Real-Time Syst (2010) 46: 251–300
DOI 10.1007/s11241-010-9101-x

A compiler framework for the reduction of worst-case execution times

Heiko Falk · Paul Lokuciejewski

Published online: 22 July 2010

Abstract The current practice to design software for real-time systems is tedious. There is almost no tool support that assists the designer in automatically deriving safe bounds of the *worst-case execution time* (*WCET*) of a system during code generation and in systematically optimizing code to reduce WCET.

This article presents concepts and infrastructures for WCET-aware code generation and optimization techniques for WCET reduction. All together, they help to obtain code explicitly optimized for its worst-case timing, to automate large parts of the real-time software design flow, and to reduce costs of a real-time system by allowing to use tailored hardware.

Keywords Real-time · WCET · Compiler · Code generation · Optimization

1 Introduction

Embedded systems often have to meet real-time constraints that make them real-time systems. Today, software development for embedded systems relies on high-level languages like C, and compilers. Modern compilers include a vast variety of optimizations. However, they mostly aim at reducing *average-case execution times* (*ACETs*). The effect of optimizations on *worst-case execution times* (*WCETs*) has not been studied in-depth up to now. In addition, even modern compilers are often unable

The research leading to these results has received funding from the European Community's ArtistDesign Network of Excellence and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement No. 216008.

H. Falk (✉) · P. Lokuciejewski
Computer Science 12, TU Dortmund University, 44221 Dortmund, Germany
e-mail: Heiko.Falk@tu-dortmund.de

P. Lokuciejewski
e-mail: Paul.Lokuciejewski@tu-dortmund.de

to quantify the effect of an optimization since they lack precise timing models (Lee 2005).

Currently, software design for real-time systems is tedious: they are often specified graphically using tools like e.g., ASCET. These tools automatically generate C code which is compiled in the next step. Since usual compilers have no integrated notion of timing, applied optimizations may lead to large WCET degradations. Therefore, it is common industrial practice to disable almost all optimizations during compilation. The code produced by the compiler is then manually fed into a WCET analyzer that computes timing information. Only after this very final step in the entire design flow, it can be verified if timing constraints are met. If not, the graphical design is changed in the hope that the resulting C and assembly codes have a lower WCET.

Up to now, no tools exist that assist the designer to purposively reduce WCETs of C or assembly code, or to automate the above design flow. In addition, hardware resources are heavily oversized due to the use of unoptimized code. Thus, it is desirable to have a WCET-aware compiler. Integrating WCET analysis into the compiler itself has the following benefits: first, it extends the compiler by a WCET timing model such that the compiler has a clear notion of a program's worst-case behavior. Second, this model is exploited by specialized compiler optimizations that reduce the WCET. Thus, the designer no longer needs to use unoptimized code, cheaper hardware platforms tailored towards the real software resource requirements can be used, and the tedious work of manually reducing the WCET of auto-generated C code is taken from the designer. Third, manual WCET analysis is no more required since this is done transparently by the compiler, using its tight integration of a WCET analyzer.

This article presents concepts, infrastructures and optimizations for WCET-aware code generation. All techniques discussed in this article are implemented and altogether form the WCET-aware C Compiler *WCC* (WCET-aware Compilation 2010), the first and currently only fully functional compiler which aims at fully automated WCET reduction at both source code and assembly code level.

1.1 Motivation

Typically, an executable program exhibits a certain variability of execution times influenced by input data and interference from the environment. Among all possible execution times of a program, the absolute maximum is the longest execution time a program can ever take. This time is called worst-case execution time. Unfortunately, it is in general very difficult or even impossible to determine the actual WCET of a program since this would include to solve the halting problem. Instead of computing the actual WCET, reliable upper bounds have to be determined by sound methods.

Two different approaches are used to estimate WCET bounds. The first approach is *measurement-based WCET analysis*. Here, the program under analysis is executed or simulated using some representative input values. A safety margin of e.g., 20% is added to the measured execution times and the resulting value is considered as the WCET. This approach is highly unsafe since no guarantee can be deduced that the inputs used during measurement really lead to the program's worst-case behavior.

If safe WCET guarantees for hard real-time systems are needed, *static program analyses* are used. The overall workflow of the leading static WCET analyzer aiT

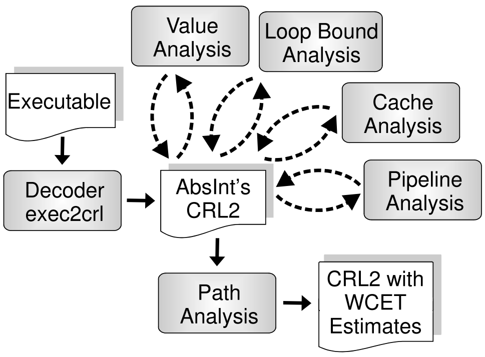


Fig. 1 Workflow of the static WCET analyzer aiT

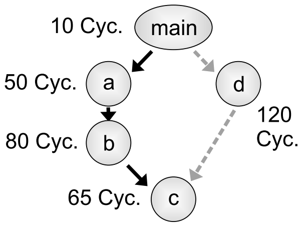


Fig. 2 Original example CFG

(AbsInt Angewandte Informatik GmbH 2010) is shown in Fig. 1. aiT applies static analyses on its intermediate format for executable code (CRL2) to e.g., estimate register values, loop iteration counts, and cache and pipeline states. The *Path Analysis* stage computes a program's global WCET. For each block on a path P from a program's entry point to its end point, its maximum execution time T is given after *Pipeline Analysis*. Using the determined loop iteration counts, a block's maximum number of executions C is estimated. The WCET of P is the sum of the products $T * C$ over all blocks of P. A program's WCET is computed by finding the maximum path WCET for all feasible paths. This maximization problem is modeled and solved using *integer linear programming* (*ILP*).

This path within a program's *control flow graph* (*CFG*) which has the maximal WCET is called the *worst-case execution path* (*WCEP*). Hence, the WCET of a program is equal to the WCET of its WCEP. In the following, a path's WCET will also be called the path's length. To reduce WCETs by a WCET-aware compiler, optimizations must exclusively focus on those parts of the program that lie on the WCEP. Optimization of parts of the program aside the WCEP are ineffective, since they do not shorten the WCEP and thus do not reduce the WCET. Therefore, optimization strategies for WCET reduction must have detailed knowledge about the WCEP. Static WCET analysis as shown above provides information about a program's WCEP, but solely knowing the WCEP is still insufficient for effective WCET reduction.

Consider the CFG of a function `main` in Fig. 2 that consists of five basic blocks. Each of them has the indicated WCET given in processor cycles. As can be seen, the longest path through this CFG is `main`, `a`, `b`, `c`. This WCEP, highlighted with solid arrows in the figure, leads to an overall WCET of 205 cycles.

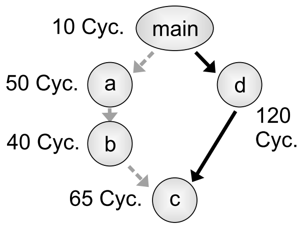


Fig. 3 CFG after optimization of `b`

We assume that some WCET-aware optimization reduces the WCET of basic block `b`, that lies on the WCEP, from 80 cycles down to 40 cycles (cf. Fig. 3). As can be seen, the WCEP after optimization of `b` is `main`, `d`, `c`. Additionally, reducing `b`'s WCET by 40 cycles does not reduce the overall WCET by 40 cycles. Instead, the overall WCET now amounts to 195 cycles which corresponds to an overall saving of only 10 cycles.

This example shows that the WCEP is unstable—it can switch from one path within the CFG to a completely different one due to a decision taken by some optimization. Thus, a WCET-aware compiler is faced with the following challenges which turn the development of WCET-aware optimizations into an even more demanding area of research compared to traditional compiler optimization:

– During the entire optimization process, WCET-aware optimizations must have detailed knowledge of the current WCEP at any point in time.
– They must be aware of the fact that the WCEP may switch in the course of an optimization and they thus have to recompute the WCEP whenever necessary.
– Additionally, optimization decisions should not only rely on local WCET data for a single code block, since local WCET savings for a single block do not necessarily translate into global WCET savings of the same order of magnitude.

This article is the first one to present a holistic approach and infrastructure for WCET-aware code generation. The key contributions are that the proposed compiler

– is equipped with a precise WCET timing model during code optimization,
– applies static WCET analysis automatically in the background, without requiring the compiler user to reason about assembly code structures that influence WCET analysis. Instead, the user is urged to support WCET analysis at source code level,
– features various optimizations which are explicitly tailored towards WCET reduction and thus overcome the challenges listed above,
– applies WCET optimizations both at source code and at assembly code level. This structure is advantageous since it helps to exploit the benefits of these different abstraction levels individually. For example, optimizations that consider function calls and function arguments are much more difficult to realize at assembly code level. Likewise, it is difficult to apply memory allocation optimizations as proposed in this article at the source code level since highly precise data on code sizes or register interference is usually only available at assembly code level.

1.2 Related work

A very first approach to integrate WCET techniques into a compiler was presented in Börjesson (1996). Flow facts used for WCET analysis were annotated manually via source-level pragmas. A fully pragma based approach is not promising since manual annotations are tedious and error-prone. Additionally, the compiler targets the Intel 8051 which is an inherently simple and predictable machine without pipeline and caches etc.

While mapping high-level code to object code, compilers apply various optimizations so that the correlation between high-level flow facts and the optimized object code becomes very vague. To keep track of the influence of compiler optimizations on high-level flow facts, co-transformation of flow facts is proposed in Engblom (1997). 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.

Kirner and Puschner (2001) present techniques to transform program path information which keep high-level flow facts consistent during 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 ANSI C, and the modeled target processor lacks pipelines and caches.

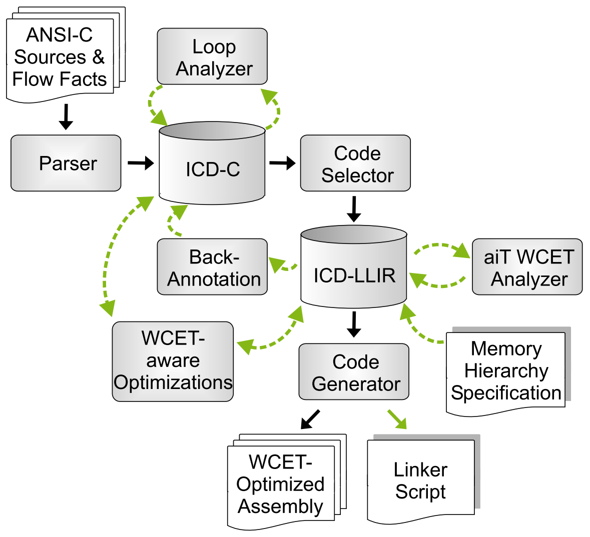
In Zhao et al. (2004, 2005), the integration of a proprietarily developed WCET analyzer into a compiler is presented. The compiler operates on a low-level *intermediate representation* (*IR*). Control flow information is passed to the timing analyzer which computes the WCET of paths, loops and functions and passes this data back to the compiler. This approach has the following limitations. First, the WCET analyzer works with very coarse granularity since it only computes WCETs of paths, loops and functions. WCETs for basic blocks or single instructions are unavailable. Thus, aggressive optimization of smaller units like single basic blocks is infeasible. Second, WCET-related data which is not the WCET itself is unavailable, too. This excludes e.g., execution frequencies of basic blocks, value ranges of registers, predicted cache behavior etc. Finally, WCET optimization at higher levels of abstraction like e.g., source code level is infeasible since WCET-related data is not provided at source code level.

1.3 Overall compiler infrastructure and article outline

The *WCET-aware C Compiler WCC* (WCET-aware Compilation 2010) described in this article is a C compiler for the Infineon TriCore TC1796 and TC1797 processors that are heavily used in the automotive industry. WCC's overall structure is depicted in Fig. 4. Those modules of the compiler connected with solid arrows resemble a typical optimizing compiler:

Parser: The parser is fully compliant with ANSI C. It accepts several C source files within a single compiler run and creates a high-level IR called ICD-C from them.

Fig. 4 WCC compiler infrastructure



ICD-C: The ICD-C framework (Informatik Centrum Dortmund e. V. 2010b) is a data structure that provides a machine-independent IR for C code. It features machine-independent code analyses and optimizations. WCC uses ICD-C's code selector interface to couple the front-end with a tree-pattern matching based code selector for the TriCore processors.

Code Selector: The code selector translates ICD-C to a representation of TriCore assembly. WCC's grammar for the TC1796 / TC1797 consists of ca. 24,000 lines of C++ code which results in the generation of highly efficient machine code.

ICD-LLIR: ICD-LLIR (Informatik Centrum Dortmund e. V. 2010a) is a data structure providing a retargetable low-level IR for compiler back-ends. It includes various assembly-level analyses and optimizations. WCC's TriCore processor description for ICD-LLIR consists of ca. 14,500 lines of C++ code which capture all aspects of the complex TriCore architecture.

Code Generator: The code generator finally emits valid assembly code from the class structures of the TriCore ICD-LLIR within WCC's back-end.

The key components which turn WCC into a unique WCET-aware C compiler are depicted with dashed arrows in Fig. 4. Sections 2–6 describe these modules in more detail. They deal with WCC's memory hierarchy specification, integration of the aiT WCET analyzer, flow facts, loop analyzer and back-annotation.

Based on this infrastructure, WCET-aware source code level optimizations (procedure cloning and positioning) are presented in Sects. 7–8, followed by WCET-aware assembly code level optimizations (scratchpad and register allocation) in Sects. 9–11. Section 12 summarizes this article and gives an outlook on future work.

2 Specification of memory hierarchies

The performance of many systems in use today is largely dominated by the memory subsystem. Due to the large speed gap between slow memories and fast processors, execution times of software widely depend on the characteristics of the memory hierarchy on the one hand, and on the characteristics of memory accesses performed by the software on the other hand. Obviously, the WCET estimates produced by a static WCET analyzer as described in Sect. 1.1 also heavily depend on the memories.

In the compiler environment described in this article where the WCET analyzer is tightly integrated into the code generation process, it is in the duty of the compiler to provide the WCET analyzer with detailed information about the underlying memory hierarchy in order to obtain safe and tight WCET estimates. For this reason, WCC includes an infrastructure to specify memory hierarchies.

But WCC uses this memory hierarchy infrastructure not only for WCET analysis. In addition, WCC features many optimizations that exploit memory hierarchies by moving parts of a program's code and data onto fast memories to reduce WCETs. These optimizations also rely on precise knowledge of the memory subsystem which is provided by the infrastructure described in this section. Section 2.1 presents related work, and Sect. 2.2 describes WCC's memory hierarchy infrastructure.

2.1 Related work

Previous work on *architecture description languages* (*ADLs*) primarily focused either on efficiently building cycle-true instruction set simulators, or on code generation in a synthesizable *hardware description language* (*HDL*). Lisa (Hoffmann et al. 2001) originally aims at the automatic generation of application-specific hardware and of corresponding simulators and low-level tools. It has also been extended towards automatic compiler generation. For this purpose, a semantic instruction set model was added. Thus, Lisa system specifications need to provide cycle-true timing models as well as semantical information about a processor's instruction set.

ArchC (Azevedo et al. 2005) was designed to support processor architecture description. Recently, means to design memory hierarchies have been added. In analogy to Lisa, ArchC also covers structural and behavioral aspects of a system model. In contrast to Lisa, ArchC builds upon SystemC in order to specify timing and concurrency. Due to the strong relationship between ArchC and SystemC, it is relatively straightforward to create an instruction set simulator or to generate synthesizable HDL code from ArchC.

The Target Description Language TDL (Kästner 2003) focuses on retargetable optimization of assembly code. It uses a structural description of a system's resources that includes memory and cache hierarchies. A behavioral instruction set description is the second key part of TDL. Due to its vicinity to code optimization, TDL is the approach coming closest to the memory hierarchy specification infrastructure of our WCC compiler.

2.2 Memory hierarchy specification

The approaches for architecture or memory hierarchy specification described in Sect. 2.1 base on powerful and retargetable ADLs. They feature sophisticated structural and behavioral information that includes detailed timing models. WCC's memory hierarchy infrastructure differs from previous work in the following aspects:

– Due to the way how worst-case timing models are integrated into our compiler (cf. Sect. 3), there is no need to equip WCC's memory hierarchy specifications with sophisticated timing models. For our purpose of supporting code optimizations, only some key parameters like e.g., memory access latencies are sufficient.
– Since we do not focus on synthesizable HDL code generation, our infrastructure does not need to provide precise, cycle-accurate timings, again. This turns WCC's memory hierarchy specification into a very lightweight infrastructure.
– Previous approaches were retargetable in that sense that they generated tools or HDL code for different target processor architectures. For this purpose, ADLs are usually equipped with detailed semantical information about the meaning of a processor's instruction set. In contrast, WCC does not require such semantical instruction set information since we only consider the memory hierarchy. Thus, retargetability of the memory hierarchy within WCC is achieved by simply updating and reconfiguring a processor's memories as described in the following.

