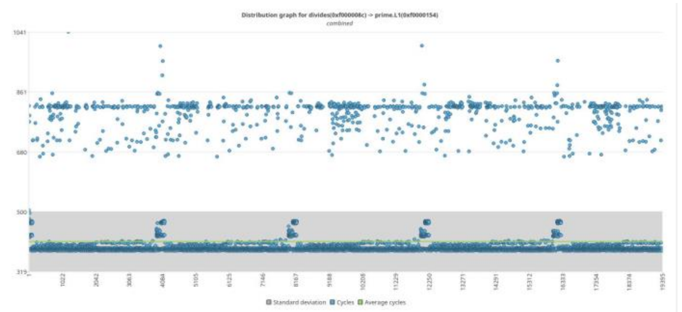

Fig. 2. Sample distribution graph of a trace segment

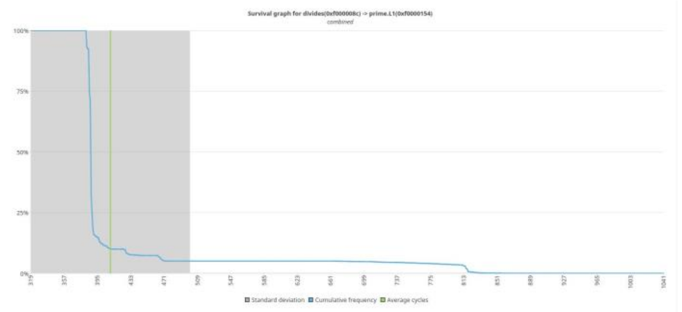

Fig. 3 Sample survival graph of a trace segment

### D. WCET Estimate Extrapolation

As mentioned above, TimeWeaver computes the global WCET estimate based on the observed execution times of trace segments. The times are maximized per trace segment and the maximized times are composed to identify the worst-case path with respect to those figures.

Where in general, one would need to measure all possible execution paths (which is impractical on real-world applications) of the analyzed program for coverage reasons, TimeWeaver allows to compute an upper bound on the global execution time of the analyzed program based on the trace segment times extracted from the input traces.

This way, it is only necessary to trace all possible execution paths between two consecutive trace points. By inserting custom trace points, the user can further decrease the required number of measurements. Fig. 4 illustrates this by showing three consecutive trace points (TP1, TP2, and TP3) and the possible execution paths between each of them. TimeWeaver composes the WCET estimate for the time between TP1 and TP3 by the sum over the maximized trace segment time between TP1→TP2 and the maximized trace segment time between TP2→TP3. Thus, the measurements need to cover the four execution paths between TP1→TP2 as well as between three execution paths between TP2→TP3. Without that time composition, all 12 execution paths between TP1→TP3 need to be measured.

### E. Loop Scaling

For loops, there might be a gap between the maximum of the observed iteration counts in the input traces (*traced bound*) and the statically possible maximum iteration count (*analyzed bound*) which is computed by the value analysis. The bound

actually used for the ILP generation is the so-called *effective bound* which is the analyzed bound if it is finite and applicable (cf. scaling conflicts below) and otherwise the traced bound. Per user request, the intersection of analyzed and traced bound is used.

If the effective bound is higher than the traced bound, the maximum observed execution time (context-sensitively) for one loop iteration is scaled up to the effective bound. This overcomes the necessity to trace each loop in the analyzed task with its worst-case iteration count, which might be hard to achieve because loop conditions often are data-dependent and thus can be complex to trigger.

However, loop scaling as described above is not always directly applicable. It requires each trace to pass a trace point inside the loop body. If there is at least one traced execution path through the loop body without a trace point, scaling cannot be done and only the traced bounds are used for this loop. Such a situation is called an event loop scaling conflict. The solution is to either trace the worst-case loop iteration count or to ensure that each traced path through the loop body passes a trace point (by inserting custom trace points).

There is another situation which triggers a loop scaling conflict: if due to the context settings of the analysis a loop is virtually unrolled more times than the corresponding loop body has been executed in the trace, scaling cannot be applied, as well. The reason is that the scaling is applied in the last loop context, i.e., in that context which represents the last loop iteration(s). In that case, there is no traced loop body time in the trace mapped to this context which prevents scaling. Such a conflict is called a unroll loop scaling conflict. To solve this conflict, one can either trace the worst-case iteration count of the corresponding loop or the (virtual) loop unroll during analysis of this particular loop can be decreased to the traced bound.

