A Retargetable Framework for Multi-objective WCET-aware High-level Compiler Optimizations *

Sascha Plazar, Paul Lokuciejewski, Peter Marwedel
Computer Science 12
TU Dortmund University
D-44221 Dortmund, Germany
FirstName.LastName@udo.edu

Abstract

The worst-case execution time (WCET) is a key parameter in the domain of real-time systems and its automatic compiler-based minimization becomes a challenging research area. Although today's embedded system applications are written in a high-level language, most published works consider low-level optimizations which complicate their portability to other processors. In this work, we present a framework for the development of novel WCET-driven high-level optimizations. Our WCET-aware compiler framework provides a multi-target support as well as an integration of different non-functional objectives. It enables multi-objective optimizations, thus opens avenues to a state-of-the-art design of predictable and efficient systems. In addition, the multi-target support provides the opportunity to efficiently evaluate the impact of different compiler optimizations on various processors.

1. Introduction

Embedded systems must often meet real-time constraints. Besides the correctness of the results, the knowledge about the worst-case execution time is crucial. Especially for safety-critical systems, e. g. in the avionic and automotive domain, the adherence of the WCET must be ensured to avoid system failure leading to a disaster. The precise knowledge of this key parameter is also required for scheduling algorithms or the development of hardware platforms which have to satisfy critical timing constraints.

Due to the complexity of today's embedded systems, the software development relies on both a high-level language, predominantly C, and a compiler. Modern compilers offer a vast variety of optimizations with the objective to minimize the average-case execution time *(ACET)* or energy dissipation. In contrast to these optimizations, a compiler-guided minimization of the WCET is still a novel research area. WCET-driven compiler optimizations require the integration of a static WCET analyzer into a compiler framework providing timing information that is exploited for an effective minimization of the program's WCET.

Typically, WCET-aware compiler frameworks model single processors. This strict binding to a particular hardware involves two disadvantages. First, any developed WCET-aware optimizations can be exclusively evaluated for the supported processor. Thus, assumptions about the effectiveness of the optimization w.r.t. other processors are difficult or even impossible. A later evaluation might even reveal that an optimization performing well on the processor used during the optimization's design has a negative impact on another processor.

Second, a WCET-aware compiler framework that supports a single processor does not benefit from synergies in the development of WCET-aware optimizations. Each generic software module implemented to assist optimizations for a particular processor must be partially or even completely rewritten when ported to another WCET-aware compiler framework. In contrast, a framework producing code for multiple processors might significantly shorten the development time of new processor-specific optimizations since generic modules can be reused.

Another requirement for an effective compiler framework is motivated by changing design goals. Today's embedded system designers are forced to consider a reconciliation of both design goals, predictability and efficiency. These goals pose a conflict in realistic systems with bounded resources which have to be shared for logical and temporal reasons. To cope with this problem, a trade-off analysis allowing the determination of pareto-optimal designs is required. In a WCET-aware compiler

*The research leading to these results has received funding from the European Community's ARTIST2 Network of Excellence and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n^{o} 216008.

framework, the trade-off is found by multi-objective optimizations which consider, besides the WCET, other non-functional objectives. To realize such a conceptual framework and to enable new design approaches, generic data structures to hold different objectives must be provided.

In this paper, we present a novel conceptual framework that opens avenues to a state-of-the-art design of predictable and efficient systems. The paper's main contributions are:

1. We present concepts for a retargetable WCET-aware compiler framework where processors involved in a static WCET analysis can be easily exchanged. This allows the evaluation of WCET-aware optimizations on different processors.
2. We introduce the *Back-annotation*, a methodology to bridge the high-level and low-level intermediate representation (IR) of the program and to enable a transformation of objectives between both levels.
3. Our Back-annotation is extended by generic data containers that store arbitrary objectives to provide opportunities for multi-objective optimizations.

The rest of this paper is organized as follows: Section 2 gives a survey of the related work. The concepts of the retargetable, multi-objective WCET-aware compiler framework are discussed in Section 3. Finally, Section 4 concludes and gives directions for future work.

2. Related Work

Recently, the minimization of energy dissipation as an optimization goal of compilers has moved into the focus of research. However, compiler-based WCET minimization is sparsely dealt within today's literature. These approaches rely on a communication between a compiler and a static WCET analyzer to gather the WCET as cost function. Besides the supported processors, the main difference between the involved frameworks is the internal code representation. Almost all frameworks work on a low-level IR, lacking the possibility to develop high-level optimizations.