Due to their focus on cycle-accurate timing, retargetability and HDL code generation, previous ADLs are very powerful but also very heavy tools. WCC's memory hierarchy infrastructure is designed to be lightweight and to only support optimizations which move parts of a program across different memories. This kind of memory allocation is usually performed by the linker during the final step of generating an executable. Thus, the information usually available only while linking needs to be provided already to the WCC compiler itself. This is because it is up to WCC in our setup to decide on a program's memory layout, and no longer up to the linker.

In a conventional environment where information on the memories is only available to the linker, the compiler is fully unaware of the available memories and thus can not optimize the code for a given memory hierarchy. Making information about a processor's memories already available to the compiler has the advantage that all tools involved in analysis and generation of machine code now have detailed knowledge on the memory hierarchy. In our environment, the WCC compiler is fully aware of the memories and can exploit them during optimization. Furthermore, WCC passes this memory hierarchy information on to the WCET analyzer such that the computed WCETs always reflect the actual memory layout as decided by the compiler. Finally, WCC passes memory-related data on to the linker that produces executable code which, again, reflects exactly the memory layout determined during compilation.

Making the compiler statically determine a program's memory layout has the drawback that dynamic code relocation, which is one of the key functions of modern linkers, is infeasible. However, this is not a serious limitation for embedded hard real-time systems, since dynamic relocation is not used there. Instead, the machine codes of different real-time tasks produced by a compiler are combined with the kernel of a *real-time operating system* (*RTOS*) during compile time so that a fully static executable that includes the RTOS and its tasks is finally produced. In such systems, application code is never loaded dynamically so that relocation is not an issue.

WCC provides a simple text file interface to specify memory hierarchies. Such a memory specification describes different regions of a processor's physical memory hierarchy. For each physical memory region, the following attributes can be defined:

Fig. 5 Example for memory specification within WCC

```
# Data SRAM (DMU)
[DMU-SRAM]
origin = 0xc0000000
length = 0x10000 # 64K
attributes = RWA # read/write/allocatable
cycles = 6
sections = .data.sram
```

– the region's base address and absolute length,
– access attributes like e.g., read, write, executable, allocatable,
– memory access times, specified in processor cycles,
– assembly-level sections that are allowed to be mapped to a memory region.

For caches, various attributes like e.g., absolute sizes, line sizes or associativity can be specified, too. Figure 5 shows a fragment of WCC's memory hierarchy specification for the Infineon TriCore TC1796 processor.

Now that WCC is aware of the processor's physical memories, program fragments need to be moved to the present memories within the compiler's back-end. The low-level IR ICD-LLIR used by WCC maintains a set of assembly-level sections that serve as containers for e.g., program code, uninitialized or pre-initialized data, constants etc. `sections` directives in a memory specification (cf. Fig. 5) define a mapping to which physical memory region an assembly section can be moved.

Memory allocation of program fragments is done by simply assigning ICD-LLIR objects to such sections. Currently, functions, basic blocks and data objects like e.g., global variables or arrays can be assigned to sections. Our infrastructure provides a convenient API to do such memory assignments of code and data. Symbol tables allow to retrieve physical memory addresses per function, basic block, or data object.

Finally, the memory allocation decisions taken by WCC must be respected by all subsequent WCET analysis and linkage stages. On the one hand, Sect. 3 describes how WCET analysis within WCC adheres to a program's actual memory layout. On the other hand, the binary executable generated by WCC must exactly match the memory layout decided by WCC. Since the executables are produced outside WCC by an external linker, WCC automatically generates a GNU *ld* compatible linker script and invokes the linker using this linker script. This way, the binary executable is fully equivalent to the memory layout determined by WCC's optimizations.

3 Integration of static WCET analysis into the compiler

Accurate WCET timing models are available in static WCET analyzers (cf. Sect. 1.1). To include such models in the WCC compiler, they should not be re-implemented from scratch inside the compiler. Rather, timing experts should develop timing analyzers, while compiler developers should generate efficient code using aggressive optimizations. Hence, WCC and the WCET analyzer aiT are two separate tools that are tightly coupled at the compiler back-end, enabling a seamless exchange of information. After providing an overview of related work, the integration of aiT's timing model into WCC is presented. For more details, please refer to Falk et al. (2006).

3.1 Related work

Most of the present WCET-aware compilation frameworks do not provide a seamless integration of WCET analysis into a compiler. Kirner and Puschner (2001) present transformations of program path information during compiler optimization. However, compilation and WCET analysis are fully decoupled. The assembly output of the compiler is passed to the WCET analyzer together with further mandatory information on the program's control flow. Additionally, the proposed compiler only processes a limited subset of ANSI C, and the modeled target processor lacks pipelines and caches.

The interactive compilation system *VISTA* (Zhao et al. 2004) translates a C source code into a low-level IR used for code optimizations. It includes a proprietary static WCET analyzer that supports simple processors without caches like the StarCore SC100. VISTA contains a loop analysis which is only able to detect simply structured loops, hence most loop iteration counts must be provided manually. Unlike WCC, recursive code can not be analyzed. The used WCET analyzer has a limited scalability, enabling the analysis of only small program codes. In contrast to WCC, VISTA lacks a high-level IR, therefore no WCET-aware source code optimizations can be developed.

Heptane (Colin and Puaut 2001) is a static WCET analyzer with multi-target support for simple processors like StrongARM 1110 or Hitachi H8/300. It expects a C source code as input that is parsed into a high-level IR. Next, the code can be translated into a low-level IR. Heptane solely supports WCET-driven assembly optimizations, e.g., predictable page allocations. The WCET can be computed either at source code level via a tree-based approach using combination rules for source code statements or via an ILP-based method that operates on a CFG extracted from the task's binary. Since Heptane does not support a detection of infeasible paths, the derived upper bounds may be considerably overestimated, as compared to WCC's integrated timing analysis. Moreover, compiler optimizations are not supported and must be disabled to avoid a mismatch between the syntax tree and the control flow graph.

SWEET (Gustafsson et al. 2006) is a static WCET analyzer with a research focus on flow analysis. It incorporates different techniques for the calculation of loop iteration counts and the detection of infeasible paths. Due to the missing import of WCET data into a compiler framework, the development of compiler optimizations that aims at a WCET reduction is not possible. To avoid a mismatch between the high-level IR, where the flow analyses are performed, and the object code used for the WCET analysis, assembly level optimizations are not allowed. In addition, SWEET is coupled to a research compiler which is only able to process a subset of ANSI C. The supported pipeline analysis is limited to in-order pipelines and does not consider timing anomalies. SWEET's target architectures are the ARM9 and NEC V850E processor.

An integration of a static WCET analyzer into a compiler framework called *TUBOUND* was presented in Prantl et al. (2008). It allows to apply source code optimizations since flow facts specified as pragmas in the ANSI C code are automatically updated. This is achieved by extending supported optimizations by a mechanism that keeps flow facts consistent. This approach resembles the handling of flow facts in the WCC framework. However, in contrast to WCC's 27 flow fact aware source code optimizations, Prantl et al. (2008) reports about three supported optimizations. Assembly optimizations are not available in TUBOUND due to a missing compiler back-end support. Currently, TUBOUND supports the simple C167 processor, which lacks caches and a pipeline.

3.2 Conversion from LLIR to CRL2

WCET analysis takes place at the assembly/binary level since processor-specific information and machine code is unavailable at higher abstraction levels. Thus, the WCET analyzer aiT is coupled to the WCC compiler at the LLIR level (cf. Fig. 4).

CRL2 is aiT's exchange format which stores the application under WCET analysis and all of aiT's analysis results. Since both LLIR and CRL2 are low-level IRs, a mutual translation of their CFGs is straightforward. The CFGs of both IRs consist of functions. Each function is a list of basic blocks connected via edges. Basic blocks in turn are a sequence of instructions. In both IRs, an instruction consists of several operations to express the implicit parallelism of e.g., VLIW machines. Due to the analogy of both IRs, it is basically sufficient to traverse the LLIR CFG and to generate corresponding CRL2 components to construct an equivalent CRL2 CFG.

The conversion of LLIR to CRL2 is complicated by the fact that CRL2 is generated from a binary executable, i.e., it relies on information produced by an assembler and a linker. In contrast, LLIR is an assembly level IR that lacks this information. This becomes apparent when converting LLIR to CRL2 operations. The latter requires a unique opcode that denotes the machine code of the operation. However, this opcode is in general unavailable at assembly level and must be computed by WCC, taking operation characteristics like the involved operands, operation size or addressing modes into account. More details about the respective algorithm can be found in Falk et al. (2006).

Another key difference between both IRs is that CRL2 relies on physical addresses while LLIR uses symbolic names for addresses. To bridge this semantic gap, the IR conversion exploits WCC's memory hierarchy specification (cf. Sect. 2) which provides the required physical information at assembly level. Using WCC's memory hierarchy API, physical addresses for LLIR basic blocks and operations are computed. In addition, branch targets of jump operations, which are represented by symbolic block labels, are translated into physical addresses. Similarly, symbolic labels involved in accesses to global variables via load/store operations are converted.

3.3 Transparent invocation of aiT

Using the conversion from LLIR to CRL2, WCC produces a CRL2 file that represents the program for which WCET timing data is required. Fully transparent to the user, WCC invokes aiT on this CRL2 file. The compiler takes control over the WCET analyzer and performs its value, loop bound, cache, pipeline, and path analysis.

As a consequence, the WCET analyzer is completely encapsulated in WCC. The compiler user is unaware of the fact that timing analysis is performed in the background. The user does not get in touch with the configuration of parameters mandatory to run a static WCET analysis. The burden of setting up a valid run-time environment for aiT is taken away from the user and is completely managed by WCC.

Also, WCC can automatically compute data that increases the precision of the WCET analysis, e.g., possible addresses of memory accesses, and pass them to aiT. Otherwise, these specifications require a tedious and error-prone definition by the user.

3.4 Import of worst-case execution data

After aiT is invoked, the analyzer's results are inserted into a final CRL2 file which represents a program's CFG enriched with all the WCET data computed by aiT. The last step for the complete integration of aiT's timing model into WCC consists of traversing this final CRL2 file, extracting its WCET data and importing this data into WCC's back-end by attaching it to the ICD-LLIR. The following list gives an overview about the WCET-related data made available within the WCC this way:

– WCET of the entire program, of each function, and each basic block,
– worst-case call frequency per function,
– worst-case execution frequency per basic block,
– worst-case execution frequency per CFG edge,
– execution feasibility of each CFG edge,
– safe approximation of register values,
– encountered I-cache misses per basic block.

Currently, WCC does not make use of context-sensitive information. All provided compiler optimizations have a static view of the code where different calling contexts are not distinguished. Hence, context-sensitive data computed by aiT is accumulated over all calling contexts and imported into WCC as context-insensitive information.

4 Flow fact specification and transformation

A program's execution time (on a given hardware) is strongly determined by its control flow, i.e., the execution order of instructions or basic blocks, as modeled by the CFG. Usually, constructs like e.g., loops or (conditional) branches express control flow. Static WCET analysis (Heckmann and Ferdinand 2004) is undecidable since it is undecidable to compute how many times a general loop iterates. Since loop iteration counts are crucial for a precise WCET analysis, and since they can not be computed for arbitrary loops in general, they need to be specified by the user of a static WCET analyzer.

Besides loops known from high-level programming languages, any cycle in a program's CFG needs to be annotated manually by the user. These user-provided annotations that specify the control flow are usually called *flow facts*. This section explains the mechanisms for flow fact specification and transformation within WCC.

Static WCET analysis can be divided into the following three areas (Puschner and Burns 2000):

1. Program execution paths should be defined at source code level, since a manual or automatic creation of this data at low abstraction levels is tedious and error-prone. WCC's ways for flow fact specification is subject of Sect. 4.1.

2. The transformation of this information from the source code level to the machine code level, where the actual static WCET analysis takes place, has to be automated. Section 4.2 describes WCC's mechanisms for flow fact updates.
3. Computation of WCET estimates for a program has to be done at a low level of abstraction close to the target architecture (cf. Sect. 1.1).

4.1 Specification of flow facts

Flow facts describe the set of possible execution paths of a program (Kirner 2003). To make WCET analysis feasible, the available flow facts must limit the execution count of every statement of a program. User-provided flow facts should be specified inside the source code since this way, only the code base needs to be maintained, and not the source codes plus some external flow fact files which are potentially forgotten.

In general, a static WCET analyzer requires the following kinds of flow facts to perform safe and precise WCET analysis:

– Loop iteration counts
– Recursion depths
– Execution frequency of an instruction, relative to some other instruction

The WCC compiler fully supports source-level flow facts by means of ANSI C pragmas. The WCC user can annotate C source codes using either *Loop Bound* (cf. Sect. 4.1.2) or *Flow Restriction* (cf. Sect. 4.1.3) flow facts. A survey of work related to the area of flow fact specification is provided in the following section.

4.1.1 Related work

Static WCET analyzers do timing analysis of executable code. Thus, machine code level flow facts are required. WCET analyzers usually include loop analyzers, but they determine loop iteration counts only for simple classes of loops. Hence, WCET analyzers rely on user-provided flow facts. For WCET analysis, the user must provide the machine code to be analyzed and a specification file that contains (among other annotations) flow facts. Using hexadecimal addresses in the specification file, a flow fact is related to those pieces of code it actually describes. Obviously, flow fact specification at machine code level is a very tedious and cumbersome issue.

In Engblom and Ermedahl (2000), scopes are defined as hierarchical groups of basic blocks such that a scope can be reached at most once via its header node within a program's CFG. Flow facts in Engblom and Ermedahl (2000) are a triple `(scope, context, constraint)` where `scope` refers to a scope for which a flow fact is to be specified, `context` is a particular calling context of the scope, and `constraint` is an inequation over the execution frequencies of basic blocks. Since it relies on basic blocks, this approach is feasible for low-level flow fact specification. However, it does not assist a user in high-level program analysis, which is the key motivation of WCC's flow facts presented in this section.

An ANSI C extension to specify flow facts is proposed in Kirner (2000, 2001). Using markers and scopes, the user of wcetC specifies loop iteration counts and inequations that relate the execution count of one code fragment to the execution count of some other piece of code. The main drawback of wcetC is its incompatibility with the ANSI C standard which prevents wcetC to be compiled with any available ANSI C compiler.

All currently known approaches have in common that flow facts are meant to specify execution counts of CFG nodes. Internally, however, flow facts are always transformed and are finally attached to CFG edges. During this conversion, a loss of either precision or of expressiveness of the specified information can be expected. In addition, previous approaches are unable to transform and to keep flow facts consistent during all the optimizations applied by an optimizing compiler (Engblom et al. 1999; Kirner 2003).

In contrast to related work, WCC's flow facts fully comply with the ANSI C standard, since ANSI C pragmas are used to specify flow facts. The user can annotate execution frequencies of ANSI C statements (i.e., CFG nodes), and WCC internally attaches this data to CFG nodes to avoid conversions that possibly degrade precision.

4.1.2 Loop bounds

Loop bounds specify limits of iteration counts of regular loops. Here, regular loops are *for*-, *while-do*- and *do-while*-loops of ANSI C with the following properties:

– they have only one single entry node (single-entry loops), and
– they must have a well-defined termination condition.

For such loops, loop bound flow facts allow to specify the minimum and maximum iteration counts. In its current state of implementation, loop bounds have to be unsigned integer values—symbolic constants are currently not supported:

`LOOPBOUND`	|= `loopbound min` <u>`NUM`</u> `max` <u>`NUM`</u>
`NUM`	|= *Non-negative Integer*

For example, the following snippet of C code specifies that the shown loop body is executed exactly 100 times:

```
_Pragma( "loopbound min 100 max 100" )
for ( i = 1; i <= 100; i++ )
  Array[ i ] = i * fact * KNOWN_VALUE;
```

In the future, loop bound flow facts could be extended by an equality operator such that only one single value needs to be specified for exact loop iteration counts. However, allowing to provide a minimum and maximum loop iteration count enables to annotate data-dependent loops. If e.g., `maxIter` is some data-dependent function parameter that ranges from 50 to 100, a data-dependent loop is annotated as follows:

```
_Pragma( "loopbound min 50 max 100" )
for ( i = 1; i <= maxIter; i++ )
  Array[ i ] = i * fact * KNOWN_VALUE;
```

4.1.3 Flow restrictions

For irregular loops (e.g., multi-entry loops, loops without explicit termination condition or loops that use *goto*-statements), loop bound annotations are inapplicable.

