A Fast and Precise Static Loop Analysis based on Abstract Interpretation, Program Slicing and Polytope Models

Paul Lokuciejewski, Daniel Cordes, Heiko Falk, Peter Marwedel
Computer Science 12
Embedded Systems Group
D-44221 Dortmund, Germany
FirstName.LastName@tu-dortmund.de

***Abstract*—A static loop analysis is a program analysis computing loop iteration counts. This information is crucial for different fields of applications. In the domain of compilers, the knowledge about loop iterations can be exploited for aggressive loop optimizations like Loop Unrolling. A loop analyzer also provides static information about code execution frequencies which can assist feedback-directed optimizations. Another prominent application is the static worst-case execution time (WCET) analysis which relies on a safe approximation of loop iteration counts.**

In this paper, we propose a framework for a static loop analysis based on Abstract Interpretation, a theory of a sound approximation of program semantics. To accelerate the analysis, we preprocess the analyzed code using Program Slicing, a technique that removes statements irrelevant for the loop analysis. In addition, we introduce a novel polytope-based loop evaluation that further significantly reduces the analysis time. The efficiency of our loop analyzer is evaluated on a large number of benchmarks. Results show that 99% of the considered loops could be successfully analyzed in an acceptable amount of time. This study points out that our methodology is best suited for real-world problems.

I. Introduction

Knowledge about loop iteration counts is mandatory for a large number of different static program analyses. One of the most prominent candidates are compiler-based loop transformations. Since programs spend most of their execution time in loops, their optimization is considered to be one of the most effective compiler tranformations. This optimization class typically relies on the knowledge of how often a loop is executed. Examples for loop transformations are *Loop Unrolling* [1], *Loop Tiling* or *Software Pipelining* [2]. Without knowing the loop iteration counts at compile-time, these optimizations can not be applied and optimization potential is not explored.

A related domain of compiler optimizations which profit from the availability of loop iteration counts is *feedback-directed optimization (FDO)*. FDOs are typically based on external data about execution frequencies gathered at program run time. A compiler exploits this profiling information to focus its optimization efforts on the most frequently executed program portions. An increasing academic interest as well as a growing commercial acceptance of FDO emphasizes the optimizations' future potential [3]. However, the main problem with profiling information is its program model which is valid for a particular set of input data. Thus, for different input sets, the program must be re-compiled [4] to gather valid profiling information. One solution to this problem is the application of a static loop analyzer which computes an approximation of possible program values valid for all possible input data. These input-invariant approximations allow the computation of execution frequencies which are valid for varying input sets.

The third major class of static program analyses which benefits from a reliable loop analysis is the static worst-case execution time analysis. The WCET is a key parameter for systems that must meet real-time constraints. Especially for safety-critical systems, e. g. in the avionic and automotive domain, the adherence of the WCET must be ensured to avoid system failure potentially leading to a disaster. Also for scheduling algorithms and an effective hardware design, the WCET is a crucial issue. A static WCET analysis relies on safe upper bounds of loop iterations in order to derive a finite WCET estimation. The majority of static WCET analyzers tries to detect loops in the object code by pattern matching. However, these pattern-based approaches are limited and highly depend on the compiler used for the code generation. Moreover, they often fail in the analysis of more complex loops. Thus, WCET analyzers typically offer the possibility to provide loop information manually by the user. In order to overcome this tedious and error-prone process, an automatic and universal loop analysis is highly desired [5]. The analysis has to meet strict requirements. On the one hand, the results must be safe, i. e. underestimated loop iteration counts are not acceptable since this would yield an invalid WCET estimation. On the other hand, the loop analysis results should be as precise as possible in order to keep the overestimation of the estimated WCET minimal.

In this paper, we present an automatic static loop analysis

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

for a high-level code representation which can be employed to either assist compiler optimizations or to provide mandatory information for a WCET analysis. It is based on a theory of a sound approximation of the program's semantics called *Abstract Interpretation* [6] which considers all potential values a program variable might have at any program point. Since all possible input data is assumed, the analysis is not sensitive to a particular input set. Thus, it is more reliable than profiling information and generates safe parameters for a WCET analysis. The main contributions of this paper are as follows:

1) We introduce a fast static loop analysis which produces highly precise results about loop iteration counts and execution frequencies of code elements.
2) Our analysis overcomes the restrictions of pattern-based approaches by performing an interprocedural data-flow analysis on source code level.
3) A technique called *Interprocedural Program Slicing* is applied in order to simplify the analyzed code and allow a faster analysis.
4) The analysis exploits mathematical concepts of polytopes to significantly accelerate the conventional loop analyses found in the literature.
5) For the first time, a static loop analyzer is applied to a wide collection of different real-world benchmarks. We show that our analyzer generates precise results in an acceptable amount of time. This can be considered as a proof of concepts and indicates the loop analyzer's practical use for other projects.

The rest of this paper is organized as follows: Section II gives a survey of the related work. An introduction to the theory of Abstract Interpretation which is the basic framework for our loop analysis is presented in Section III. Section IV introduces the concepts of the interprocedural Program Slicing, followed by our novel concepts of the polytope-based loop evaluation in Section V. In Section VI we describe our experimental environment and show how the previously described techniques are combined. Results achieved on different real-world benchmark suites are presented in Section VII. Finally, Section VIII summarizes this paper and gives directions for future work.

II. Related Work

Most of the previous works relying on known loop iteration counts implicitly assume the existence of a loop analyzer. However, static loop analysis is only sparsely dealt within today's literature. An overview of related work is presented in the following.

[7] presents a pattern-based approach to determine the loop iteration counts for an assembly program. The analysis exclusively evaluates instructions which represent loops while other instructions are ignored. In that way, loops which rely on function parameters can not be analyzed. To overcome this problem, the authors provide a mechanism which allows the specification of value ranges for unknown variables making their analysis semi-automatic.

The approach developed in [7] has been adapted to programs written in the high-level language C by [8]. Again, the loop analysis does not automatically succeed for all types of loops. Mandatory information that can not be extracted during the static analysis must be provided by the user in the form of source code annotations.

In contrast to the pattern-based loop analyses, [9] uses an interprocedural data-flow based loop analysis at assembly level. The advantage of this approach is that the success of the static analysis does not strictly rely on pre-defined code patterns a particular compiler generates but on the semantics of the instruction set for a specific target machine. As stated by the authors, their analysis works best for well-structured loops and where the loop counter is modified only by a simple addition.

