Design of a WCET-Aware C Compiler [*]

Heiko Falk
Computer Science 12
University of Dortmund
D-44221 Dortmund
Heiko.Falk@udo.edu

Paul Lokuciejewski
Computer Science 12
University of Dortmund
D-44221 Dortmund
Paul.Lokuciejewski@udo.edu

Henrik Theiling
AbsInt Angewandte Informatik
Science Park 1
D-66123 Saarbrücken
theiling@absint.com

Abstract

This paper presents techniques to tightly integrate worst-case execution time (WCET) information into a compiler framework. Currently, a tight integration of WCET information into the compilation process is strongly desired, but only some ad-hoc approaches have been reported currently. Previous publications mainly used self-written WCET estimators with very limited functionality and preciseness during compilation. A very tight integration of a high quality industry-relevant WCET analyzer into a compiler was not yet achieved up to now. This work is the first to present techniques capable of achieving such a tight coupling between a compiler and the WCET analyzer aiT. This is done by automatically translating the assembly-like contents of the compiler's low-level intermediate representation (LLIR) to aiT's exchange format CRL2. Additionally, the results produced by the WCET analyzer are automatically collected and re-imported into the compiler infrastructure. The work described in this paper is smoothly integrated into a C compiler environment for the Infineon TriCore processor. It opens up new possibilities for the design of WCET-aware optimizations in the future.

The concepts for extending the compiler infrastructure are kept very general so that they are not limited to WCET information. Rather, it is possible to use our structures also for multi-objective optimization of e. g. best-case execution time (BCET) or energy dissipation.

1. Introduction

In contrast to general-purpose systems, embedded systems often have to meet some real-time constraints making them real-time systems. The correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the results are produced [5]. Besides the necessity of safeness of real-time systems, the market demands high performance, energy efficient and low cost products. Without knowledge about the worst-case timing of a real-time application, the designer tends to use oversized hardware in order to guarantee the safeness of the real-time system. The knowledge of the worst-case execution time *(WCET)* gives the designer the opportunity to use or to design a hardware platform which is tailored towards the software resource requirements like memory or clock rate. Thus, the production costs can be reduced significantly, while still guaranteeing the safeness of the real-time system.

Today, software development for embedded systems is done using high-level languages like C, requiring the existence of a suitable compiler. Modern compilers are equipped with a vast variety of optimizations. However, these optimizations aim at minimizing e. g. average-case execution time *(ACET)* [14] or energy dissipation [15]. The influence of compiler optimizations on WCET is unknown in almost all cases. Currently, the binary executable generated by the compiler is manually fed into a WCET analyzer computing the required information. Using this WCET data, it can be determined whether real-time constraints are met. If this is not the case, the program source code has to be tuned, compiled and optimized again in another cycle of the design flow.

As can be seen, it is highly desirable to have a compiler that is aware of WCET properties. In an integrated WCET-aware compilation environment, it will be possible to integrate and to apply optimizations with the objective of WCET minimization. WCET information available within the compiler can be used to determine those parts of the code that lie on the worst-case path. Specialized complex optimizations could be applied in the future only to these code portions in order to minimize WCET aggressively. If the compiler is able to support multiple optimization objectives at the same time (e. g. energy consumption and WCET), automated trade-offs can be applied by the compiler such that the most energy efficient code is generated that still meets real-time constraints.

This paper is the first one to present a tight integration of an existing WCET analyzer into a compiler infrastructure. Using the techniques of this work, it is possible to feed the assembly-like contents of the compiler's low-level intermediate representation *(LLIR)* into the WCET analyzer automatically. The results produced by the WCET analyzer are automatically re-imported into the compiler infrastructure and are available for future optimizations as described above. Currently, this includes the WCET of an entire application, of functions and basic blocks as well as calling contexts, execution counts and feasibility information. The techniques presented in this work are part of a C compiler environment and were validated for the Infineon TriCore [9] processor.

In addition, a very powerful and flexible mechanism is presented enabling to attach not only WCET-related data to the LLIR, but also to store arbitrary information used by optimizations targeting different objectives than WCET. This approach will be useful in order to perform automated trade-offs between different optimization goals.