Instead, WCC provides *flow restriction* annotations which allow to relate the execution frequency of one C statement with that of other statements.

In order to use flow restrictions, some auxiliary annotations called *markers* are required which attach an identifying string to some source code statement. WCC's markers are identical to labels known from ANSI C or assembly code:

```
MARKER            |= marker NAME
NAME              |= Identifier
```

For example, the following piece of code attaches the identifier `outermarker` for further use to a source code statement:

```
_Pragma( "marker outermarker" )
Statement A;
```

Using the identifiers specified by markers, complex flow restriction annotations can be defined according to the following EBNF syntax:

```
FLOWRESTRICTION  |= flowrestriction SIDE COMPARATOR SIDE
COMPARATOR       |= >= | <= | =
SIDE             |= SIDE + SIDE | NUM * REFERENCE
REFERENCE        |= NAME | Function Name
```

Flow restrictions allow to specify linear dependencies between arbitrary positions in the C source code. E.g., the flow restriction below annotates a triangular loop:

```
_Pragma( "marker outermarker" )
Statement A;

for ( i = 0; i < 10; i++ )
  for ( j = i; j < 10; j++ )
    _Pragma( "marker innermarker" )
    Statement B;

_Pragma( "flowrestriction 1*innermarker <= 55*outermarker" );
```

It states that the execution frequency of the code marked by `innermarker` is at most 55 times larger than that of statement *A* marked by marker `outermarker`.

Similarly, recursion depths are specified via flow restrictions. For example, the C code below shows how to annotate a recursion that computes Fibonacci within WCC:

```
int fib( int i )              int main()
{                             {
   if ( ( i == 0 ) ||           int In = fib( 7 );
        ( i == 1 ) )            _Pragma( "marker recursion");
     return 1;                  _Pragma( "flowrestriction \
                                           1*fib <= 41*recursion");
   return fib( i - 1 ) +
          fib( i - 2 );         return In;
}                             }
```

4.2 Flow fact transformation

Flow facts must be transformed by the WCC compiler whenever it changes the code's abstraction level or it applies control flow changes. This is supported by techniques called flow fact translation (cf. Sect. 4.2.1) and flow fact update (cf. Sect. 4.2.2).

4.2.1 Flow fact translation

Due to the fact that source-level flow facts are highly desirable, there is a semantic gap between the place where flow facts are specified (C code) and where they are actually used for static WCET analysis (assembly code). WCC is inherently aware of this semantic gap and closes it using *flow fact translation*. Whenever WCC lowers the level of abstraction during compilation, a *flow fact manager* is responsible for the translation of all flow facts from the previous higher abstraction level to the lower level. More precisely, the flow fact managers are active during the following compiler stages:

From ANSI C to ICD-C: The first flow fact manager extracts loop bound, marker and flow restriction pragmas from the C source codes and attaches these flow facts to objects of ICD-C. All required classes of the ICD-C IR are made flow fact-aware and thus allow to hold user-specified flow facts.

From ICD-C to ICD-LLIR: Code selection translates the source-level IR ICD-C into ICD-LLIR machine code. Another flow fact manager thus translates all ICD-C flow facts to ICD-LLIR flow facts and attaches them to the corresponding LLIR objects. During this stage, it is guaranteed that the ICD-LLIR flow facts are semantically equivalent to the ICD-C flow facts.

From ICD-LLIR to CRL2: The third flow fact manager within WCC takes care of translating all ICD-LLIR flow facts to equivalent CRL2 flow facts so that the static WCET analyzer aiT is able to perform a precise WCET analysis.

4.2.2 Flow fact update

Flow fact translation by itself is insufficient to guarantee that the flow facts passed to aiT are semantically equivalent to the specifications provided at source code level. This is caused by the optimizations WCC applies at ICD-C and ICD-LLIR level.

Currently, WCC includes 42 different optimizations. 27 of them take place within ICD-C, the other 15 ones in ICD-LLIR. Many optimizations restructure loops to increase performance, but loop optimizations are particularly critical when flow facts are present. This is because restructuring of a loop potentially yields changed iteration counts which, in turn, have to be reflected by the attached flow facts. E.g., unrolling the following loop by a factor of two invalidates attached flow facts completely:

```
_Pragma( "loopbound min 100 max 100" )
for ( i = 1; i <= 100; i++ )
  Array[ i ] = i * fact * KNOWN_VALUE;
```

The following loop would result from unrolling:

```
_Pragma( "loopbound min 100 max 100" )
for ( i = 1; i <= 100; i += 2 ) {
  Array[ i ] = i * fact * KNOWN_VALUE;
  Array[ i + 1 ] = (i + 1) * fact * KNOWN_VALUE;
}
```

The flow fact states that the unrolled loop body is executed exactly 100 times which is not true since it is executed only 50 times. As a result, heavily overestimated WCET bounds can be expected from static timing analysis for this example.

Thus, all optimizations of WCC are made fully flow fact-aware using built-in *flow fact update* techniques. They ensure that safe and precise flow facts are maintained for each individual optimization. For the above example, the update mechanisms produce the flow fact `_Pragma( "loopbound min 50 max 50" )` after loop unrolling. WCC's update mechanisms support some fundamental operators on flow facts like e.g.,

- creation, copying and deletion of loop bounds and flow restrictions,
- displacement of the min/max interval of a loop bound,
- replacement of a flow restriction by another equivalent flow restriction, and
- replacement of a flow restriction by an inequivalent flow restriction if no fully equivalent replacement can be determined. This inequivalent flow fact is computed conservatively, such that it leads to an overapproximation of execution frequencies, but not to an unsafe underapproximation.

All basic operations of ICD-C and ICD-LLIR that create, delete or move statements or basic blocks were extended to automatically update flow facts via the techniques described above. WCC's optimizations were finally made flow fact-aware by using the aforementioned basic flow fact operators and by explicitly adjusting flow facts whenever such basic operations are not sufficient.

5 Automated loop bound analysis

WCC's goal is to fully automatically reduce WCETs, and WCET analysis requires the existence of flow facts. Manual flow fact annotation (cf. Sect. 4) becomes tedious and even infeasible even at the source code level if the program to be annotated is long, or if it is automatically generated by some high-level specification tool. To relieve the user from this burden and to establish an automated framework for WCET reduction, a static loop analyzer that produces flow facts for ICD-C was integrated.

Our loop analyzer bases on *abstract interpretation* (Cousot and Cousot 1977), a theory of a sound approximation of program semantics. It is applied at source code level since this level of abstraction provides valuable information, such as data types, which is lost when code is translated into a low-level IR. To accelerate loop analysis, the analyzed code is preprocessed using *program slicing* (Horwitz et al. 1988), a technique that excludes statements irrelevant for the loop analysis. Moreover, we introduce a novel *polyhedral loop evaluation* that further decreases analysis times. WCC's loop analyzer has proven to be of superior quality—among all tools participating in the WCET Tool Challenge 2008 (Holsti et al. 2008), it was the only one which solved all flow facts related analysis problems.

First, we give a survey of related work in Sect. 5.1 and introduce abstract interpretation in Sect. 5.2. Program slicing and our novel polyhedral evaluation are presented in Sects. 5.3 and 5.4, resp., followed by results achieved on real-life benchmarks in Sect. 5.5. A detailed description of the analysis can be found in Lokuciejewski et al. (2009).

5.1 Related work

Static loop analysis is crucial for different fields of applications. Besides WCET analysis, the knowledge of loop iteration counts can be used for aggressive loop optimizations or to assist feedback-directed compiler optimizations. In Healy et al. (1998), a pattern-based approach to determine loop iteration counts of assembly programs is presented. It exclusively analyses the parts of the assembly code that represent loops, while the remaining instructions are ignored. This way, loops relying on function parameters can not be analyzed. To solve this problem, the authors provide a mechanism that allows to specify value ranges for unknown variables, making their analysis semi-automatic.

The approach developed in Healy et al. (1998) has been adapted to programs written in the high-level language C by Kirner (2006). Again, loop analysis does not automatically succeed for all types of loops. Mandatory information that can not be extracted during the static analysis must be provided by the user in the form of source code annotations.

In contrast to pattern-based analyses, Cullmann and Martin (2007) use an interprocedural data-flow based loop analysis at assembly level. This has the advantage that the loop analysis does not strictly rely on pre-defined code patterns a particular compiler generates, but on the semantics of the instruction set of a specific target machine. As stated by the authors, the analysis works best for well-structured loops and supports only a simple modification of the loop counter by exclusively allowing additions of constant intervals.

A different approach for a fully automatic static loop analysis at source code level was described in Ermedahl and Gustafsson (1997). The authors involve a data flow analysis based on abstract interpretation. Representing values by intervals, a loss of precision is introduced making the concrete program semantics decidable. Based on this approximation, a determination of loop bounds is enabled. This work was used in Gustafsson et al. (2006) to assist static WCET analysis. It was extended to determine bounds of nested loops as well as to detect *infeasible paths*, i.e., paths that are not taken in particular execution contexts and which should thus be excluded from WCET analysis to avoid WCET overestimation. To further improve and accelerate this loop analysis, the authors combine different standard program analyses like *program slicing* and *invariant analysis* (Ermedahl et al. 2007).

5.2 Abstract interpretation

Static loop analysis includes solving the halting problem and is thus undecidable. For concrete program semantics, an automatic loop analysis that determines loop iteration counts for all types of loops is not feasible. However, by introducing abstract semantics, which is a superset of the concrete program semantics covering all possible concrete cases, the loss of information makes the analysis computable. The abstraction is accomplished by a technique called abstract interpretation.

The fundamental idea of abstract interpretation is to find a compromise between analysis precision and analysis run time. A reduction of information is achieved by mapping a possibly infinite set of program states, typically consisting of the value of the program counter (program point) and a set of variables (or memory locations), into a finite set of *abstract states*. A static analysis using abstract interpretation aims at assigning sets of possible variable values (abstract states) to CFG edges.

The main drawback of abstract interpretation is its iterative behavior in loops which might slow down the analysis such that it becomes impractical. In particular, such an explosion of analysis times can be observed during the analysis of loops with high iteration counts where each loop iteration is interpreted individually.

WCC's loop analyzer combines abstract interpretation with mechanisms to avoid its iterative behavior. They rely on interprocedural program slicing and polyhedral loop analysis that determine loop iteration counts and variable values by examining the loop body exactly once. If these advanced techniques succeed in computing loop bounds, classical abstract interpretation is omitted for this loop, leading to an accelerated analysis. Otherwise, classical abstract interpretation needs to be applied.

5.3 Interprocedural program slicing

Program slicing (Weiser 1979) is a static analysis that finds statements of a program that are relevant for a particular computation, defined by the *slicing criterion*. A slicing criterion is a pair $\langle q, V\rangle$ where q is a program point and V is a subset of program variables at q. The slice w.r.t. $\langle q, V\rangle$ is a subset of the program with all statements that might affect the variables in V, i.e., variables that might either be used or defined at q.

WCC's loop analysis uses loop exit conditions as slicing criterion. By taking all relevant data and control dependencies into account, the resulting program slice contains all statements that are relevant to determine loop iteration counts. Slicing is supported by a context-sensitive pointer alias analysis. Contexts introduce a distinction between different calls to a given function, enabling a more precise analysis.

Slicing is run before the actual loop analysis for two reasons. First, it accelerates loop analysis (Sandberg et al. 2006), since slicing the code in advance strips all superfluous statements. Considering the relevant subset of the program, the fixed-point iteration during abstract interpretation usually finds a solution in less time. Second, the innovative polyhedral loop evaluation (cf. Sect. 5.4) requires simple loop bodies to infer final abstract states without repetitive iterations. Bodies of original loops are often too complex for this static evaluation but after slicing, the required prerequisites are met.

5.4 Polyhedral loop evaluation

A *polyhedron* P is an N-dimensional geometrical object defined as a set of linear inequations: $P := \{x \in \mathbb{Z}^N \mid Ax = a, Bx \geq b\}$ for $A, B \in \mathbb{Z}^{m\times N}$ and $a, b \in \mathbb{Z}^m$ and $m \in \mathbb{N}$. A polyhedron is called a *polytope* if $\|P\| < \infty$. Polytopes are often employed in compiler optimizations to represent loop nests and affine condition expressions. Their formal definition enables efficient code transformations. Typical fields of application are program execution parallelization or the optimization of nested loops (Falk and Marwedel 2003).

WCC's polyhedral loop evaluation is motivated by the observation that a large number of loops consists of statements not affecting the calculation of loop iterations. Typical examples are initialization procedures found in many embedded applications. The main task of such procedures is to initialize arrays and other data structures. Afterwards, this initialized data is used to compute output data, e.g., an output stream of an image compression algorithm, but it is not influencing the execution frequency of loops. Slicing recognizes those meaningless statements for loop analysis and does not evaluate them further. This often results in loops with almost empty loop bodies.

Loops to be analyzed by the polyhedral evaluation must meet certain constraints that specify the structure of the loop and the type of statements in the loop body. The first class of requirements concerns the structure of loops including their conditional statements, e.g., *if*-statements. These restrictions are imposed by the polytope models and their violation would make a polytope evaluation infeasible. The requirements concern loop exit conditions which must either depend on a constant or a program variable which is not modified within the loop body. Moreover, it must be ensured that all condition statements are affine expressions. It should be noted that these constraints are often met by well-structured loops found in many applications, thus they do not inhibit a successful application of WCC's non-iterative loop evaluation. The second class of constraints refers to the loop body statements. Assignment expressions in a sliced loop body need to be transformable to the ANSI C assignment operators $=$, $+=$ or $-=$ with variables or constants as right-hand-sides of the assignments.

If the conditions are met, the loop iteration counts required for the evaluation of statements are statically determined in the next step. Results from this phase allow a fast static evaluation of statements in a single step without the need to analyze the statements iteratively. The problem of finding the loop iteration counts is equivalent to computing the number of integer points in a (parametric) polytope. To efficiently count the integer points, *Ehrhart* polynomials (Verdoolaege et al. 2004) are used.

Considering all integer points of a polytope might yield an over-approximation. The total number of integer points represents the number of loop iterations if the loop counter is incremented by one, thus other modifications to the counter must be adequately modeled. Also, additional exit edges that affect the control flow in the loop body, e.g., in the case of *break* or *continue* statements, must be taken into account. They are modeled as further polytopes and their intersection with the polytope representing the loop nest yields the precise solution space. For some loops found in real-life benchmarks having an empty loop body after program slicing, counting of integer points is already sufficient to determine the loop iteration counts statically.

Using these loop iteration counts, execution frequencies of condition-dependent basic blocks, which might obviously differ from the loop iteration counts, are computed. The conditions are modeled by polytopes and an intersection with the loop polytope allows to compute execution frequencies for both the `then` and `else`-part.

The last step is the static evaluation of statements within the loop based on the loop iteration counts and basic block execution frequencies from the previous step. The goal is to evaluate modifications of variables within the loop like `b+=a` without a repetitive abstract interpretation. These final variable values are used to determine iteration counts for loops that are analyzed afterwards.

WCC's loop analysis can be used in two different ways. It can either be used as a stand-alone tool that produces loop information in a human-readable form, or as a module integrated into WCC. In the latter case, the loop analyzer automatically generates flow facts and passes them to the flow fact manager relieving the user from manually annotating flow facts. Figure 4 (cf. p. 256) depicts the analyzer's integration into WCC. It operates on the ICD-C IR and starts with slicing that marks statements relevant for loop bound computations. After that, loop analysis using the modified abstract interpretation and polytope models is done. At this point, the loop bound information for the program under analysis can be generated.

5.5 Results

To show the efficacy of our static loop analyzer, a total of 96 benchmarks was extensively analyzed and evaluated. The benchmarks come from the test suites MRTC, DSPstone, MiBench, MediaBench, UTDSP, and our own set of real-life benchmarks. The different types of the suites were chosen to point out that our loop analysis can successfully handle applications of different domains. All measurements were performed on a single core of an Intel Xeon CPU with 2.40 GHz and 8 GB RAM. For the sake of clarity, we provide a comprehensive overview of the results and discuss more interesting cases in more detail in the following.

5.5.1 Determination of loop iteration counts

Table 1 presents the evaluation of the loop analysis precision. The table shows for each benchmark suite the number of benchmarks, the number of contained loops, the percentage of loops that were successfully analyzed (column *Analyzable*) and the percentage of loops for which our loop analysis produces exact non-over-approximated results (column *Exact*). All percentages of Table 1 relate to column *Loops*.

The 96 benchmarks contain 707 loops in total. On average, 99% of those loops could be successfully analyzed. This means that for those loops, loop analysis produced safe results in terms of loop iteration counts which are never under-approximated, but might be over-approximated. The small fraction of 1% loops that could not be analyzed is mainly due to technical restrictions of our alias analysis.