A different approach for a fully automatic static loop analysis at source code level was described in [10]. The authors involve a data flow analysis which is based on the theory of *Abstract Interpretation* that enables an automatic extraction of information about all possible program executions. Their data flow analysis determines a precise approximation of variable values which are exploited to calculate the number of loop iterations. The approximation is achieved by representing actual values as intervals. By this loss of precision, the concrete program's semantics become decidable and make an answer to the question about loop iterations possible.

This work was employed in [5] to assist a static WCET analysis. It was extended by supporting a determination of loop bounds for nested loops as well as a detection of *infeasible paths*, i. e. paths that are not taken in particular execution contexts of the program and which should thus be excluded from the WCET analysis to avoid WCET overestimation.

Further improvements to this loop analysis were presented in [11]. The authors combine different standard program analyses like *Program Slicing* and *Invariant Analysis* techniques in order to accelerate the loop analysis. This publication is close to the work presented in this paper. We also use Program Slicing to reduce the amount of code to be analysed. However, we additionally exploit Slicing to leverage our novel polytope-based loop evaluation as will be described in more detail in Section V. Moreover, unlike our loop analyzer the authors state that their analysis can not handle recursive code and that their analysis fails for more complex loops.

III. Abstract Interpretation

Abstract Interpretation is a theory of sound approximation of mathematical structures. In computer science, it is mainly used to approximate semantic models of computer systems. The theory's main field of application is the static program analysis which exploits the fact that undecidable or very complex problems can be solved when incomplete results are tolerated [6].

In general, the static loop analysis is not decidable since it includes the proof of termination. Thus, for concrete program semantics an automatic loop analysis which determines loop iteration counts for any type of loops is only feasible for a restricted set of programs. However, by introducing abstract semantics, which is a superset of the concrete semantics of the program that covers all possible concrete cases, the loss of information makes the analysis computable. We take advantage of this fact and build our static loop analysis upon the theoretical concepts of the classical Abstract Interpretation discussed next.

A. Classical Abstract Interpretation

The fundamental idea behind Abstract Interpretation is to find an appropriate compromise between the precision of the analysis and analysis time. This reduction of information is achieved by mapping a possibly infinite set of program states typically consisting of the value of the program counter (program point) and a set of variables (or memory locations) into a finite set of *abstract states*. This mapping is achieved by the function α called *abstraction function*:

$$\alpha : L \rightarrow M \tag{1}$$

while *L* and *M* are complete lattices. Typically, computations on *M* are less complex than on *L*. The counterpart to α is the *concretization function* γ with

$$\gamma : M \rightarrow L \tag{2}$$

It maps elements of lattice *M* into lattice *L*. During a static program analysis, computations are performed on the more abstract sets being part of *M* before finally transforming them into the original (concrete) states via the concretization function. The abstract states which represent *M* are specified by their *abstract domain*. Different domains are known in literature like the convex polyhedra domain [12], the congruence domain [13] or the octogan domain [14] which provide different representations of the variable values held in an abstract state. These domains represent different trade-offs between precision and complexity. For our loop analysis, we use the interval domain [6] which represents a set of variable values by a single interval I:

$$I = \{\bot\} \cup \{[z_1, z_2] \mid z_1 \leq z_2\} \tag{3}$$

for $z_1 \in \mathbb{Z} \cup \{-\infty\}, z_2 \in \mathbb{Z} \cup \{\infty\}$ and $\bot$ representing an empty interval.

Besides the representation of values, the *abstract operators* define computations in the abstract domain. As for the concrete semantics of a program, which might modify their values with n-ary operators, its abstract domain generated via the abstraction function α must provide equivalent operators. In case of our interval domain, the abstract operators are based on interval arithmetics.

The objective of a static analysis based on Abstract Interpretation is to assign sets of possible variable values (abstract states) to edges of a control flow graph *(CFG)*. The CFG consists of five different node types which represent program points:

1) A designated *start* and *end* node representing the beginning and end point of a CFG.
2) *Expression nodes* representing different expression types found in a concrete semantic model.
3) *Condition nodes* representing forks in a control flow, i. e. this type of nodes has one incoming and two outgoing edges.
4) *Join nodes* merging two paths of the CFG, i. e. these nodes have two incoming and one outgoing edge.

Based on the CFG, a transition system is constructed. It defines how an abstract state is transfered into another state at program point $\mathcal{P}$:

$$\tau : \mathcal{P}(STATES) \rightarrow \mathcal{P}(STATES) \tag{4}$$

The transition system τ is used to construct a system of equations which define the assignment of abstract states to program points. A solution is found by a fixed-point iteration. It begins with the least possible assignment $\tau(\bot)$ where $\bot$ is the *least element* representing $\emptyset$. The fixed-point iteration continues as long as a further application of τ does not compute a new state:

$$\tau^{n-1} = \tau^{n} \tag{5}$$

The problem of Abstract Interpretation is that a termination of the fixed-point iteration can not be guaranteed. Moreover, due to the nature of Abstract Interpretation which iteratively simulates each state transition, the fixed-point iteration can consume a significant amount of time for loops with large iteration counts. To overcome both problems, the *widening operator* ∇ [6] can be applied. It allows the substitution of an infinite sequence of abstract states $(\hat{S}\ ((\hat{S}^{i})^{i \in \mathbb{N}})$ of a program point q by a finite and safely approximated sequence $((\hat{S}^{m}_{\nabla})^{m \in \mathbb{N}})$. Its application typically enlarges the abstract states during the fixed-point iteration leading to a correct but also over-approximated solution which might become infeasible as result for many applications. Thus, a *narrowing operator* $\triangle$ was introduced by [15] trying to restrict the over-approximation afterwards.

B. Modified Abstract Interpretation

The classic widening operator can not be applied for our static loop analysis since it yields significantly over-approximated loop iteration counts, in the worst-case even ∞, which are insufficient for compiler optimizations or a static WCET analysis. In addition, the iterative behavior of the classical Abstract Interpretation might slow down the analysis such that it becomes impractical. In particular, this can be observed for the analysis of program loops with high iteration counts for which each loop iteration is separately simulated. This section proposes a solution for both problems.

The main innovation of this work is the development of a static loop analysis that on the one hand exploits the formal framework of Abstract Interpretation but on the other hand avoids its iterative behavior during the loop analysis. We propose a non-iterative static loop evaluation which determines loop iteration counts and variable values with a purely static analysis by iterating through the loop body exactly once. If this efficient approach succeeds for the analysis of a loop, the classical Abstract Interpretation is omitted for this loop, otherwise the classical approach is applied.

