View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Hertfordshire Research Archive

Real-Time Syst (2010) 45: 72–105
DOI 10.1007/s11241-010-9091-8

Transforming flow information during code optimization for timing analysis

Raimund Kirner · Peter Puschner · Adrian Prantl

Published online: 9 April 2010
© The Author(s) 2010. This article is published with open access at Springerlink.com

Abstract The steadily growing embedded-systems market comprises many application domains in which real-time constraints must be satisfied. To guarantee that these constraints are met, the analysis of the worst-case execution time (WCET) of software components is mandatory. In general WCET analysis needs additional control-flow information, which may be provided manually by the user or calculated automatically by program analysis. For flexibility and simplicity reasons it is desirable to specify the flow information at the same level at which the program is developed, i.e., at the source level. In contrast, to obtain precise WCET bounds the WCET analysis has to be performed at machine-code level. Mapping and transforming the flow information from the source-level down to the machine code, where flow information is used in the WCET analysis, is challenging, even more so if the compiler generates highly optimized code.

In this article we present a method for transforming flow information from source code to machine code. To obtain a mapping that is safe and accurate, flow information is transformed in parallel to code transformations performed by an optimizing compiler. This mapping is not only useful for transforming manual code annotations but also if platform-independent flow information is automatically calculated at the source level.

R. Kirner (✉) · P. Puschner
Institut für Technische Informatik, Vienna University of Technology, Treitlstraße 3/182/1, 1040 Vienna, Austria
e-mail: raimund@vmars.tuwien.ac.at

P. Puschner
e-mail: peter@vmars.tuwien.ac.at

A. Prantl
Institut für Computersprachen, Vienna University of Technology, Argentinierstraße 8/185/1, 1040 Vienna, Austria
e-mail: adrian@complang.tuwien.ac.at

We show that our method can be applied to every type of semantics-preserving code transformation. The precision of this flow-information transformation allows its users to calculate tight WCET bounds.

Keywords Worst-case execution time analysis · Real-time languages · Compiler optimizations · Code transformation · Abstract interpretation · Graph transformation

Abbreviations

WCET	Worst-case execution time
BCET	Best-case execution time
IPET	Implicit path enumeration technique
LCFG	Control-flow graph with loop-scope information

1 Introduction

The calculation of a WCET bound is necessary to guarantee the timeliness of real-time computer systems. To calculate a tight WCET bound one has to consider information about the possible *control-flow paths* of the program. We call this information *flow information*, typically given as program annotation. Due to the complexity of program analysis, it is not possible to extract all flow information directly from the source code (Kirner and Puschner 2003).

There are two reasons for using program annotations for WCET analysis at the source-code level: (1) When calculating flow information automatically, the static program analysis can take advantage of the abstract system view of the source-code level with availability of explicit information about the program, like control-flow structure or alias information. This allows to use compiler-independent generation of flow information. (2) When manually written flow information is added to refine the automatically generated flow information, it is much more convenient for the developer to specify them at the source-code level. Annotating the machine code is not a desirable solution as it requires that the programmer studies the compiler-generated machine code and understands the code transformations done by the compiler. Therefore, if there is need for annotations, the preferred solution is to annotate the program source with flow information and to transform the flow information automatically, while optimizing the code (Kirner and Puschner 2001; Engblom et al. 1998).

However, in contrast to this plea for the source-code level, WCET analysis has to be performed at the machine-code level to be able to calculate tight upper WCET bounds. At the machine-code level the exact timing of all operations and the exact addresses of memory locations accessed is known (Wilhelm et al. 2008; Ferdinand et al. 2001), and therefore an accurate modeling of the timing behavior is possible.

When code is translated by means of an optimizing compiler it is in general not possible to find a safe and precise match of the source-code flow information to the machine code (Engblom et al. 1998). Annotating flow information at the machine-code level is the state-of-the-art workaround to this problem in industry.

Performing code optimizations for real-time systems is important, because real-time systems typically use embedded processors with restricted processing resources.

The requirement to restrict processing resources is due to the cost pressure from mass-market applications or low power consumption requirements for mobile devices. For example, our project partners from the automotive industry sector expressed the wish to support the annotation of source code for optimizing compilation. Typical code optimizations that improve performance are loop optimizations, since most of the execution time is typically spent within loops. Examples of loop optimizations are *loop blocking* for improving the locality of memory accesses and *loop unrolling*, which enables *loop scheduling*. Such transformations change the structure of the program significantly and require non-trivial adaptions of flow information. Marlowe et al. reported on a case in which the application of several code transformations for reducing the average execution time had serious impacts on code timing and caused deadline misses (Marlowe and Masticola 1992; Younis et al. 1996).

In general, a code optimization consists of an applicability check (verifying its precondition) and a code transformation (establishing its postcondition). Our safe update of flow information depends only on the performed code transformation but not on the operations performed to check the precondition. In contrast to empiric flow information gathered by code profiling (Ball and Larus 1996), we focus on the precise update of flow information during compilation.

2 Related work

While the research community has spent lots of efforts to develop WCET analysis methods and to model many different types of target processors and hardware architectures, the implications of using optimizing compilers for WCET analysis have received little attention.

Mok et al. (1989) use special event markers to keep a mapping between the C source and the assembly code. These event markers are automatically inserted as annotations into the source code by a tool. A modified compiler is used to transform the annotations to assembly code and to generate a timing analysis language (TAL) script. The approach demands considerable high user interactions, as the generated TAL script has to be edited manually to specify flow information.

Park et al. modified the GNU C compiler to perform WCET analysis on programs written in a subset of the C language (Park and Shaw 1991; Park 1993). The analysis is done at source code by predicting the code that is generated by the compiler with deactivated optimizations. This WCET-analysis approach works well only with the absence of code optimizations performed by the compiler.

Extensive research has been done on using *debug information* to perform symbolic analysis of optimized code (Jaramillo et al. 1998). However, the use of such transformed debug information does not help to perform WCET analysis on optimized code because symbolic debugging uses only a structural mapping, but does not reflect the exact changes of control-flow logic. Thus, the support of symbolic debugging is not sufficient to reconstruct flow information at machine-code level.

Ferdinand et al. (2001) describe a WCET framework that allows the programmer to express flow information like loop/recursion bounds at the source code. Since the

authors use an external tool for the transformation of flow information, without support by the compiler, the applicability of their approach requires to restrict the set of permitted code optimizations.

Compilers have also been used to output information describing the control flow or memory-access addresses (Lim et al. 1995; Healy et al. 1999). These described approaches mostly focus on providing information to model the target hardware for WCET analysis, whereas they do not describe the modification of flow information in case of structure-changing code optimizations. Healy et al. (2000) extended a compiler to calculate loop bounds automatically for certain types of loops and to calculate certain branch constraints (Healy and Whalley 2002). Thus, their approach reduces the number of required code annotations. A similar loop analysis has been implemented by Cullmann and Martin (2007). More generic methods for calculating flow information have been developed by Gustafsson et al. (2006), Gustafsson and Ermedahl (1998) and Lokuciejewski et al. (2009). Both approaches are based on *abstract interpretation*. Such calculation techniques for deriving flow information are valuable and helpful, but they cannot calculate all types of flow information, i.e., they do not completely eliminate the need for manual flow-information annotations and automatic transformation of flow information.

Vrchoticky (1994) developed a fully integrated code compilation and WCET analysis for the Modula/R language. The optimizations performed by the compiler are rather simple and do not support structure-changing code optimizations. Lokuciejewski (2007) extended a research compiler to minimize the WCET based on feedback from WCET-analysis.

Lim et al. (1998) let the compiler generate information that characterizes the optimizations performed during compilation. Their WCET analysis method is based on the *extended timing schema* and loop bounds are the only type of supported flow information. By using labels and transformation rules, their approach is powerful enough, for example, to model the construction of a new loop from two loops in the original code. However, instead of calculating the new loop bound automatically, this approach requires to user to compute the new loop bound manually.

Engblom et al. (1998) published a more advanced approach for *compiler-generated optimization traces*, called *co-transformation*. They designed an *optimization description language* (ODL) to describe the code optimizations performed by the compiler. A more detailed discussion about the capabilities of ODL is given in Engblom (1997).

The co-transformer of Engblom et al. is currently the most advanced related approach published that deals with updates of flow information that reflect code transformations. The approach presented in this article provides the following advances:

- By linking flow information to basic blocks, the co-transformer is incapable of supporting many common code optimizations, for example, branch optimization.

  The technique we present in this article links flow information to control-flow edges, which in combination with the transition rules described in Sect. 5 allows us to support arbitrary code transformations for which it is possible to bound the iteration count for all loops in the transformed program.
- The flow-information domain of the co-transformer allows to express bounds on the execution frequency of basic blocks. This flow-information domain does not allow to express other forms of flow information to describe infeasible paths.

We support a wider range of flow information based on linear flow constraints, as described in Sect. 4. Further, our approach supports the update of flow information for hierarchical program representations by introducing special predicates that represent all relevant information about flow information within composed blocks. The latter enables the precise support of code optimizations like, for example, *loop unrolling*.

Our flow transformation framework works on the inter-procedural control-flow graph, which enables the support of inter-procedural program optimizations.

3 WCET calculation with flow information

As we will in the following focus on transformations of flow information, this section describes the use of flow information that is supported.

The WCET analysis method used with the proposed framework is based on the *implicit path enumeration technique* (IPET) (Li and Malik 1995; Puschner and Schedl 1997). This method is quite flexible and allows to model the whole program or function as one IPET problem. This method allows to specify flow information that describes dependencies between the execution frequency of arbitrary parts of a program.

For calculating a WCET bound via IPET, the structure of a program's control-flow graph (CFG) is translated into a set of flow constraints on a graph. Solving the resulting IPET problem yields a WCET bound. The solution can be calculated using standard methods like *integer linear programming*. We give a short introduction to IPET-based WCET analysis to demonstrate the expressiveness of the described flow information.

The structure of a program is given by its control-flow graph $CFG = \langle N, E, s, t\rangle$, where N is the set of program nodes (basic blocks) and $E \subseteq N \times N$ is the set of control-flow edges. We assume a unique entry node s and exit node t. An edge $e = AB \in E$ denotes control-flow passing from node A to node B.[1]

Each control-flow edge $e_i \in E$ is labeled with the maximum execution time τ_i of the corresponding code statements. The flow (total execution count) of an edge $e_i \in E$ is denoted as f_i. Using the IPET method, the WCET is expressed by the following target function using the variables f_i and the constants τ_i:

$$WCET = \max \sum_{e_i \in E} f_i \cdot \tau_i \tag{1}$$

A flow variable f_i represents the overall execution count of the edge e_i after one execution of the program. Note that in case of complex processors it would be necessary to split the execution count into different contexts, like cache hit or miss (Ferdinand et al. 2001).

The CFG is modeled by a set of constraints over the flow variables f_i. Furthermore, it is assumed that each such flow variable holds a positive integer.

[1] In practice it can happen that temporarily there exists more than one edge between two nodes during code transformations. In such cases it would be necessary to model edges with an additional identifier: $E \subseteq N \times N \times ID$. For simplicity of presentation we omit such an identifier and simply use $E \subseteq N \times N$.