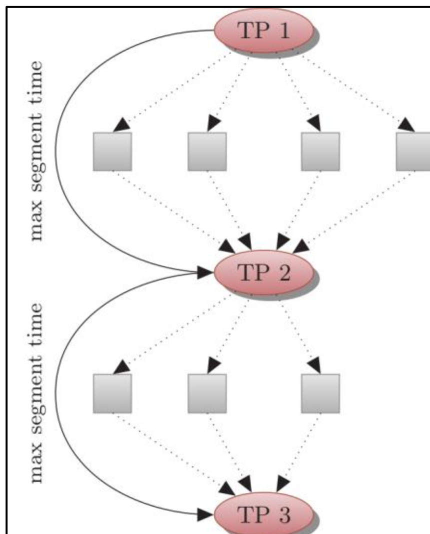


Fig. 4 Execution paths between trace points

## V. Experimental Results on TimeWeaver

To evaluate TimeWeaver for PowerPC, we recorded program executions on an NXP T1040 [25] evaluation board using a Lauterbach PowerDebug Pro JTAG debugger.

### A. Loop scaling

Execution times for loops can be scaled up from the maximum observed execution time of the loop body. This can be seen in the analysis of the following program:

```
1 volatile int sensor;
2
3 int helper (int x)
4 {
5   int result = x;
6   result += sensor;
7   return result + 3;
8 }
9
10
11 int main (void)
12 {
13   int result = 0;
14
15   result += helper(256);
16
17   int i;
18   int loop_bound = (sensor-0xDEADBEEF)+5;
19
20   /* Loop with statically unknown bound */
21   for (i=0 ; i<loop_bound ; ++i) {
22     result += 1024;
23     result += helper(128);
24   }
25
26   result += helper(256);
27
28   return result + 1;
29 }
```

Before starting measurements, we stored the value `0xDEADBEEF` in the memory cell representing the variable `sensor`, so that the traced loop bound is 5 due to the assignment in line 18. However, the code has been written in a way that the static analysis cannot compute a finite loop bound for this loop because the value for `sensor` is not known statically. Without any user input, TimeWeaver uses the traced loop bounds in its analysis and computes a WCET estimate of 8.5 µs. Analyzing again with a loop bound of 10 yields an estimate of 16.7 µs. This is expected as TimeWeaver now scales up the time for the loop body to 10 rounds. The WCET estimate does not precisely doubles because of the contribution of the surrounding code to the execution time.

### B. Precision

To show the precision of the computed WCET estimates, we compared TimeWeaver results with the maximum end-to-end times seen in those input traces we have fed to TimeWeaver. For comparison reasons, loop bounds in the analysis have been chosen equal to the traced loop bounds. The following table shows the results.

| Application | Trace [cycles] | Estimate [cycles] | Diff [%] |
|---|---|---|---|
| crc | 809068 | 829039 | 2.47 |
| edn | 4788025 | 4791420 | 0.07 |
| eratosthenes sieve | 368345 | 369803 | 0.40 |
| dhrystone | 168093 | 177314 | 5.49 |
| md5 | 127857 | 131718 | 3.02 |
| nestedDepLoops | 2747357 | 2747359 | 0.00 |
| sha | 23426161 | 23815350 | 1.66 |
| Avionics Task | 420677 | 498028 | 18.38 |
| Automotive Task 1 | 65058 | 71964 | 10.62 |
| Automotive Task 2 | 27215 | 28967 | 6.44 |
| Automotive Task 3 | 17386 | 18595 | 6.95 |
| Automotive Task 4 | 101749 | 109302 | 7.42 |

Tab. 1. TimeWeaver Result Comparison

For each application, the maximum observed end-to-end time has been extracted from the traces and compared with the WCET estimate computed by TimeWeaver. The difference represents the overestimation of TimeWeaver resulting from the composition of trace segment times to a global estimate. In average, the TimeWeaver results from the table above are 5.24% above the maximum observed end-to-times from the traces.

## VIII. CONCLUSION

In this article we have given a definition of timing predictability and discussed hardware features which increase the difficulty of obtaining safe and precise worst-case execution time bounds, both on single-core and multicore processors. We have described the methodology of static worst-case execution time analysis which can provide guaranteed WCET bounds on complex processors, if the timing behavior of the processor is well specified, and asynchronous interferences can be controlled or bounded. Hybrid worst-case execution time analysis allows to obtain worst-case execution time bounds even for systems where these conditions are not met. We have given an overview of the hybrid WCET analyzer TimeWeaver which combines static value and path analysis with timing measurements based on non-intrusive instruction-level real-time traces. The trace information covers interference effects, e.g., by accesses to shared resources from different cores, without being distorted by probe effects since no instrumentation code is needed. The analysis results include the computed WCET bound with the time-critical path, and information about the trace coverage obtained. They provide valuable feedback for optimizing trace coverage, for assessing system safety, and for optimizing worst-case performance. Experimental results show that with good trace coverage safe and precise WCET bounds can be efficiently computed.