In the following, we first describe the extensions to the basic concepts of the classical Abstract Interpretation which are required for the integration of the fast non-iterative loop evaluation. The latter will be described in Section V.

First of all, the classical set of five node types described previously must be extended by further node types which are required for the analysis of programs written in an imperative high-level language like ANSI-C. The new node types comprise for example compound statements, function calls or loop statements.

Moreover, the classical approach begins with the construction of a complete transition system for the five node types. This is a closed model for which the solution is found by one particular approach, the fixed-point iteration. In contrast, our analysis requires a more flexible model of the transition system. We do not consider the analyzed CFG as a closed monolithic system but subdivide the graph into smaller chunks for which a separate transition system is constructed and solved. This enables an easy exchange of the solvers. In our framework, our novel non-iterative static loop evaluation is applied to transition systems that represent loops.

Like the classical approach, our analysis also begins at the *start* node of the CFG and traverses the graph during its static program analysis. Depending on the encountered node type, a particular set of rules which is based on Abstract Interpretation is applied. The rules manage the computation of safe abstract states at the given program point. Also, more complex node types like those which represent compound statements might consist of nested statements requiring their own rules. Thus, our approach can be considered as a simulation based on Abstract Interpretation which analyses the program in a hierarchical manner. Due to the isolated consideration of particular programming language constructs like loops, the methods used to compute the abstract states can be freely interchanged. This gives us the freedom to either evaluate loops with Abstract Interpretation or perform the evaluation with our more efficient non-iterative static approach.

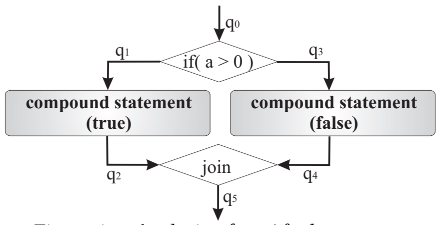


Figure 1. Analysis of an if-$else$ statement

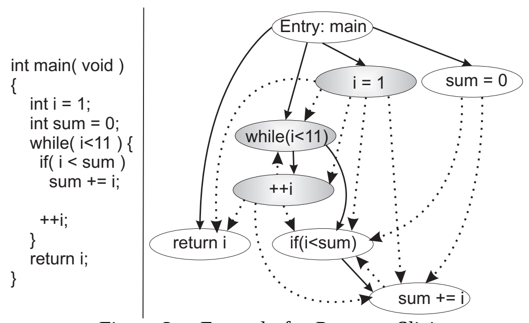
Figure 1 gives an example for an `if-else` statement based on our modified model. q_n represent different program points. Our analysis starts with the abstract state $\hat{S}_{q_0}$. For state $\hat{S}_{q_1}$, all values from $\hat{S}_{q_0}$ which meet the condition $if(a > 0)$ are assigned. Equivalently, $\hat{S}_{q_3}$ is handled. These two abstract states are passed as input to the compound statement nodes representing the condition's `then`- and `else`-part, respectively. For these complex nodes, further rules are applied and finally their outputs represented by the abstract states $\hat{S}_{q_2}$ and $\hat{S}_{q_4}$ are merged at the *join* node leading to the computation of the abstract state $\hat{S}_{q_5}$.

To determine the number of loop iterations, a counter variable is incremented each time our abstract simulation enters the first program point within the loop body. If the CFG edge leaving the loop is taken for the first time, the current value of the counter variable represents a safe lower bound for the number of loop iterations. The counter is incremented as long as the loop body is entered and a fix point for its abstract states is not achieved. If the fixed-point iteration has found a solution, the value of the counter variable represents a safe approximation for the upper bound of the loop iterations. When the number of loop iterations depends on a variable represented by an interval with unequal bounds, e. g. generated by a *join* node, then the number of loop iterations might be variable in the sense that the lower and upper bounds are different.

IV. Interprocedural Program Slicing

Program Slicing is a program analysis which finds statements of a program that are relevant for a particular computation. The technique was introduced in [16]. It defines how a given program can be sliced w.r.t. a *slicing criterion*. By definition, a slicing criterion is defined by a pair $\langle q, V\rangle$ where q is a program point, as already described in the last section, and V is a subset of program variables at q. The slice for a given program w.r.t. $\langle q, V\rangle$ defines a subset of the program containing all statements which might affect the variables in V, i. e. variables that might either be used or defined at q. The fields of applications for Slicing are manyfold. Nowadays, Program Slicing is used for debugging [17], [18], software maintenance [19], compiler optimizations [20] or for flow analyses [21].

An intermediate program representation that enables an efficient Program Slicing is the *Program Dependence Graph*



```
int main( void )
{
  int i = 1;
  int sum = 0;
  while( i<11 ) {
    if( i < sum )
      sum += i;

    ++i;
  }
  return i;
}
```

Figure 2. Example for Program Slicing

(PDG) [22]. It is a directed graph that makes both control and data dependencies visible. Data dependencies represent essential data flow relationships of a program while the control dependencies, usually derived from a CFG, serve to indicate relevant control flow relationships. Taking both types of dependencies into account allows the efficient determination of a program slice with respect to a slicing criterion.

An example for Program Slicing is depicted in Figure 2. The PDG of the program on the left-hand side of the figure consists of nodes representing statements and edges modelling data dependencies (dotted arrows) as well as control dependencies (solid arrows). By definition, each PDG contains a distinguished entry node that serves as a starting point for the analysis. The program is sliced with respect to the $while$ loop exit condition $i < 11$, i. e. i is a variable of the slicing criterion. The corresponding program slice can be found by reversing all edges and performing a *Depth First Search* algorithm starting at the node specified by the slice criterion. All nodes that are visited, are part of the slice. They are marked in grey in the figure and are those statements being relevant for the loop analysis. Statements in the remaining nodes can be omitted for the loop analysis.

The main drawback of the PDG is its restriction to an intraprocedural slicing. Since most real applications typically consist of numerous procedures, they can not be handled by PDGs. To overcome this problem, [23] extended the PDG by concepts which enable to cross the boundaries of procedures. The interprocedural program representation is called *System Dependence Graph (SDG)*. It is based on a PDG and models procedure calls as well as the transfer of values via function parameters and return values.