The remainder of this paper is structured as follows: Section 2 gives a survey of related work. Section 3 presents the entire structure of our WCET-aware C compiler. It is fol-

[*]This work is partially funded by the European IST FP6 Network of Excellence ARTIST2.

lowed by a discussion on the proposed compiler structure in Section 4. Finally, Section 5 summarizes this paper and gives an outlook on future work.

2. Related Work

[4] presented a very first simple approach to integrate WCET techniques into a commercial compiler. Flow facts required for WCET analysis have to be annotated manually using source-level pragmas. The compiler backend generates code for the Intel 8051 which is an inherently simple and predictable machine without pipeline and caches etc. The entire work was not finished and tested, and results are unavailable. A fully pragma based approach is not promising since manual annotations are tedious and error-prone.

While mapping high-level code to object code, compilers perform various optimizations so that the correlation between high-level flow facts and the optimized object code becomes very vague. In order to keep track of the influence of compiler optimization on high-level flow facts, [7] proposes co-transformation of flow facts. However, the co-transformer has never reached a fully working state, and several standard compiler optimizations can not be modeled at all due to insufficient data structures.

[12] presents techniques for transforming program path information during compiler optimization. The authors are able to keep high-level flow facts consistent while performing GCC's standard optimizations. Their approach was thoroughly tested and led to precise WCET estimates. However, compilation and WCET analysis are done in a decoupled way. The assembly file generated by the compiler is passed to the WCET analyzer together with the transformed flow facts. Additionally, the proposed compiler is only able to process a subset of the C programming language, and the modeled target processor lacks pipelines and caches.

In [17, 18], the integration of a proprietarily developed WCET analyzer into a compiler is presented. The compiler operates on a low-level intermediate representation *(IR)*. Control flow information is passed to the timing analyzer which computes the WCET of loops and functions and passes this data back to the compiler. However, this approach has several limitations. First, the WCET analyzer works with very coarse granularity since it only computes WCETs of loops and functions. WCETs for basic blocks or single instructions are unavailable. Second, WCET-relevant data which is not the WCET itself is unavailable, too. This includes e. g. execution contexts, execution frequencies of basic blocks, value ranges of registers, predicted cache behavior etc. Third, the lack of a high-level IR within the compiler requires to costly re-generate valuable high-level flow facts that are only available at the source code level. Finally, the considered processor is quite simple as it has a simple pipeline and no caches.

A compiler guided trade-off between WCET and code size is presented in [13]. The authors observe that applications using the 16-bit THUMB instruction set of an ARM7 processor typically are smaller but run slower than when using the 32-bit ARM instruction set. [13] compiles a program using both instruction sets at the same time with the goal to reduce WCET at the expense of code size and vice versa. To obtain WCET information, a simple timing analyzer was implemented from scratch assuming the absence of caches and a simple pipeline structure. [13] presents a WCET-guided compiler optimization, but it does not homogeneously integrate WCET analysis into the compiler itself. Compilation and WCET analysis are completely decoupled, WCET data is not fed back to the compilation stages.

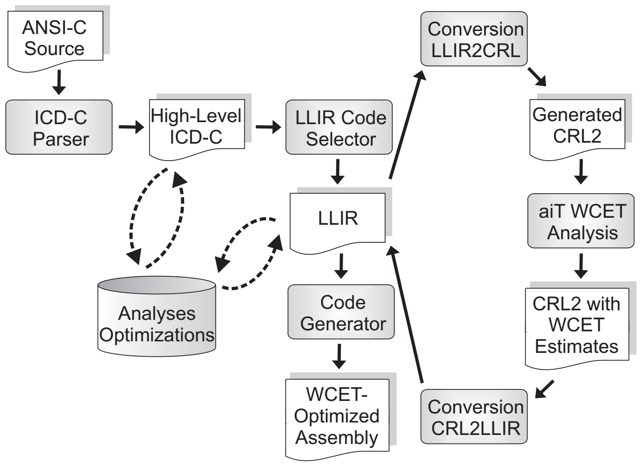

Figure 1. WCET-aware C Compiler (WCC)

An optimization allocating both functions and data elements of an application to a scratchpad memory is presented in [16]. The authors report a significant WCET reduction, since scratchpads are much faster than other types of memory, and their use is fully guided by the compiler so that very precise flow facts can be passed to the WCET analyzer. However, this approach also does not establish a tight integration of WCET data into the compiler. The memory allocation optimization is not based on WCET data. Instead, the optimization minimizes energy dissipation when using scratchpad memories. WCET timing is measured after compilation by analyzing the resulting binary executable.