The last column of Table 1 shows the percentage of loops for which exact iteration counts were computed. On average, our analysis produced exact results for 96% of the loops. The remaining 4%, including the non-analyzable 1% of loops of the previous column, could not be exactly analyzed, i.e., the loop iteration counts were afflicted with an over-approximation. The main reason for the imprecision comes from the analysis of pointers which can not always be precisely evaluated in a static analysis. However, most of the over-approximations introduced only a marginal error ranging between 8% and 51% w.r.t. the exact results. Thus, the results are still acceptable.

Slicing was successfully applied to all benchmarks. The number of statements irrelevant for loop analysis ranges from 2% to 88%, showing that computations in

Table 1 Precision of loop analysis

Benchmark suite	# Benchmarks	# Loops	Analyzable	Exact
MRTC	32	152	100%	99%
DSPStone	37	152	98%	93%
MediaBench/MiBench	6	162	99%	98%
UTDSP	14	88	100%	88%
Misc.	7	153	100%	100%
Total/Average	96	707	99%	96%

Table 2 Run times of loop analysis

Benchmark	Benchmark suite	Basic	Slicing	Polytope
`matmul`	MRTC	8.4 s	2.4 s(28%)	0.8 s (1%)
`hamming`	Misc.	0.4 s	0.3 s (80%)	0.2 s (62%)
`g721`	DSPstone	80.2 s	70.5 s (88%)	71.3 s (89%)
`fft`	DSPstone	920.7 s	119.7 s (13%)	110.5 s (12%)
`matrix1`	DSPstone	0.8 s	0.09 s (12%)	0.03 s (4%)
`mult_10_10`	UTDSP	4.6 s	3.6 s (78%)	3.7 s (80%)

many programs do not affect loop iteration counts. 21% of the loops were analyzable via the innovative polytope-based loop evaluation, which shows that the prerequisites of this polyhedral evaluation are not too restrictive and are often met in real-life code.

5.5.2 Analysis time

Besides the precision of the analysis, the second crucial issue for static program analyses is their complexity expressed in terms of analysis time. In general, the analysis times highly depend on the program structure and the loop iteration counts. If our polyhedral loop evaluation can not be applied, the analysis based on abstract interpretation must consider each loop iteration separately. On average, smaller benchmarks require a few seconds for the analysis, while the analysis time for larger benchmarks such as MiBench's *GSM encoder* takes on average less than 4 minutes.

The impact of the different techniques on the analysis run time of some example benchmarks is shown in Table 2. Column *Basic* represents the absolute run time of the basic loop analysis based on abstract interpretation. The fourth column (*Slicing*) depicts the analysis run time after program slicing, while the last column (*Polytope*) indicates the measured run times after the application of the polytope-based fast loop evaluation (including slicing). In addition, values in parentheses found in the fourth and fifth column represent the relative run times w.r.t. the third column.

Table 2 shows that slicing significantly decreases analysis times. For `matmul`, a reduction of 72% was achieved. `matmul` also benefits from the polytope approach. It contains some loops that can be statically evaluated using the polyhedral model that leads to a further reduction in time of 27%. For other benchmarks like `mult_10_10`, slicing reduces the analysis time by 22%. For `mult_10_10` the test whether the polytope approach can be applied was negative, thus slightly increasing the analysis time by 2% and forcing the analysis to switch back to the basic (iterative) loop evaluation.

Considering all 96 evaluated benchmarks, 38 benchmarks benefit from program slicing leading to a decreased analysis time. For 13 of these benchmarks, the analysis time could be further improved by switching from the iterative approach based on abstract interpretation to the polytope-based non-iterative approach.

6 Back-annotation of WCET data

WCC's infrastructure described so far allows the effective WCET reduction by optimizations applied at ICD-LLIR level where WCET estimates are imported from aiT and made accessible to the compiler. Still, high-level WCET-aware optimizations that take place at the source code level are not yet supported due to the lack of WCET timing information at the level of the ICD-C IR. However, high-level optimizations that focus on function call and loop transformations exhibit a large potential for WCET reduction. Thus, a WCET model for ICD-C is highly desired. To transform WCET timing data from assembly to the source code level, a bridge between both abstraction levels of the code is required, which is realized by WCC's *back-annotation*.

6.1 Mapping of low-level to high-level structures

To raise the abstraction level of the WCET timing model from assembly to source code level, a connection between ICD-LLIR and ICD-C must be established. Mapping between coarse-grained objects, e.g., compilation units (source code files) and functions, is trivial. Each ICD-C compilation unit has a unique file name and its translation into machine code results in one ICD-LLIR compilation unit. This relationship is exploited and mapping between compilation units of the two IRs is done with the file name as a key. Mapping between functions of both abstraction levels is achieved using the unique function names as key. Care needs to be taken only for functions that have *static* storage in the sense of ANSI C. Since several static functions with the same name may exist, both function and file name is used as mapping key.

Mapping of basic blocks from ICD-LLIR to ICD-C is more complicated since a 1:1 mapping does not always exist. By definition, a basic block is a code fragment with a single entry and exit point where jumps can only occur at the block's end. Function calls, which implicitly modify the control flow, can be handled in two different ways. They can either represent a basic block boundary, i.e., a new basic block begins after a function call, or they are considered as regular statements/instructions that do not explicitly modify the control flow. The former definition is used within ICD-LLIR, while the latter is used by ICD-C. Due to the varying definitions and assembly-level optimizations that modify the basic block structure, the relationship of basic blocks represents an *n*:*m* mapping in general. The following relationships between assembly- and source-code basic blocks (*ICD-LLIR*:*ICD-C*) may occur:

1:1 Relation: Sequential code with no control flow modification as shown on the right is represented in both IRs as a single basic block, thus a mapping is again obvious. For all ICD-LLIR basic blocks for which such a bijective 1:1 relation holds, a mapping to ICD-C basic blocks can be achieved using the block label as key.

```
{
  c += 2;
  b += c;
  res = a + b;
}
```

***n*:1 Relation:** A source code fragment with a function call as depicted right is represented by a single ICD-C basic block. It corresponds to two ICD-LLIR basic blocks due to the call of `foo`. Similar *n*:1 situations occur in the presence of the logical AND (`&&`), OR (`||`), and conditional (`?`) operators of ANSI C since they implicitly modify the control flow. They are typically used in complex conditions with multiple comparisons which are covered by a single ICD-C block. In contrast, each comparison is represented in ICD-LLIR by an individual basic block. Thus, mapping of ICD-LLIR basic blocks to ICD-C blocks becomes surjective for *n*:1 relations. Using basic block labels, several ICD-LLIR blocks are mapped to a single ICD-C block.

```
{
  a = a / 2;
  a += 100;
  a = foo( a );
  return a;
}
```

1:*m* Relation: For the code shown right, two ICD-C blocks represent the loop body and the exit condition. This loop is modeled by only one ICD-LLIR block since the computations of the loop body, the test of the exit condition and the conditional jump back to the loop header is a sequence of code without any control flow modifications in between.

```
do {
  a--;
  sum += a;
} while ( a > 0 );
```

WCET data of ICD-LLIR blocks with a 1:m relation must not be attached to all m ICD-C blocks. Otherwise, the m-fold storage of equal WCET data in ICD-C would falsify the ICD-C timing model and lead to a global WCET of a program that is larger than that computed by aiT. Thus, WCET data is attached to only one of the m ICD-C blocks, called the reference block. The remaining $m-1$ ICD-C blocks simply point to that reference block to enable forwarding of requests of back-annotation data.

Since WCC's code selector is the interface between the source- and assembly-level IRs, it also determines the relationships between ICD-C and ICD-LLIR blocks and the corresponding mappings. We extended all WCC optimizations applied after code selection to automatically update all mappings when modifying ICD-LLIR blocks. The integration of the back-annotation is depicted in Fig. 4 (cf. p. 256).

6.2 Transformed data during back-annotation

After establishing the connection between assembly- and source-level basic blocks using the mappings presented above, WCET timing data attached to ICD-LLIR basic blocks can be back-annotated to ICD-C. In addition, information from the compiler back-end is imported. The following data is transformed during back-annotation:

- WCET for the entire program, functions, and basic blocks
- Information whether an ICD-C block lies on the WCEP
- Worst-case execution frequency per CFG edge
- Execution feasibility of blocks and CFG edges
- Number of I-cache misses per basic block encountered during WCET analysis
- Code size and amount of spill code per assembly-level basic block

After back-annotation, detailed WCET timing data is present in ICD-C. To show the effectiveness of this mechanism, this data is used by WCC's WCET-aware, high-level optimizations *Procedure Cloning* and *Procedure Positioning* in the following.

7 WCET-aware procedure cloning

Procedure cloning is a standard optimization of functions that are often called with constants as arguments. If the caller invokes a callee with constant arguments, the callee can be cloned, the constant parameters are removed from the parameter list and are instead imported into the clone. This is beneficial for two reasons. First, it may enable further optimizations like e.g., constant propagation or folding in the clone. Second, calling overhead is reduced since the constant parameters are no longer passed between caller and callee. WCC applies cloning at source code level, early in the optimization process, to enable potential for a large number of following optimizations. In addition, changing function calls and parameters is easier at this abstraction level.

In contrast to previous work, the impact of procedure cloning on the WCET of embedded real-time applications was studied for the first time using WCC (Lokuciejewski et al. 2007, 2008).

7.1 Impact of procedure cloning on WCET estimation

Typical embedded real-time source codes often contain loops l whose iteration counts depend on a parameter p of the function f that surrounds this loop. In addition, such a function f can be called from various places, with different values for p. This code structure has a negative impact on the WCET computed by a timing analyzer.

This is due to the flow fact annotation of such loops l. Since f is called from many places with possibly different arguments p, the effective iteration counts of l can vary seriously, depending on the context with which parameters f is called. Many WCET analyzers apply context-sensitive analyses that take context information of each call into account. If a data-dependent loop can be statically analyzed, information on constant parameter values is used to compute precise WCET data for each individual context of l. However, real-life loops are often too complex to be analyzed at assembly level. Thus, this loop analysis only succeeds for a limited class of loops.

For this reason, most real-life loops require flow fact specification either manually by the user or by WCC's loop analyzer. The flow facts for such data-dependent loops must cover all possible contexts in which the loop may be executed in order to result in safe WCET estimates. Hence, the upper bound of such flow facts must represent the global maximum of iterations executed by such a loop over all contexts in which f is called, equivalently the same holds for the lower bound. Since such flow facts for data-dependent loops do not consider possible different execution contexts of a function f, the flow facts are safe but lead to a highly overestimated WCET.

WCC applies WCET-aware cloning if a caller invokes a callee f that has data-dependent loops l, and if the iterations of l depend on a parameter p of f that is constant. Cloning creates a specialized version f' of f that has constant loop bounds w.r.t. p. The data-dependence of l is broken by cloning so that highly precise flow facts for f' result. Hence, cloning is a way to express different calling contexts at the source code level that eliminates the need to maintain context-sensitive data (cf. Sect. 3.4). It thus enables high-precision WCET analysis of such clones. Consider the following code before cloning and its annotated flow facts (cf. Sect. 4):

```
int f( int *x, int n, int p ) {               int main() {
  _Pragma( "loopbound min 2 max 2000" )         ... f( y, 2000, 5 ); ...
  for ( i = 0; i < n; ++i ) {                   ... f( z, 2, 5 );
    x[i] = p * x[i];                            return f( a, 2, 5 );
    if ( i == 10 ) { ... }                    }
  }
  return x[n];
}
```

`f` contains a loop that depends on the function parameter `n`. Within `main`, `f` is called three times, once with `n` = 2,000 and twice with `n` = 2. To obtain safe WCET estimates, `f`'s loop must be annotated with a minimum of 2 and a maximum of 2,000 iterations. A WCET analyzer has to compute safe results, thus 2,000 iterations are assumed in the worst case for each execution of this loop. For each call of `f` with `n` equal 2, a significant overestimation is introduced leading to imprecise WCET estimates.

The application of our WCET-aware procedure cloning transforms this code snippet into a code that is better accessible for high-precision WCET analyses:

```
int f( int *x, int n, int p ) {        int f_2_5( int *x ) {
  _Pragma( "loopbound min 2000 \         _Pragma( "loopbound min 2 \
                      max 2000" )                            max 2" )
  for ( i = 0; i < n; ++i ) {              for ( i = 0; i < 2; ++i ) {
    x[i] = p * x[i];                         x[i] = 5 * x[i];
    if ( i == 10 ) { ... }                   if ( i == 10 ) { ... }
  }                                        }
  return x[n];                             return x[2];
}                                        }

int main() {
  ... f( y, 2000, 5 ); ...
  ... f_2_5( z );
  return f_2_5( a );
}
```

Cloning yields more precise loop bound annotations for the original function `f`. In addition, a specialized version `f_2_5` of `f` for the values 2 and 5 of parameters `n` and `p` is created. The loop in `f_2_5` is no longer data-dependent. Hence, precise flow facts now state that the loop iterates exactly twice. This way, WCC's cloning helps to produce high-quality flow facts for static WCET analysis, which considerably improves the tightness of WCET estimates. This optimization is WCET-aware since it is only applied to those functions that enable a more precise WCET analysis of loops in order to keep code size increases resulting from cloning small. Moreover, functions are sorted in advance by their WCETs to clone those functions first that promise the largest WCET reduction. The required data is provided by WCC's back-annotation.

7.2 Results

Figure 6 shows the impact of procedure cloning on the WCET estimates of three complex real-life benchmarks. The 100% base line denotes the WCET estimates of the original code optimized using constant folding, constant propagation and dead

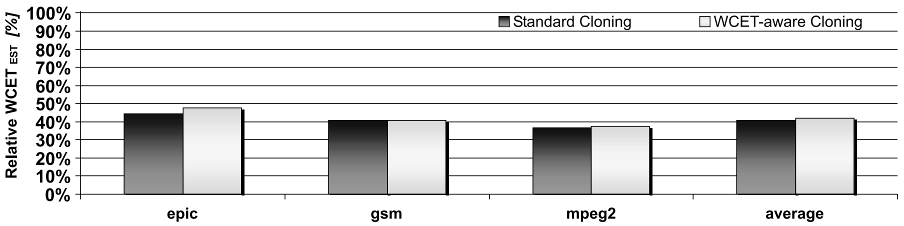


Fig. 6 Relative WCETs after procedure cloning

code elimination. These optimizations are applied to simplify the code structure and to remove dead code which can be detected even without cloning. The dark bars represent the WCET when the code is additionally optimized by standard cloning, while the light bars show results for WCET-aware procedure cloning.

As can be seen, standard and WCET-aware cloning reduce WCET estimates comparably. Maximal reductions were observed for `mpeg2` that contains two functions that were cloned. The first one realizes a full-search motion detection. In this function, another procedure is called that computes the distance between frame blocks. After cloning, the code contains a dedicated version of the Fullsearch algorithm for each block size. The loop bounds in the nested functions can again be defined more precisely. For standard and WCET-aware cloning, WCET estimates are reduced by 63.6% and 62.7%, resp. For other benchmarks, similar improvements were achieved.

On average, standard cloning outperforms WCET-aware cloning merely by 1.3%. This is due to the fact that it clones more functions compared to WCET-aware cloning since also functions that do not allow a more detailed specification of flow facts are transformed. These clones can be further optimized by other standard optimizations.

However, this extensive cloning significantly increases code sizes. `epic` exhibits the maximal code size increases by 693.7% after standard cloning. Since WCET-aware cloning only transforms functions that promise a WCET reduction, it increases code size by only 357.9%. This still high increase is due to a very uncommon structure of `epic`—it contains 32 functions that were cloned many times. Unlike this extreme case, the code size of `gsm` remained almost unchanged, while WCET-aware cloning increased the code size of `mpeg2` by 127.0%. It can be seen that cloning represents a trade-off between WCET reduction and code expansion. To cope with the risk of an undesirable code size increase, the new optimization provides a parameter to control the maximally permitted code expansion during the transformation.

8 WCET-aware procedure positioning

Procedure Positioning aims to improve I-cache behavior by reducing the number of cache conflict misses. Caches reduce the average memory access time by exploiting spatial and temporal locality. The former refers to the reference of contiguous memory locations. The latter means that particular memory locations are accessed within a short period of time. Due to an inappropriate layout of a piece of code in memory, temporal locality may, however, degrade cache performance, if memory locations being accessed temporally close to each other are mapped to the same cache lines. This leads to an eviction of cache contents and repetitive cache refills. Procedure positioning uses call frequencies to reorder functions and reduce cache thrashing. This section gives an overview of our approach for WCET-aware procedure positioning (Lokuciejewski et al. 2008).