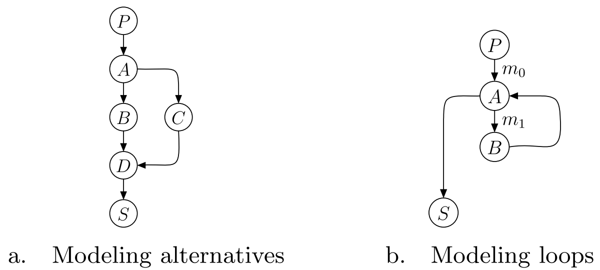


Fig. 1 Modeling program structures

The structure of the resulting IPET constraints is explained with help of two examples, with P standing for predecessor node and S for successor node (Fig. 1).

Figure 1a gives an example of a CFG that represents conditionals. Besides the constraints describing the static structure, no additional flow information is required to model the control flow of this CFG. The possible control flow can be expressed by the IPET constraints given in (2). These constraints simply express that for each node the incoming flow is equal to the outgoing flow.

$$\begin{aligned} f_{PA} &= f_{AB} + f_{AC} \\ f_{AB} &= f_{BD} \\ f_{AC} &= f_{CD} \\ f_{BD} + f_{CD} &= f_{DS} \end{aligned} \tag{2}$$

Let us assume, analysis of the code determined that the execution count of edge AB is at most half of the execution count of edge AC. This information can be expressed by the following additional flow constraint:

$$2 \cdot f_{AB} \le f_{AC}$$

An example of a loop is given in Fig. 1b. Besides the constraints describing the loop structure, WCET analysis requires at least some flow information that describes the upper loop bound of each loop. The *lower loop bound* (*LLB*) and the *upper loop bound* (*ULB*) are a lower and upper bound of a loop's iteration count:

$$0 \le LLB \le loop_iteration \le ULB$$

The flow equations of the loop construct are as follows:

$$f_{PA} + f_{BA} = f_{AB} + f_{AS}$$

$$f_{AB} = f_{BA}$$

The upper loop bound *ULB* is expressed by the following flow constraint:

$$f_{AB} \le ULB \cdot f_{PA}$$

In addition to upper loop bounds, flow information can be used to express further knowledge about the control-flow behavior of the code. Such flow information can be derived, for example, by static code analysis or from explicit annotations given by the programmer who has additional knowledge about the input parameters. A programming language that allows to specify flow information in the form of source-code annotations is described in Kirner (2002).

To summarize, the control-flow structure and all available flow information is translated into IPET constraints. A WCET bound is calculated by searching the maximum flow fulfilling these IPET constraints.

4 The flow-information domain of the transformation framework

In this section we present the domain of flow information we support by our flow-information transformation framework.

To demonstrate the application of the flow-information domain we use the small sample code given in Fig. 2a. This code contains two loops and a simple conditional statement. For the outer loop we assume that static code analysis has found the possible number of iterations to be within the interval $[1\dots4]$. The inner loop has a hard-coded loop bound of 8. The function `even(n)` $\mapsto \{0, 1\}$ returns 1 iff the argument n is a multiple of 2. The corresponding CFG for the code is shown in Fig. 2b.

4.1 The abstract program representation

A code optimization requires a precondition to be fulfilled for applying a code transformation F_t. The transformation of flow information does not depend on all the low-level operation details of the concrete code transformation, like individual rewritings, duplications, or placements of code. Instead, the update of the flow information only needs to deploy knowledge about the precondition of the code optimization and the transformation of the control-flow structure.

To filter out unneeded complexity, we focus on an abstraction $\widetilde{F_t}$ ($\widetilde{F_t} : LCFG \rightarrow LCFG$) of the program transformation, operating on the control-flow structure of the code. This abstraction works on the control-flow domain $LCFG$ shown in Table 1, where CFG is simply the control-flow graph, either in intra-procedural form or inter-procedural form. The latter is needed if we want to support inter-procedural code

Fig. 2 Example code including loop and conditional

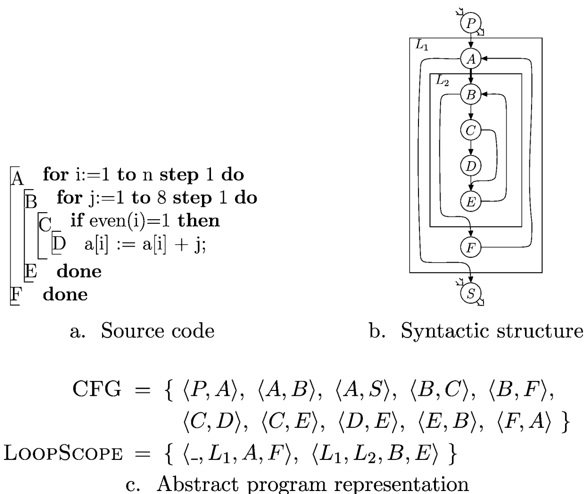


Table 1 *LCFG*—the control-flow domain

LCFG	CFG×LOOPSCOPE	
CFG	℘(EDGE) ∪ ℘(CALL)	
LOOPSCOPE	℘(LID×LID×LBEGIN×LEND)	
EDGE	BB×BB	... *control-flow edge*
CALL	BB×BB	... *call edge*
LBEGIN, LEND	BB	... *first/last basic block of a loop*
BB		... *reference to basic block*
LID		... *loop-scope identifier*

optimizations. The CFG nodes BB of EDGE refer to single basic blocks of the program $\mathcal{P}$. The two CFG nodes BB of CALL describe a call edge, denoted by the basic block ending with the function call and the basic block of the callee entry. The loop scope information LOOPSCOPE is used to describe the code boundaries of loops. LOOPSCOPE has a tree structure that represents the nesting levels of loops. LID×LID are the unique identifiers of the enclosing loop scope and the current loop scope. LBEGIN×LEND is used to identify the code of the loop body in address ordering. Non-structured loops are also supported, since the loop body identification with LBEGIN and LEND does not restrict the structure of the loop, e.g., the number of loop entries.

Typically, *LCFG* does not have to be calculated explicitly since in most compiler architectures it is implicitly given by the internal representation of intermediate code. In most cases it will suffice to augment the existing CFG with the loop scope information.

Example Consider the program code given in Fig. 2a with the corresponding CFG given in Fig. 2b: The syntactic program structure CFG and the loop scope information LOOPSCOPE are given in Fig. 2c. For an outermost loop the identifier of the surrounding loop scope is written as "_".

The domain *LCFG* contains only information that can be extracted from the code structure. The CFG is constructed by parsing the code. LOOPSCOPE can be constructed from the syntactic code structure or can also be calculated by domination relations in case of reducible loops.[2] In case of nonreducible loops more complicated algorithms are needed (Ramalingam 2002).

4.2 Supported flow information

The set of possible execution paths of a program $\mathcal{P}$ is described by its control-flow structure *LCFG* together with the flow information *fi*. We define the domain of flow information as shown in Table 2. This domain represents all the information that is needed for a safe update of flow information by the induced function F_{fi} ($F_{fi}: \mathrm{FI} \rightarrow \mathrm{FI}$).

[2] A node X is said to dominate a node Y in a CFG if every possible path from s to Y goes through X.

Table 2 FI—the domain of flow information

FI	CONSTR × LBOUND
CONSTR	$\wp$(TERMS × REL × TERMS)
TERMS	$\wp$(NUM × EDGE ∪ NUM)
REL	$\{=,<,\leq\}$
LBOUND	$\wp$(LID × LOWER × UPPER)
LOWER, UPPER	NUM (non-negative integers)

FI represents the flow information and consists of a set of flow constraints (CONSTR) and a set of lower and upper loop bounds (LBOUND). The flow information can be used to calculate both the WCET and the BCET. Note that loop bounds could be represented the same way as all other flow constraints, i.e., the loop bounds given in LBOUND could be expressed indirectly by flow constraints. However, we decided to handle loop bounds separately: Explicit knowledge of loop bounds is needed for the precise transformation of flow information in case of code optimizations like *loop interchange* or *loop blocking* (Kirner 2008). If we represent loop bounds only as a set of flow constraints, it would be necessary to recompute the explicit loop bounds before transforming the flow information in case of such code transformations. Keeping the loop bounds explicit thus improves the performance of the flow-information transformation, since the recalculation of the loop bounds would require to solve an optimization problem for each loop that is to be optimized. Finally, when constructing the IPET problem, we translate loop bounds into a set of flow constraints. While flow constraints tend to be global flow information, i.e., constraints being able to relate the overall execution count of arbitrary code locations in the program, the loop bounds are always local to their loop, even if expressed by flow constraints. In case of nested loops with non-rectangular iteration spaces (Muchnick 1997) there are, besides loop bounds, additional flow constraints necessary to tightly express the iteration count of the inner loop.

The structure LBOUND includes an upper bound (UPPER) and a lower bound (LOWER) of the iteration count of each loop. UPPER and LOWER are of type NUM, which represents the non-negative integer values. For the final calculation of a WCET bound only the upper loop bound and for a BCET calculation only the lower loop bound is necessary. But for the calculation of flow information and the safe update of flow information in case of certain loop transformations, the lower and the upper loop bounds are both needed. For example, for *loop unrolling* new constraints based on the lower bound as well as the upper bound of the loop are necessary to maintain the original precision of the flow information (Kirner 2003).

Example To give an example for the application of the flow-information domain, the program code given in Fig. 2a is used. The loop scope information together with the intervals $[1\ldots4]$ and $[8\ldots8]$ as boundaries of the loop iterations is modeled by the data domain LBOUND:

$$\text{LBOUND} = \{\langle L_1, \langle 1,4\rangle\rangle,\ \langle L_2, \langle 8,8\rangle\rangle\}$$

The data domain LBOUND contains the additional flow information required to build the *LCFG* of the code.

By analyzing the code of Fig. 2a it is possible to refine the information about (in)feasible paths. The following linear flow constraints can be added:

- $2 \cdot f_{CD} \leq 8 \cdot f_{AB}$. This constraint models the fact that the *then*-path (edge "*CD*": `even(i)=1`) of the *if*-statement in the inner loop (with a constant iteration count of 8) is executed in at most half of the loop iterations of the outer loop (edge "*AB*"). This is because the outer loop starts with an uneven index. Note that the same flow information could be equivalently expressed as $2 \cdot f_{CD} \leq 1 \cdot f_{BC}$; we chose the first variant to maintain a symmetric form to the flow information described next.
- $2 \cdot f_{CE} \leq 8 \cdot f_{AB} + 8 \cdot f_{PA}$. This constraint models the fact that the *else*-path (edge "*CE*", `even(i)=0`) of the *if*-statement is executed at most $\lceil \frac{lic}{2} \rceil \cdot 8$ times, where *lic* denotes the concrete *loop iteration count* of the outer loop. The least upper bound in $\lceil \frac{lic}{2} \rceil \cdot 8$ is necessary to compensate the case where *lic* is uneven, since the else-path of the if-statement is executed every second iteration of the outer loop, starting with the first iteration. This least upper bound is taken into account by the flow variable f_{PA}, which is equivalent to the entry count of the outer loop.

As a result, the set CONSTR contains the following flow constraints:

$$\begin{aligned}\textsc{Constr} = \{&\langle\langle\langle 2, CD\rangle\rangle, \leq, \langle\langle 8, AB\rangle\rangle\rangle,\\ &\langle\langle\langle 2, CE\rangle\rangle, \leq, \langle\langle 8, AB\rangle, \langle 8, PA\rangle\rangle\rangle\}\end{aligned}$$

4.3 Transformation of flow information

Having defined the domain of flow information we now present the foundations for the transformation of flow information.