In our framework, we apply Program Slicing w.r.t. loop exit conditions. By taking all relevant data and control dependencies into account, the resulting program slice contains all statements that are involved in the determination of loop iteration counts. Our slicing is accompanied by a context-sensitive alias analysis to support pointers. Contexts introduce a distinction between different calls to a particular function, thus allowing a more precise analysis.

Program Slicing is run before the actual loop analysis with Abstract Interpretation for two purposes. First, like [21], we want to accelerate the loop analysis. By slicing the code in advance, all superfluous statements are eliminated. By considering the relevant subset of the program, the fixed-point iteration can usually find a solution in a reduced amount of time. Second, the integration of our fast non-iterative loop evaluation (see Section V) requires simple loop bodies to infer final abstract states without repetitive iterations. Loop bodies of original applications are often too complex for this static evaluation but after a code simplification via Slicing, the required prerequisites are met. Thus, Program Slicing can be considered as a mandatory step to establish potential for the application of our extended loop evaluation described in the following.

V. Polyhedra-based Static Evaluation

A *polyhedron* is an N-dimensional geometrical object defined as a set of linear inequations:

$$P := \left\{ x \in \mathbb{Z}^N \mid Ax = a, Bx \geq b \right\} \tag{6}$$

for $A, B \in \mathbb{Z}^{m \times N}$ and $a, b \in \mathbb{Z}^m$ and $m \in \mathbb{N}$. A polyhedron is called a *polytope* if

$$|P| < \infty \tag{7}$$

Polytopes are often employed in compiler optimizations since they can be exploited to represent loop nests and affine condition expressions. Their formal definition enables efficient code transformations. Typical fields of application are program execution parallelization or the optimization of nested loops [24].

In this work, polytope models are applied for two purposes. On the one hand, they allow a precise computation of abstract states after processing a *condition* node (see Figure 1) which splits the control flow. On the other hand, for loops that are modelled by a polytope, its number of iteration counts can be determined statically. This knowledge is exploited by our non-iterative loop evaluation. The remainder of this section describes these two issues in more detail.

A. Efficient Polytope-based Condition Evaluation

The practical use of this approach will be motivated by an example. Assume that this condition is given:

$$\textit{if}(\overline{(2 * i + 2 \leq j \text{ NAND } j > 15)})$$

and that the Abstract Interpretation calculated the following abstract state before the evaluation of the condition:

$$\hat{S}_q = \{i \rightarrow [5..15]\,, j \rightarrow [10..20]\}$$

The evaluation of the condition $2 * i + 2 \leq j$ yields for the state $\hat{S}_{q_{true}}$ and the values of $\hat{S}_q$ the condition $[12..32] \,\hat{\leq}\, [10..20]$. By definition, the result of the abstract operator $\hat{\leq}$ is neither $true$ nor $false$, thus an explicit computation is not possible. It must be assumed that both outgoing edges of the *condition* node are taken. Due to the missing result of $\hat{\leq}$, the value ranges for i and j in the states

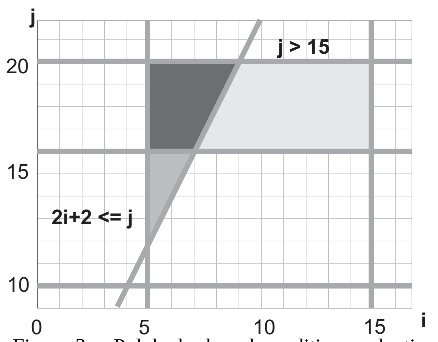


Figure 3. Polyhedra-based condition evaluation

$\hat{S}_{q_{true}}$ and $\hat{S}_{q_{false}}$ can not be precisely approximated. Thus, a conservative approximation for i and j in both states is to assume that their value ranges correspond to those computed for $\hat{S}_q$. Such a result is overestimated and often useless for further computations.

Since an effective determination of variable values is not feasible with the means of classical Abstract Interpretation, we make use of polytopes to determine precise values for the variables i and j. To compute polytopes, they must be transformed into the normalized form from Definition 6. This transformation is performed in several steps which are described in the following.

Transformation of Conditions

The first step is a transformation of the conditions into an equivalent form of type $if(C_1 \oplus C_2 \oplus ... \oplus C_n)$ where C_x are conditions. The set of operators $\oplus$ consists of the elements $\{\wedge, \vee\}$. In addition, each expression can be negated by the NOT operator ($\overline{x}$). With the means of *propositional calculus* and *De Morgan's laws*, it is possible to transform each condition in this equivalent form.

In our example condition, the $NAND$ operator does not belong to the set $\oplus$ of supported operators and must thus be eliminated by the transformation:

$$x \text{ NAND } y = \overline{x \wedge y}$$

Applying this rule to our example, we obtain the transformed condition:

$$\textit{if}(\overline{\overline{(2*i+2 \leq j \wedge j > 15)}})$$

According to De Morgan's laws:

$$\overline{(a \wedge b)} = \overline{a} \vee \overline{b} \text{ and } \overline{(a \vee b)} = \overline{a} \wedge \overline{b}$$

the example condition can be further transformed into

$$\textit{if}(2*i+2 \leq j \wedge j > 15)$$

This condition corresponds to the form $if(C_1 \oplus C_2 \oplus ... \oplus C_n)$, thus finishing the first transformation phase.

In a second step, each condition C_i must have the normalized form $C_x = \sum_{l=1}^{N}(c_l * i_l) + c \geq 0$ for constant values c_l, c with c_l, $c \in \mathbb{Z}$ and i_l representing variables within the conditions. Each affine condition using the comparators $\oplus \in \{<, \leq, >, \geq, =\}$, can be transformed into this normalized form [25]. Normalizing our example condition results in

$$\textit{if}(-2*i+j-2 \geq 0 \wedge j-16 \geq 0)$$

Its form allows the construction of the two polytopes $-2*i+j-2 \geq 0$ and $j-16 \geq 0$ which are combined by the intersection operator $\cap$. These conditions are depicted in Figure 3 as polytopes and their intersection is marked as dark area.

The final step is to approximate valid variable values for the abstract state that is entered if the condition is evaluated as $true$. The final polytope (intersection) encompasses all combinations of variable values that satisfy the condition. A safe approximation of values is defined by the extrema of the polytope, namely its corner points which are $(5,16), (7,16), (9,20), (5,20)$. Performing a join over the coordinates, the abstract state $\hat{S}_{q_{true}}$ is defined as:

$$\hat{S}_{q_{true}} = \{i \rightarrow [5..9], j \rightarrow [16..20]\}$$

The state $\hat{S}_{q_{false}}$ is computed equivalently. Compared to the solution found by the classic Abstract Interpretation which defines the complete value range for i and j, our approach significantly increases the precision.