3. Design of the WCET-Aware C Compiler

Based on the results of the previous publications presented in Section 2, we propose the structure depicted in Figure 1 for WCET-aware compilation. As can be seen, the compiler we call *WCC* relies on two IRs, namely ICD-C [10] being close to ANSI-C, and LLIR [11] modeling code at the assembly level. The code selector is responsible for the translation of ICD-C to LLIR. Analyses and optimizations take place both at the high and the low level. Currently, all optimizations done by the compiler are not yet WCET-aware. The focus of this paper clearly lies on the integration of WCET analysis into the compiler, and not on WCET optimizations which are part of our future work.

Today, there are only few vendors providing commercial WCET analyzers. One of the leading analyzers is the aiT [1] WCET analyzer developed by AbsInt. aiT is available for various processors, including ARM7, Power-PC, TI TMS320C33 and the Infineon TriCore v1.3. In this paper, the Infineon TriCore processor is considered as target architecture for compilation and WCET analysis.

WCET analysis can only take place at the assembly / binary level since processor-specific information and machine code is unavailable at higher levels of abstractions. Thus, the WCET analyzer aiT is coupled to our compiler at the LLIR level. Our compiler is able to translate its LLIR representation to the CRL2 format [2] which is the IR of aiT. This way, both aiT and WCC use the same format to ex-

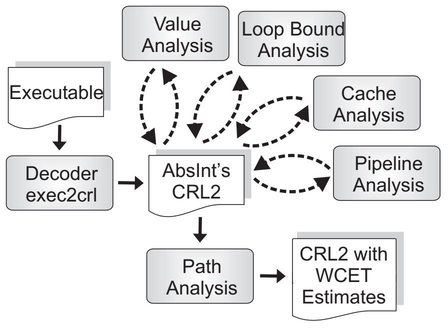

Figure 2. Workflow of aiT

change information. Hence, aiT can be provided with CRL2 accurately modeling the application under analysis, and all results computed by aiT are also available in the CRL2 format after WCET analysis is done. Using a conversion from CRL2 to LLIR, we are able to import all data computed by aiT into our compiler.

3.1. The WCET Analyzer aiT

In contrast to numerous other WCET analyzers, aiT performs a highly accurate analysis of the processor pipeline and available caches. When being used outside our WCC compiler framework, aiT needs to be provided with a binary executable of the program to be analyzed as well as with a specification file. To estimate the WCET of the binary executable, several analysis steps are taken (cf. Figure 2). The decoder exec2crl reads the executable and reconstructs its control flow graph. This control flow graph is translated into aiT's intermediate format CRL2 [2]. CRL2 is used as exchange format storing the application under analysis as well as analysis results generated by the individual substeps of aiT. In the WCC setup, aiT's decoder exec2crl is skipped.

Value analysis determines potential values in the processor registers for any possible program point. These values are frequently used within aiT. Cache analysis requires predicted values to identify possible addresses of memory accesses. In addition, the predicted values are used to determine infeasible paths resulting from conditions being true or false at any point of the analyzed program. Finally, tight value ranges are required to determine loop bounds.

Since a program spends most of its execution time in loops, the iteration counts play an important role for WCET estimation. aiT relies on precise bounds to be able to perform a WCET analysis at all. The detection of loop bounds during *loop bound analysis* succeeds only for simple loops and demands their external annotation outside aiT.

The *cache analysis* of aiT statically analyzes the cache behavior of a program using a formal cache model. Accesses to main memory are examined by an algorithm distinguishing between sure cache hits and unclassified accesses. A proper cache analysis relies on the value ranges of processor registers obtained from the value analysis.