We induce the construction of the flow-information transformation function $fi' = F_{fi}(fi)$ directly from the control-flow transformation $LCFG' = \widetilde{F}_t(LCFG)$ by abstracting from the concrete program transformation F_t to its corresponding control-flow transformation $\widetilde{F}_t$. This abstraction helps to reduce the complexity of deriving F_{fi}. The upper part of Fig. 3 forms a commutative diagram for the calculation of the updated control-flow structure *LCFG*. Transforming the concrete program (F_t) and abstracting it then to its control-flow structure (α) is equivalent to first abstracting the concrete program to its control-flow structure (α) and then performing the abstract transformation of the control-flow structure ($\widetilde{F}_t$). This is because the transformation of flow information is independent of the low-level implementation details of a code

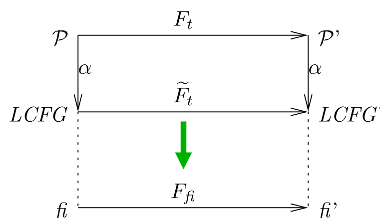


Fig. 3 Derivation of the flow-information update F_{fi} from the control-flow transformation $\widetilde{F}_t$

Table 3 IndTrans—a framework for induced update of flow information ($F_{fi} = \text{IndTrans}(\langle \text{Opt}, \widetilde{F}_t \rangle)$)

Opt	*... identification of code optimization*
$\widetilde{F}_t$	$LCFG \longrightarrow LCFG$
F_{fi}	TransConstr × TransLBound
TransConstr	Terms ⟶ Terms
TransLBound	LBound ⟶ LBound

transformation. The lower part of Fig. 3 visualizes that F_{fi} is induced from the control-flow update $\widetilde{F}_t$.

Based on the assumption that the given flow information is safe according to Definition 4.1, our goal is to derive flow information for the transformed program that is also safe. Note that this definition of safe flow information is equivalent to being *timing invariants* at the corresponding *annotation layer* (Kirner et al. 2008).

Definition 4.1 (Safe Flow Information) A flow information *fi* of a program $\mathcal{P}$ is *safe* iff for all possible initial states of program $\mathcal{P}$ the flow information *fi* is fulfilled for the execution of $\mathcal{P}$.

Typical compiler optimizations consist of a program analysis phase and a subsequent program transformation phase which can also be interleaved. By using the abstract program transformation function $\widetilde{F}_t$ one can find whether different code optimizations fall into the same class of abstract *LCFG* transformations. This fact simplifies the design of a transformation function F_{fi} that is both complete and safe. Code optimizations at the instruction level do not require an update of flow information. Only transformations that change the control flow, i.e., the data domain *LCFG*, affect the flow information.

5 A framework for flow-information transformation

To specify the flow-information updates induced by a code transformation, we have developed a graph transformation framework that supports graph hierarchies with "boundary-crossing" edges (Drewes et al. 2002).

The fact that the flow-information update function F_{fi} is induced from knowledge about the concrete code transformation Opt and the *LCFG* update is denoted as $F_{fi} = \text{IndTrans}(\langle \text{Opt}, \widetilde{F}_t \rangle)$. The signatures of the transformation functions are summarized in Table 3. In the following, each of these basic components of the induced flow-information update framework is described.

We use *loop interchange* (Fig. 4) as a running example within this section. *Loop interchange* is a code transformation that exchanges the position of two loops within a loop nest. The loop nest has to be a *perfect loop nest*.[3] A typical application of *loop interchange* is to increase data-access locality or to produce loop-invariant expressions of the inner loop. Another application domain is to enable vectorization of the innermost loop on vector architectures.

[3] In a *perfect loop nest* the body of every loop, except the innermost one, consists of only the nested loop.

Original Program

```
for i:=1 to n step 1 do
  for j:=1 to 8 step 1 do
    if even(i)=1 then
      a[i] := a[i] + j;
    else
      skip;
```

Transformed Program
(with interchanged loops)

```
for j:=1 to 8 step 1 do
  for i:=1 to n step 1 do
    if even(i)=1 then
      a[i] := a[i] + j;
    else
      skip;
```

Fig. 4 Example of loop interchange

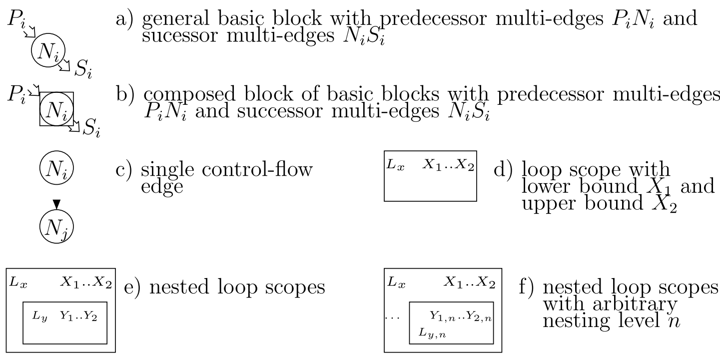


Fig. 5 *LCFG* Representation symbols for modeling the transformation $\widetilde{F}_t$

5.1 $\widetilde{F}_t$: specification of *LCFG* transformations

The specification of the performed *LCFG* transformation $\widetilde{F}_t$ is given by a graph representation supporting hierarchic transformations. Its modules are shown in Fig. 5. The graph representation greatly simplifies the matching of control-flow edges between the original and the transformed program.

The specification of a basic block N_i with edges to arbitrary *predecessor nodes* P_i and arbitrary *successor nodes* S_i that are not modified during the transformation uses the symbols of Fig. 5a. The number of such arbitrary edges is not specified and may be even zero. To modify these multi-edges, we use the generic name for the predecessor nodes P_i and the successor nodes S_i.

We use graph hierarchies to specify *composed blocks* (Fig. 5b) Such blocks can consist of an arbitrary subgraph of composed or basic blocks. Composed blocks use the same notation for arbitrary edges as basic blocks.

A *single edge* between two nodes involved in a transformation is shown in Fig. 5c. It is important to note that each node is assumed to have only the edges explicitly given by single edges or arbitrary predecessor and successor multi-edges.

A loop scope with loop identifier L_x and lower/upper loop bound is given in Fig. 5d. The direct nesting of such loop scopes is shown in Fig. 5e. Arbitrary nesting levels of loop scopes are denoted by the symbol given in Fig. 5f. The individual

Fig. 6 Special highlighting for control-flow edges of the *LCFG*

a) *control-flow edge* that is not subject to flow-information updates or structural changes.

b) *control-flow edge* that is subject to a change of its possible execution count range or is created/deleted by the code transformation with flow information assigned to it. The first case requires at least to update of scaling constants in the flow constraints and the second case requires at least to change references to control-flow edges within the flow constraint to reflect the new CFG structure.

c) *control-flow edge* that is created by the code transformation and has *no* flow information assigned to it.

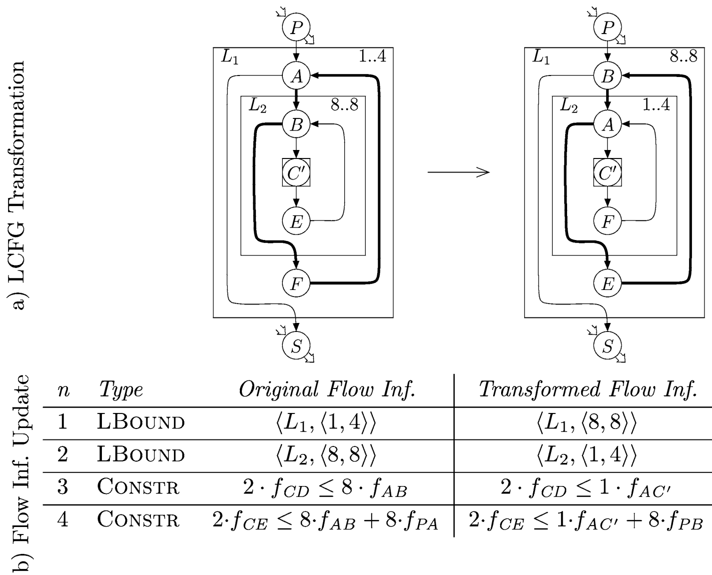
nested loop scopes $L_{y,i}$ are marked by an additional index i with $1 \le i \le n$. For example, if we have a loop scope $L_{y,n}$ with loop nesting level n relative to a loop scope L_x then there exists a chain $\{L_{y,i} | 1 \le i \le n-1\}$ of nested loops between them. The iteration count of the body of loop $L_{y,n}$ is therefore a multiple within the interval $[\prod_{i=1}^{n} Y_{1,i} \ldots \prod_{i=1}^{n} Y_{2,i}]$ of the loop bound of the body of loop L_x. These arbitrary loop nesting levels are used to graphically describe code transformations that work on loops with arbitrary nesting levels without modifying the nested loops between them.

The structural description of code transformations by the graphical notation given in Fig. 5 is used to reflect the structural control-flow changes. Additional semantic information about the possible execution-frequency changes of control-flow edges is given implicitly by the type of the performed code optimization. This semantic information is available to the compiler and will be used to induce the flow-information update function F_{fi}. To further support the visual development of F_{fi} we defined a graphical notation that discerns different types of control-flow edges involved in the described code transformation. The different types of control-flow edges and their meanings are summarized in Fig. 6.

To demonstrate the use of the abstract code transformation $\widetilde{F}_t$ based on the notation of Figs. 5 and 6 we use our running example of *loop interchange* (Fig. 4). The transformation of the LCFG for loop interchange is shown in Fig. 7a. From Fig. 7a we see that the control-flow structure of the program does not change, but the loop bounds of the two loops are swapped.

The flow information of the original program, as described in Sect. 4.2, is shown on the left of Fig. 7b using the syntax defined in Table 2. The right side of Fig. 7b shows the updated flow information after *loop interchange*. The formal flow-information transformation rules that have to be applied for *loop interchange* are described by (3) in Sect. 7.1. The meaning of the assumed flow information is as follows:

- Flow information 1 and 2 represent the loop bounds that got swapped due to the loop interchange.
- Flow information 3 and 4 are the additional flow constraints as described in Sect. 4.2. These two flow constraints refer both to edges *CD* and *CE*, which are



n	*Type*	*Original Flow Inf.*	*Transformed Flow Inf.*
1	LBOUND	$\langle L_1, \langle 1, 4\rangle\rangle$	$\langle L_1, \langle 8, 8\rangle\rangle$
2	LBOUND	$\langle L_2, \langle 8, 8\rangle\rangle$	$\langle L_2, \langle 1, 4\rangle\rangle$
3	CONSTR	$2 \cdot f_{CD} \leq 8 \cdot f_{AB}$	$2 \cdot f_{CD} \leq 1 \cdot f_{AC'}$
4	CONSTR	$2{\cdot}f_{CE} \leq 8{\cdot}f_{AB} + 8{\cdot}f_{PA}$	$2{\cdot}f_{CE} \leq 1{\cdot}f_{AC'} + 8{\cdot}f_{PB}$

Fig. 7 LCFG transformation and flow-information update on loop interchange

hidden in the LCFG transformation given in Fig. 7a. Since the execution count of the body of the inner loop is not changed by loop inversion, the execution count of edges in the inner loop body does not need to be changed. Thus, the details of the body can be hidden for simplicity.