B. Polyhedra-based Non-iterative Loop Evaluation

A weak point of the classic Abstract Interpretation is its iterative behavior. To overcome this problem, we determine the number of loop iterations statically based on a polyhedral model and use this knowledge to infer final values of variables modified in the loop without iterating over them repetitively. Another solution to this problem was presented in [26]. The authors use a graph formulation and adapt chaotic iteration to propagate information along edges of the graph until a fixpoint is reached.

The motivation for our new approach comes from the observation that a large number of loops consists of statements that do not affect the calculation of loop iterations. A typical example are applications with initialization procedures found in many embedded system applications. The main task of such procedures is the initialization of arrays and other data structures. After the initialization, this data is involved in the computation of output data, e. g. an output stream of an image compression, but is not influencing the execution frequency of loops. Using Program Slicing as described in Section IV, those statements are recognized to be meaningless for the loop analysis and are removed for further evaluation. The result are often loops with almost empty loop bodies. Unlike Abstract Interpretation that has to simulate each loop execution with the fixed-point iteration, our static evaluation requires just a single iteration over the loop body.

Loops that are analyzed using the static evaluation must meet particular constraints specifying the structure of the loop and the type of statements within the loop body. Before the static evaluation begins, loop headers as well as their bodies are scanned and if all conditions are satisfied, the evaluation is performed. The goal of the evaluation is to determine the final state of a variable, which is iteratively modified within a loop, in a single step. The important prerequisites for such a *single-step evaluation* are discussed in the following.

Constraints for Single-Step Evaluation

The first class of requirements concerns the structure of loops including their conditional statements, e. g. if statements. These restrictions are imposed by the polytope models and their violation would make a polytope evaluation infeasible, i. e. it would be not possible to determine how often a loop iterates and how often loop body statements within a condition are executed. This inhibits our single-step statement evaluation. The requirements of this class concern loop exit conditions which must either depend on a constant, a non-modified variable within the loop, or a single variable. Moreover, it must be ensured that all condition statements are affine expressions. It should be noted that these constraints are often met by well-structured loops found in many applications, thus they do no inhibit a successful application of our non-iterative loop evaluation.

The second class of constraints refers to the loop body statements. If they are assignment statements, they must be transformable into the form `=`, `+=` or `-=` to ensure that the variable is increased in each loop iteration by a constant value. Moreover, structs and pointers are not supported in our current version. Again, these requirements must be met to ensure that a single-step evaluation can be performed. Even though these constraints seem to be highly restrictive, our experiments have shown that sliced loop bodies often satisfy them.

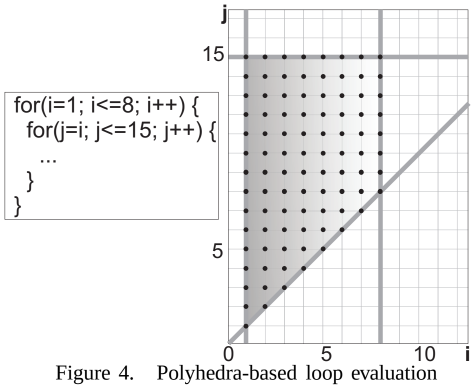
Ehrhart Polynomial Evaluation

If the conditions are met, the loop iteration counts required for the evaluation are statically determined in the next step. This step serves to determine the execution frequencies of loop statements in order to allow the evaluation of their iterative execution in a single analysis step. The problem of finding the loop iteration counts is equivalent to computing the number of integer points in a (parametric) polytope. To efficiently count the integer points, *Ehrhart* polynomials [27] are used. Before counting integer points, the polytope P of the dimension n used to represent a loop is transformed into a parametric polynomial E:

$$E_P(k) = a_n * k^n + a_{n-1} * k^{n-1} + ... + a_0$$

for $k \in \mathbb{Z}$ since a set of integer points is searched.

Taking all integer points of a polytope might yield an over-approximation for many loops. The total number of integer points represents the number of loop iterations if the loop counter is incremented by one. For other modifications to the counter, the number of counted points must be appropriately



```
for(i=1; i<=8; i++) {
  for(j=i; j<=15; j++) {
    ...
  }
}
```

Figure 4. Polyhedra-based loop evaluation

adjusted. Also, additional exit edges that affect the control flow in the loop body, e. g. in the case of $break$ or $continue$ statements, must be taken into account. They are modelled as further polytopes and their intersection with the polytope representing the loop nest yields the precise solution space. For some loops found in real-world benchmarks having an empty loop body after Program Slicing, counting of integer points was already sufficient to determine the loop iteration counts statically.

Example: An example for the determination of iteration counts for a nested loop is depicted in Figure 4. The iteration space for the inner loop depends on the outer loop and its corresponding polytope is indicated by the grey area. Using Ehrhart polynomials, the number of integer points for this polytope representing the inner loop's iterations is 92. For the outer loop, the one-dimensional polytope holds 8 points.

Based on this knowledge of loop iteration counts, the execution frequencies of condition-dependent basic blocks, which might obviously differ from the loop iteration counts, are determined. The conditions are represented as polytopes and an intersection with the loop polytope allows the calculation of their $true$- and $false$-parts.

Static Statement Evaluation

The last step is the static evaluation of statements within the loop based on the loop iteration counts and basic block execution frequencies from the previous step. The goal is to evaluate modifications of variables within the loop like `b+=a` without a repetitive simulation. These final variable values are used to determine iteration counts for loops that are analyzed afterwards. For this purpose, a dependence graph for all variables is constructed and the graph is topologically sorted. Based on this, we can infer a sorted order of executions indicating in which order variables should be statically evaluated. For example, if variable b is incremented by variable a, we first compute a safe approximation for a before evaluating b. For more details, please refer to [28].