Pipeline analysis models the processor's pipeline behavior and is based on the current state of the pipeline, the resources in use, the contents of prefetch queues and the results obtained during cache analysis. It aims at finding a WCET estimate for each basic block of a program. Each basic block is analyzed by taking possible pipeline states from preceding basic blocks into account. After processing each instruction of the currently analyzed block, the longest time this block takes to execute is computed.

Using the data provided by the previous analysis steps, *path analysis* computes a program's global WCET. A path within the control flow graph is a sequence of successive basic blocks starting at the entry point of a program and ending at its end point. For each block on a path, its maximum execution time T can be expressed based on the previous analyses. Using the loop bound information, a block's maximum number of executions C can be determined. The WCET of a path can be expressed as the sum of the products $T * C$ over all edges of a path. A program's global WCET is determined by finding the maximum path WCET for all feasible paths. This maximization problem is expressed and solved using integer linear programming.

All analyses of aiT take *execution contexts* into account. A context indicates a particular way of calling a program's function R. If R is called several times, contexts are used to distinguish between the different program states representing the different flows of control through the program to function R. Contexts improve the precision of analysis if analysis results are computed individually for each program state represented by a context.

Since aiT is directly invoked with a CRL2 IR generated by our compiler, the decoder exec2crl shown in Figure 2 is skipped in our WCC setup. In the following subsections, the conversions between LLIR and CRL2 and vice versa are described as well as a generic mechanism for storing arbitrary data beyond WCET information within LLIR.

3.2. Conversion LLIR $\leftrightarrow$ CRL2

Since both LLIR and CRL2 are low-level IRs representing a program as control flow graph *(CFG)*, it is easy to translate these IRs into each other. At the top level, the control flow graph of both IRs consists of functions. Each function contains a list of basic blocks connected via edges. Basic blocks in turn consist of a sequence of instructions. In both IRs, an instruction can consist of several operations in order to express the implicit parallelism present in e. g. a VLIW machine. In addition to basic blocks, information on execution contexts is attached to functions within CRL2. In LLIR, this kind of information does not exist.

Due to the analogy of both IRs, the translation steps basically need to traverse the CFG of one IR recursively and need to generate equivalent CFG components of the other IR. However, there are two issues making the translation from LLIR to CRL2 and back much more complex than this simple recursive traversal.

Determination of *op_id*s

First, we need to solve the problem that LLIR is an assembly-level IR, whereas CRL2 is a binary-level IR. As a consequence, any information additionally created during assembly and linking is unavailable within LLIR, but available to CRL2. This difference between both IRs mainly comes into play when converting LLIR operations into CRL2 operations. An LLIR operation is represented by its assembly mnemonic and belonging operands, whereas CRL2 requires a unique identifier (*op_id*) representing the binary machine code of the operation together with its operands. Unfortunately, there is no direct translation between an LLIR opcode and a corresponding *op_id*.

For example, the TriCore ISA contains four different machine operations with the mnemonic AND [9]:

AND Dc, Da, Db AND Dc, Da, *const9*
AND Da, Db AND D15, *const8*

All these AND operations differ by the number and types of parameters, namely data registers denoted by Dx or D15, or constants. As can be seen, the mere use of the mnemonic does not yield an unambiguous CRL2 operation. As a consequence, more key characteristics of an LLIR operation need to be taken into account.

In addition to its mnemonic, the operation's *opcode format* is considered next. Depending on the amount and type of parameters, the binary machine code of an operation has various formats. For example, the first AND operation listed above has the 32-bit wide format RR. The second operation with a constant parameter is of format RC. The last two operations with only two parameters have the 16-bit opcode formats SRR and SC, respectively.

Using the alignment and the type of parameters of an LLIR operation, its opcode format is determined. In almost all cases, these first two steps of mnemonic and opcode format matching are sufficient to obtain an unambiguous *op_id*. However, the addressing mode has to be considered as third criterion in a few cases. The addressing mode specifies the calculation of an effective memory address of a certain operation parameter, using values held in registers and constants. Mainly load and store operations make extensive use of the various addressing modes. The TriCore architecture provides eight different modes such as *Pre-* and *Post-increment* or *Short* and *Long Base plus Offset* addressing.