In this example the transformation of the flow information was without loss of precision. However, if we study the flow-information transition rules of loop interchange given in (3) of Sect. 7.1, we will see that this would not be the case if the iteration count of the inner loop had been variable, instead of iterating constantly 8 times.

5.2 F_{fi}: induced transformation of flow-information

In the following we describe the two components of the flow-information update F_{fi} = TRANSCONSTR × TRANSLBOUND. Examples for their application to concrete code optimizations are given in Sect. 7.

5.2.1 TRANSCONSTR*: update of flow constraints*

The transition $\xrightarrow{C}$ changes constraint terms $\langle n_0 \cdot AB\rangle \in$ TERMS of flow constraints. The specification of TRANSCONSTR has the following syntax:

$$\langle n_0 \cdot f_{AB}\rangle \xrightarrow{C} \{\langle n_1 \cdot f_{CD}\rangle,\ \langle n_2 \cdot f_{EF}\rangle, \ldots\}$$

The semantics of this transition is to replace the term $\langle n_0 \cdot f_{AB}\rangle$ on the left and right side of all flow constraints by the list of terms $\{\langle n_1 \cdot f_{CD}\rangle, \langle n_2 \cdot f_{EF}\rangle, \ldots\}$. If the term $\langle n_0 \cdot f_{AB}\rangle$ does not occur in any flow constraint, such a transition has no effect.

To give an example, lets assume we have the use update rule $\langle n \cdot f_{CE}\rangle \xrightarrow{C} \{\langle n \cdot f_{GH}\rangle, \langle 3{\cdot}n \cdot f_{LM}\rangle\}$. The flow constraint "$5 \cdot f_{AB} \leq 2 \cdot f_{BC}$" will not be changed, since the edges AB and CE do not occur within the left side of the transition rule. In contrast, the flow constraint "$2 \cdot f_{CE} \leq 5 \cdot f_{RS}$" will be transformed into "$2 \cdot f_{GH} + 6 \cdot f_{LM} \leq 5 \cdot f_{RS}$".

If there is more than one transition with the same term on the left side, the semantics is to *create copies* of these flow constraints so that all term updates are visible. All transitions have to be applied simultaneously. If we want to delete a constraint term, we write

$$\langle n \cdot f_{AB}\rangle \xrightarrow{C} \emptyset$$

For example, the flow constraint "$5 \cdot f_{AB} + 1 \cdot f_{CE} = 1 \cdot f_{GH}$" will be transformed by above rule into "$1 \cdot f_{CE} = 1 \cdot f_{GH}$".

If any of the two term sets of a flow constraint is empty, it is implicitly replaced by the constant "0".

Scaling constraint terms by an interval If the scaling value of $\xrightarrow{C}$ is a single value, it can be directly applied to a constraint term by changing its multiplication value.

For some code transformations the relative change of the iteration count cannot be bounded by a single scaling value. Instead, it is expressed by an interval giving the lower and upper bound for the relative change. For example, when one moves a block out of a loop scope where the lower and upper loop bound of the involved loop are not equal, the relative change of the iteration count needs to be represented by an interval.

The resulting transition has the following form:

$$\begin{aligned} \langle n_0 \cdot f_{AB}\rangle \xrightarrow{C} \{&\langle [n_{11} \ldots n_{12}] \cdot f_{CD}\rangle, \\ &\langle [n_{21} \ldots n_{22}] \cdot f_{EF}\rangle, \ldots\} \end{aligned}$$

The semantics of this transition depends on the type of relation REL used by the flow constraint in which the replacement is made:

REL $\in$ {"$<$", "$\leq$"}: If the term $\langle n_0 \cdot f_{AB}\rangle$ occurs in the term list on the left of the relation we use the lower bound of the interval in the new term. Analogously, we use the upper interval bound if the term to be updated is in the right term list.

To give an example, lets assume we have to use the update rule $\langle n \cdot f_{CE}\rangle \xrightarrow{C} \{\langle [n{\cdot}3 \ldots n{\cdot}7] \cdot f_{GH}\rangle\}$. The flow constraint "$5 \cdot f_{AB} \leq 2 \cdot f_{CE}$" will be transformed into "$5 \cdot f_{AB} \leq 14 \cdot f_{CE}$". In contrast, the flow constraint "$2 \cdot f_{CE} \leq 5 \cdot f_{RS}$" will be transformed into "$6 \cdot f_{CE} \leq 5 \cdot f_{RS}$".

REL = "=": Terms in flow constraints with an equality relation ("=") cannot be directly updated by one of the two bounds of the scaling interval of the transition rule. Before applying the transformation rule we first have to normalize the equality constraint into two inequality constraints, i.e., a constraint of the form "$A = B$" is replaced by the two constraints "$A \leq B$" and "$B \leq A$". After this normalization the transformation semantics for constraints with inequality relations applies.

A further example of how to update constraint terms by an interval is given in Sect. 7.1 for *loop interchange*.

Handling composed blocks The function *bedges*(B) represents all control-flow edges in a *composed block* B. We can replace all terms referring to edges of block B by other terms referring to edges of block C or just delete them.

The direct replacement of terms referring to edges in B by terms referring to edges in C works if both blocks have the same syntactical structure and execution counts. The transition of constraint terms referring to edges of the composed block B to terms referring to edges in C is written as:

$$\langle n_0 \cdot bedges(B)\rangle \xrightarrow{C} \langle n_1 \cdot bedges(C)\rangle$$

The transition of constraint terms referring to edges of a composed block B also allows to replace terms by a list of the terms of composed blocks $C, D, \ldots$:

$$\langle n_0 \cdot bedges(B)\rangle \xrightarrow{C} \{\langle n_1 \cdot bedges(C)\rangle,\ \langle n_2 \cdot bedges(D)\rangle, \ldots\}$$

The typical applications of the composed block handling are optimizations that copy a code block with optionally slight modifications. For example, *loop unrolling* with an unrolling factor k replaces the loop body B by k copies $B_1, \ldots, B_k$ and one copy B_{k+1} in the remainder loop (Kirner 2008). All $k+1$ code copies have the same internal structure as the original code block B and the execution count of every edges in B is equal to the sum of the execution count of the corresponding edges in all new copies $B_1, \ldots, B_{k+1}$:

$$\langle n \cdot bedges(B)\rangle \xrightarrow{C} \{\langle n \cdot bedges(B_1)\rangle, \ldots, \langle n \cdot bedges(B_{k+1})\rangle\}$$

Another example for the application of composed block handling is *procedure inlining*, where each inlining creates a code copy of the procedure body. Thus, as in the example with loop unrolling, any flow-constraint term referring to an edge of the subroutine has to be extended by an additional term that refers to the corresponding edge in the inlined code.

5.2.2 TRANSLBOUND: *update of loop bounds*

The transition $\xrightarrow{L}$ changes loop bounds $\langle L_x, \langle l, u\rangle\rangle \in$ LBOUND. L_x is the loop identifier and $\langle l, u\rangle$ represents a lower (l) and upper (u) iteration bound of this loop ($l \leq u$). For a compact representation, a loop frame $\langle L_x, \langle l, u\rangle\rangle \in$ LBOUND is denoted as $L_x\langle l, u\rangle$. The induced update of loop bounds is given by a transition sequence of the following form:

$$L_x\langle l_0, u_0\rangle \xrightarrow{L} \{L_y\langle l_1, u_1\rangle, L_z\langle l_2, u_2\rangle, \ldots\}$$

This transition removes the original loop bounds $L_x\langle l_0, u_0\rangle$ and creates the new loop bounds $\{L_y\langle l_1, u_1\rangle, L_z\langle l_2, u_2\rangle, \ldots\}$ instead. If no loop bounds with the key L_x exists,

the transition has no effect. If the loop bounds are to be deleted, we write:

$$L_x\langle l_0, u_0\rangle \xrightarrow{L} \emptyset$$

If a new loop L_x with loop bounds $\langle l, u\rangle$ is introduced, we write:

$$\emptyset \xrightarrow{L} L_x\langle l, u\rangle$$

The basic operations defined above are used to compose the flow-information update function $F_{fi} = \text{TRANSCONSTR} \times \text{TRANSLBOUND}$.

5.3 Grouping flow-information transitions for each code optimization

Let us assume that the following two constraint-term transitions belong to the same code optimization:

$$\langle n \cdot f_{AB}\rangle \xrightarrow{C} \langle n{\cdot}k \cdot f_{BC}\rangle$$

$$\langle n \cdot f_{BC}\rangle \xrightarrow{C} \{\langle n \cdot f_{BC}\rangle, \langle n \cdot f_{CD}\rangle\}$$

Executing these two transitions in sequence yields an illegal scaling of the constraint terms that could be expressed by the following unintended transition:

$$\langle n \cdot f_{AB}\rangle \xrightarrow{C} \{\langle n{\cdot}k \cdot f_{BC}\rangle, \langle n{\cdot}k \cdot f_{CD}\rangle\}$$

For this reason, all transitions belonging to the flow-information update of a single code optimization have to be executed simultaneously. The compiler has to generate the set of flow-information transitions and group them for each code optimization.

5.4 Specification of flow transformations

To derive a correct and precise flow-information transformation F_{fi} (as described in Sect. 5.2) from the *LCFG* transformations (described in Sect. 5.1) is a non-trivial task. We do this in a systematic way with the following steps:

1. Determine the loop bound transitions $\xrightarrow{L}$ for the transformed program based on the flow information of the original program and the knowledge about the code transformation.
2. Determine the constraint-term transitions $\xrightarrow{C}$ for the transformed program based on the knowledge about the code transformation.
3. Delete flow information that cannot be adequately transformed. The deletion of flow information is also necessary if the transformation framework has been implemented only partially. Note: this may result in a loss of precision and higher pessimism of the successive WCET analysis. However, no flow information necessary for computing a WCET bound will be deleted.
4. Create new flow constraints for the transformed program based on the knowledge about the code transformation.

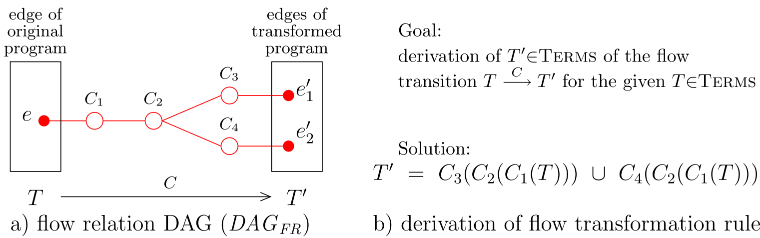


Fig. 8 Example: deduction of flow transformations from flow relations

5. Annotate the code of the program transformation with the transformation function F_{fi}.

Based on our experience the derivation of $\xrightarrow{C}$ in step 2 is the most complex part in determining the transformation function F_{fi}. Thus we developed a method to derive the constraint-term transitions $\xrightarrow{C}$ from flow relations between the original and the transformed program.

A *flow relation* is a directed acyclic graph (DAG) starting at an edge of the original program and its leaves ending either at edges of the transformed program or at so-called absolute flow declarations. Each intermediate node of a flow relation DAG (DAG_{FR}) consists of a term rewrite rule C_i : TERMS $\rightarrow$ TERMS. The derivation of the transition rule $\xrightarrow{C}$ is done in a hierarchical way to cover all paths in the DAG_{FR}. For example, given the DAG_{FR} in Fig. 8a, the resulting transition rule $\xrightarrow{C}$ is calculated with the formula given in Fig. 8b. Note that we do not draw the directions of the DAG_{FR} edges because by convention all edges are directed from left (original program) to right (transformed program). And if we interpret a code transformation in the inverse direction then by convention all DAG_{FR} edges are directed from right to left.