## ACKNOWLEDGMENT

This work was funded within the project ARAMiS II by the German Federal Ministry for Education and Research (BMBF) with the funding ID 01IS16025B, and within the BMBF project EMPHASE with the funding ID 16EMO0183. The responsibility for the content remains with the authors.

## REFERENCES


[1] AbsInt GmbH. aiT Worst-Case Execution Time Analyzer Website. http://www.AbsInt.com/ait.

[2] AbsInt GmbH. TimeWeaver Website. http://www.AbsInt.com/-timeweaver.

[3] ARM Ltd. Coresight$^{TM}$ Program Flow Trace$^{TM}$ PFTv1.0 and PFTv1.1 architecture specification, 2011. ARM IHI 0035B.

[4] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems*, 13(4):82:1–82:37, 2014.

[5] Certification Authorities Software Team (CAST). Position Paper CAST-32A Multi-core Processors, November 2016.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4$^{th}$ POPL*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

[7] Christoph Cullmann. *Cache persistence analysis for embedded real-time systems*. PhD thesis, Universitaet des Saarlandes, Postfach 151141, 66041 Saarbruecken, 2013.

[8] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of Embedded Real Time Software and Systems*, pages 36–42, May 2010.

[9] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[10] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[11] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer-Verlag, 2001.

[12] Christian Ferdinand and Reinhard Wilhelm. Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[13] Freescale Inc. *QorIQTM P4080 Communications Processor Product Brief*, September 2008. Rev. 1.

[14] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: Definition and challenges. *SIGBED Rev.*, 12(1):28–36, March 2015.

[15] IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.

[16] IEEE-ISTO. IEEE-ISTO 5001$^{TM}$-2012, The Nexus 5001$^{TM}$ Forum Standard for a Global Embedded Processor Debug Interface, 2012.

[17] Infineon Technologies AG. *AURIXTM TC27x D-Step User's Manual*, 2014.

[18] D. Kästner, M. Schlickling, M. Pister, C. Cullmann, G. Gebhard, R. Heckmann, and C. Ferdinand. Meeting Real-Time Requirements with Multi-Core Processors. *Safecomp 2012 Workshop: Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR)*, September 2012.

[19] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[20] Lauterbach GmbH. Lauterbach Website. http://www.lauterbach.com.

[21] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[22] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium (RTSS)*, December 1999.

[23] NASA Engineering and Safety Center. Technical Support to the National Highway Traffic Safety Administration (NHTSA) on the Reported Toyota Motor Corporation (TMC) Unintended Acceleration (UA) Investigation, 2011.

[24] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *ECRTS'14: Proceedings of the 26th Euromicro Conference on Real-Time Systems*, July 2014.

[25] NXP Semiconductors. *QorIQTM T1040 Reference Manual*, 2015.

[26] Radio Technical Commission for Aeronautics. Formal Methods Supplement to DO-178C and DO-278A, 2011.

[27] Radio Technical Commission for Aeronautics. RTCA DO-178C. Software Considerations in Airborne Systems and Equipment Certification, 2011.

[28] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[29] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In Frank Mueller, editor, *International Workshop on Worst-Case Execution Time Analysis (WCET)*, July 2006.

[30] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing predictability on multi-processor systems with shared resources. In *Workshop on Reconciling Performance with Predictability (RePP), 2010*, October 2009.

[31] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios, and Reinhold Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

[32] Stefan Stattelmann and Florian Martin. On the Use of Context Information for Precise Measurement-Based Execution Time Estimation. In B. Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 64–76. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[33] Henrik Theiling. *Control Flow Graphs for Real-Time System Analysis. Reconstruction from Binary Executables and Usage in ILP-Based Path Analysis*. PhD thesis, Saarland University, 2003.

[34] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, December 1998.

[35] Stephan Thesing. *Safe and Precise Worst-Case Execution Time Prediction by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[36] Simon Wegener. Towards Multicore WCET Analysis. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[37] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Markus Pister, Marc Schlickling, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future time-critical embedded architectures. *IEEE TCAD*, 28(7):966–978, July 2009.