As example, the LD.W operation loading a word from memory is considered. The TriCore instruction set includes only one operation with the opcode LD.W:

LD.W Da, <*mode*>

Its first parameter designates the register to be loaded, the second one specifies the addressing mode. Depending on this mode, various CRL2 *op_id*s can be used for the LD.W operation. It is easy to see that mnemonic and opcode format are insufficient in this case to obtain a unique *op_id*.

But even after considering addressing modes, there are still some operations which can not be unambiguously identified so far. In particular, halfword arithmetical operations have equivalent mnemonics and opcode formats, but do not rely on addressing modes at all. The only difference between these halfword operations is a specifier like e. g. UL denoting that an upper halfword of a register is combined with a lower halfword of another register. However, these few cases of halfword arithmetic can be handled easily by performing particular checks for these halfword operations.

But even after this fourth stage of halfword operation matching, there still exist LLIR operations which are not unambiguously mapped to CRL2 *op_id*s. The TriCore instruction set still contains few operations which can not be classified unambiguously at all using the information available within LLIR. For instance, there are two versions of the MOV operation:

MOV Dc, Db MOV Da, Db

The parameters of these two operations are arbitrary data registers in both cases. However, the first operation is 32 bits wide, whereas the second operation is a 16-bit operation. Since the LLIR does not distinguish the bit-width of operations, it is impossible to classify these two opcode for-

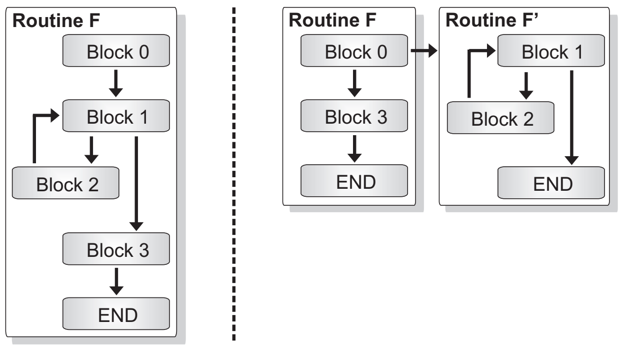

Figure 3. Loop Transformation Stage of aiT

mats correctly. As a consequence, a decision on the underlying opcode format has to be taken during translation from LLIR to CRL2. In such a situation, our compiler framework decides to assume the 32-bit version of an operation.[1] After this decision is finally taken, all LLIR operations are assigned a unique CRL2 *op_id*. Hence, a complete translation of our compiler's LLIR into aiT's IR is achieved.

Attention needs to be paid that the decision taken during the previous phase described above is also considered during the subsequent compilation steps. When writing out the contents of the LLIR into an assembly file, a line just like

```
MOV D9, D8
```

will be dumped. Eventually, the assembler processing the assembly file next treats this line as a 16-bit operation. Hence, the binary code produced by the assembler and the CRL2 file analyzed by aiT would differ, leading to incorrect WCET estimates. In order to prevent this situation, the decision taken on the bit-width of an operation is passed to the assembler using particular assembler directives [6]:

```
.code32
MOV D9, D8
```

Loop Transformation

Second, the conversions from LLIR to CRL2 and back need to be aware of a control flow transformation inherently performed by aiT. As already mentioned previously, execution contexts provide higher precision of WCET estimates. Due to the structure of CRL2, context information can only be attached to CRL2 functions. However, context-related information is not only relevant for functions, but also for loops within functions. For example, it is useful to distinguish individual iterations of a loop during WCET analysis and to compute different WCET information for loop iterations, depending on e. g. cache states. This can only be achieved if contexts are also applicable to loops. This is done within aiT by moving loops out of their original function into a new dedicated CRL2 function which calls itself recursively during each iteration (cf. Figure 3). This way, aiT is able to attach context-related information to loops by simply annotating the newly created loop functions.

However, this loop transformation stage has the effect that the CFG structures of LLIR and CRL2 differ. Since a CRL2 CFG may contain more functions than its LLIR counterpart, there is no direct correspondence at the function level. When traversing the CRL2 CFG in order to import all WCET data computed by aiT into the WCC compiler, CRL2 functions may be reached for which no LLIR

[1]We could also assume a 16-bit operation – the actual width does not matter. However, it is important to feed the decision to the assembler.

function exists. Hence, the question arises where to store all the WCET data attached at CRL2 functions within LLIR.

