Compilation for Real-Time Systems - An Overview of the WCET-Aware C Compiler WCC

Dominic Oehlert
Institute of Embedded Systems
Hamburg University of Technology
Germany
dominic.oehlert@tuhh.de

Arno Luppold
Institute of Embedded Systems
Hamburg University of Technology
Germany
arno.luppold@tuhh.de

Heiko Falk
Institute of Embedded Systems
Hamburg University of Technology
Germany
heiko.falk@tuhh.de

***Abstract*—Traditionally, design of embedded hard real-time software and timing analysis are decoupled from each other, leading to complicated design flows involving human interaction. Furthermore, traditional compilers optimize for average-case performance so that no tool support exists supporting the designer to systematically reduce Worst-Case Execution Times in case that deadlines are missed. The WCET-aware C Compiler WCC improves this situation by tightly schedulability analyses) into the compilation and optimization flow. Furthermore, the compiler features dedicated real-time aware optimizations and exploits detailed architectural knowledge so that schedulable code meeting deadlines can be generated automatically, even for multitask or multicore systems.**

I. Introduction

Developing for hard real-time systems can be a tedious task. Programs running on hard real-time systems always have to meet their temporal deadlines, otherwise the system is not real-time capable. While the worst-case execution time (WCET) of a program strongly depends on several, mostly low-level, characteristics of the architecture and the program's machine code, the source code is typically written in an abstract high-level programming language. In case it turns out a program does not meet its deadline, the programmer has to return to the source code and try to optimize program parts manually which may be part of the worst-case execution path (WCEP). WCET analyzers, such as AbsInt's aiT [1] or OTAWA [2], help the user to find out which particular parts of the program actually lie on the WCEP. Yet, this is an additional manual process and requires to generate architecture- and program-specific configuration files. This manual optimization cycle of compiling, analyzing and modifying can be lengthy and is easily error-prone.

On the other hand, the programmer can choose from a plethora of optimizations in modern mainstream compilers. Anyhow, these optimizations typically exploit heuristics and aim at reducing the average-case execution time (ACET) of a program. As a consequence, these ACET optimizations are not guaranteed to improve the timing or may even deteriorate the WCET of a program.

To tackle these issues and to introduce a unified design flow for hard real-time systems, the WCET-aware C compiler (WCC) [3] consists of sophisticated WCET-oriented analyses and optimizations. It offers automated WCET or schedulability analyses and optimizations for several architectures, such that the user does not need to manually interact with the different tools. In the following, we will give a brief overview of the general structure, analyses and optimizations of the WCC. The paper is organized as follows: Section II gives a general overview of the structure of the WCC. Section III presents featured analyses as well as an exemplary set of WCET-oriented optimizations. Section IV presents novel features currently being implemented and concludes this paper.

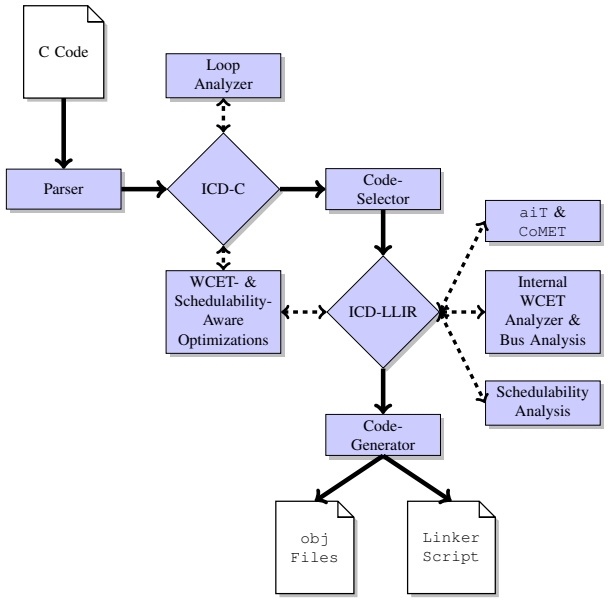

Fig. 1. Overall WCC Workflow

II. Structure

The overall structure and workflow of the WCC can be seen in Fig. 1. Solid lines depict the workflow of a typical compiler, whereas the dashed lines show WCC specific parts. The WCC is a C compiler currently targeting the Infineon TriCore TC1796 and TC1797 processors as well as the ARM7TDMI processor. The platform-independent C-like high-level intermediate representation (ICD-C) is transformed to the assembler-like low-level intermediate representation

(ICD-LLIR) via the code-selector. Furthermore, it is possible to annotate information from the low-level intermediate representation back to the corresponding high-level constructs through a back-annotation. This enables the user to perform typical high-level optimizations (such as, e.g., loop unrolling) with detailed low-level knowledge of the generated code (e.g., timings or code size). In order to describe the actual memory layout of the chosen processor, the WCC features an adaptable memory hierarchy specification. Architecture-dependent characteristics such as memory regions, their worst-case latencies, caches and bus bandwidths are described within.