Example: To illustrate the combination of both techniques, the exploitation of Ehrhart polynomials and the static statement evaluation, the static analysis of the example code from Figure 5 is briefly discussed. The code contains a conditional assignment statement within a nested loop,

```
for( i =0; i <5; i ++) {
  for( j =0; j <10; j ++) {
    if ( i > 2 )
      a +=2;
  }
}
```

Figure 5. Static polytope-based evaluation

and the abstract state $a \rightarrow [0,5]$ before the outer loop is assumed. From the Ehrhart polynomials we know that the inner loop is executed 50 times, while the basic block within the if-condition is executed 20 times. This means that variable a is incremented 20 times by the constant 2 resulting in $a \rightarrow [40,45]$ after the outer loop. Using Abstract Interpretation, the assignment statement `a+=2` would have to be considered 20 times and not just once.

The novel static evaluation is embedded into our framework based on the modified Abstract Interpretation. The extension should be not considered as a new type of a static loop analysis which is supposed to completely replace the approach based on Abstract Interpretation described in Section III-B. Rather, this static evaluation should be seen as an extension which enables a fast loop analysis and (if possible) is employed instead of the classical approach. If the analysis successfully determines loop iteration counts and if a valid abstract state for the program point following the loop was found, these results are adopted for further computation and the loop analysis proceeds with the next node in the CFG. The determination of a safe and precise approximation of variable values modified in the loop is crucial since the variables might be influencing following loops. Otherwise, if the loop does not meet the aforementioned constraints, the fast evaluation is omitted and the standard approach is applied for this loop.

The loop analysis algorithm is sketched in Figure 6. In the first step, interprocedural Program Slicing is performed with loop exit conditions as slicing criteria. In the next step, all global variables are initialized with the value 0 as defined by ANSI-C. In line 6, the program's entry point is determined which is typically the first statement of the $main$ routine. Next, all statements are analyzed in the order of their execution as long as the entry routine has not been left. Function $analyzeInExecutionOrder$ returns the succeeding statement of statement $current$. Within function $analyzeInExecutionOrder$, the abstract state from the proceeding statement is taken and a modified abstract state is returned depending on the type of statement passed to $analyzeInExecutionOrder$. When the current statement is a loop or condition statement, a polyhedra-based static evaluation as described in Section V is performed. The algorithm returns the program P for which each loop statement is annotated with the determined loop iteration counts.

VI. Experimental Environment

One of the objectives of this paper is to point out that a static loop analysis is an application of practical relevance,

```
 1 Input: Program $P$
 2 Output: Annotated $P$ with Loop Counts
 3 begin
 4   performProgramSlicing($P$)
 5   initializeGlobalVariables($P$)
 6   IR_Stmt $current$ := getProgramEntry($P$)
 7   while($curren \neq$ getProgramExit($P$)
 8     $current$:=analyzeInExecutionOrder($current$)
 9   return $P$
10 end
```

Figure 6. Algorithm for loop analysis

i. e. the implementation of its theoretical concepts based on Abstract Interpretation and polyhedral models performs well for real-world problems. We show not only that the analysis can be employed for most benchmarks in an acceptable time, but also that the majority of considered loops could be precisely analyzed. As far as we know, this paper is the first which provides an overview of results of a loop analysis for a wide range of benchmarks. Other works usually restrict their evaluation to a small set of benchmarks.

To indicate the efficacy of our static loop analyzer, evaluations on a large number of different benchmarks were performed. The benchmarks come from the test suites MRTC WCET Benchmark Suite [29], DSPstone [30], MiBench [31], MediaBench [32], UTDSP Benchmark Suite [33], and our own collection of real-world benchmarks containing miscellaneous applications, e. g. an H263 coder or a G.721 encoder. The different types of the suites were chosen to point out that our loop analysis can successfully handle applications of different domains.

The 96 considered benchmarks are currently also used to test our WCET-aware compiler that will be discussed later. Their complexity ranges between less than one hundred and several thousands lines of code, e. g. 3178 lines in case of the *GSM encoder*. Due to the complexity of some MiBench and MediaBench benchmarks which can not be handled by today's state-of-the-art static WCET analyzers, those benchmarks are excluded from our compiler test suite and are also not evaluated with our loop analyzer.

Our loop analysis can be either used as a stand-alone tool to generate loop information in a human-readable form, or it can be integrated into a compiler and assist compiler optimizations or a static WCET analysis. Figure 7 depicts the workflow for both configurations. After parsing and transforming the C source code into a high-level intermediate representation (IR), the ICD-C IR [34], the static analysis is performed. It begins with Program Slicing that marks statements relevant for loop bound computations. In the following step, the loop analysis based on the modified Abstract Interpretation and polytope models using the library *Barvinok* [35] is performed. At this point, the loop bound information concerning the program under analysis can be generated.

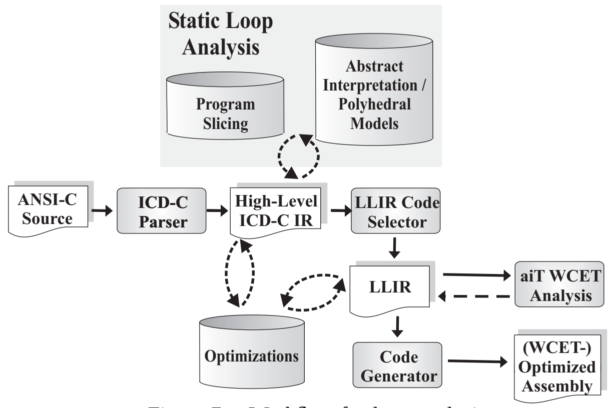


Figure 7. Workflow for loop analysis

The loop data can be also passed to the optimizer that exploits it for high-level optimizations. If desired, this data can be also provided to the low-level intermediate representation which provides it together with the code to a static WCET analyzer. In our framework, the low-level IR and the static WCET analyzer are represented by the tools *LLIR* [36] and aiT [37], respectively. It should be noted that our compiler framework contains mechanisms to hold loop iteration counts consistent. Whenever a code transformation modifies a loop, e. g. Loop Unrolling, the previously determined loop iteration counts will be automatically adjusted without a further invocation of the loop analyzer. More details about the WCET-aware C compiler WCC can be found in [38].

VII. Results

In this section, we give an overview of the evaluation of benchmarks from different benchmark suites. The results show how many loops could be precisely analyzed. We also measured the processor times required for the purely static loop analysis (Section III-B), the analysis combined with Program Slicing (Section IV) and the analysis time for the most advanced mode combining the basic analysis, Program Slicing and the polytope-based non-iterative loop evaluation described in Section V-B. All measurements were performed on a single core of an Intel Xeon CPU with 2.40 GHz and 8 GB RAM. In total, 96 benchmarks were extensively analyzed and evaluated. For the sake of clarity, we provide a comprehensive overview of the results and discuss more interesting cases in more detail in the following.