By studying typical code optimizations we identified a number of useful flow-relation operators to be used as DAG_{FR} intermediate nodes. These flow-relation operators and their semantics given as term rewrite rules are shown in Fig. 9.

Each DAG_{FR} consists of exactly one of the flow-relation operators given in Figs. 9a–c, which partitions the DAG_{FR} into two parts, a left-hand and a right-hand part. The two parts themselves are composed by zero or more flow-modifier nodes as given in Figs. 9d–g. The numerical identifier aside the flow-relation operators in Figs. 9a–c is used to identify its resulting flow-information transition rule.

Equal flow between the original and the transformed program is specified with the symbol given in Fig. 9a. If the flow of the original program cannot be exactly matched to a flow in the transformed program, we have to bound the original flow by the two inequality symbols given in Figs. 9b and 9c.

To build matching flow relations for code transformations that change the execution counts of some edge(s), the flow of control-flow edges can be multiplied by a constant k, as shown in Fig. 9d.

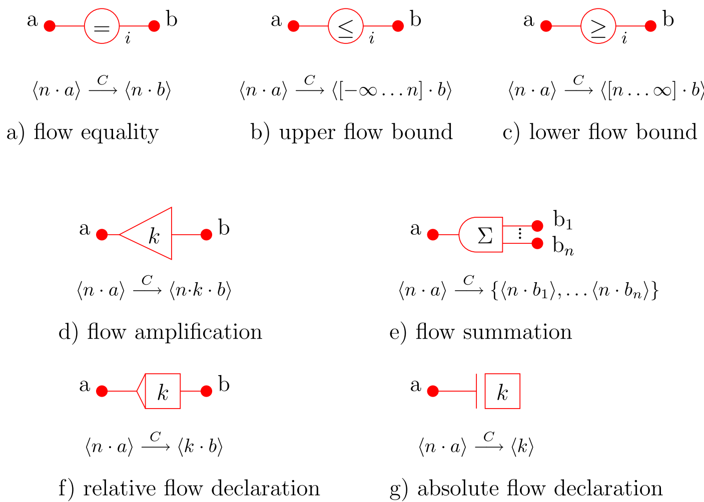


Fig. 9 Semantics of symbols for modeling flow transformations

If the flow of an edge is split into several edges, this can be modeled by the symbol given Fig. 9e. This symbol can also be applied in the opposite direction, i.e., if a code transformation merges the flow of several edges into a single edge.

If the original flow is to be replaced with the flow of a new edge independently of the original flow, this can be modeled by the symbol of Fig. 9f. Similarly, if the original flow is to be replaced with an absolute flow declaration independently of the flow in the new edges, this can be modeled by the symbol of Fig. 9g.

The flow-relation nodes presented above allow to derive any $\xrightarrow{C}$ (as described in Sect. 5.2) where the constraint term transition is based on linear scalings and absolute and relative flow declarations. This has proven to be sufficient to derive the constraint-term transitions for the many different code transformations (Kirner 2008) we investigated.

5.4.1 *Examples of DAG_{FR} and $\xrightarrow{C}$ derivation*

Once the flow relations between the original and the transformed program have been specified, the constraint-term transitions $\xrightarrow{C}$ described in Sect. 5.2 can be calculated automatically.

Figure 10 shows some examples of DAG_{FR} and the resulting constraint-term transitions $\xrightarrow{C}$. Since a code transformation can also be applied in the reverse direction, the figure shows how the derived flow-information transitions are calculated for both directions.

Relation 1 shows the update of a constraint term if the flow of an edge e in the original program relates to the flow of an edge e' in the transformed program by a rational number $\frac{k_1}{k_2}$, i.e., $e \cdot \frac{k_1}{k_2} = e'$. If the flow of the edge e equals the flow of edge e', we have the special case of $k_1 = 1$ and $k_2 = 1$.

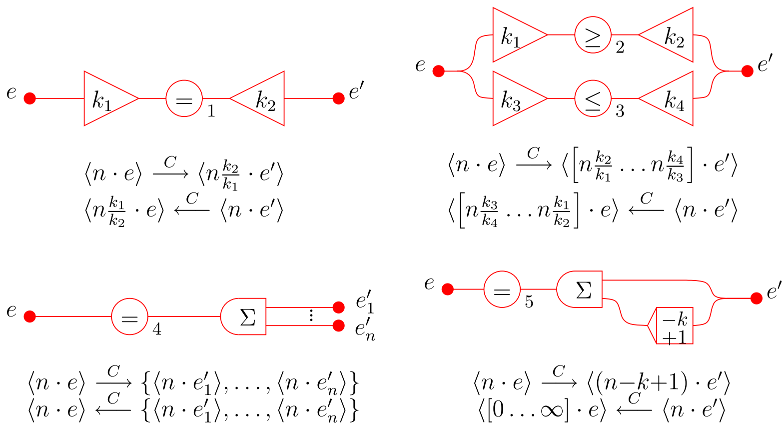


Fig. 10 Examples of flow transformations derived from flow relations

As we see in Relation 2 and 3 of Fig. 10, any DAG_{FR} with inequalities is a source of precision loss during the transformation, as it will result in a scaling of constraint terms by an interval, as described in Sect. 5.2.1. These two flow relations are grouped together, because each of them alone would result in a scaling interval either without upper bound or lower bound.

The flow relations of Fig. 10 that include a flow summation (Relation 4) can be directly translated from left to right. The other direction is only precise when every original flow constraint that contains one of the terms of the right side also contains all the other terms of the right side. If it does not contain of the given terms, then the reverse direction of these flow relations will result in a precision loss. Flow Relation 5, which is based on a relative flow declaration, shows an extreme case where the interpretation of the DAG_{FR} from right to left results in a scaling interval with no upper bound. This means that any original flow in equation $(<, \leq)$ that contains the term $\langle n \cdot e' \rangle$ on its right side becomes completely noneffective after its transformation.

5.4.2 Modeling flow relations of loop interchange

To demonstrate the use of our flow relations, we again use our *loop interchange* example. Adding the flow relations to the specification of the *LCFG* transformation given in Fig. 7, we get the flow relations shown in Fig. 11.

For each code transformation rule of the compiler the flow relations have to be specified based on the semantics of the transformation. In our example, Relation 1 says that the flow of edge *PA* in the original program is equal to the flow of edge *PB* in the transformed program. Relation 2 says that the flow of edge *AB* in the original program multiplied by the constant Y_2 (the upper loop bound of the inner loop) is at least as high as the flow of edge AC' in the transformed program. As a complement to Relation 2, Relation 3 provides an upper bound for the flow of edge *AB* in the original program: Relation 3 says that the flow of edge *AB* in the original program multiplied by the constant Y_1 (the lower loop bound of the inner loop) is at most as high as the flow of edge AC' in the transformed program. Relation 2 and 3 together represent the

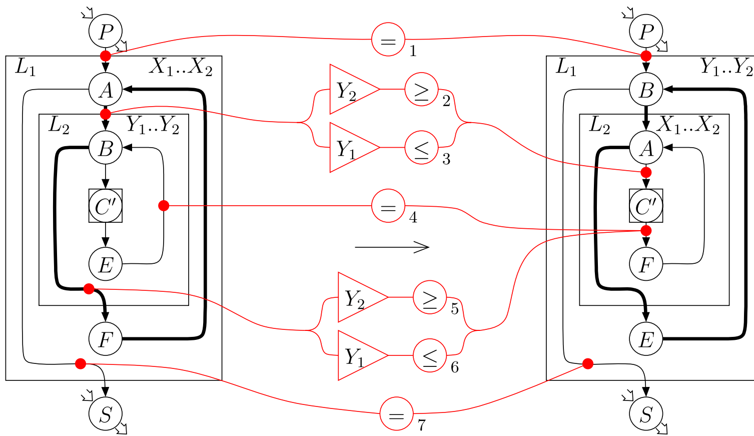


Fig. 11 Flow relations on loop interchange

fact that no precise flow scaling for edge AB of the original program has been found, but a lower and upper bound have been identified in the transformed program. The meaning of Relations 4 to 7 is analogous.

Concrete examples for the calculation of constraint term transitions $\xrightarrow{C}$ from flow relations of several code optimizations are given in Sect. 7.

6 The completeness of the approach

Having defined the flow-information transformation framework we want to answer the question if a safe transformation of flow information can be induced for any type of code transformation (completeness). This section shows the completeness of the presented flow-information update framework F_{fi}.

The transition $\xrightarrow{C}$ is the key operation to describe the update of information about infeasible paths. With this operation it is possible to replace information about the control flow of a specific control-flow edge by an arbitrary set of control-flow edges, each edge scaled by an individual scaling value. By using intervals for scaling, the transition type $\xrightarrow{C}$ can also be used to model the flow-information update for the transformation of code with variable execution counts. Examples of code with variable execution counts are conditional constructs or loops with a variable iteration count. For the construction of safe and accurate flow-information transitions in case of variable execution counts of control-flow edges, the extended format of $\xrightarrow{C}$ is used, which allows to scale constraint terms by intervals.

The transition $\xrightarrow{C}$ is powerful enough to model arbitrary control-flow transformations of the code provided that the iteration count of all loops in the transformed program can be bounded. The interesting question for modeling flow-information updates is how it is possible to know which control-flow transformation has been performed. As already described, there is sufficient information available to the compiler:

- Information about the structure of the code. For several optimizations a certain code structure or semantics is a precondition to ensure the safe application of a given code transformation.
- Semantic information about the applied code transformation. This semantic information includes information about the control flow update. This information is a static property of the applied code optimization and is therefore known by the compiler.

  For example, a code transformation can be formally described by its preconditions, postconditions, and invariants.

Lemma 6.1 *The transition* $\xrightarrow{C}$ *can express the split of control flow from an edge* e *in the original program* $\mathcal{P}$ *into the edges* $e'_1, e'_2, \ldots$ *in the transformed program* $\mathcal{P}' = F_t(\mathcal{P})$ *by a flow-information transition of the form*

$$\langle n_0 \cdot f_e \rangle \xrightarrow{C} \{\langle n_1 \cdot f_{e'_1} \rangle,\ \langle n_2 \cdot f_{e'_2} \rangle, \ldots\}$$

where the constants $n_i > 0$ *have to be set appropriately to obtain a safe and precise update of flow information.*

Lemma 6.2 *The transition* $\xrightarrow{C}$ *can express the merge of a set of control-flow edges* $S = \{e_1, e_2, \ldots\}$ *in the original program* $\mathcal{P}$ *into a single edge* e' *in the transformed program* $\mathcal{P}' = F_t(\mathcal{P})$ *by using a separate flow-information transition for each edge* $e \in S$ *of the form*

$$\langle n \cdot f_e \rangle \xrightarrow{C} \langle [0 \ldots n] \cdot f_{e'} \rangle$$

where the interval $[0 \ldots n]$ *is a safe default approximation that has to be refined by information available in the semantics of the code transformation* F_t*. If there is no information available in* F_t *to refine the interval* $[0 \ldots n]$ *then the interval* $[0 \ldots n]$ *is already the most precise scaling interval that can be obtained from the structure of a program* $\mathcal{P}$*.*

Theorem 6.3 shows the completeness of the flow-information transformation framework.