8.1 Related work

I-caches mainly profit from a code reorganization at procedure and basic block level. Tomiyama and Yasuura (1997) propose two code placement methods for basic blocks to reduce the cache miss rate using ILP. Hwu and Chang (1989) propose a compiler with an integrated instruction placement algorithm that reduces page faults. In Lebeck and Wood (1994), a cache profiling system identifies hot spots by providing cache performance information at source code level. After an automatic classification into compulsory, capacity and conflict misses, the profiler suggests appropriate standard program transformations to improve cache performance. The work of Mendlson et al. (1994) does not use profiling data but static information and additionally, in contrast to the previously cited works, requires the exact knowledge of the cache architecture. Their idea is to prevent different segments of code executed in a loop to be mapped into the same cache area by code replication.

Static cache analysis is essential for a WCET analyzer for cache-based processors. Its goal is to classify each memory access as a cache hit or miss. Ferdinand uses *must* and *may analysis* based on abstract interpretation (Ferdinand et al. 2001). The former determines if a memory access is always a cache hit while the latter computes if the access may be a hit. This approach is also used in aiT, the WCET analyzer integrated into WCC.

8.2 WCET-centric call graph-based positioning

Procedure Positioning uses a call graph whose set of nodes represents program procedures. Edges denote calling relationships between procedures and are weighted with call frequencies which, for ACET optimization, are gained using profiling.

In contrast to the standard, profiling-based optimizations, we extract input data for the call graph from a WCET analyzer. This fundamental difference makes our approach more reliable. Profiling data is critical since it reflects the program execution for a particular set of input data, i.e., profiling the program under test with varying inputs may yield different results. For more complex programs that consist of numerous input-dependent execution paths, it is almost infeasible to find representative input values. This may lead to a call graph that is annotated with profiling data that does not represent some particular program executions. The optimized code will possibly not improve cache behavior and may even suffer performance degradation.

Our approach does not rely on representative input data. Edge weights are computed by a WCET analyzer and are invariant for all program executions. They are used for the construction of our WCET-centric call graph. Those edges with the heaviest weight potentially combine the most promising functions for optimization. These functions are reordered and are placed next to each other in the ICD-C IR. In the next step, the code selector processes the IR function-wise, i.e., each source code function is translated into an equivalent assembly function while preserving the order of the functions. This function order in the ICD-LLIR yields the desired memory layout.

8.3 Greedy WCET-aware positioning approach

It is well-known that the impact of a memory layout modification on caches is hardly predictable. Therefore, a greedy approach that evaluates the impact of a particular procedure rearrangement on the WCET seems promising. In case a WCET reduction was achieved, this novel memory layout is considered as a new starting point for the next optimization cycle, and the next most promising function for positioning is considered. Hence, the approach successively reduces the WCET and guarantees that no degradation of the WCET is accepted. The greedy approach is an iterative algorithm that processes a single edge of the WCET-centric call graph during each iteration cycle. Each cycle performs a WCET analysis to update the WCET timing data to be used during the next iteration and to keep track of possible WCEP changes.

8.4 Heuristic WCET-aware positioning approach

Due to the possibly large number of time-consuming WCET analyses of the greedy approach, a fast heuristic was developed that just uses the data of the WCET-centric call graph for the initial input program. In contrast to the greedy approach, the heuristic performs exactly one WCET analysis to construct the initial call graph.

The speed advantages come at the cost of efficacy. First, the reordering of procedures is based exclusively on the initial call graph and is performed without re-evaluating its impact on the WCET. Hence, also undesired WCET increases are accepted. Second, WCEP switches are not considered. Since the call graph is not updated, the heuristic approach operates on an outdated WCET-centric call graph if the WCEP changes. The applied positionings would then possibly not affect the WCET.

8.5 Results

Figure 7 shows the results for greedy and heuristic procedure positioning. 100% correspond to the WCET estimates of the original code of different real-life benchmarks from the MRTC and MediaBench suites compiled with optimization level *-O3*. It can be seen that a WCET reduction was achieved for most benchmarks. The greedy algorithm achieved an average WCET reduction by 10%, while the heuristic reduced the WCET by 4% on average.

The results strongly depend on the initial order of the benchmarks' procedures. If the original memory layout already yields a good cache performance, positioning might lead to smaller improvements than for benchmarks that lead to more cache conflict misses. Moreover, tiny benchmarks whose text section is small enough to fit entirely into the cache (e.g., `expint`) do not profit from this optimization since no conflict misses can occur. However, applications that completely fit into the (usually) small I-cache of a resource-restricted embedded system are very uncommon.

For all benchmarks, greedy positioning achieved better results since it does not allow a degradation of the WCET. This might result in a local optimum missing the

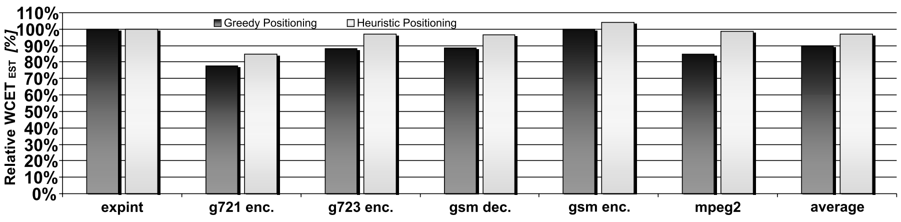


Fig. 7 Relative WCETs after greedy and heuristic procedure positioning

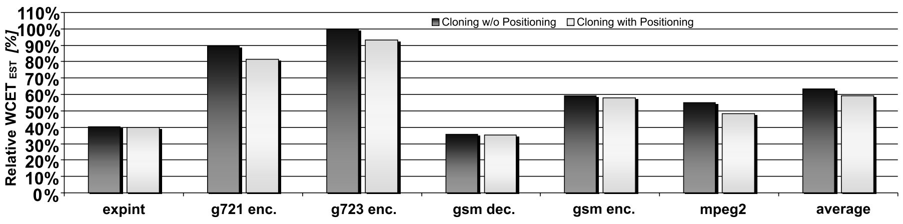


Fig. 8 Relative WCETs after positioning and cloning

global minimum as could be potentially achieved by the heuristic approach. However, for the considered benchmarks, this case did not arise. The heuristic approach might worsen the WCET as experienced for `gsm enc`. Hence, it is worthwhile to invest time for the optimization to achieve best results, as done by our greedy approach.

In addition, we combined WCET-aware procedure cloning (cf. Sect. 7) with procedure positioning. The results of Fig. 6 show WCETs when cloning is done for a system with disabled caches. Any newly created function clone was placed behind the last function in the code with no regard to cache effects. Figure 8 shows results for a combination of procedure cloning and WCET-aware procedure positioning particularly for cache-based systems. 100% correspond to the WCETs of the benchmarks in their unoptimized versions. Combined procedure cloning and procedure positioning achieves WCET reductions of up to 64%. These results allow two conclusions.

First, procedure cloning and positioning are best suited in a cache-based system. Although inserting clones increases code size, the benefits of the improved WCETs exceed the drawbacks that may emerge from more cache conflict misses due to the increased working set. Second, combined cloning and positioning achieves better results for most benchmarks. Both techniques in concert aim at compensating conflict misses due to the function clones. Obviously, the achieved benefits are smaller than those of Fig. 7 since positioning was only applied to the clones instead of all functions in the program. However, the complexity of positioning is negligible so that it should be applied to all functions in combination with cloning, for even better results.

Unlike the WCET, procedure positioning does not influence the code size since the applied re-allocations of functions do not require any additional code.

9 WCET-aware scratchpad allocation of program code

Caches are problematic for hard real-time systems. Due to their hardware control, it is difficult to predict memory access latencies for many popular cache architectures, and statically analyzed WCET estimates may be heavily overestimated. Even techniques like e.g., procedure positioning (cf. Sect. 8) do not eliminate this inherent problem. Thus, designers of safety-critical real-time systems often disable caches, which leads to a low average-case performance since each memory access is served by the slow main memory. *Scratchpad memories* (*SPMs*) have both a good average- and worst-case performance. This section presents a WCET-aware static SPM allocation of program code, where the scratchpad contents is pre-computed at compile time and remains unchanged during run time. Due to the use of ILP, our SPM allocations are optimal and result in a minimal WCET for architectures without I-caches.

9.1 Related work

Compiler-guided SPM allocation to reduce ACET or energy dissipation has been studied intensely in the past. Due to the vast amount of related literature, we only refer to Wehmeyer and Marwedel (2006) and Verma and Marwedel (2007) where various ILP-based approaches for SPM allocation are presented. This section thus lists only contributions related to worst-case execution times.

In Wehmeyer and Marwedel (2005), the impact of SPMs on WCET prediction is studied. Based on an ILP for energy minimizing SPM allocation, the effect of this energy reduction strategy on WCET is evaluated. Even though significant WCET reductions were reported, that work is not a true WCET-aware optimization and does not consider WCEPs at all.

Software controlled caches that allow to lock loaded cache lines so that they are not evicted, behave like SPMs. In Campoy et al. (2005), a genetic algorithm for static I-cache locking is used. However, this approach does not necessarily yield optimal results. An explicit search for the CFG's WCEP is performed in Falk et al. (2007) and the I-cache is locked along the found WCEP. This approach repeatedly examines the CFG and is thus expensive to perform. In contrast, Puaut (2006) applies multiple optimization steps along the current WCEP without recomputing the WCEP. After a couple of optimization steps, the partially optimized CFG is analyzed, the WCEP is updated and optimization resumes.

In Suhendra et al. (2005), a fully ILP-based solution for static SPM allocation of data that reduces WCET is presented. It serves as basis for WCC's SPM allocations. However, it does not allocate code onto SPMs and suffers from several limitations that prevent it from optimizing real-life programs. This literature study shows that no unified ILP-based SPM allocation for program code currently exists which is able to reduce WCET.

9.2 Structure of the ILP for program code scratchpad allocation

Section 9.2.1 presents the ILP model of a function's control flow. Section 9.2.2 extends the ILP to allocate consecutive blocks. A program's global control flow, capacity constraints and objective function are subject of Sects. 9.2.3 to 9.2.5, resp.

9.2.1 ILP constraints modeling the control flow of a function

In the following, that part of our ILP for SPM allocation of program code is presented that moves individual basic blocks in their entirety onto the SPM. This is done under simultaneous consideration of possibly switching WCEPs by formulating ILP constraints that inherently model the longest path which starts at a certain basic block.

The following equations represent ILP variables using lowercase letters whereas constants use uppercase letters. The ILP uses one binary decision variable x_i per basic block b_i of a program (cf. (1)). x_i states if b_i is allocated to main memory

(mem_{main}) or to the SPM (mem_{spm}). A block b_i of a function F causes some costs c_i (cf. (2)), i.e., b_i's WCET depending on whether b_i is allocated to main memory or to the SPM.

$$x_i = \begin{cases} 1 & \text{if basic block } b_i \text{ is assigned to } mem_{spm} \\ 0 & \text{if basic block } b_i \text{ is assigned to } mem_{main} \end{cases} \tag{1}$$

$$c_i = C^i_{main} * (1 - x_i) + C^i_{spm} * x_i \tag{2}$$

$$w^L_{exit} = c^L_{exit} \tag{3}$$

$$\forall b_i \in V \setminus \{b^L_{exit}\} : \forall (b_i, b_{succ}) \in E : w_i \geq w_{succ} + c_i \tag{4}$$

$$c_L = w^L_{entry} * C^L_{max} \tag{5}$$

For reducible CFGs, an innermost loop L of F has exactly one back-edge that turns it into a cyclic graph. Not considering this back-edge turns L's CFG into an acyclic graph. This acyclic graph without L's back-edge is denoted as $G_L = (V, E)$ in the following. Each node of G_L represents a single basic block. Without loss of generality, there is exactly one unique exit basic block b^L_{exit} of loop L in G_L and one unique entry node b^L_{entry}. The WCET w^L_{exit} of b^L_{exit} is set to the costs of b^L_{exit} (cf. (3)). The WCET of a path from a node b_i (different from b^L_{exit}) to b^L_{exit} must be greater or equal than the WCET of any successor of b_i in G_L, plus b_i's costs (cf. (4)).

Variable w^L_{entry} thus represents the WCET of all paths of the innermost loop L if L is executed exactly once. To model several executions of L, all CFG nodes $v \in V$ of G_L are merged to a new super-node v_L. The costs of v_L are equal to L's WCET if executed once, multiplied by L's maximal loop iteration count C^L_{max} (cf. (5)).

Replacing a loop L by its super-node v_L in a function's CFG may turn another loop L' of function F that immediately surrounds L into an innermost loop with acyclic CFG G'_L. Hence, (3)–(5) can be formulated analogously for L'. This way, the innermost loops of F are successively collapsed in the CFG so that ILP constraints that model F's control flow are created from the innermost to the outermost loops.

A program's WCEP can switch during optimization only at a basic block b_i that has more than one successor because only there, forks in the control flow are possible. Since (4) is created for each successor of b_i, variable w_i always reflects the WCET of any path starting from b_i—irrespective of which of the successors actually lies on the current WCEP. This way, (4) realizes the implicit consideration of (switching) WCEPs in the ILP. The structure of the ILP constraints of (2)–(5) was originally proposed by Suhendra et al. (2005). However, these basic constraints of Suhendra et al. need to be refined substantially to obtain a functional SPM allocation for program code. The following sections describe our extensions to the original ILP formulation.

9.2.2 *ILP constraints allocating consecutive basic blocks*

The variables x_i allow to place a block b_i in the SPM independent of the allocation of any other block b_j. This independence is particularly problematic for typical processors. If a block b_i is allocated to main memory and an immediate successor b_j of b_i is placed on the SPM, jumps must ensure that b_i still reaches b_j. Due to the limited offset that can be encoded as target of jump operations, and due to the usually

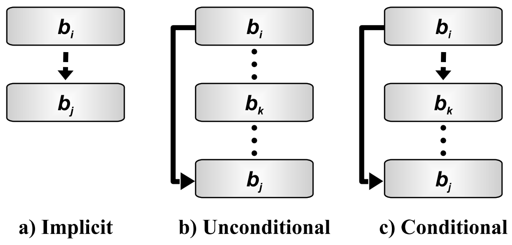


Fig. 9 Typical jump scenarios

too large distance between the address spaces of SPM and main memory, a single jump operation is often insufficient to jump from b_i to b_j. Instead, the jump target address must be computed and stored in an address register so that a register-indirect jump can be used. Thus, branching from b_i to b_j may require several machine operations which constitute a severe jumping overhead. This overhead is avoided if both b_i and b_j are placed in the same memory. Thus, the ILP should consider this jumping overhead and allocate consecutive blocks to the same memory to reduce jumping overhead.

Processors usually support different *jump scenarios* (cf. Fig. 9). An *implicit jump* transfers control from b_i to b_j without a jump operation in b_i. An *unconditional jump* always jumps from b_i to b_j. Finally, a *conditional jump* branches from b_i to either b_j unconditionally or to b_k implicitly.

The variables x_i, x_j and x_k for the basic blocks b_i, b_j and b_k, resp., provide the information whether jumping overhead needs to be considered within the ILP or not.

If a jump from b_i to b_j is implicit and b_i and b_j are placed in different memories, a penalty should be added since this jump across different memories leads to a large jumping overhead. In contrast, no penalty is added if both b_i and b_j lie in the same memory, because both blocks are allocated adjacently so that no jump is required. The jump penalty for implicit jumps from b_i to b_j is thus defined in (6). The operator $\otimes$ denotes the Boolean XOR of two binary variables. XOR can be modeled in an ILP, but we omit these constraints for the sake of brevity. P_{high} is a constant that penalizes jumps across different memories due to their large jumping overhead.

$$jp^i_{impl} = (x_i \otimes x_j) * P_{high} \tag{6}$$

$$\begin{aligned} jp^i_{uncond} &= (x_i \otimes x_j) * P_{high} \\ &\quad + \overline{(x_i \otimes x_j)} * \left(1 - \prod_{b_k \in \text{Fig. 9b)}} (x_i \otimes x_k)\right) * P_{low} \end{aligned} \tag{7}$$

$$\begin{aligned} jp^i_{cond} &= (x_i \otimes x_k) * P_{high} + (x_i \otimes x_j) * P_{high} \\ &\quad + \overline{(x_i \otimes x_j)} * \left(1 - \prod_{b_k \in \text{Fig. 9c)}} (x_i \otimes x_k)\right) * P_{low} \end{aligned} \tag{8}$$

$$jp_i = \begin{cases} jp^i_{impl} & \text{if jump of } b_i \text{ is } \textit{implicit} \\ jp^i_{uncond} & \text{if jump of } b_i \text{ is } \textit{unconditional} \\ jp^i_{cond} & \text{if jump of } b_i \text{ is } \textit{conditional} \\ 0 & \text{else} \end{cases} \tag{9}$$

$$w_{exit}^L = c_{exit}^L + jp_{exit}^L \tag{10}$$

$$\forall b_i \in V \setminus \{b_{exit}^L\} : \forall (b_i, b_{succ}) \in E : w_i \geq w_{succ} + c_i + jp_i \tag{11}$$

An unconditional jump from b_i to b_j bypasses some other blocks b_k (cf. Fig. 9b) that must also be considered, since they state if a jump from b_i to b_j is needed at all. If b_i and b_j are placed in different memories, P_{high} is applied again. If they are placed in the same memory *mem*, and if no other block b_k that originally laid between b_i and b_j is allocated to *mem*, b_i and b_j are adjacent in *mem*. Thus, no jump from b_i to b_j is needed at all and no penalty is applied. If any b_k is placed between b_i and b_j in *mem*, an unconditional jump from b_i to b_j is needed that is penalized by P_{low} which is lower than P_{high}. The penalty for unconditional jumps is thus given in (7).