A. Determination of Loop Iteration Counts

As can be seen in Table I, we have evaluated the precision of our loop analysis. The table indicates for each benchmark suite the number of benchmarks (in brackets), the number of contained loops, the relative portion of loops that could be successfully analyzed (column *Analyzable*) and the relative portion of loops for which our loop analysis produces exact non-over-approximated results (column *Exact*).

The 96 benchmarks contain 707 loops in total. On average, 99% of those loops could be successfully analyzed. This means that for those loops the loop analysis produced results

Table I
Precision of Loop Analysis

Benchmark Suite	# Loops	Analyzable	Exact
MRTC (32)	152	100 %	99.3 %
DSPStone (37)	152	98 %	93 %
MediaBench/ MiBench (6)	162	99 %	98 %
UTDSP (14)	88	100 %	88 %
Misc. (7)	153	100 %	100 %
Total (96) / Average	707	99 %	96 %

in terms of loop iteration counts which might however be over-approximated. The small fraction of 1% loops that could be not analyzed is mainly due to technical restrictions of our alias analysis.

The last column of Table I presents the ratio of loops that could be exactly analyzed, i. e. loops for which precise loop iteration counts were gathered. On average, our analysis produced exact results for 96% of the loops. This was verified by comparing the results with manually determined loop annotations. For the remaining 4%, which include the non-analyzable 1% of loops from the previous column, 3% of the 707 loops could not be exactly analyzed, i. e. the loop iteration counts were afflicted with an over-approximation. The main reason for the imprecision comes from the analysis of pointers which can not always be precisely evaluated in a static analysis. However, most of the over-approximations introduced only a marginal error ranging between 8% and 51% w.r.t. the exact results. Thus, the results are still acceptable. A typical example for an over-approximation is encountered in the MRTC benchmark *fir*. Our analysis computed as lower and upper loop bounds the interval $[8..26]$, while the exact loop bounds are $[17..26]$. In contrast, other works [11] report that they could only analyze 51% of the loops in the MRTC Benchmark suite exactly.

Program Slicing could be successfully applied to all benchmarks. The number of statements irrelevant for the loop analysis ranges between 23% and 88% showing that computations in many programs do not affect the loop iteration counts. 21% of the 707 loops could be analyzed with the polytope-based non-iterative loop evaluation. This number indicates that the constraints for a successful application of this evaluation, as described in Section V-B, are not too restrictive and are often met in real-world applications.

B. Analysis Time

Besides the results about the precision of the analysis, the second crucial issue for static program analyses is their complexity expressed by the analysis time. In general, the analysis times highly depend on the program structure and the loop iteration counts. If our non-iterative loop evaluation (see Section V-B) can not be applied, the analysis based on Abstract Interpretation must consider each loop iteration separately. On average, smaller benchmarks require a few seconds for the analysis, while the analysis time for larger benchmarks such as MiBench's *GSM encoder* takes on average less than 4 minutes.

A time measurement was difficult for some benchmarks, especially for those with a small number of statements to analyze and with a few loop iterations. Due to our efficient implementation of the analysis, which was one of the main goals during the development phase, the analysis time for those benchmarks was a fraction of a second. For those cases, a comparison between different configurations of the analysis combined with Program Slicing or the non-iterative loop evaluation is difficult to measure. Thus, significant speed-ups of the conventional analysis based on the modified Abstract Interpretation via Slicing and the polytope evaluation become best visible for benchmarks with larger analysis times.

Table II
ACCELERATED LOOP ANALYSIS

Benchmark	Basic	Slicing	Polytope
matmul - MRTC	8.4 s	28 %	1 %
hamming - Misc.	0.4 s	63.7 %	62.0 %
g721 - DSPstone	80.2 s	88 %	89 %
fft - DSPstone	920.7 s	13 %	12 %
matrix1 - DSPstone	0.8 s	12 %	4 %
mult_10_10 - UTDSP	4.6 s	78 %	80 %

Table II shows for some example benchmarks the influence of the different techniques on the analysis run time. The second column of the table represents the run time of the basic loop analysis based on the modified Abstract Interpretation. The third column depicts the relative analysis run time after Program Slicing, while the last column indicates the measured run time after the application of the polytope-based fast loop evaluation w.r.t. the run time in the second column. It can be seen that Slicing significantly decreases the analysis time. In case of the *matmul* benchmark from the MRTC suite, a reduction of 72% was achieved. This benchmark also benefits from the polytope approach. It contains some loops that can be statically evaluated without iterating over all loop executions leading to a further reduction in time of 27%. For other benchmarks like *mult_10_10* from UTDSP, Slicing could reduce the analysis time by 21.8%. For *mult_10_10* the test whether the polytope approach can be applied was negative, thus slightly increasing the analysis time by 2% and forcing the analysis to switch back to the basic (iterative) loop evaluation.

Considering all 96 evaluated benchmarks, 40% of the benchmarks could benefit from Program Slicing which led to a decreased analysis time. For 13% of these benchmarks, the analysis time could be further improved by switching from the iterative approach based on Abstract Interpretation to the polytope-based non-iterative approach.

A general conclusion that can be drawn based on the results is that Program Slicing is an approach that yields a significant reduction in the analysis time. Its success highly depends on the size of the program slice, i. e. how many statements can be eliminated without affecting the computation of loop iteration counts. Slicing is also an enabling technique for a successful application of the non-iterative loop evaluation. If the original loop can be sufficiently simplified, the fast loop evaluation can take place leading to further remarkable accelerations of the loop analysis.

Our loop analysis computed safe loop iteration counts for all loops, i. e. we never encountered under-estimations that are unacceptable for many fields of applications. Taking the timing results into account, a good strategy is to always combine the basic loop analysis with the extensions of Program Slicing and the polytope-based evaluation. The computation time for these techniques is negligible and most benchmarks profit from them while we had no cases where these extensions significantly degrade the analysis.

In summary, it can be said that the results emphasize the reliability of our analysis. Almost all of the encountered loops could be precisely analyzed in an acceptable time producing beneficial information for further analyses. Besides using the results for compiler optimizations, like Loop Unrolling, we exploit the results for our WCET-aware compiler framework. The manual annotation of loops with iteration counts mandatory for a static WCET analysis is a tedious and error-prone task. Using our automatic loop analysis, we were able to verify those values and even correct some erroneous annotations. Moreover, a static loop analysis is compulsory to turn a WCET-optimizing compiler into a fully automatic system. In this configuration, new programs can be optimized w.r.t. to their WCET without any further user interaction.