Theorem 6.3 *For any program transformation* $F_t : \mathbb{P} \to \mathbb{P}$*, for which the iteration count of loops in the transformed program* $\mathcal{P}' = F_t(\mathcal{P})$ *can be bounded relative to the execution count of control-flow edges or to loop bounds in the original program* $\mathcal{P}$*, the flexible applicability of the constraint term transition* $\xrightarrow{C}$ *together with knowledge of the structure of the original program* $\mathcal{P}$ *allows to specify updates of flow information in a* safe *and* precise *way.*

Proof First, we show the support for safety. We have to bear in mind that a flow variable f_e represents the execution count of its control-flow edge $e \in$ CFG. The code transformation maps the set CFG of control-flow edges of the original program code into a new set CFG'. Since we only focus on the WCET analysis of programs

with finite execution time there exists always a lower bound b_e ($b_e \geq 0$) and upper bound c_e ($c_e < \infty$) for the execution count of each control-flow edge $e \in$ CFG:

$$\forall e \in \text{CFG}: \quad b_e \leq f_e \leq c_e$$

Given that the original program $\mathcal{P}$ has finite execution time then based on the assumption used in the theorem it follows that the execution count of each control-flow edge of the transformed program $\mathcal{P}'$ is also guaranteed to be bounded:

$$\forall e' \in \text{CFG}': \quad b_{e'} \leq f_{e'} \leq c_{e'}$$

It is assumed that based on the knowledge of the original program $\mathcal{P}$ and the semantics of the code transformation $F_t : \mathbb{P} \to \mathbb{P}$ it is possible to bound the iteration count of any loop in the transformed program $\mathcal{P}' = F_t(\mathcal{P})$ relative to the execution count of control-flow edges or to loop bounds in the original program $\mathcal{P}$. Thus, we can use for the safe update of the flow-information a constraint-term transition of the form

$$\langle n \cdot f_e \rangle \xrightarrow{C} \sum_{e \in \text{CFG}'} \langle [l_{e'} \ldots u_{e'}] \cdot f_{e'} \rangle$$

where the interval bounds $l_{e'}$,$u_{e'}$ of each involved target edge $e' \in$ CFG' are to be set such that the following *safety condition* is met:

$$\sum_{e' \in \text{CFG}'} l_{e'} \cdot f_{e'} \leq n \cdot f_e \leq \sum_{e' \in \text{CFG}'} u_{e'} \cdot f_{e'}$$

It is a safety condition because it ensures that constraints of the original program may only lose precision after the transformation (due to the scaling interval) but never become unsafe according to Definition 4.1.

Second, we show the precision with respect to the information that is available during code transformation. Using above safety condition, a constraint-term transition is precise if it minimizes the following precision term:

$$\sum_{e' \in \text{CFG}'} |u_{e'} - l_{e'}| \cdot f_{e'}$$

The maximum execution count f_e of any control-flow edge $e \in$ CFG is not known at the transformation time of the code. But the execution-count is approximated by the relative execution counts of the changed CFG edges, which are derived from the structure of the code and the semantic precondition of the specific code transformation. Thus, the precision criteria can be fulfilled by directly setting the proper scaling values of $\xrightarrow{C}$ to minimize above precision term. Code transformations that do not involve loops can only split and join control flow. Following from Lemmas 6.1 and 6.2 such transformations can be directly expressed by the flow-information transition $\xrightarrow{C}$, where the precision is set by the scaling values on the right side of the transition. Since the scaling values are freely chosen, the most precise scaling values based on the semantics of the code transformation F_t can be selected. If loops

are involved in the code transformation, then any split or merge of control flow can be modeled with the maximal possible precision by taking into account the relative iteration counts of involved loops. This is based on the assumption that the iteration count of loops in the transformed program $\mathcal{P}'$ can be derived relative to the control flow in the original program. □

Concluding, the framework is able to precisely update arbitrary linear flow constraints on the control-flow of a program. The framework is applicable to all code transformations where loop bounds of the transformed program can be obtained from the original program for arbitrary code structures, which is the case for almost all code optimizations. Flow information about loop bounds is managed explicitly rather than expressing them also as linear constraints, because this allows to transform flow information more precisely in case of optimizations that change the iteration space of loops. Although the specification of loop bounds and also the application of certain loop transformations is only suitable for *natural loops*,[4] the flow-information transformation framework can be applied to generic code structures as well by using general constraints instead of loop bounds.

The framework in its current form cannot be applied to program transformations which introduce loops in the code that cannot be bounded from knowledge about the structure of the original program and the semantics of the code transformation.

7 Examples of flow-information transformation rules

The application of the flow-information transformation framework is demonstrated by constructing the flow-information transformation rules for our *loop interchange* example. Furthermore, the flow-information transformation for code optimizations that introduce new loops is briefly discussed.

7.1 Loop interchange

The abstract code transformation $\widetilde{F}_t$ for *loop interchange* (Fig. 4) is given in Fig. 7a. It describes the structural control-flow transformation together with the execution-count changes caused by the specific code optimization.

By analyzing the abstract code transformation function $\widetilde{F}_t$ we get the flow relations specified in Fig. 11. Using these flow relations and the calculation schema described in Sect. 5.4 we directly get the following set of flow-information transitions for the

[4] *Natural loops* are loops with only a single entry point.

flow-information transformation function F_{fi}:

$$
\begin{aligned}
&L_1\langle X_1, X_2\rangle \xrightarrow{L} L_1\langle Y_1, Y_2\rangle, \qquad L_2\langle Y_1, Y_2\rangle \xrightarrow{L} L_2\langle X_1, X_2\rangle \\
&\langle n \cdot f_{AB}\rangle \xrightarrow{C} \left\langle \left[n\frac{1}{Y_2} \ldots n\frac{1}{Y_1} \right] \cdot f_{AC'} \right\rangle \quad \text{(Relations 2, 3)} \\
&\langle n \cdot f_{BF}\rangle \xrightarrow{C} \left\langle \left[n\frac{1}{Y_2} \ldots n\frac{1}{Y_1} \right] \cdot f_{C'F} \right\rangle \quad \text{(Relations 5, 6)} \\
&\langle n \cdot f_{PA}\rangle \xrightarrow{C} \langle n \cdot f_{PB}\rangle \quad \text{(Relation 1)} \\
&\langle n \cdot f_{BC'}\rangle \xrightarrow{C} \langle n \cdot f_{AC'}\rangle \quad \text{(Relation 4)} \\
&\langle n \cdot f_{AS}\rangle \xrightarrow{C} \langle n \cdot f_{BS}\rangle \quad \text{(Relation 7)}
\end{aligned}
\tag{3}
$$

Using the transformation rules of (3) provides the concrete flow-information update shown in Fig. 7b.

7.2 Introduction of new loops

Code transformations that change the iteration count of loops or even introduce new loops are illustrative examples to demonstrate the capability of the presented flow-information transformation framework. As discussed in Sect. 6, the new flow information is calculated from the current flow information by considering the structure of the code and the semantic information of the applied code transformation.

For code transformations that introduce new loops we discuss the update of the loop bounds (see Sect. 5.2.2). The flow-information update of the control-flow edges can be constructed in a similar way. The method is based on the assumption that for every introduced loop a loop bound can be derived from the loop bounds of existing loops and the semantics of the code optimization. New loops are inserted by, for example, *loop blocking*, *loop distribution*, *loop peeling*, *loop unswitching* and sometimes also by *procedure inlining* (Muchnick 1997; Kirner 2008). For this type of code optimizations it is possible to derive the loop bound of newly introduced loops from the original code and loop bounds.

However, there are also some code transformations that introduce loops for which the loop bound cannot be derived directly from the original code and loop bounds, for example, inserting a busy-waiting loop to wait for the write cycle of an flash memory. In this case the worst-case loop bound to be used has to be specified by the compiler developer, which could be made adjustable via compiler switches describing the target platform.

As an example for the introduction of new loops we take *loop blocking*, which is a code transformation that splits the iteration space of a loop into a sequence of smaller iteration spaces of length k by inserting an extra loop (Kirner 2008). The abstract code transformation $\widetilde{F}_t$ for this form of *loop blocking* is given in Fig. 12.

By analyzing the abstract code transformation function $\widetilde{F}_t$ we get the flow relations specified in Fig. 13. This flow relations are relatively simple, since loop blocking allows to find a direct equality flow relation for each edge of the original program.

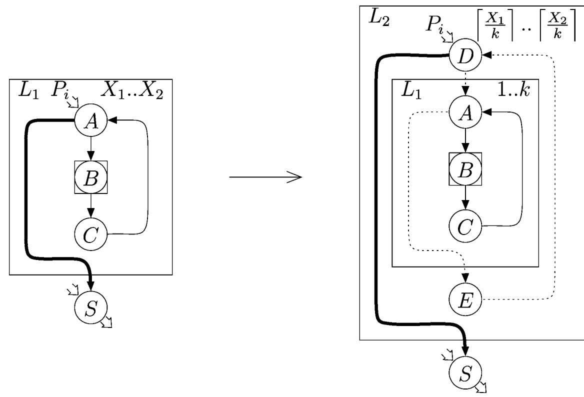


Fig. 12 CFG transformation on loop blocking

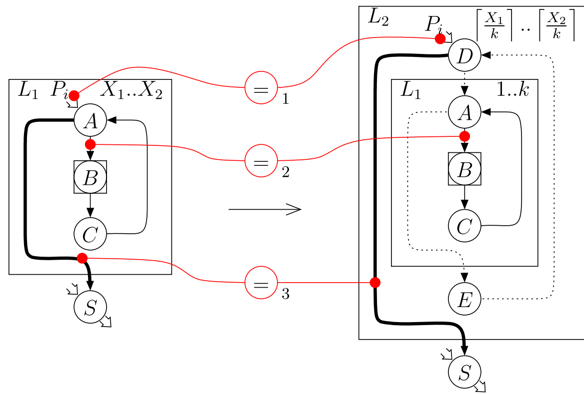


Fig. 13 Flow relations on loop blocking

Using these flow relations and the calculation schema described in Sect. 5.4 we directly get the following set of flow-information transitions for the flow-information transformation function F_{fi}:

$$L_1\langle X_1, X_2\rangle \xrightarrow{L} L_1\langle 1, k\rangle, \qquad L_2\left\langle \left\lceil \frac{X_1}{k} \right\rceil, \left\lceil \frac{X_2}{k} \right\rceil \right\rangle$$

$$\langle n \cdot f_{P_iA}\rangle \xrightarrow{C} \langle n \cdot f_{P_iD}\rangle \quad \text{(Relation 1)}$$

$$\langle n \cdot f_{AS}\rangle \xrightarrow{C} \langle n \cdot f_{DS}\rangle \quad \text{(Relation 3)}$$

The following two flow constraints are added to reuse the loop bound information of the original loop:

$$1 \cdot f_{AB} \geq X_1 \cdot f_{P_iD}$$

$$1 \cdot f_{AB} \leq X_2 \cdot f_{P_iD}$$

This example demonstrates that even for complex control-flow transformations the update of flow information is determined by the abstract code transformation

function $\widetilde{F}_t$. Further examples of flow-information transformation rules for concrete code optimizations can be found in Kirner (2003, 2008). First results with a prototype implementation have shown that supporting code optimizations for WCET analysis yields WCET bounds that can be more than three times smaller than when code optimizations are disabled (Kirner and Puschner 2003).