Since a conditional jump combines implicit and unconditional jumps (cf. Fig. 9c), its penalty is the combination of (6) and (7) (cf. (8)). Depending on b_i's jump scenario, the overall jump penalty jp_i is defined in (9). jp_i is added to the basic control flow constraints (cf. (3) and (4)) as defined in (10) and (11).

9.2.3 ILP constraints modeling the global control flow

Up to this point, the ILP of (1)–(11) only models the control flow of a single function F. Without loss of generality, each function F has one dedicated entry block b_{entry}^F. For b_{entry}^F, the ILP variable w_{entry}^F denotes the WCET of any path that starts at b_{entry}^F, assuming that F is called exactly once. However, a block b_i of a function F' may call a function F. Here, F's WCET (i.e., variable w_{entry}^F) has to be added to b_i's WCET. Also, a function call penalty is added to b_i's WCET since branching overhead in analogy to Sect. 9.2.2 occurs if b_i and b_{entry}^F reside in different memories. As a result, the overall function call penalty cp_i for a block b_i is defined in (12). cp_i is finally added to the control flow constraint of (11) as shown in (13).

$$cp_i = \begin{cases} w_{entry}^F + (x_i \otimes x_{entry}^F) * P_{high} & \text{if } b_i \text{ calls } F \\ \quad + \overline{(x_i \otimes x_{entry}^F)} * P_{low} & \\ 0 & \text{else} \end{cases} \tag{12}$$

$$\forall b_i \in V \setminus \{b_{exit}^L\} : \forall (b_i, b_{succ}) \in E : w_i \geq w_{succ} + c_i + jp_i + cp_i \tag{13}$$

$$s_i = \begin{cases} (x_i \wedge \overline{x_j}) * S_{impl} & \text{if jump of } b_i \text{ is } \textit{implicit} \\ (x_i \wedge \overline{x_j}) * S_{uncond} & \text{if jump of } b_i \text{ is } \textit{unconditional} \\ (x_i \wedge \overline{x_k}) * S_{impl} & \text{if jump of } b_i \text{ is } \textit{conditional} \\ \quad + \underline{(x_i \wedge \overline{x_j})} * S_{uncond} & \\ (x_i \wedge \overline{x_{entry}^F}) * S_{call} & \text{if } b_i \text{ calls } F \\ 0 & \text{else} \end{cases} \tag{14}$$

$$\sum_{b_i} (S_i * x_i + s_i) \leq S_{spm} \tag{15}$$

$$w_{entry}^{\mathtt{main}} \rightsquigarrow min. \tag{16}$$

Equation (13) is the constraint which is finally generated for our ILP per block b_i and per successor b_{succ} of b_i. Equation (13) assumes non-recursive functions. Due to the practical irrelevance of recursion for embedded real-time software (Engblom 1999), this assumption holds even if support of recursion could be added to the ILP.

9.2.4 Scratchpad capacity constraint

For a valid SPM allocation, the size of all blocks put on the SPM must not exceed the SPM's size. Previous work on ILP-based SPM allocation universally assumed a fixed block size so that it can be used as constant factor in the ILP. However, this is an over-simplification. Section 9.2.2 discussed that different jump operations are needed, depending on the jump scenario of a block b_i that jumps to b_j: no jump is needed if b_i and b_j are adjacently placed in the same memory. A conventional jump is needed if b_i and b_j are placed in the same memory, but not adjacently. Finally, complex address computations and register-indirect jumps realize jumps across different memories.

Obviously, these jump scenarios influence a block b_i's size so that it depends on the ILP's decision variables in practice. To cope with such variable block sizes, we fall back to the jump scenarios (cf. Fig. 9). In the ILP, we only consider b_i's size if b_i is put on the SPM since we assume a main memory large enough to hold the entire program. For a block b_i placed in the SPM, a new variable s_i denotes b_i's growth in size in bytes if b_i's successors b_j are kept in main memory. Depending on b_i's jump scenario, or if b_i contains a function call, s_i is computed as shown in (14).

For each jump scenario, dedicated constants S_{impl}, S_{uncond} and S_{call} are used that represent b_i's growth in bytes for the different jump scenarios. Using s_i, the SPM capacity constraint which ensures the validity of an SPM allocation is defined as shown in (15). In (15), S_i denotes b_i's byte size in its original form without any cross-memory jumps. S_{spm} represents the available SPM size in bytes.

9.2.5 Objective function

The ILP aims to minimize a program's WCET by assigning basic blocks to the SPM. Due to (12) and (13), variable w^F_{entry} models the WCET of function F which includes the WCETs of all functions called by F, plus some abstract jump penalties. Since function `main` is the unique entry point of an entire program, variable $w^{\texttt{main}}_{entry}$ denotes the WCET of a program including all penalties. As a consequence, the value of this decision variable needs to be minimized by the ILP as shown in (16).

This objective function seems surprising since WCET analysis usually relies on a maximizing ILP due to the *Implicit Path Enumeration Technique* (*IPET*) (Li and Malik 1995). It is a very general technique that models execution paths not as ordered sequences of basic blocks, but instead using the blocks' successor/predecessor relations and their respective execution counts. Due to this generality that arbitrary paths in the CFG can be taken, IPET searches the maximal flow through a CFG using a maximizing ILP.

In contrast, the ILP presented in this section assumes that loops are reducible and have the single-entry/single-exit property. Thus, we do not need to consider arbitrary CFG paths. Instead, this assumption allows to simply model paths that start at a certain node b_i and end at an exit node b^L_{exit} (cf. (4)). Since (4) uses the greater-equal operator to specify a safe upper bound of block b_i's WCET, our ILP can safely minimize its objective function, in contrast to traditional WCET analysis.

9.3 Implementation issues

Since the ILP for SPM allocation of program code relies on information about the size and jump scenario of each block, it is evident that this optimization is realized at assembly code level within WCC. WCC's infrastructure (cf. Sects. 2–5) is employed to turn the ILP of Sect. 9.2 into a fully functional optimization. In particular, it serves to extract all constants used by the ILP from the assembly code.

Equation (2) depends on constants C^i_{main} and C^i_{spm} that represent a block b_i's WCET if it is put either in main memory or in the SPM, resp. C^i_{main} is obtained within WCC by placing the whole program in main memory and performing a WCET analysis as described in Sect. 3. Afterwards, the whole program is virtually assigned to the SPM using WCC's memory hierarchy infrastructure (cf. Sect. 2). Another WCET analysis of this program yields the values C^i_{spm}. These two required WCET analyses are performed within WCC prior to solving the ILP for SPM allocation. Thus, WCET analyses and SPM allocation are decoupled from each other so that the seemingly antagonistic objective functions of their ILPs do not interfere with each other.

Equation (5) relies on a loop's maximal iteration count C^L_{max}. In our compiler, C^L_{max} can stem from user-specified flow facts or from our loop analyzer (cf. Sect. 5). Irrespective of its origin, flow fact mechanisms (cf. Sect. 4) keep C^L_{max} up to date during WCC's optimizations so that always correct values are used by our ILP.

The jump penalties P_{high} and P_{low} (cf. Sect. 9.2.2) do not rely on our compiler infrastructure. WCET analyses of code for the different jump scenarios revealed that the values 16 and 8 are appropriate for the considered TriCore architecture.

Equation (14) uses constants S_{impl}, S_{uncond} and S_{call} that model the size of the additional code required to jump from b_i to b_j if b_i is allocated to the SPM but b_j is not. Due to the TriCore-specific jumping code and the different jump scenarios of (14), S_{impl} and S_{uncond} equal to 10 bytes and S_{call} is equal to 12 bytes.

The size S_i (cf. (15)) of a basic block without consideration of a jump operation at the end of b_i can be computed easily by enumerating all instructions of b_i and accumulating their sizes. The totally available scratchpad size S_{spm}, however, is extracted again from WCC's memory hierarchy infrastructure (cf. Sect. 2).

After solving the ILP, the decision variables x_i specify where to place each block b_i. Using WCC's memory hierarchy API, the program code is finally transformed such that it adheres to the allocation decisions taken by the ILP. In addition, WCC emits a linker script to generate an executable that reflects the ILP's SPM allocation.

9.4 Evaluation

This section evaluates the ILP for WCET-aware SPM allocation of code. For benchmarking, WCC's optimization level *-O2* with 35 different optimizations was activated so that our SPM allocation was applied to already highly optimized code. The TriCore TC1796 includes a 48 kB large program code SPM. From these 48 kB, 1 kB is reserved for system code so that 47 kB remain. A program SPM access takes one cycle and an access to the uncached main memory (i.e., program Flash) takes 6 cycles.

We applied our ILP to 73 different real-life benchmarks. The simplest ones contain 4 basic blocks, the most complex one 585. Code sizes range from 52 bytes up to

18 kB with an average of 2.8 kB per benchmark. Since these code sizes are much smaller than the totally available SPM size, we artificially limit the available SPM space for benchmarking. For each benchmark, SPM sizes of 10%, 20%, ..., 100% of the benchmark's code size were used. Our results show the WCET estimates of all benchmarks produced by the WCET analyzer aiT that result from our WCET-aware SPM allocator as a percentage of the WCET when not using the program SPM at all.

Figure 10 shows the impact of our SPM allocation on the WCET estimates of two representative benchmarks. `g721_encode` (size: 3,204 bytes) exhibits a steady WCET decrease with increasing SPM size. Already for tiny SPMs of only 10% of the program's size, the WCET after our optimization amounts to 71% of the original WCET, i.e., WCET was reduced by 29%. If the benchmark entirely fits into the SPM, the resulting WCET is 52.2% of the original WCET which leads to savings of 47.8%.

`cover` (size: 2,670 bytes) exhibits stepwise WCET reductions. At 40%, 70% and 100% of SPM size, our ILP moves exactly those loops with the highest savings onto the SPM. Thus, WCET savings of 10.2%, 34.9% and 44.3% were achieved, resp.

On average over all 73 benchmarks, steadily decreasing WCETs were observed for increasing SPM sizes (cf. Fig. 11). Already for tiny SPMs, WCETs decrease to 92.6% of the WCET without any SPM which corresponds to a WCET reduction of

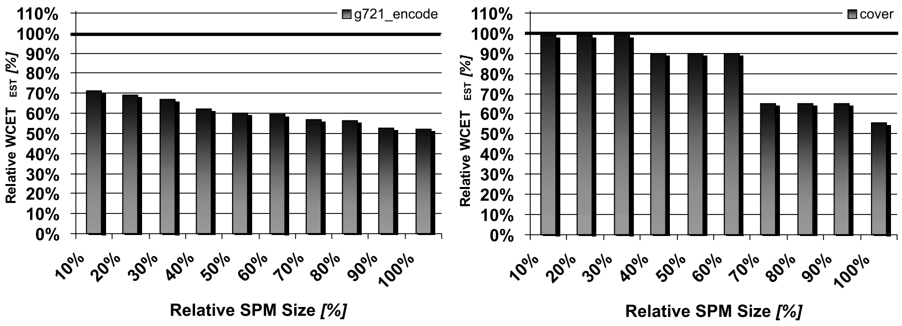
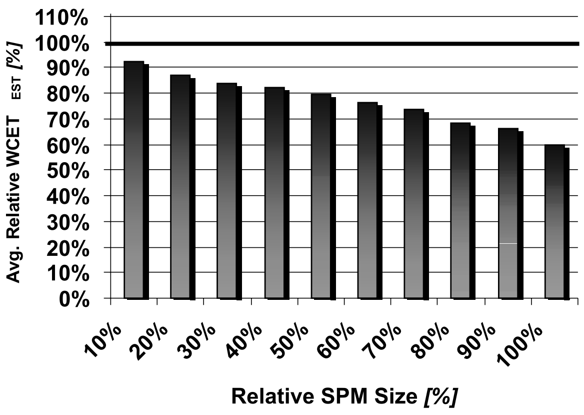


Fig. 10 Relative WCETs after WCET-aware SPM allocation of code for representative benchmarks

Fig. 11 Average WCETs after WCET-aware SPM allocation of code



7.4%. For SPMs large enough to hold entire benchmarks, average WCETs of only 60% of the original WCET were obtained which leads to overall savings of 40%.

A basic block's size depends on its memory allocation so that the ILP potentially changes the benchmarks' code sizes (cf. Sect. 9.2.4). It turned out that these changes are negligible. We observed a maximal code size increase by 128 bytes for our benchmarks. On average over all 73 benchmarks, code sizes increased by 0.02%.

The complexity of our ILP-based SPM allocator is negligible, too. For all 73 benchmarks, the ILP solver *cplex* only takes one or two CPU seconds on an Intel Xeon machine that runs at 2.4 GHz. Compared to this, the two WCET analyses required to generate the constants C^i_{spm} and C^i_{main} (cf. Sect. 9.3) are more expensive, but they also terminate within a few CPU minutes for our largest benchmarks.

10 WCET-aware scratchpad allocation of program data

In analogy to Sect. 9, this section presents an ILP for SPM allocation of program data. Here, program data denotes global data or local data with static storage. The ILP described in the following also relies on the techniques proposed by Suhendra et al. (2005) and is thus very similar to that of Sect. 9. Therefore, we just provide a very compact description of this ILP without a further detailed survey of related work.

Instead, we just briefly highlight that on the one hand, Suhendra's work suffers from several limitations that prevent it from being applied to real-life programs. E.g., a way to automatically determine which data is accessed by each basic block is missing, which is mandatory to formulate an ILP for SPM allocation of data.

On the other hand, Deverge and Puaut (2007) present a hybrid approach for dynamic SPM allocation of data that combines an ILP with an iterative heuristic. First, the current WCEP is computed, and an ILP tailored for this particular WCEP determines which data to place on the SPM. Next, the WCEP is updated and some more SPM contents is computed using ILP. In contrast, the following section presents a fully ILP-based SPM allocation. As was done in Sect. 9, we assume a processor without D-cache.

10.1 Structure of the ILP for program data scratchpad allocation

A binary variable y_i per data object d_i of a program specifies if a data object is put in main memory or in SPM (cf. (17)). For SPM allocation of data, a block b_j's WCET depends on the placement of all data objects accessed by b_j. Each block b_j causes some costs c_j. c_j reflects b_j's WCET depending on whether the data objects accessed by b_j are put in main memory or in the SPM (cf. (18)). Here, C_j denotes b_j's WCET if all data objects accessed by b_j are placed in main memory. $G_{i,j}$ is a constant that denotes the WCET reduction of b_j if data object d_i is put on the SPM.

As was done in Sect. 9.2.1, the WCET of a loop's exit node b^L_{exit} is set to the costs of b^L_{exit} (cf. (19)) and the WCET of a path from b_j to b^L_{exit} must be greater or equal than the WCET of any successor of b_j, plus b_j's costs (cf. (20)). For an entire loop L with entry node b^L_{entry}, variable w^L_{entry} denotes the WCET of all paths in L if L is executed exactly once. Again, the innermost loop L is collapsed, a new super-node v_L is created and v_L's costs are defined as shown in (21). Equation (20) is formulated for each successor of b_j so that variable w_j reflects the WCET of any path that starts at b_j. Thus, (20) realizes the implicit consideration of (switching) WCEPs in the ILP.

$$y_i = \begin{cases} 1 & \text{if data object } d_i \text{ is assigned to } mem_{spm} \\ 0 & \text{if data object } d_i \text{ is assigned to } mem_{main} \end{cases} \tag{17}$$

$$c_j = C_j - \sum_{d_i \in \text{data objects}} G_{i,j} * y_i \tag{18}$$

$$w^L_{exit} = c^L_{exit} \tag{19}$$

$$\forall b_j \in V \setminus \{b^L_{exit}\} : \forall (b_j, b_{succ}) \in E : w_j \geq w_{succ} + c_j \tag{20}$$

$$c_L = w^L_{entry} * C^L_{max} \tag{21}$$

$$cp_j = \begin{cases} w^F_{entry} & \text{if } b_j \text{ calls } F \\ 0 & \text{else} \end{cases} \tag{22}$$

$$\forall b_j \in V \setminus \{b^L_{exit}\} : \forall (b_j, b_{succ}) \in E : w_j \geq w_{succ} + c_j + cp_j \tag{23}$$

$$\sum_{d_i \in \text{data objects}} (S_i * y_i) \leq S_{spm} \tag{24}$$

$$w^{\texttt{main}}_{entry} \rightsquigarrow min. \tag{25}$$

For a function F with entry node b^F_{entry}, w^F_{entry} denotes the WCET of any path that starts at b^F_{entry}, if that F is called exactly once. If a block b_j calls a function F, a call penalty cp_j per block b_j is introduced (cf. (22)). cp_j is added to the control flow constraint of (20), and the resulting equation (23) is the constraint that is finally generated for our ILP per block b_j and per successor b_{succ} of b_j.