The high number of exactly analyzable loops also shows that our analysis is not only able to cope with small (toy) applications but can also be utilized for complex real-world applications. This indicates its practical use for different purposes. Motivated by the positive results, we consider our methodology as a proof of concepts.

VIII. Conclusions and Future Work

In this paper, we presented concepts for a static loop analysis which generates useful information about loop iteration counts for different applications, e. g. compiler optimizations or a static loop analysis. After introducing the theory behind Abstract Interpretation, which is the basis for our static analysis and ensures the correctness of computed results, our work focuses on techniques aiming at an acceleration of the loop analysis. This objective is achieved by combining Abstract Interpretation with interprocedural Program Slicing which removes statements irrelevant for the computation of loop iterations. In addition, we developed a static non-iterative evaluation of loops which is based on mathematical concepts of polytopes and allows a more efficient computation of loop iteration counts than the conventional Abstract Interpretation. Our framework is evaluated with a large number of real-world benchmarks which contain 707 loops. For 96% of these loops, precise loop iteration counts were determined in analysis times ranging from less than one second up to few minutes in the most cases. This leads us to the conclusion that our static loop analysis is of practical use for the adoption in real-world problems.

In the future, we want to exploit the results of the loop analysis for different compiler optimizations, especially for those that aim at an automatic WCET reduction. We also want to extend our polytope-based loop analysis to ease the constraints described in Section V-B. This would allow its exploitation for further benchmarks. Another interesting extension of the current analysis is the exploration of other abstract domains, e. g. the octogan or convex polyhedra domain, to study their strengths and weaknesses in the context of static loop analyses. An interesting compiler application of our loop analysis is also its exploitation for a value analysis which might even compute possible values depending on a particular calling context.

REFERENCES

[1] S. S. Muchnick, *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler Transformations for High-performance Computing," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, 1994.

[3] M. D. Smith, "Overcoming the Challenges to Feedback-Directed Optimization," in *Proc. of DYNAMO*, January 2000.

[4] P. Berube and J. N. Amaral, "Aestimo: A Feedback-Directed Optimization Evaluation Tool," in *Proc. of ISPASS*, March 2006.

[5] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis using Abstract Execution," in *Proc. of RTSS*, Dec. 2006.

[6] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *POPL*, January 1977.

[7] M. S. C. Heal and D. Whalley, "Bounding Loop Iterations for Timing Analysis," in *Proc. of RTAS*, June 1998.

[8] M. Kirner, "Automatic Loop Bound Analysis of Programs Written in C," Master's thesis, Technische Universität Wien, 2006.

[9] C. Cullmann and F. Martin, "Data-Flow Based Detection of Loop Bounds," in *Proc. of WCET*, July 2007.

[10] A. Ermedahl and J. Gustafsson, "Deriving Annotations for Tight Calculation of Execution Time," in *Proc. of Euro-Par*, August 1997.

[11] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper, "Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis," in *Proc. of WCET*, July 2007.

[12] P. Cousot and N. Halbwachs, "Automatic Discovery of Linear Restraints among Variables of a Program," in *Proc. of POPL*, January 1978.

[13] P. Granger, "Static Analysis of Arithmetical Congruences," *International Journal of Computer Mathematics*, vol. 30, no. 30, pp. 165–190, 1989.

[14] A. Miné, "The Octagon Abstract Domain," *Higher Order Symbol. Comput.*, vol. 19, no. 1, pp. 31–100, 2006.

[15] P. Cousot and R. Cousot, "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation," in *Proc. of PLILP*, August 1992.

[16] M. D. Weiser, "Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method," Ph.D. dissertation, Ann Arbor, MI, USA, 1979.

[17] J. Lyle and M. Weiser, "Automatic Program Bug Location by Program Slicing," in *Proc. of International Conference on Computers and Applications*, Feburary 1987.

[18] T. Reps and G. Rosay, "Precise Interprocedural Chopping," in *Proc. of SIGSOFT*, October 1995.

[19] S. Bates and S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," in *Proc. of POPL*, Jan. 1993.

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and its Use in Optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[21] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper, "Faster WCET Flow Analysis by Program Slicing," *SIGPLAN Not.*, vol. 41, no. 7, pp. 103–112, 2006.

[22] K. J. Ottenstein and L. M. Ottenstein, "The Program Dependence Graph in a Software Development Environment," *SIGSOFT Softw. Eng. Notes*, vol. 9, no. 3, pp. 177–184, 1984.

[23] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," in *Proc. of PLDI*, June 1988.

[24] H. Falk and P. Marwedel, "Control Flow driven Splitting of Loop Nests at the Source Code Level," in *Proc. of DATE*, Munich, Germany, March 2003.

[25] H. Falk, *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*. Kluwer Academic Publishers, 2004.

[26] Z. Su and D. Wagner, "A Class of Polynomially Solvable Range Constraints for Interval Analysis without Widenings," *Theor. Comput. Sci.*, 2005.

[27] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe, "Analytical Computation of Ehrhart Polynomials: Enabling More Compiler Analyses and Optimizations," in *Proc of. CASES*, September 2004.

[28] D. Cordes, "Loop Analysis for a WCET-optimizing Compiler Based on Abstract Interpretation and Polylib (in German)," Master's thesis, TU Dortmund University, 2008.

[29] Mälardalen WCET Research Group, "WCET Benchmarks," http://www.mrtc.mdh.se/projects/wcet, September 2008.

[30] C. S. V. Zivojnovic, J. Martinez and H. Meyr, "DSPstone: A DSP-Oriented Benchmarking Methodology," in *Proc. of ICSPAT*, January 1994.

[31] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," 2001.

[32] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *Proc. of MICRO*, December 1997.

[33] "UTDSP Benchmark Suite," http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html, September 2008.

[34] Informatik Centrum Dortmund, "ICD-C Compiler framework," http://www.icd.de/es/icd-c, September 2008.

[35] S. Verdoolaege, "Barvinok," http://http://www.kotnet.org/~skimo/barvinok, September 2008.

[36] *ICD Low Level Intermediate Representation Backend Infrastructure (LLIR) – Developer Manual*. Informatik Centrum Dortmund, 2008.

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

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