8 Implementation in a compiler system

Proof-of-concept implementations of the flow-information transformation framework described in this article were performed in the following two compiler systems:

- The first implementation was done for the GNU C Compiler (GCC) version 2.7.2 (Kirner 2003). This implementation has been able to show only some basic capabilities of the flow-information transformation framework, since GCC 2.7.2 at that time implemented only a small number of code transformations that change the control flow of a program significantly.
- In more recent work, Schulte integrated flow-information transformation into a research compiler that is also able to perform WCET-aware code optimizations (Schulte 2007). Schulte added transformations for additional annotations that are supported by the WCET analysis tool aiT.

Both of the above implementations tightly integrate the handling and transformation of flow information with the compiler. Such an approach involves a substantial engagement in the compiler internals and is only feasible when it is integrated with the main development branch of the compiler. Experiences with the first implementation show that maintaining flow-information handling as an add-on patch can be very time-consuming. These complications inspired the development of a more portable solution: Considering that not every optimizing transformation alters the control flow graph, the optimizations can be divided into two groups, control-flow-invariant and control-flow-modifying. A majority of the control-flow-modifying transformations are loop optimizations (Kirner 2003), which can be implemented effectively as source-to-source transformations, an approach traditionally taken by Fortran compilers (Allen and Kennedy 2002). Recent versions of GCC also use a near-source internal representation to perform high-level loop optimizations (Pop et al. 2006). By carrying out control-flow-modifying optimizations as source-to-source transformations, the subsequent target compiler needs only to perform a direct translation to machine code, leaving the control flow intact. It is still safe to apply control-flow-invariant transformations in the target compiler.[5]

The prototypical work-flow of the high-level source-to-source compiler is shown in Fig. 14 (Prantl 2007a): In the first step, annotations and source code are separated. Each annotation is associated a unique label that identifies its corresponding location in the source code. In the second step, the source-to-source optimization is performed.

[5]In current compiler systems like GCC or LLVM this can be achieved by starting with the `-O0` optimization level and manually enabling safe transformations using the respective command line flags.

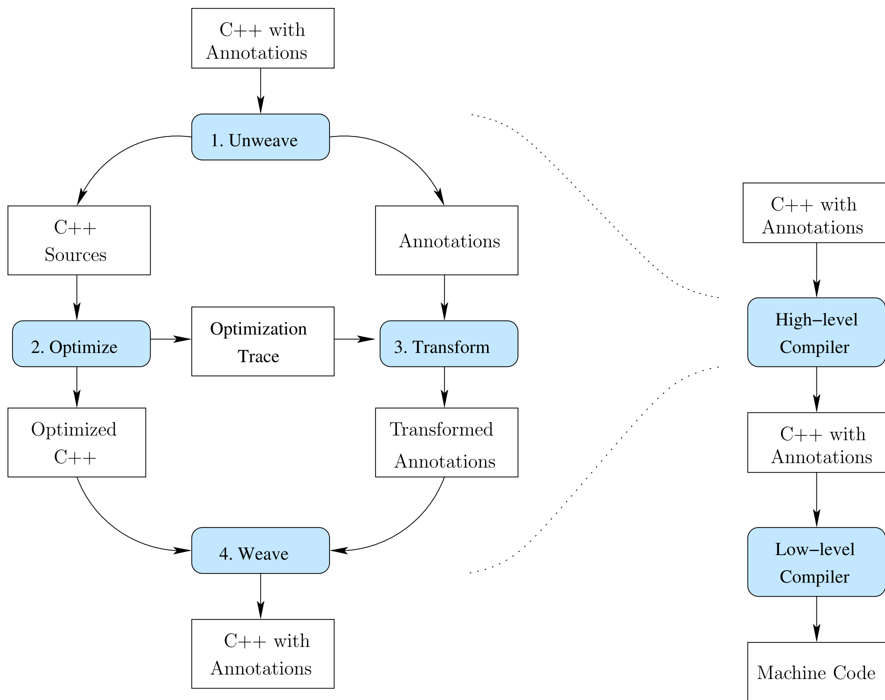


Fig. 14 Work-flow of the source-to-source high-level compiler

The optimizer generates a trace of the performed program transformations. The optimization trace together with the original annotations is then the input for the flow information transformation engine. This engine contains a rule base describing the flow constraint update for each type of optimization (cf. Sect. 7). In the final step, the transformed annotations are merged with the transformed source code.

8.1 Experiences with the TuBound implementation

The TuBound WCET analysis tool contains an implementation of this source-to-source work flow, based on a C++ port of the Fortran D loop optimizer included with the source-to-source compiler ROSE (Quinlan et al. 2004). Even though the performed high-level optimizations target the average-case execution time, first measurements indicate a positive effect also on the analyzed worst-case performance of the optimized programs (Prantl 2007b; Prantl et al. 2008).

8.1.1 Transformation rules example

While the source-to-source infrastructure is targeting a subset of C++, the low-level compiler and rest of the tool chain is currently restricted to C as input language. The analysis results (loop bounds, flow constraints) found by TuBound are annotated into the source code as `#pragma` directives. The concrete syntax of these annotations was

```
% loop interchange
% -----------
% interchanged(+Loop1, +Loop2, +Annotations, +OldAnnotation, -NewAnnotations)
%
interchanged(Loop1, Loop2, _, annotation(Loop1, wcet_loopbound(Lo..Up)),
                              [annotation(Loop2, wcet_loopbound(Lo..Up))]).

interchanged(Loop1, Loop2, _, annotation(Loop2, wcet_loopbound(Lo..Up)),
                              [annotation(Loop1, wcet_loopbound(Lo..Up))]).

interchanged(Loop1, Loop2, Annotations,
             annotation(M_Annot, wcet_constraint(Lhs=<Rhs)),
             [annotation(M_Annot, wcet_constraint(Lhs1=<Rhs1))]) :-
 member(annotation(Loop2, wcet_loopbound(Lo..Up)), Annotations),
 replace(Lhs, Loop1, Loop2/Lo, Lhs1),
 replace(Rhs, Loop1, Loop2/Up, Rhs1).
```

Fig. 15 Flow transformation specification for loop interchange

designed such that each annotation is also a legal Prolog term. This makes it possible to use Prolog as specification language for the flow information update rules.

Figure 15 shows the rules for *loop interchange*: Processing the optimizations in the trace, these rules are applied to each annotation. The first two arguments contain information from the optimization trace (the labels of the interchanged loops `Loop1` and `Loop2`). The third argument is the list of all annotations before the transformation. This is followed by the annotation to be transformed: An annotation is associated a location (=a marker) and a body: Valid bodies are loopbounds, constraints and markers. Markers are labels to basic blocks or specific edges in the CFG and follow a unique hierarchical naming scheme that encodes the location in the abstract syntax tree of the program. The last argument is unified with a list of annotations generated by the transformation rule. The first two clauses in Fig. 15 swap the loopbounds of the interchanged loops. The third clause updates flow constraints similar to (3). The helper predicate `replace` is used to replace all occurrences of `Loop1` on the left-hand-side with `Loop2` divided by its lower loop bound and vice versa. After the rules are applied, the resulting constraints are normalized to remove the division operator and allow further processing by other tools.

8.1.2 Benchmarks

To give an impression of the potential of the high-level optimizations, Fig. 16 shows results for the standardized set of WCET analysis benchmarks collected by Mälardalen University (MRTC Benchmarks 2009). Using the benchmarks that could be fully annotated at source level by an unassisted TuBound, the diagram shows the WCET bound for several combinations of optimizations. The WCET bound calculation was done using the $calc_wcet_{C167}$ back end of TuBound. This back end uses the 16-bit Infineon C167 microcontroller as target hardware (INFINEON 2000). The C167 processor features a four-stage pipeline and a jump cache. The jump cache remembers the last jump target and is used for branch prediction. Due to the lack of a data cache, alignment optimizations such as loop blocking do not improve the performance on the C167. From left to right the diagram shows four columns for each benchmark:

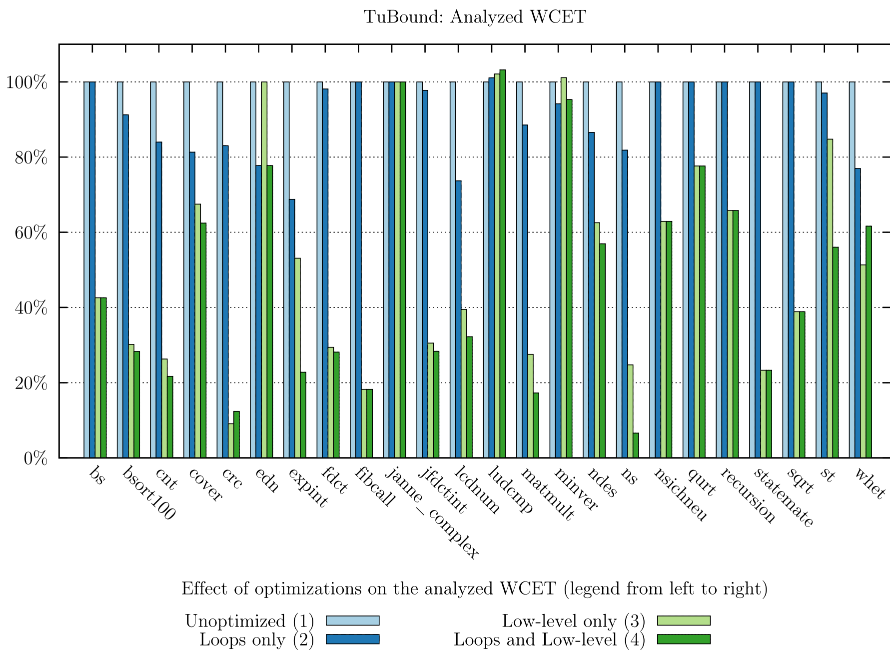


Fig. 16 Benchmark results (*vertical bars* show analyzed WCET relative to the unoptimized program)

1. WCET bound of the unoptimized program.
2. WCET of the high-level loop optimized program.
3. WCET with low-level control-flow insensitive optimizations.
4. WCET when combining both types of optimizations.

Each value is scaled by the WCET bound of the unoptimized program (column 1). Loop optimizations are performed by the source-to-source optimizer which uses the upper and lower loop bound information found by TuBound. For this reason the applied loop optimizations improve the analyzed WCET in most cases. The low-level optimizations are performed by the target compiler and do not to alter the control flow any more.

The benchmarks indicate that the potential for optimizations is significant, especially when keeping in mind that the used high-level optimizations (loop unrolling, fusion, interchange, splitting), generally target the average case performance. Outliers like `whet` show that careful selection of the different optimization phases is very important. This process, however, can be supported by an automatic WCET analysis, which can be used to guide the optimizer by judging the improvement of a program transformation (Lokuciejewski and Marwedel 2009).

The described format of the flow information in the flow-information transformation framework is suitable to annotate intra-procedural flow information. For simplification we use a notation were flow variables represent the overall execution of a program location. To get the most benefit from inter-procedural flow information, the syntax of the flow information has to be extended to support call contexts (Kirner

et al. 2008). Typical inter-procedural code optimizations that require the update of inter-procedural flow information are *procedure cloning* (Lokuciejewski et al. 2008) or *procedure inlining*.