In contrast to Sect. 9.2.4, allocating data objects does not change the objects' size. Thus, the data objects' sizes are true constants in our ILP that can easily be computed by WCC. Hence, the SPM capacity constraint is defined as shown in (24). S_i denotes the byte size of object d_i, and S_{spm} is the available data SPM size in bytes.

In analogy to Sect. 9.2.5, the above ILP optimally reduces WCETs by minimizing variable $w^{\texttt{main}}_{entry}$ that denotes the entire program's WCET (cf. (25)).

10.2 Implementation issues

As for the SPM allocation of code, the ILP for SPM allocation of data is realized at assembly code level. Equation (18) uses the constants C_j and $G_{i,j}$. C_j represents the WCET of block b_j and is obtained within WCC by allocating all data objects to main memory and performing a WCET analysis as described in Sect. 3. $G_{i,j}$ models the gain achieved for block b_j if a data object d_i is moved from main memory to the SPM. To obtain $G_{i,j}$, it must be known how often b_j accesses d_i. This information is generally difficult to compute and requires massive support by WCC's infrastructure.

Since modern processors usually are load-store machines, address registers keep addresses, and dedicated load/store operations use them to access memory. To obtain the required data access information per block b_j, one needs to know the address registers' contents for each load/store in b_j. The WCET analyzer aiT already provides such an address register analysis (cf. Sects. 1.1 and 3.4). Hence, the WCET analysis used to obtain the constants C_j also provides register analysis results. If aiT determines that a certain load/store accesses an address range that belongs to exactly one data object, we have highly precise data access information for that load/store.

Unfortunately, aiT can not always compute address register values this precisely. To obtain more precise information, WCC includes a pointer alias analysis. Since aiT works on assembly code level, we complement its low-level analysis by an alias analysis at the ICD-C source code level. WCC's alias analysis has the following features:

- interprocedural analysis that considers data flow between functions,
- context-sensitive function argument/return value analysis using summaries (Nystrom et al. 2004),
- inclusion-based analysis for high precision (Andersen 1994),
- field-sensitive analysis that provides data on element accesses of composed types.

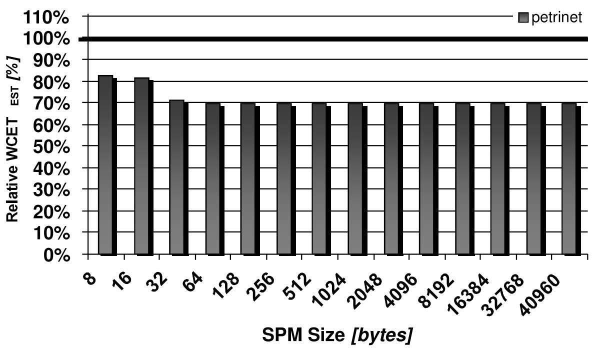
Using this alias analysis, the ICD-C IR contains information about the location a pointer variable points to for each pointer dereferencing expression. This information is preserved during code selection so that points-to data is available in WCC's backend. Additionally, the code selector creates points-to data for array and struct accesses using the regular C operators `.` and `[]`. This points-to information complements aiT's analysis and is used to determine which data object is accessed by a load/store. Accumulating the points-to information over all operations of a block b_j yields how many times a data object d_i is accessed by b_j. Multiplying this value by the speed difference between main memory and SPM finally yields the needed constants $G_{i,j}$.

All other constants required by the ILP for SPM allocation of program data (i.e., C^L_{max}, S_i and S_{spm}) are determined by the WCC compiler in analogy to Sect. 9.3.

10.3 Evaluation

The Infineon TriCore TC1796 features a data SPM of 56 kB accessible in one cycle and 64 kB of data main memory with 6 cycles access latency. It does not feature a D-cache per se. From the available data SPM, 12 kB are reserved for the stack and 4 kB are used for the TriCore's Context Save Area so that a total of 40 kB remains for free use by our SPM allocation. Our ILP was applied to only those benchmarks that contain global data or local data with static storage. During benchmarking, SPM sizes between 8 bytes and 40 kB were used (cf. Fig. 12). Our results show the WCET estimates of all benchmarks that result from our WCET-aware SPM allocator as a percentage of the WCET when not using the data SPM at all.

Figure 12 shows the impact of our WCET-aware data SPM allocation on the WCET estimates of two representative benchmarks. `petrinet` is a Petri net simulation that uses 6 global variables of only 72 bytes size in total. Some of these variables are accessed very frequently, and our ILP clearly identifies how often each variable



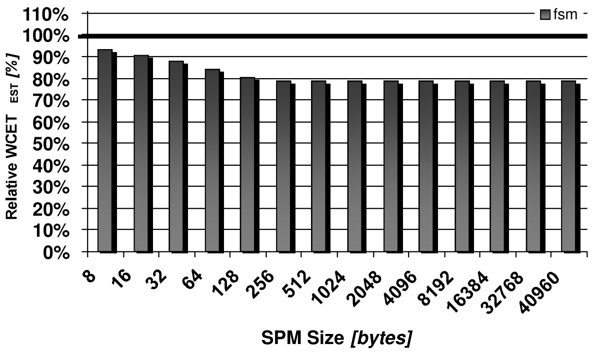


Fig. 12 Relative WCETs after WCET-aware SPM allocation of data for representative benchmarks

is accessed by which basic block and moves the most beneficial variables onto the data SPM. Hence, already a tiny data SPM of 8 bytes leads to WCETs after our optimization of only 82.5% of the original WCET, i.e., WCET was reduced by 17.5%. A 32 bytes large SPM leads to a relative WCET of 71.4% which corresponds to a total WCET reduction of 28.6%. If the entire global data of `petrinet` fits into the SPM, a WCET of 69.7% was achieved which translates to an overall reduction by 30.3%.

`fsm` (electric window lifter control) exhibits a more steady WCET reduction. `fsm` uses 98 global variables to save the automaton's state. All these variables are at most 4 bytes large, some of them are accessed frequently, some only rarely. For an 8 bytes SPM, a relative WCET of 93.5% was observed that leads to a WCET reduction of 6.5%. Increased SPM sizes translate to reduced WCETs as shown in Fig. 12. The best results were obtained for SPMs of size 256 bytes or larger. Here, our optimization achieves relative WCETs of 78.6% which corresponds to improvements of 21.4%.

On average over all benchmarks, WCETs decreased steadily for increasing data SPM sizes (cf. Fig. 13). Already for small SPMs, average WCETs decrease to 97.4% of the WCET without any SPM which corresponds to a WCET reduction of 2.6%. For the real TriCore architecture with its 40 kB SPM, average WCETs of only 79.8% of the original WCET were obtained which leads to overall savings of 20.2%.

For the considered TriCore architecture, no additional assembly instructions have to be generated by this optimization. Instead, only the memory locations of the variables assigned to the SPM need to be adjusted. Thus, our SPM allocation of data does not modify code size at all. In analogy to Sect. 9, the complexity of the ILP for data

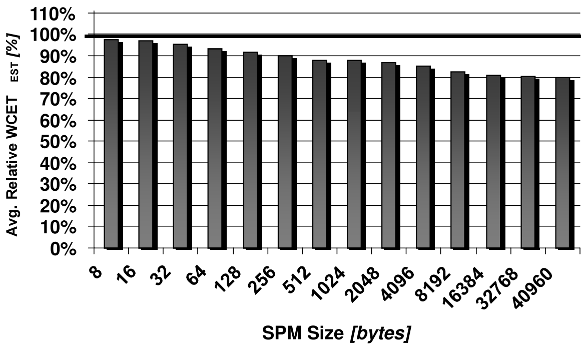


Fig. 13 Average WCETs after WCET-aware SPM allocation of data

SPM allocation is negligible in practice. Solving times of at most two CPU seconds were observed on an Intel Xeon machine that runs at 2.4 GHz.

11 WCET-aware register allocation

Register allocation is considered to be the most important compiler optimization. Its goal is to use a processor's registers most efficiently to reduce slow main memory accesses. Due to the increasing speed gap between processors and memories, register accesses are orders of magnitudes faster than memory accesses. However, memory accesses can not be totally avoided, since the amount of temporary variables (aka. *virtual registers VREGs*) at a certain place in a program can exceed the number of available *physical processor registers* (*PHREGs*). In such a situation, *spill code* is inserted during register allocation that swaps out a register to memory and back.

Current register allocators usually decide heuristically where to insert spill code. Due to a lack of precise models, compilers are unaware of the impact of spill code on a program's execution time. Especially for real-time systems, badly placed spill code can have a dramatic impact on a program's WCET. The following sections present a WCET-aware graph coloring register allocator. Its main contributions are its explicit use of WCET data during optimization and the automatic update of WCET data in the course of the optimization to cope with the inherent instability of the WCEP.

Since WCC is the very first compiler to include a technique for WCET-aware register allocation, no related work currently exists. All previously published approaches for register allocation only focus either on ACET or on code size. Nowadays, graph coloring is the standard register allocation technique. Due to its outstanding importance, it is discussed in more detail in the following section.

11.1 Traditional graph coloring

Traditional graph coloring based register allocation (*GC*) was originally published in Chaitin et al. (1981). An *interference graph* $G = (V, E)$ contains a node for each VREG of a function and for each of the C available PHREGs. An undirected edge $e = \{v, w\}$ is added to E whenever nodes v and w interfere, i.e., if they either represent

VREGs which are simultaneously alive and thus should not share the same PHREG, or if a VREG v must not be allocated to PHREG w for architectural reasons.

GC assigns one of C colors that denote the PHREGs to each node $v \in V$ such that no two adjacent nodes have the same color. According to Chaitin et al. (1981), this is done as follows:

Build: Construct the interference graph $G = (V, E)$.
Simplify: Iteratively remove each $v \in V$ from G that has a degree $< C$, push v onto stack S. This step removes all nodes from G which are always colorable due to the small amount of adjacent nodes.
Spill: After *simplify*, each node v has degree $\geq C$. Select one node $v \in V$, mark v as *potential spill*, remove v from G, push v onto S. If $V \neq \emptyset$, continue with *simplify*.
Select: Iteratively pop nodes v from S, re-insert them into G. If v is not a potential spill, assign a free color c to v. If v is a potential spill, there may be a free color c. If this is the case, assign c to v. Else, don't color v and mark v as *actual spill*.
Start over: If there are actual spills $v \in V$, insert a load operation before each use of v and a store operation after each definition of v and continue with *build*.

By inserting spill operations before uses and after definitions of an actual spill v, v's lifetime is split into smaller intervals in the hope that these smaller intervals will only interfere with lifetimes of fewer VREGs in the next iteration of the above algorithm. This register allocator has proven to have high quality and has a complexity of $\mathcal{O}(n \log n)$, where n is the number of a program's instructions.

A crucial issue of this allocator is which node v to choose as potential spill during the *spill* stage. Related literature proposes several heuristics for this purpose:

– Select nodes according to the order in which VREGs occur in the compiler's IR.
– Select the node v with highest degree, since spilling this node reduces the degree of many other nodes in G so that it is more likely to maximize the number of nodes with degree $< C$ after spilling v.
– Select a node v depending on the degree of v, on the number of operations o that use or define v, on the register pressure around o and on the loop nesting level of each such operation o.

The above list shows that no formal timing model is used during spilling. Due to a lack of such models, heuristics try to estimate the impact of spilling on code quality. Not surprisingly, one heuristic is better in some cases, and another heuristic is better in other cases. Due to the missing link between the heuristic's estimates and actual timing data, a register allocator may be guided into a wrong direction.

11.2 WCET-aware graph coloring

Due to the spill heuristics discussed in Sect. 11.1, current register allocators have no direct control over where spill code is generated, since only simplified measures are used. This can have severe effects on a program's WCET, because traditional spill heuristics may now lead to spill code generation along the WCEP.

A WCET-aware register allocator needs to know the worst-case execution frequencies per CFG node. Unfortunately, static WCET analysis can not be applied to obtain this data for a program P. This is because P is not executable since it uses VREGs instead of PHREGs. Hence, there are cyclic dependencies between register allocation and WCET analysis—in addition to the requirements discussed in Sect. 1.1—which must be broken in order to obtain a WCET-aware register allocator.

Conventional register allocators try to keep as many VREGs in PHREGs as possible and move a VREG to memory only if really necessary. The traditional way of thinking thus assumes optimistically that all VREGs fit into the physical register file and that only exceptionally, a VREG is moved to memory. Graph coloring (cf. Sect. 11.1) also follows this strategy: it first removes all colorable nodes from the interference graph, and only after that, a decision on one single potential spill is taken.

However, this strategy is infeasible for WCET-aware register allocation. The intermediate code produced during all the steps and iterations of traditional graph coloring is not executable and thus, no WCEP can be determined. For WCET-aware graph coloring, we propose the opposite way of thinking: we assume pessimistically that all VREGs are kept in memory. Our register allocator thus moves VREGs from memory to PHREGs. This has the advantage that the IR generated in the course of register allocation is always executable so that WCEPs can be determined.

The WCET-aware graph coloring algorithm is shown in Fig. 14. In a loop, it handles one basic block per iteration (lines 2 to 13). For a program P's IR that is input to register allocation, the algorithm maintains a copy P' which is fully spilled, i.e., where all VREGs of P are marked as actual spills and load/store operations are inserted for spilling (lines 3 and 4). This fully spilled IR is statically analyzed by the WCC compiler to obtain the current WCEP, which is feasible since P' does not contain any VREGs (line 5).

Among all blocks on the current WCEP, the block b' with highest spill code execution in the worst case is chosen. Worst-case spill code execution is the product of the number of inserted spill operations per block and the block's worst-case execution frequency as computed by the WCET analyzer (line 8). For b' within the fully spilled IR P', its counterpart b in the IR P still which contains VREGs is searched (line 9).

Fig. 14 Algorithm for WCET-aware graph coloring

```
 1 IR WCET-GC-RA( IR P ) {
 2   while ( true ) {
 3     IR $P'$ = P.copy();
 4     $P'$.spillAllVREGs();

 5     set<basicBlock> WCEP = computeWCEP( $P'$ );
 6     if ( getVREGs( WCEP ) == $\emptyset$ )
 7       break;

 8     basicBlock $b'$ = getMaxSpillCodeBlock( WCEP );
 9     basicBlock b = getBlockOfOriginalP( $b'$ );

10     list<virtualRegister> vregs = getVREGs( b );
11     vregs.sort( occurrences of VREG in b );
12     traditionalGraphColoring( P, vregs );
13   }
14   traditionalGraphColoring( P, getVREGs( P ) );

15   return P;
16 }
```

Block b is the most critical one along the current WCEP. Hence, all VREGs of b should be kept in PHREGs. However, if register pressure is too high, spilling of some of b's VREGs should lead to only minimal spill code execution in b in the worst case. Therefore, all VREGs v of b are sorted by their number of occurrences in b (line 11). Since spill code generation always inserts a load before (store after) each use (definition) of v, v's occurrences in b correlate with the amount of spill code needed in b to spill v. This sorting is a precedence which VREGs are better candidates for spilling and which ones are not. It is passed to a standard graph coloring allocator (line 12) that actually maps these VREGs to PHREGs. After that, b' is put in a black-list to prevent it from being selected again by line 8 during a later iteration. For the sake of simplicity, this black-listing is omitted in Fig. 14. After basic block b is processed, the allocation loop iterates and updates the current WCEP again (line 5).

If the current WCEP does not contain any more VREGs, the allocation loop is left (lines 6 and 7). However, the IR P might still contain VREGs after leaving the loop. This happens e.g., for basic blocks in the CFG which have never been on the WCEP—such blocks have never been considered during the allocation loop. However, they still need to be allocated. To obtain a fully allocated IR, all remaining VREGs in P are passed to a final run of the traditional graph coloring register allocator (line 14).