The interactive compilation system called *VISTA* [12] translates a C source code into a low-level IR used for code optimizations. By binding the compiler to a proprietary developed static WCET analyzer supporting two simple processors without caches, the obtained WCET is exploited for WCET-driven optimizations, e. g. block reordering [13].

The open-source tool *Heptane* [3] is a static WCET analyzer with multi-target support. It expects a C source code as input parsed into a high-level IR. Next, the code is translated into a low-level IR where the WCET analysis takes place. Heptane exclusively supports WCET-driven low-level optimizations, e. g. predictable page allocations [5].

The single-target WCET-aware C compiler *WCC* [4] incorporates a high- and low-level IR. To obtain WCET objectives, the compiler is coupled to the static WCET analyzer *aiT* [1], a sophisticated analysis tool supporting various state-of-the-art processors. *WCC*'s timing model, currently only available for the Infineon TriCore 1796 processor, is exploited to conduct WCET-driven low-level optimizations, e. g. Procedure Positioning [9]. Unlike other approaches, the *WCC* also enables the development of WCET-driven high-level optimizations, e. g. Procedure Clonining [8]. However, the compiler supports a rudimentary Back-annotation transforming WCET information exclusively for functions from the low-level to the high-level IR.

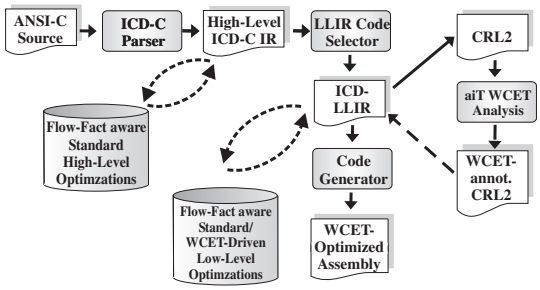

Figure 1: Workflow of the single-target WCET-aware compiler

Besides WCET-optimizing compilers, other tools serve exclusively for the estimation of the WCET. *OTAWA* [2] is a generic multi-target low-level framework dedicated to the development of static analyses to compute the WCET based on a binary executable. The goal of the open-source tool is to provide a system where different WCET computation models can be integrated for evaluation purposes. *Bound-T* [6] is a commercial static WCET analyzer with multi-target support performing the analysis on a binary executable. In contrast to the static analyzers, *RapiTime* [10] is a measurement-based WCET tool. These three tools have not been exploited for WCET-driven compiler optimization.

3. Framework Architecture

In this section, the conceptual framework for processor-indepedent, multi-objective WCET-aware compiler optimizations is presented. The goal of this work is to extend our WCET-aware compiler framework *WCC* by a Back-annotation which supports arbitrary objective types at a finer grained level, namely basic blocks. Moreover, we extend *WCC* by multi-taget support. Before the required extensions to realize this novel framework architecture are discussed, we introduce the compiler's current state. Figure 1 shows its workflow. The workflow begins with parsing and translating the C source code into a high-level intermediate representation called *ICD-C IR* [7]. At this level, the compiler front-end provides miscellaneous standard compiler analyses, e. g. control- and data-flow analyses, as well as a large set of ACET optimizations, e. g. *Dead Code Elimination* or *Loop Unrolling*.

In the next phase, the *LLIR Code Selector* translates the high-level IR into the low-level IR, the retargetable *ICD-LLIR* [7] which also provides standard low-level optimiza-

tions. Due to the single-target support, the code at this level is specific for the TC1796 processor. By coupling the compiler back-end to the static WCET analyzer *aiT*, the objective WCET is imported into the *WCC* and exploited for WCET-driven low-level optimizations. Since *aiT* internally works on its own code representation, this coupling is realized by converting the ICD-LLIR into an equivalent CRL2 IR which is AbsInt's machine-code level intermediate representation. After the WCET analysis, *aiT* provides a CRL2 file annotated with WCET information that is used for the objective import into the ICD-LLIR.