The WCC also supports user-provided information concerning the control flow of a program. These so-called flow facts can be directly annotated into the source code of the program and describe, e.g., the maximum (resp. minimum) number of loop iterations or the maximum recursion depth. Besides, loop bounds may also be derived automatically using an integrated loop analyzer.

III. Features

The WCC contains plenty of useful features in order to compile, optimize and analyze a single task or a set of tasks for an embedded hard real-time system.

A. Analyses

As embodied by the name, the WCC is dedicated to perform a WCET-aware compilation and optimization. In order to do so, it includes several analyses. The WCC is tightly coupled with AbsInt's WCET analyzer aiT, which enables the user to analyze the WCET of a single task automatically while compiling. The analysis is fully automated, such that the user can call the WCC with a corresponding flag such that a program is compiled and a WCET analysis is carried out afterwards. Required configuration files (containing, e.g, architecture-dependent latencies or program-dependent information like loop bounds) are generated and the required timing analysis tools are called completely transparent to the user. The resulting values, such as the global WCET, worst-case execution counts, cache hits and misses, etc., are annotated back to the compiler's intermediate representations. Based on this information further WCET-oriented optimizations can be performed.

Recently, a holistic schedulability analysis framework was incorporated into the WCC [4]. It is based on the event model by Gresser [5], allowing for the analysis of systems featuring arbitrary activation patterns and deadlines. The framework supports both preemptive fixed priority scheduling algorithms like Deadline Monotonic Scheduling as well as dynamic priority scheduling like Earliest Deadline First. WCC's low-level analysis techniques allows to automatically include detailed information on timing overheads into the analysis framework. This does not only comprise penalties due to the system's real-time scheduler, but also context switching costs and cache-related preemption delays.

Additionally, the WCC has an own internal WCET analyzer featuring a precise timing estimation for complex ARM7TDMI-based multicore systems [6]. This analyzer is capable of analyzing a system with up 8 parallel homogeneous cores connected via configurable bus. Supported bus arbitration policies include Time Division Multiple Access (*TDMA*), Round Robin, Fixed Priority and Priority Division [7]. Beside shared and private memories, the analyzer is also able to analyze hierarchal cache structures, both shared and private. The analyzer comprises multicore-specific configuration settings (such as, e.g., a may-happen-in-parallel analysis on a basic block level) to offer an adjustable trade-off between tightness and speed.

Bridging the gap between system-level analysis and low-level code analysis, the WCC is capable to extract event arrival functions [8] from a given program [9]. Event arrival functions allow to model the dynamics of a real-time system, even for arbitrarily triggered events. They are used in system-level analysis to limit the number of events and analyze the extent of induced interference. Such an abstract event can be defined as, e.g., the access to a specific memory region. The WCC then extracts a corresponding safe upper (or lower) event arrival function with an adjustable level of granularity, offering a flexible choice between tightness of the resulting curve and required runtime for the extraction. The resulting curves can be saved either in a CSV file or be directly used for further optimizations.

For ACET-oriented optimizations or in order to acquire reference timings, the WCC also has a tight coupling with the cycle-accurate instruction set simulator CoMET [10] from Synopsys. The actual generation of configuration files and calling of the corresponding tool happens completely transparent to the user as well. Similar to the integration of WCET analyzers in the WCC, also the profiling results are annotated back to the internal structures.

B. Optimizations

The aforementioned analyses are used to perform sophisticated WCET-oriented optimizations. In the following, we will give a brief overview on WCET-oriented static memory allocation optimizations integrated into the WCC as an exemplary set of optimizations.

The static allocation of certain program parts to a faster, yet typically smaller memory, can be exploited to reduce a program's WCET. The faster but smaller memories are commonly known as scratchpad memories (*SPM*). Determining by hand which parts of the program should go into the faster memory and which should reside in the larger, yet slower, memory in order to achieve a lower WCET is tedious and will most likely not provide an optimal result. To counter this issue, the WCC offers several dedicated optimizations to find optimal static WCET-aware allocations. The optimizations differ in the type of allocation (instruction or data) and in their coverage of architecture-specific characteristics.

WCET-aware static instruction allocation optimizations are typically carried out on a basic block granularity by the WCC. This enables a good trade-off between optimization runtime and optimization quality. The optimization is not limited to a

non-cached singletask-singlecore system [11], but can also be applied to systems including instruction caches [12], multitask-singlecore systems [4] or singletask-multicore systems [13]. These specific optimizations take the architecture-dependent characteristics into account. Overall, these optimizations automatically decide which program parts should remain in the larger, yet slower, memory region and which parts should be allocated to the faster SPM in order to receive a minimum WCET or to make a system schedulable in case of a multitask system. As these optimizations are based on integer linear programming (*ILP*), the resulting allocation will be optimal with regard to the underlying model. The actual allocation to the corresponding memory regions and the potentially required control flow repair is done automatically by the compiler.