9 Summary and conclusion

So far, the industrial acceptance of WCET analysis is limited due to the fact that a developer is forced to examine and understand the *machine code* in case the WCET tool needs additional code annotations to guide the analysis. In this article we presented a novel method to support precise WCET analysis at the machine-code level by taking advantage of the flow information provided at the *source-code level*. We described a transformation framework that is capable of transforming more generic flow information from source-code level to machine-code level than previous approaches with even better precision. In our framework it is possible to support arbitrary code transformations. Since this approach is based on the semantics of the performed code transformations it can be systematically integrated into a compiler. With the TuBound tool we have shown how to use the approach with a commercial-off-the-shelf (COTS) compiler by using source-to-source code transformations and deactivating control-flow changing optimizations in the COTS compiler.

Another benefit of the presented approach is that the update of flow information for every code transformation is expressed by a short sequence of primitive flow-information update functions. Therefore, this approach requires relatively small implementation effort. All that is needed is a compiler that maintains an abstract representation of the code in form of a standard control-flow graph with an additional loop scope hierarchy. As every code transformation is handled in a safe way, this approach allows the timing analysis tool to calculate safe bounds for both the WCET and BCET.

The presented flow-information transformation framework supports inter-procedural program transformations and flow constraints. To take more advantage of this feature, it is advisable to include support for specifying call-context sensitive flow information. Before extending the framework, appropriate types of flow information have to be developed. As a first step in this direction, we announced the *WCET Annotation Language Challenge* in 2007 to get feedback from the community towards a common annotation language (Kirner et al. 2007).

The currently used flow-information update functions have been developed manually with the help of a graphical transformation-description language that we developed for that purpose. As a future research it would be useful to automate the construction of flow-information update functions based on a formal description of the code transformation.

Acknowledgement The research leading to these results has received funding from the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Compiler-Support for Timing Analysis" (COSTA) and within the research project "Sustaining Entire Code-Coverage on Code Optimization" (SECCO).

The authors would like to thank Martin Schöberl for his valuable comments on earlier versions of this article.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

References

Allen R, Kennedy K (2002) Optimizing compilers for modern architectures. Morgan Kaufmann, San Mateo

Ball T, Larus JR (1996) Efficient path profiling. In: Proc IEEE international symposium on microarchitecture, pp 46–57

Cullmann C, Martin F (2007) Data-flow based detection of loop bounds. In: Proc 7th international workshop on worst-case execution time analysis, Pisa, Italy

Drewes F, Hoffmann B, Plump D (2002) Hierarchical graph transformation. J Comput Syst Sci 64:249–283. Short version in Proc FOSSACS 2000, LNCS, vol 1784

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

Engblom J, Ermedahl A, Altenbernd P (1998) Facilitating worst-case execution time analysis for optimized code. In: Proc 10th Euromicro real-time workshop, Berlin, Germany

Ferdinand C, Heckmann R, Langenbach M, Martin F, Schmidt M, Theiling H, Thesing S, Wilhelm R (2001) Reliable and precise WCET determination for a real-life processor. In: Proc of the 1st international workshop on embedded software (EMSOFT 2001), Tahoe City, CA, USA, pp 469–485

Gustafsson J, Ermedahl A (1998) Automatic derivation of path and loop annotations in object-oriented real-time programs. Parallel Distrib Comput Pract 1(2)

Gustafsson J, Ermedahl A, Sandberg C, Lisper B (2006) Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In: Proc 27th IEEE real-time systems symposium (RTSS 2006), Rio de Janeiro, Brazil

Healy CA, Arnold RD, Mueller F, Whalley D, Harmon MG (1999) Bounding pipeline and instruction cache performance. IEEE Trans Comput 48(1)

Healy CA, Sjödin M, Rustagi V, Whalley D, van Engelen R (2000) Supporting timing analysis by automatic bounding of loop iterations. Real-Time Syst 18:121–148

Healy CA, Whalley DB (2002) Automatic detection and exploitation of branch constraints for timing analysis. IEEE Trans Softw Eng 28:763–781

INFINEON (2000) C167CR derivatives. 16-bit single-chip microcontroller. User's manual, Version 3.0. Infineon Technologies AG

Jaramillo CI, Gupta R, Soffa ML (1998) Capturing the effects of code improving transformations. In: Proc international conference on parallel architectures and compilation techniques (PACT'98), Paris, France, pp 118–123

Kirner R (2002) The programming language WCETC. Technical Report 02/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr 1-3/182-1, 1040 Vienna, Austria

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

Kirner R (2008) Compiler support for timing analysis of optimized code: precise timing analysis of machine code with convenient annotation of source code. VDM Verlag, Saarbrücken

Kirner R, Puschner P (2001) Transformation of path information for WCET analysis during compilation. In: Proc 13th IEEE Euromicro conference on real-time systems, Delft, The Netherlands, pp 29–36

Kirner R, Puschner P (2003) Timing analysis of optimised code. In: Proc 8th IEEE international workshop on object-oriented real-time dependable systems (WORDS 2003), Guadalajara, Mexico, pp 100–105

Kirner R, Knoop J, Prantl A, Schordan M, Wenzel I (2007) WCET analysis: the annotation language challenge. In: Proc 7th international workshop on worst-case execution time analysis, Pisa, Italy, pp 83–99

Kirner R, Kadlec A, Puschner P, Prantl A, Schordan M, Knoop J (2008) Towards a common WCET annotation language: essential ingredients. In: Proc 8th international workshop on worst-case execution time analysis, Prague, Czech Republic, pp 53–65

Li Y-TS, Malik S (1995) Performance analysis of embedded software using implicit path enumeration. In: Proc 32nd ACM/IEEE design automation conference, pp 456–461

Lim S-S, Bae YH, Jang GT, Rhee B-D, Min SL, Park CY, Shin H, Park K, Moon S-M, Kim C-S (1995) An accurate worst case timing analysis for RISC processors. Softw Eng 21(7):593–604
Lim S-S, Kim J, Min SL (1998) A worst case timing analysis technique for optimized programs. In: Proc 5th international conference on real-time computing systems and applications (RTCSA), Hiroshima, Japan, pp 151–157
Lokuciejewski P (2007) A WCET-aware compiler. Design, concepts and realization. VDM Verlag, Saarbrücken
Lokuciejewski P, Marwedel P (2009) Combining worst-case timing models, loop unrolling, and static loop analysis for WCET minimization. In: Proc 21st Euromicro conference on real-time systems, Dublin, Ireland
Lokuciejewski P, Falk H, Marwedel P, Theiling H (2008) WCET-driven, code-size critical procedure cloning. In: Proc 11th international workshop on software and compilers for embedded systems, Munich, Germany, pp 21–30
Lokuciejewski P, Cordes D, Falk H, Marwedel P (2009) A fast and precise static loop analysis based on abstract interpretation program slicing and polytope models. In: Proc international symposium on code generation and optimization, Seattle, USA
Marlowe TJ, Masticola SP (1992) Safe optimization for hard real-time programming. In: Proc 2nd international conference on systems integration (ICSI), pp 436–445
Mok AK, Amerasinghe P, Chen M, Tantisirivat K (1989) Evaluating tight execution time bounds of programs by annotations. In: Proc 6th IEEE workshop on real-time operating systems and software, Pittsburgh, PA, USA, pp 74–80
MRTC Benchmarks (2009) Mälardalen research and technology centre WCET benchmarks. Web page http://www.mrtc.mdh.se/projects/wcet/. Accessed online in January 2010
Muchnick SS (1997) Advanced compiler design & implementation. Morgan Kaufmann, San Mateo
Park CY (1993) Predicting program execution times by analyzing static and dynamic program paths. Real-Time Syst 5(1):31–62
Park CY, Shaw AC (1991) Experiments with a program timing tool based on a source-level timing schema. Computer 24(5):48–57
Pop S, Cohen A, Bastoul C, Girbal S, Silber GA, Vasilache N (2006) GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In: Proc 4th GCC developer's summit, pp 179–198
Prantl A (2007a) The CoSTA transformer: integrating optimizing compilation and WCET flow facts transformation. In: Proceedings 14 Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'07)
Prantl A (2007b) Source-to-source transformations for WCET analysis: the CoSTA approach. In: Proc 24. Workshop der GI-Fachgruppe Programmiersprachen und Rechenkonzepte, Christian-Albrechts-Universität zu Kiel, Bad Honnef, Germany
Prantl A, Schordan M, Knoop J (2008) TuBound—a conceptually new tool for worst-case execution time analysis. In: Proc 8th international workshop on worst-case execution time analysis, Prague, Czech Republic
Puschner P, Schedl AV (1997) Computing maximum task execution times—a graph-based approach. Real-Time Syst 13:67–91
Quinlan DJ, Schordan M, Miller B, Kowarschik M (2004) Parallel object-oriented framework optimization. Concurr Comput Pract Exper 16(2–3):293–302
Ramalingam G (2002) On loops, dominators, and dominance frontiers. ACM Trans Program Lang Syst 24(5):455–490
Schulte D (2007) Flow Facts für WCET-optimierende Compiler: Modellierung und Transformation. VDM Verlag, Saarbrücken
Vrchoticky A (1994) Compilation support for fine-grained execution time analysis. In: Proc ACM SIGPLAN workshop on language, compiler and tool support for real-time systems, Orlando, FL
Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckman R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenstrom P (2008) The worst-case execution time problem—overview of methods and survey of tools. ACM Trans Embed Comput Syst (TECS) 7(3)
Younis MF, Marlowe TJ, Tsai G, Stoyenko AD (1996) Toward compiler optimization of distributed real-time processes. In: Proc 2nd international conference on engineering of complex computer systems, pp 35–42

Raimund Kirner is Assistant Professor at the Institute of Computer Engineering of the Vienna University of Technology. The research focus of Kirner is on system reliability, especially worst-case execution time analysis of real-time programs, including compiler support and design methodologies to make systems predictable. Currently, Raimund Kirner is principal investigator of three research projects. R. Kirner chaired the PC of WDES 2006 and WCET 2008. He is a member of the IEEE Computer Society, the ACM, the IFIP WG 10.2 (Embedded Systems) and the Austrian Computer Society (OCG).

Peter Puschner is a professor in computer science at Vienna University of Technology. His main research interest is on hard real-time systems for safety-critical applications, with a focus on the worst-case execution time (WCET) analysis of real-time programs and software/hardware architectures for time-predictable computing. He has published more than 80 refereed conference and journal papers and was a guest editor for the special issue on WCET analysis of the Kluwer (now Springer) International Journal on Real-Time Systems in 2000. P. Puschner chaired the PC of ISORC 2003 and ECRTS 2004 and was the general chair of the Euromicro Conference on Real-Time Systems 2002 and ISORC 2004. P. Puschner is a member of the IEEE Computer Society, IFIP working group 10.2 on Embedded Systems, Euromicro, the OCG (Austrian Computer Society), and the Marie-Curie Fellowship Association.

Adrian Prantl is Ph.D. student at the Institute of Computer Languages at Vienna University of Technology. He is currently employed for the FWF-funded project "Compiler Support for Timing Analysis" (CoSTA) were he is working on high-level analysis and optimizing program transformations. He is the main author of the TuBound WCET analysis tool, which supports source-based flow annotations and annotation-aware source-to-source program optimizations. He is also a contributing author of SATIrE, an open-source framework for static program analysis of C++ programs.