

Analysis of path exclusion at the machine code level

Ingmar Stein and Florian Martin
AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
{stein,florian}@absint.com, <http://www.absint.com>

Abstract

We present a method to find static path exclusions in a control flow graph in order to refine the WCET analysis. Using this information, some infeasible paths can be discarded during the ILP-based longest path analysis which helps to improve precision. The new analysis works at the assembly level and uses the Omega library to evaluate Presburger formulas.

1 Introduction

A commonly used method to calculate worst-case execution times (WCET) for a program is to maximize

$$t_G = \sum_{n \in N} c(n) \cdot t(n)$$

where $G = (N, E, s, x)$ is the control flow graph representing the program, $c(n)$ is the execution count of a basic block n and $t(n)$ is the runtime of n . This optimization problem can be formulated as an Integer Linear Program (ILP) and solved by widely available ILP solvers.

The result of an ILP-based path analysis is a path that represents a safe upper bound of the execution time. However, it is possible that this path can never occur at runtime. At a fork in the control-flow graph, the decision which of the successor nodes will be executed next often depends on the path that leads to the fork. Depending on the execution history, only one of two successors might be feasible. Those dependencies are not accounted for in the ILP, and the path analysis views both nodes as possible successors. This situation can lead to a drastic overestimation of the real WCET.

In this paper, we introduce an extension of the aiT [1] analyzer that incorporates those dependencies into the ILP, which in turn improves the WCET prediction. The analysis produces additional ILP constraints that can exclude several classes of infeasible paths.

The example in Figure 1 illustrates how flow facts can be beneficial for the WCET computation. In this example, the path analysis has to select the successor nodes with the highest costs for both of the branches A and D . The resulting WCET is the sum of the costs associated with the edges constituting the critical path, i.e. $100 + 100 = 200$.

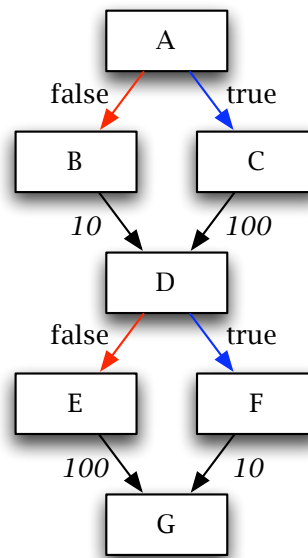


Figure 1. A control flow graph

However, if the analysis finds out that a positive outcome of the branch condition at A implies a positive outcome of the branch condition at D and vice versa, it creates a flow fact which allows only the paths $ACDFG$ and $ABDEG$. As a result, the new critical path has a WCET of $100 + 10 = 110$.

Such constructs as in the example often occur in code generated by code-generators such as SCADE [2] or in mode-driven code where many execution paths are controlled via relatively few flags.

2 Overview

The input for the flow constraint analysis is the control-flow graph of the program. While traversing this graph, each conditional branch is visited and an expression describing the branch condition is built. This step is trivial for high-level programming languages where the conditions are given in the source code, but as we are facing machine code, we have to reconstruct this information. Using a slicing component which operates on the assembly level, we find a set of instructions and variables that contribute to the branch conditions. If all instructions contained in that set can be mapped to arithmetic or comparison operations, we can build a boolean expression representing the branch condition.

In a second step, the expressions are transformed into another representation suitable for a solver library (Omega). The solver is used to compare two expressions, i.e. to check whether one expression implies the other or whether they are even equivalent. Beforehand, we test whether the two expressions can actually occur on the same path because not every implication allows for a sensible statement about the program.

The results of the comparisons are used to create new ILP constraints that are added to the ILP for the path analysis. This leads to a higher precision of the WCET prediction, i.e. a predicted worst-case execution time that is lower than the predicted WCET without the flow constraint analysis, but still is a safe upper bound of the real WCET.

3 The Flow Constraint Analysis

The flow constraint analysis traverses the control-flow graph and inspects all conditional branches, i.e. all inner nodes with more than one successor that are not call nodes.

If value analysis finds the exact (singleton) value of the condition register at a conditional branch, it marks one of the two outgoing edges as infeasible, and additional flow facts cannot improve the situation any more. Hence, only those branches where value analysis cannot deduce the value of the condition register are relevant for the flow-fact generation; the ones whose outcome is already determined by the value analysis are skipped.

A backward slice is computed for each considered conditional branch using the condition register as the initial target. A slice is a set of program points that directly or indirectly participate in the computation of the slicing criterion. A method how to compute slices is presented in [5].