If the user chooses to run a cache-aware static instruction allocation optimization, the optimization will also predict additional new cache hits or misses due to the updated memory layout of the program. Architecture-dependent parameters like the cache size are automatically adapted from WCC's target memory layout.

For multicore architectures, the WCC offers a dedicated static instruction allocation optimization as well. It assumes a private SPM per core and a shared Flash memory, whereas the architecture is connected using a TDMA-scheduled bus. The optimization then predicts the additional timing penalties or gains due to the TDMA schedule by estimating the timing offsets for each shared memory access.

When compiling for a multitask system, the WCC offers another dedicated static instruction allocation. This optimization derives the required scheduling information given by the user and tries to find an allocation for all tasks (sharing the same memories), such that a previously non-schedulable system becomes schedulable. This optimization also includes a schedulability test, meaning if the optimization succeeds, the resulting system is guaranteed to be schedulable and therefore meets all its deadlines.

Additionally, the WCC also offers a static instruction allocation optimization based on evolutionary algorithms. This optimization invokes a WCET analysis for each individual. As the WCET analysis considers all characteristics from the given system, the optimization itself also inherently considering all system properties. Although not guaranteeing to find an optimal allocation, it will likely find a solution close to the global optimum due to the nature of evolutionary algorithms.

Similarly, the WCC comprises an ILP-based WCET-oriented memory region allocation for data objects. In analogy, the compiler finds an optimal allocation for the existing data objects in a program, such that a minimal WCET is achieved. The optimization decides which data objects are allocated to which memory region. According to this decision, a corresponding linker file is generated as depicted in Fig. 1. The WCC then automatically assembles and links the program, resulting in a final binary with an optimized WCET.

IV. Outlook

A part of future work is the integration of new architectures into the WCET-aware C Compiler. Beside supporting TriCore TC1796, TC1797 and ARM7TDMI architectures, the WCC is currently being expanded in order to support the ARM Cortex M and the LEON3 processors. Analogous to the existing architectures, the platform-dependent parameters of the LEON3 architecture will be integrated as well as the coupling with the corresponding timing analysis tools, enabling real-time aware optimizations.

Besides, we are currently integrating a coupling with the real-time calculus toolbox [14] in order to further utilize the possibility to extract tight and safe event arrival functions using the WCC. This will allow WCET-oriented system-level optimizations in combination with low-level optimizations, leveraging both levels of abstraction. Additionally, even complex distributed architectures may be analyzed using modular performance analysis via real-time calculus.

References

[1] AbsInt Angewandte Informatik, GmbH. (2018) aiT Worst-Case Execution Time Analyzers.

[2] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: An Open Toolbox for Adaptive WCET Analysis," in *Proceedings of International Workshop on Software Technologies for Embedded and Ubiquitous Systems (2010)*, 2010.

[3] H. Falk and P. Lokuciejewski, " A compiler framework for the reduction of worst-case execution times ," *Real-Time Systems*, vol. 46, no. 2, 2010.

[4] A. Luppold and H. Falk, "Schedulability-aware SPM Allocation for preemptive hard real-time systems with arbitrary activation patterns," in *Proceedings of Design, Automation Test in Europe Conference Exhibition (2017)*, 2017.

[5] K. Gresser, "An Event Model for Deadline Verification of Hard Real-Time Systems," in *Proceedings of Euromicro Conference on Real-Time Systems (1993)*, 1993.

[6] T. Kelter and P. Marwedel, "Parallelism analysis: Precise wcet values for complex multi-core systems," *Science of Computer Programming*, vol. 133, 2017, Formal Techniques for Safety-Critical Systems.

[7] H. Shah, A. Raabe, and A. Knoll, "Priority division: A high-speed shared-memory bus arbitration with bounded latency," in *Proceedings of Design, Automation Test in Europe (2011)*, March 2011.

[8] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century.*, 2000.

[9] D. Oehlert, S. Saidi, and H. Falk, "Compiler-based Extraction of Event Arrival Functions for Real-Time Systems Analysis," in *Proceedings of Euromicro Conference on Real-Time Systems (2018)*, 2018.

[10] Synopsys, Inc. (2018) Synopsys CoMET-METeor.

[11] H. Falk and J. C. Kleinsorge, "Optimal Static WCET-aware Scratchpad Allocation of Program Code," in *Proceedings of Annual Design Automation Conference (2009)*, 2009.

[12] A. Luppold, C. Kittsteiner, and H. Falk, "Cache-Aware Instruction SPM Allocation for Hard Real-Time Systems," in *Proceedings of International Workshop on Software and Compilers for Embedded Systems (2016)*, 2016.

[13] D. Oehlert, A. Luppold, and H. Falk, "Bus-Aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems," in *Proceedings of Euromicro Conference on Real-Time Systems (2017)*, 2017.

[14] E. Wandeler and L. Thiele, "Real-Time Calculus (RTC) Toolbox," 2006. [Online]. Available: http://www.mpa.ethz.ch/Rtctoolbox