For this final run, the applied spill heuristic does not matter. This is because even in the worst case where all remaining VREGs would be spilled, the blocks that still contain VREGs in line 14 will never ever lie on the WCEP and thus never influence P's global WCET. If they lay on the WCEP, they would have been captured by the allocation loop—in contradiction to the loop's exit condition. Hence, it is sufficient to pass an arbitrary precedence list of VREGs to the standard graph coloring allocator. For the sake of simplicity, we just use the order in which VREGs occur in P here.

11.3 Evaluation

Since register allocation needs information about the liveness and interference of VREGs and about a processor's physical registers, it can only be applied at assembly code level. The TriCore's register file has 16 data and 16 address registers. However, not all of them can be used freely by a register allocator. Several registers are used e.g., to realize function calling conventions or serve as stack or return address pointer so that we use a total of 14 data and 10 address registers. Benchmarking used WCC's optimization level *-O3* such that register allocation is always applied to already highly optimized code. In all experiments, spilling uses the TriCore's highly efficient SPM memory with 1 cycle access latency.

We used 46 different benchmarks from various domains to evaluate our WCET-aware register allocator: some are small filter and sorting routines, others are large and complex audio/video codecs. Their basic block counts range from 7 to 808. All benchmarks have in common that register pressure is high so that spill code must be generated. Figure 15 shows the WCETs of all benchmarks after WCET-aware register allocation as a percentage of the WCETs that result from traditional graph coloring (cf. Sect. 11.1) which selects the node with highest degree as spill heuristic.

WCET-aware register allocation reduces the WCETs of all benchmarks considerably. For `qurt`, the WCET after WCET-aware register allocation is 93.1% of the

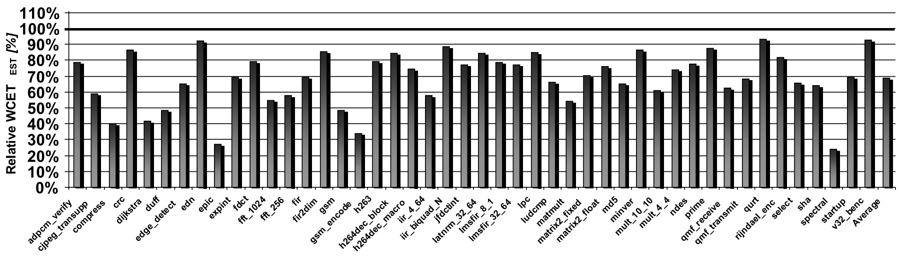


Fig. 15 Relative WCETs after WCET-aware register allocation

original WCET, i.e., WCET was reduced by 6.9%. For all other benchmarks, even higher gains were observed. `spectral` exhibits the largest WCET reductions: the WCET after our register allocation is only 24.1% of the original WCET which leads to savings of 75.9%. On average over all 46 benchmarks, a WCET of 68.8% of the original WCET was achieved so that WCETs were reduced by 31.2% on average.

For all 46 benchmarks, we observed an average increase of the benchmarks' text section of 29.8%, with a maximal increase of 298% for `dijkstra`. However, this is the only benchmark with such extreme increases. `dijkstra` is a very small code so that the insertion of only few additional spill instructions leads to excessive percental code size increases. These increases stem from the fact that our WCET-aware register allocator generates more spill code if this helps to keep the WCEP free of spill code.

Even though our register allocator performs a WCET analysis for the allocation of each basic block along the WCEP which leads to a total of 1,979 WCET analyses during allocation of all 46 benchmarks, the run times of our algorithm are still moderate. Register allocation of all benchmarks took a total of 12:15 CPU-hours on an Intel Xeon at 2.4 GHz. Of course, this is much longer than the overall 7:40 CPU minutes used by graph coloring, but it is still acceptable if high code quality for hard real-time systems is required.

12 Conclusions and future work

This article presents the WCET-aware C Compiler WCC that aims at code optimization to reduce WCETs. WCET-aware optimization needs a complex compiler infrastructure. The exploitation of memory hierarchies for WCET reduction requires detailed information about memories inside the compiler. Obviously, a tight integration of a WCET analyzer into the compiler is mandatory for WCET-aware optimization. Since WCET analysis relies on flow facts, WCC provides sophisticated mechanisms for source-level flow fact annotation. Besides user-provided flow facts, WCC includes an innovative loop analyzer that derives flow facts automatically. A back-annotation module is finally used to perform WCET-aware optimization at source code level.

On top of this infrastructure, the WCET-aware optimizations procedure cloning and positioning, scratchpad memory allocation and register allocation are integrated into WCC. Each of them reduces the WCETs of typical benchmarks between 3% and

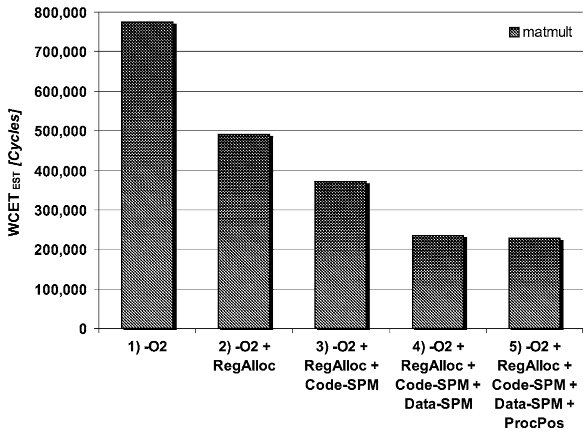


Fig. 16 WCETs after sequence of WCET-aware optimizations

48% on average. To clearly show the performance of the entire WCC framework, we applied WCET-aware register allocation, scratchpad allocation of both code and data, and procedure positioning to the `matmult` benchmark. Since this benchmark does not exhibit any potential for procedure cloning, this optimization is excluded here.

Figure 16 shows the absolute WCETs achieved by this combination of our novel techniques. All our WCET-aware optimizations are applied on top of WCC's optimization level *-O2*. The X-axis of Fig. 16 shows the different optimization sequences applied to `matmult`; they are labeled from 1) to 5) in the figure.

As can be seen, register allocation reduces the WCETs achieved by optimization level *-O2* by 284,000 cycles which corresponds to a reduction by 37.7%. The subsequent activation of our SPM allocation of program code yields a further reduction by 120,000 cycles. Compared to the previous bar 2), this translates to a reduction by 24.5%. Additionally enabling the SPM allocation of data reduces the WCET by another 135,000 cycles which—again compared to the previous bar 3)—corresponds to a percental decrease by 36.4%. Procedure positioning finally reduces the absolute WCETs by 8,000 extra cycles. Comparing this absolute reduction with the previous bar 4) of Fig. 16 shows that procedure positioning reduces WCETs by 3.4%.

This example shows that the savings individually achieved by our optimizations add up if applied in combination. In total, the optimizations described in this article are able to reduce the WCET of the `matmult` example from 775,720 cycles down to 227,794 cycles which corresponds to an overall reduction by more than 70%.

In the future, more WCET-aware optimizations will be integrated into WCC. This particularly includes function inlining, loop unswitching, dynamic SPM allocations and ILP-based register allocation. Besides pure WCET-aware optimizations, we will consider multi-objective optimizations to achieve trade-offs between real-time constraints and other optimization criteria like e.g., energy dissipation.

Acknowledgements The authors would like to thank AbsInt Angewandte Informatik GmbH for their support related to WCET analysis using the aiT framework.

References

AbsInt Angewandte Informatik GmbH (2010) aiT: worst-case execution time analyzers. http://www.absint.com/ait

Andersen LO (1994) Program analysis and specialization for the C programming language. PhD thesis, University of Copenhagen, Copenhagen

Azevedo R, Rigo S, Bartholomeu M et al (2005) The ArchC architecture description language and tools. Int J Parallel Program 33(5):453–484

Börjesson H (1996) Incorporating worst case execution time in a commercial C-compiler. Master's thesis, Uppsala University, Department of Computer Systems, Uppsala, Sweden

Campoy AM, Puaut I, Ivars AP et al (2005) Cache contents selection for statically-locked instruction caches: an algorithm comparison. In: Proceedings of ECRTS, Palma de Mallorca, Spain

Chaitin GJ, Auslander MA et al (1981) Register allocation via coloring. Comput Lang 6

Colin A, Puaut I (2001) A modular and retargetable framework for tree-based WCET analysis. In: Proceedings of ECRTS, Delft, Netherlands

Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of POPL, Los Angeles, USA

Cullmann C, Martin F (2007) Data-flow based detection of loop bounds. In: Proceedings of WCET, Pisa, Italy

Deverge J-F, Puaut I (2007) WCET-directed dynamic scratchpad memory allocation of data. In: Proceedings of ECRTS, Pisa, Italy

Engblom J (1997) Worst-case execution time analysis for optimized code. Master's thesis, Uppsala University, Department of Computer Systems, Uppsala, Sweden

Engblom J (1999) Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. In: Proceedings of RTAS, Vancouver, Canada

Engblom J, Ermedahl A (2000) Modeling complex flows for worst-case execution time analysis. In: Proceedings of RTSS, Orlando, USA

Engblom J, Ermedahl A, Sjödin M et al (1999) Towards industry strength worst-case execution time analysis. In: Proceedings of SNART, Linköping, Sweden

Ermedahl A, Gustafsson J (1997) Deriving annotations for tight calculation of execution time. In: Proceedings of Euro-Par, Passau, Germany

Ermedahl A, Sandberg C, Gustafsson J et al (2007) Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In: Proceedings of WCET, Pisa, Italy

Falk H, Marwedel P (2003) Control flow driven splitting of loop nests at the source code level. In: Proceedings of DATE, Munich, Germany

Falk H, Lokuciejewski P, Theiling H (2006) Design of a WCET-aware C compiler. In: Proceedings of ESTIMedia, Seoul, Korea

Falk H, Plazar S, Theiling H (2007) Compile time decided instruction cache locking using worst-case execution paths. In: Proceedings of CODES+ISSS, Salzburg, Austria

Ferdinand C, Heckmann R, Langenbach M et al (2001) Reliable and precise WCET determination for a real-life processor. In: Proceedings of EMSOFT, Tahoe City, USA

Gustafsson J, Ermedahl A, Sandberg C et al (2006) Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In: Proceedings of RTSS, Rio de Janeiro, Brazil

Healy C, Sjödin M, Rustagi V et al (1998) Bounding loop iterations for timing analysis. In: Proceedings of RTAS, Denver, USA

Heckmann R, Ferdinand C (2004) Worst-case execution time prediction by static program analysis. In: Proceedings of IPDPS, Santa Fe, USA

Hoffmann A, Kogel T, Nohl A et al (2001) A novel methodology for the design of application specific integrated processors (ASIP) using a machine description language. IEEE TCAD 20(11)

Holsti N, Gustafsson J, Bernat G et al (2008) WCET tool challenge 2008: report. In: Proceedings of WCET, Prague, Czech Republic

Horwitz S, Reps T, Binkley D (1988) Interprocedural slicing using dependence graphs. In: Proceedings of PLDI, Atlanta, USA

Hwu W-mW, Chang PP (1989) Achieving high instruction cache performance with an optimizing compiler. In: Proceedings of ISCA, Jerusalem, Israel

Informatik Centrum Dortmund e. V. (2010a) ICD-LLIR low-level intermediate representation. http://www.icd.de/es/icd-llir

Informatik Centrum Dortmund e. V. (2010b) ICD-C compiler framework. http://www.icd.de/es/icd-c

Kästner D (2003) TDL: a hardware description language for retargetable postpass optimizations and analyses. In: Proceedings of GPCE, Erfurt, Germany

Kirner R (2000) Integration of static runtime analysis and program compilation. Master's thesis, Technische Universität Wien, Vienna, Austria

Kirner R (2001) The programming language wcetC. Technical report, Technische Universität Wien, Vienna, Austria

Kirner R (2003) Extending optimising compilation to support worst-case execution time analysis. PhD thesis, Technische Universität Wien, Vienna, Austria

Kirner M (2006) Automatic loop bound analysis of programs written in C. Master's thesis, Technische Universität Wien, Vienna, Austria

Kirner R, Puschner P (2001) Transformation of path information for WCET analysis during compilation. In: Proceedings of ECRTS, Delft, Netherlands

Lebeck AR, Wood DA (1994) Cache profiling and the SPEC benchmarks: a case study. IEEE Comput 27(10)

Lee EA (2005) Absolutely positive on time: what would it take? IEEE Comput

Li Y-TS, Malik S (1995) Performance analysis of embedded software using implicit path enumeration. In: Proceedings of DAC, San Francisco, USA

Lokuciejewski P, Falk H, Schwarzer M, Marwedel P, Theiling H (2007) Influence of procedure cloning on WCET prediction. In: Proceedings of CODES+ISSS, Salzburg, Austria

Lokuciejewski P, Falk H, Marwedel P (2008) WCET-driven cache-based procedure positioning optimizations. In: Proceedings of ECRTS, Prague, Czech Republic

Lokuciejewski P, Cordes D, Falk H et al (2009) A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In: Proceedings of CGO, Seattle, USA

Mendlson A, Pinter SS, Shtokhamer R (1994) Compile time instruction cache optimizations. ACM SIGARCH Comput Arch News 22(1)

Nystrom EM, Kim H-S, Hwu W-mW (2004) Bottom-up and top-down context-sensitive summary-based pointer analysis. In: Proceedings of SAS, Verona, Italy

Prantl A, Schordan M, Knoop J (2008) TuBound—a conceptually new tool for worst-case execution time analysis. In: Proceedings of WCET, Prague, Czech Republic

Puaut I (2006) WCET-centric software-controlled instruction caches for hard real-time systems. In: Proceedings of ECRTS, Dresden, Germany

Puschner P, Burns A (2000) A review of worst-case execution-time analysis. Real-Time Syst 18(2/3)

Sandberg C, Ermedahl A, Gustafsson J et al (2006) Faster WCET flow analysis by program slicing. ACM SIGPLAN Not 41(7)

Suhendra V, Mitra T, Roychoudhury A et al (2005) WCET centric data allocation to scratchpad memory. In: Proceedings of RTSS, Miami, USA

Tomiyama H, Yasuura H (1997) Code placement techniques for cache miss rate reduction. ACM TODAES 2(4)

Verdoolaege S, Seghir R, Beyls K et al (2004) Analytical computation of Ehrhart polynomials: enabling more compiler analyses and optimizations. In: Proceedings of CASES, Washington, USA

Verma M, Marwedel P (2007) Advanced memory optimization techniques for low-power embedded processors. Springer, Berlin

WCET-aware Compilation (2010) http://ls12-www.cs.tu-dortmund.de/research/activities/wcc

Wehmeyer L, Marwedel P (2005) Influence of memory hierarchies on predictability for time constrained embedded software. In: Proceedings of DATE, Munich, Germany

Wehmeyer L, Marwedel P (2006) Fast, efficient and predictable memory accesses—optimization algorithms for memory architecture aware compilation. Springer, Berlin

Weiser MD (1979) Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, University of Michigan, Ann Arbor, USA

Zhao W, Kulkarni P, Whalley D et al (2004) Tuning the WCET of embedded applications. In: Proceedings of RTAS, Toronto, Canada

Zhao W, Kreahling W, Whalley D et al (2005) Improving WCET by optimizing worst-case paths. In: Proceedings of RTAS, San Francisco, California

Heiko Falk received his Ph.D. in Computer Science from the University of Dortmund (Germany) in 2004. From 2004 on until today, he works as assistant professor in the embedded systems group at the TU Dortmund.
His Ph.D. focused on high-level source code optimizations. Typical embedded multimedia applications only use a small fraction of their execution time to compute audio or video data. Most of the execution time is used to evaluate complex control flow. Motivated by this observation, Dr. Falk developed novel techniques for control flow optimization at the source code level.
In the last years, the focus of his work is on code generation and optimization for performance and predictability of safety-critical real-time systems. The WCC compiler initially established by him and developed by the research team led by Dr. Falk is the currently only known compiler which is able to systematically reduce the worst-case execution time (WCET) of programs by tightly integrating static timing analyses into the code generation and optimization stage.

Paul Lokuciejewski received the diploma degree in Applied Computer Science from the University of Dortmund, Germany, in 2005.
Afterwards, he has been working with Prof. Peter Marwedel as a member of the embedded systems group at Dortmund. In 2010, he received the Ph.D. degree in Computer Science from TU Dortmund. His research focus is on worst-case execution time aware compilation techniques for real-time systems. As key researcher in Dortmund's WCC compiler team, Dr. Lokuciejewski developed several novel WCET-aware optimizations operating at both the source code and assembly level of the program code.