In addition to hardware timing characteristics, a successful static WCET analysis relies on information about program execution flows to bound the number of instruction executions. These specifications are called *flow facts* and encompass upper bounds of loop iterations and the recursion depth. For user convenience, flow facts can be annotated at source code level. Afterwards, a *flow fact manager* (not shown in Figure due to clarity reasons) extracts them from the C code and attaches the flow facts to objects of the ICD-C IR. During code selection, the manager translates the high-level flow facts into semantically equivalent ICD-LLIR flow facts which are finally translated into CRL2 and serve as auxiliary information for *aiT*. Due to the tight integration of the flow fact mechanisms into the WCC, a simple flow fact API and highly efficient data structures and algorithms are provided.

The flow fact translation does not ensure that flow facts passed to *aiT* are semantically equivalent to the used annotations specified in the source code. This is due to compiler optimizations, e. g. Loop Unrolling, that might invalidate user-defined flow facts. As a consequence, all of the 47 WCC optimizations are made flow-fact aware, i. e. the flow fact manager takes care to adjust them during code transformations in order to preserve their semantical correctness.

3.1. Retargetable WCET-aware Compiler Framework

The first step towards a retargetable WCET-ware compiler framework is the elimination of the TriCore-specific code selector and compiler back-end ICD-LLIR. This is achieved by substituting these two components by an arbitrary compiler for a processor supported by the static analyzer. Due to the large number of supported processors and compilers by AbsInt's *aiT*, the WCET-aware compiler is highly flexible and allows the evaluation of different configurations. The employment of an arbitrary compiler that replaces the code selector from the previous scenario and provides input for the static WCET analyzer is depicted in Figure 2. Before passing the input program to an arbitrary compiler, it is processed by the ICD-C IR which serves as an source-to-source optimizer. Thus, all analyses and optimizations provided by our high-level IR are not discarded

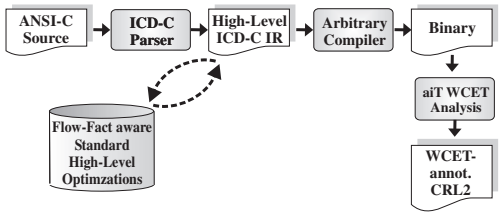

Figure 2: First step towards a multi-target WCET-aware compiler

but can be furthermore exploited. Due to the flow fact manager, all source code flow facts are correctly adjusted and are dumped together with the transformed IR into equivalent C code. Finally, this code is passed to an arbitrary compiler which generates a binary executable provided as input to the WCET analyzer. In contrast to the previous workflow where the *WCC* generated the initial CRL2 file, this approach uses *aiT*'s decoder to construct the initial input file based on the binary executable. In addition, *aiT* extracts the dumped and slightly adjusted ICD-C annotations from the source code and uses it during the WCET analysis.

3.2. Back-Annotation

The goal of the new framework architecture is to provide a platform for the development of high-level WCET-aware compiler optimizations. To accomplish this goal, a bridge between the analyzed binary executable and the high-level IR must be established in order to make WCET information visible in the ICD-C IR.

The basic difference between the *WCC* (Figure 1) and the novel workflow (Figure 2) is the different handling of the compiler back-end. The application of an arbitrary compiler without the inside into its code selector must be considered as a black box where the relationship between the source code and the binary executable is not apparent.

The problem of bridging the gap between the source code and the machine code is not new. In particular, problems arise when optimized code is debugged since an unambiguous mapping between the source code and the optimized target program becomes infeasible. This is known as the *Code Location Problem*. To overcome this problem, compiler optimizations are typically disabled. In our framework, the involved compiler produces a non-optimized binary executable annotated with *DWARF2* [11] debug information. To achieve good code quality, the preceding flow-fact aware ICD-C IR optimizations can be applied. They have no negative impact on the debugging information since the consistency between the input program and the non-optimized machine code is preserved.

The last step for a complete Back-annotation is the collection of WCET information provided by *aiT*'s annotated CRL2 file and their assignment to the proper ICD-C constructs. This final workflow denoted by dashed arrows is depicted in Figure 3. To achieve a mapping between the ma-

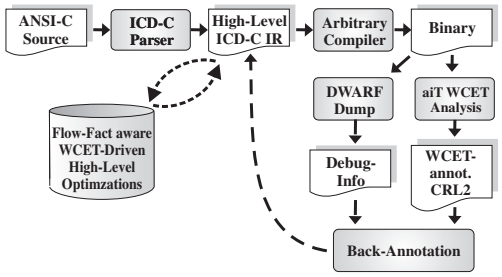

Figure 3: Workflow of a back-annotated, multi-target WCET-aware compiler

chine code and the ICD-C IR, DWARF2 information are exploited. We apply the standard tool *dwarfdump* to the binary executable that writes DWARF2 information in a human-readable format into a data structure we call *Debug-Info*. This information indicates which machine code instructions (specified by physical address) resulted from which high-level constructs (specified by source code line):