Definition 3.1. A slice is called *linear* iff the program points contained in the slice can be ordered such that each program point is dominated by its predecessor. A linear slice that is ordered like that is called an *ordered slice*.

Example 3.1 (Linear slice). Figure 2 shows two control-flow graphs. The instructions that constitute two different slices are highlighted using a bold border. The left graph represents a linear slice because the two basic blocks can be ordered as A, D and block A dominates block D . In contrast, the right graph is non-linear because block C dominates neither D nor A .

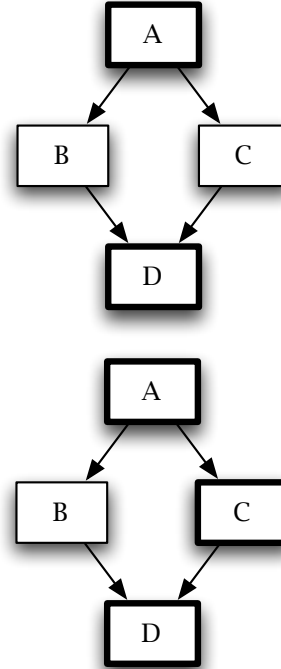


Figure 2. Linear slice (upper) and non-linear slice (lower)

We now restrict the analysis to linear slices. This excludes exactly those conditions that are built up on several different paths. The ordered slices are then transformed into slice trees. The inner nodes of a slice tree represent instructions while the leaves are either registers, memory cells, or constants (see for instance Figure 3).

Slice trees containing memory accesses whose target addresses cannot be determined statically cannot be used for the following comparisons and are therefore discarded.

A slice tree is an intermediate representation that can be transformed into other formats for different theorem provers. This process is described in the following for the Omega library.

The Omega Project is a collection of “Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs” by William Pugh and the Omega Project Team [4]. In particular, Omega offers a tautology test for

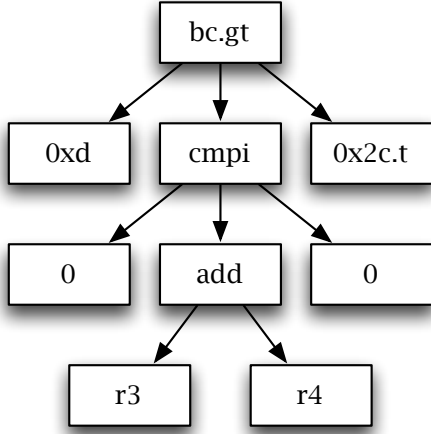


Figure 3. A slice tree

Presburger formulas that we will use to compare the branch expressions.

Definition 3.2. *Presburger arithmetic* is defined as an arithmetic with the constants 0 and 1, a function +, a relation = and the axioms

1. $\forall x: \neg(0 = x + 1)$;
2. $\forall x \forall y: \neg(x = y) \implies \neg(x + 1 = y + 1)$;
3. $\forall x: x + 0 = x$;
4. $\forall x \forall y: (x + y) + 1 = x + (y + 1)$;
5. If $P(x)$ is a formula consisting of the constants 0, 1, +, = and a single free variable x , then the following formula is an axiom

$$(P(0) \wedge \forall x: P(x) \implies P(x+1)) \implies \forall x: P(x).$$

Presburger arithmetic is a decidable fragment of arithmetic and implementations of fully automatic decision procedures (such as Omega) are readily available.

Slice trees are translated into Omega trees by mapping the semantics of the individual instructions to arithmetic or comparison operations. Instructions with unknown semantics are treated as symbolic functions. Several patterns are used during the translation of instructions into Omega operators that allow for the combination of multiple instructions into a single operator. While the inner nodes of Omega trees represent operations, the leaves are translated as follows:

- Integer constants remain constants.
- Registers and memory cells become free variables. A prefix of the variable name encodes the type of the variable as shown in Table 1.

Prefix	Type	Suffix
r	Register	Register number
m	Memory cell (word)	Memory address
h	Memory cell (halfword)	Memory address
b	Memory cell (byte)	Memory address

Table 1. Omega tree leaves

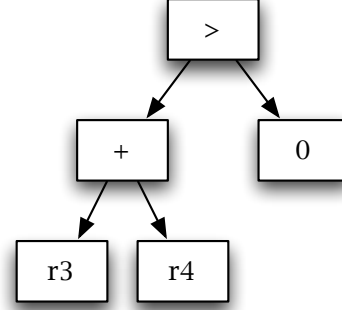


Figure 4. An Omega tree

Figure 4 shows the Omega tree resulting from the slice tree of Figure 3 using a simplified notation.