Basic blocks have a unique identifier in both IRs, namely the label denoting its starting address. WCET information stored at the level of CRL2 functions is attached to the very first basic block of this function within LLIR. For CRL2 functions F also existing within LLIR, this has the effect that all WCET data of F will also be available for F within LLIR by just checking F's first basic block. For CRL2 functions F' generated during loop transformation, the WCET data attached to F' will be stored at that position within LLIR where the loop represented by F' begins. This way, all WCET information is stored exactly at those code positions within LLIR to which the WCET data belongs.

3.3. LLIR Objectives and Handlers

In a simple approach, all WCET data extracted from CRL2 would directly be attached to the corresponding LLIR components as described in the previous section, e. g. the class of LLIR basic blocks would be extended by new attributes storing the execution count and the block's WCET. However, this approach is too inflexible for our purposes.

As already shown in Section 2, WCET-related compiler optimizations will have to deal with trade-offs between WCET and other optimization criteria, e. g. code size [13]. In the future, we intend to extend the WCC infrastructure presented in this paper towards a full multi-objective optimization engine. Besides the WCET data, the compiler will thus have to deal with other types of information representing other optimization objectives. Hence, the simple approach to just store this additional data within the LLIR core classes like e. g. basic blocks in the future will require the entire LLIR core to be extended and rebuilt. But since a compiler's IR is a complex piece of software, modifications of its core should be reduced to an absolute minimum.

In order to be able to attach arbitrary data to the LLIR core components without having to modify the core in the future, we have extended the LLIR by a modular and flexible objective and handler mechanism.

An *LLIR objective* represents a container storing all data relevant for the compiler in order to perform optimizations for a particular objective. Besides its pure data elements, methods in order to set and get the data stored within an LLIR objective are provided. Additionally, each LLIR objective carries specific type information such that a WCET LLIR objective can be distinguished from e. g. a code size objective by simply comparing their type attributes.

Using inheritance from an abstract objective class, other kinds of LLIR objectives besides WCET can be created easily. Arbitrary types of LLIR objectives can now be attached to the entire LLIR control flow graph, to LLIR functions, basic blocks and instructions. All of these LLIR components contain a so-called *objective handler* which is responsible for managing multiple LLIR objectives of different type attached to the same LLIR component. In this way, arbitrary data useful for compiler optimizations pursuing different optimization goals can be freely attached to the LLIR. The objective handler takes care that only one instance of an LLIR objective with a given type can be attached to an LLIR component, so that all data relevant e. g. for the object's WCET is not scattered among various instances of the WCET LLIR objective. The objective handler provides

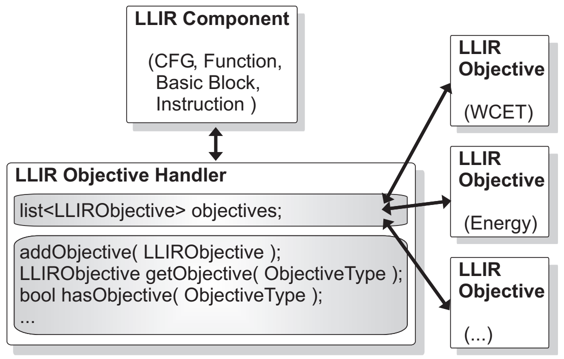
LLIR Component
(CFG, Function, Basic Block, Instruction)
LLIR Objective Handler
list<LLIRObjective> objectives;
addObjective(LLIRObjective);
LLIRObjective getObjective(ObjectiveType);
bool hasObjective(ObjectiveType);
...
LLIR Objective
(WCET)
LLIR Objective
(Energy)
LLIR Objective
(...)

Figure 4. LLIR Objective Handling

methods in order to set and get objectives of a certain objective type. The structure of the LLIR objective and handler mechanism is illustrated in Figure 4.

4. Discussion of the Structure of WCC

The proposed architecture of our WCET-aware C compiler WCC has several advantages compared to previously published approaches (cf. Table 1).