DWARFDUMP : $physical\ address \rightarrow source\ line$

To retrieve the WCET information, *aiT*'s WCET-annotated CRL2 file is analyzed. Since it was constructed from a binary executable, its internal control flow graph representation contains basic blocks that are assigned physical addresses (analogous to the binary) and the computed WCET representing the accumulated timing information for all executions of this block:

CRL2 : $physical\ address \rightarrow accumulated\ WCET$

With this information, it is straightforward to assign WCET information to the corresponding source code constructs exploiting the physical address as join point. This mapping is performed by the Back-annotation. Hereafter, WCET data can be employed to assist WCET-driven high-level optimizations.

3.3. Multi-objective Storage

Our framework allows the storage of arbitrary non-functional objectives within our high-level IR. The objectives are held in generic containers which can be assigned to any ICD-C C construct. High-level optimizations operating on the ICD-C can easily access this information. In the current state, WCET estimations are the only supported objectives. However, due to the framework's flexibility, other profilers providing ACET or energy dissipation objectives can be easily integrated and the framework opens avenues to multi-objective optimizations.

4. Conclusions and Future Work

In this paper, we presented a novel WCET-aware compiler optimization framework supporting multiple target processors as well as the storage of different non-functional compiler objectives. By the integration of WCET information, the framework offers the opportunity to develop WCET-aware high-level compiler optimizations. Due to the framework's flexibility, other objectives can be easily integrated in order to leverage state-of-the-art multi-objective design approaches. Furthermore, the multi-target support can be exploited to evaluate the impact of newly developed optimizations on different processors.

In the future, we intend to exploit the imported WCET information for the design of novel WCET-driven high-level optimizations. For example, Loop Unrolling could be controlled by the worst-case number of loop iterations. We also plan to integrate additional profilers providing new objectives for multi-objective optimizations as well as further compilers to extend the set of supported target processors. Last but not least, the development of new low-level WCET-driven optimizations is also a challenging topic. By retargeting the ICD-LLIR to other processors, similar as depicted in Figure 1 for the TC1796 processor, the development of novel multi-objective low-level optimizations would be enabled. Also, standard LLIR optimizations could be applied to the non-optimized assembly code generated by the compilers in Figure 2 to achieve better code quality.

References

[1] AbsInt Angewandte Informatik GmbH. Worst-Case Execution Time Analyzer aiT for TriCore. 2008.

[2] H. Cass and C. Rochange. OTAWA, Open Tool for Adaptative WCET Analysis. In *Poster session "University Booth" of DATE*, April 2007.

[3] A. Colin and I. Puaut. A Modular & Retargetable Framework for Tree-Based WCET Analysis. In *Proc. of ECRTS*, June 2001.

[4] H. Falk, P. Lokuciejewski, and H. Theiling. Design of a WCET-Aware C Compiler. In *Proc. of ESTIMedia*, October 2006.

[5] D. Hardy and I. Puaut. Predictable Code and Data Paging for Real Time Systems. In *Proc. of ECRTS*, july 2008.

[6] N. Holsti and S. Saarinen. Status of the Bound-T tool. In *Proc. of WCET*, June 2002.

[7] Informatik Centrum Dortmund, Embedded Systems Profit Center. http://www.icd.de/es, September 2008.

[8] P. Lokuciejewsi, H. Falk, P. Marwedel, and T. Henrik. WCET-Driven, Code-Size Critical Procedure Cloning. In *Proc. of SCOPES*, March 2008.

[9] P. Lokuciejewski, H. Falk, and P. Marwedel. WCET-driven Cache-based Procedure Positioning Optimizations. July 2008.

[10] Rapita Systems Ltd. http://www.rapitasystems.com, September 2008.

[11] The DWARF Debugging Standard. http://dwarfstd.org, September 2008.

[12] W. Zhao, P. Kulkarni, D. Whalley, et al. Tuning the WCET of Embedded Applications. In *Proc. of RTAS*, May 2004.

[13] W. Zhao, D. Whalley, C. Healy, and F. Mueller. WCET Code Positioning. In *Proc. of RTSS*, December 2004.