First, the use of both a high- and a low-level IR is extremely beneficial for WCET analysis and optimization. WCET analysis inherently requires high-level flow facts to be present. Previously published approaches recompute these flow facts from a low-level IR. This approach is cumbersome and inelegant. The availability of a high-level IR within our WCC compiler results in an overall simplification of flow fact computation since flow facts are computed directly at the corresponding level of abstraction. In addition, there are several high-level control flow optimizations potentially having a positive effect on WCET. For example, procedure cloning making use of calling contexts is easier to realize using a high-level IR than using LLIR.

Second, our WCC compiler does not rely on a proprietarily developed WCET analyzer. Instead, we were able to tightly integrate aiT into our compiler. By coupling WCC with AbsInt's aiT, we do not use simplified models of the target processor architecture at all. In contrast to various other publications, no simplifying assumptions on e. g. pipeline behavior and cache structure are made during WCET analysis. This is ensured by the high quality of aiT where the entire processor architecture is modeled with a very high degree of accuracy. As a consequence, the WCET data our compiler uses is highly accurate and precise.

Due to the accuracy and precision of the processor models used by aiT, a vast amount of WCET-related information is computed. Since both the compiler and aiT use CRL2 as common exchange format, all this data is available within WCC without limitations. Currently, we are able to import the following data from aiT into our compiler: global WCET of an entire CFG, calling contexts, context-dependent WCETs and execution counts of basic blocks and functions, feasibility of CFG edges, overall WCETs of basic blocks. Using the infrastructure presented in this paper, it is easy to import all other WCET data from aiT to WCC like e. g. value ranges of registers or pipeline and cache states.

The concept of objectives and handlers presented in this paper allows to model arbitrary trade-offs within our compiler. Previous publications have already shown that it is relevant to combine WCET optimization with other criteria like e. g. code size. Using objectives and handlers, our

	High/Low-Level IR	Tight WCET Integration	Automated Flow Facts	Compiler Optimization Support	Complex Processor	Avail. WCET Data	Multi-Objective	Tested
WCC	+	+	±	±	+	+	+	+
[4]	–	+	–	–	–	–	–	–
[7]	+	–	–	+			–	–
[12]	–	–	+	+	–	–	–	+
[17, 18]	–	+	+	+	–	–	–	+
[13]	–	–			–	–	+	+
[16]	–	–	+	+	±	–	–	+

Table 1. Comparison of WCC with Related Work

compiler is already well designed for this purpose, even if this is part of our future work.

5. Summary and Future Work

This paper is the first one describing a tight integration of an existing WCET analyzer into a compiler infrastructure. Using our techniques, it is possible to feed the assembly-like contents of our compiler's low-level IR LLIR into AbsInt's aiT automatically. This is achieved by translating LLIR into aiT's intermediate format CRL2. Since LLIR is an assembly-level representation, some information only visible at the linker level (e. g. exact opcode formats of operations having the same mnemonic) is unavailable. Hence, care has been taken during the translation from LLIR to CRL2 that the generated CRL2 representation of a given program is fully equivalent to its LLIR representation, and that the same holds for the final executable program generated by assembling and linking the LLIR representation.

After CRL2 generation, the WCET analyzer aiT is invoked automatically by our compiler. After its timing analyses are completed, the WCET data produced by aiT is imported into LLIR for further compiler optimizations.

In order to integrate WCET data within LLIR, a novel mechanism for future multi-objective compiler optimization is employed. It allows to attach arbitrary data relevant for compiler optimizations pursuing different optimization goals to almost all components of the LLIR. For example, this objective handler technique enables to store code size or energy and WCET data within LLIR at the same time in a flexible, modular and extensible way. This multi-objective design of the LLIR is motivated by the fact that the design of low-cost real-time systems requires performing automated trade-offs between different optimization criteria.

Our compiler is equipped with an extensive testbench consisting of 1,500 C files with 165,000 lines of code. The WCC compiler successfully compiles this testbench, generates valid CRL2 and performs WCET computation using aiT in the background. These tests serve as a proof of concept for the techniques presented in this paper and clearly demonstrate the reliability and quality of our concepts.

As already stated in Section 2, a good WCET analysis framework relies on the presence of high-level flow facts. To make them available within LLIR, we need to compute these flow facts within our high-level IR ICD-C, keep them up to date during optimization and pass them through the code selector down to the LLIR. Thus, the integration of flow fact computation [8] and transformation [12] into our WCC is part of the future work. Hence, the corresponding entries for WCC in Table 1 are marked just with $\pm$.

Of course, the main part of our future work is to develop WCET-aware compiler optimizations both at the high ICD-C level and at the LLIR level. This requires some kind of back annotation of the WCET data stored in the LLIR up to our high-level IR. Using our compiler structure with WCET data available at all levels, several issues noted in the WCET compiler wish list [3] can be tackled in the future.

Besides pure WCET-aware optimizations, we will consider multi-objective optimizations within our compiler in order to achieve trade-offs between real-time constraints and other optimization criteria.

Acknowledgments

The authors would like to thank Jens Wagner, Jörg Eckart and Luis Gomez who have designed the LLIR.

References

[1] AbsInt Angewandte Informatik GmbH. aiT: Worst-Case Execution Time Analyzers. *http://www.absint.com/ait*, 2005.
[2] AbsInt Angewandte Informatik GmbH. CRL Version 2. *http://www.absint.com/artist2/doc/crl2*, 2006.
[3] G. Bernat and N. Holsti. Compiler Support for WCET Analysis: a Wish List. In *Proc. of "3rd Intl. Workshop on WCET Analysis" (WCET)*, Porto, July 2003.
[4] H. Börjesson. Incorporating Worst Case Execution Time in a Commercial C-Compiler. Master's thesis, Uppsala University, Jan. 1996.
[5] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley, Boston, 2001.
[6] D. Elsner and J. Fenlason. *Using as – The GNU Assembler*. Free Software Foundation, 1994.
[7] J. Engblom. Worst-Case Execution Time Analysis for Optimized Code. Master's thesis, Uppsala University, Uppsala, Sept. 1997.
[8] J. Gustafsson, A. Ermedahl and B. Lisper. Towards a Flow Analysis for Embedded System C Programs. In *Proc. of "The 10th IEEE Intl. Workshop on Object-oriented Real-time Dependable Systems" (WORDS)*, Sedona, Feb 2005.
[9] *TriCore 1 32-Bit Unified Processor Core v1.3 Architecture – Architecture Manual*. Infineon Technologies AG, Sept. 2002.
[10] Informatik Centrum Dortmund. ICD-C Compiler framework. *http://www.icd.de/es/icd-c*, 2006.
[11] *ICD Low Level Intermediate Representation backend infrastructure (LLIR) – Developer Manual*. Informatik Centrum Dortmund, 2006.
[12] R. Kirner and P. Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. of "13th Euromicro Conference on Real-Time Systems" (ECRTS)*, Delft, Jun 2001.
[13] S. Lee, J. Lee, C. Y. Park and S. L. Min. A Flexible Tradeoff between Code Size and WCET using a Dual Instruction Set Processor. In *Proc. of "Intl. Workshop on Software and Compilers for Embedded Systems" (SCOPES)*, Amsterdam, Sept. 2004.
[14] R. Leupers. *Code Optimization Techniqes for Embedded Processors - Methods, Algorithms and Tools*. Kluwer Academic Publishers, Boston, 2000.
[15] S. Steinke, L. Wehmeyer et al. The *encc* Compiler Homepage. *http://ls12-www.cs.uni-dortmund.de/research/encc*, 2002.
[16] L. Wehmeyer and P. Marwedel. Influence of Onchip Scratchpad Memories on WCET Prediction. In *Proc. of "4th Intl. Workshop on WCET Analysis" (WCET)*, Catania, June 2004.
[17] W. Zhao, W. Kreahling, D. Whalley et al. Improving WCET by Optimizing Worst-Case Paths. In *Proc. of "11th RTAS Symposium"*, San Francisco, Mar 2005.
[18] W. Zhao, P. Kulkarni, D. Whalley et al. Tuning the WCET of Embedded Applications. In *Proc. of "10th RTAS Symposium"*, Toronto, May 2004.