If all conditional branches are annotated with Omega trees, we can compare the branch conditions of two basic blocks A and B by testing several boolean expressions using Omega: $A \implies B$, $A \implies \neg B$, $\neg A \implies B$, $\neg A \implies \neg B$ and the same expressions with A and B swapped. If Omega determines one of the expressions to be a tautology, we can derive the flow constraints according to Table 2. The names a_t , a_f , b_t , and b_f stand for the *true* and *false* successors of the two basic blocks a and b , and $c(x)$ the execution count of basic block x . The table includes expressions that are logically equivalent to cover those cases where some of the successors a_t , a_f , b_t , and b_f are unavailable.

Expression	Flow constraint
$A \implies B$	$c(a_t) \leq c(b_t)$
$A \implies \neg B$	$c(a_t) \leq c(b_f)$
$\neg A \implies B$	$c(a_f) \leq c(b_t)$
$\neg A \implies \neg B$	$c(a_f) \leq c(b_f)$
$B \implies A$	$c(b_t) \leq c(a_t)$
$B \implies \neg A$	$c(b_t) \leq c(a_f)$
$\neg B \implies A$	$c(b_f) \leq c(a_t)$
$\neg B \implies \neg A$	$c(b_f) \leq c(a_f)$

Table 2. Implications and corresponding flow constraints

4 Limitation to n bits

Omega operates on the domain of integers, therefore the variables in the Presburger formulas have no range restrictions. However, the machine arithmetic works on n bits and is thus not modelled correctly in the Omega expressions. To resolve this problem, one can introduce modulo operators in the expressions to simulate an n bit range. If C is an expression whose result is an n bit value, C is replaced by $C' = C \bmod 2^n$. Because Presburger expressions don't have a built-in modulo operator, another substitution is needed:

If a term $x \bmod c$ occurs in a constraint C' , C' is replaced by

$$\exists \gamma : c\gamma \leq x < c(\gamma + 1) \wedge C''$$

where C'' is derived from C' by replacing $x \bmod c$ by $x - c\gamma$.

5 Evaluation

In order to evaluate the effectiveness of the analysis, we have analyzed a set of test programs. All tests were performed using aiT for MPC755. Table 3 illustrates how the WCET changes if path analysis is run without or with the flow constraints ($WCET_{fc}$). The last column shows the number of generated flow facts. The runtime of the flow constraint analysis on the test programs is presented in figure 5.

6 Outlook

With the main work done, we now look at possible future enhancements and additional uses of the flow constraint analysis.

6.1 Portability.

We plan to implement the analysis for further microarchitectures besides the PowerPC platform. The ARM platform is a natural extension since the slicing component already exists for it.

6.2 Nonlinear slices.

Furthermore, it seems worthwhile to examine nonlinear slices to find out whether new opportunities for optimization arise if the linearity constraint is dropped. Nonlinear slices may be handled by using a data-flow analysis that propagates the node conditions and subsequently combines all conditions associated with a node. However, the risk is

very high that the resulting expressions grow too large for the Omega library and that the runtime increases by several orders of magnitude.

6.3 Theorem-prover interface.

In addition to this, other theorem provers could be evaluated by providing an interface to the flow constraint analysis. An alternative prover could provide a performance superior to Omega in some cases or offer more functionality such as floating-point support.

6.4 Elimination of unreachable code.

With a simple extension, flow constraint analysis is able to detect some cases of unreachable code and to exclude the respective code blocks from the subsequent analyses, e.g., pipeline analysis. For that, a condition of a child node is compared to that of its direct parent. If they are equivalent or complementary, one of the two successors of the child node can be marked as infeasible.

6.5 PAG.

Unreachable code elimination as described above is an example how the information gathered by flow constraint analysis can be used for additional purposes. Another use case is PAG-generated analyzers [3] whose precision can be improved by path exclusions.

7 Conclusion

We have presented a method to find path implications within a given control flow graph for machine code programs. This information has been used to generate constraints which are then added to the ILP of the path analysis. The so-called flow constraints contribute to an improvement of the WCET prediction by excluding paths which cannot occur at runtime.

A tool which implements the algorithm presented in this paper has been successfully integrated into a WCET framework. It represents another phase in the workflow of the aiT WCET analyzer and fits seamlessly into the existing infrastructure. The tool has been used to conduct several tests which show both the effectiveness of the flow constraint analysis (ppcbrunch) on industrial programs as well as the moderate runtime increase of the complete WCET analysis as can be seen in figure 6.

References

- [1] AbsInt Angewandte Informatik GmbH. ait: Worst-case execution time analyzers.

Program	WCET	WCET _{fc}	Improvement	Constraints
Synth. example 1	1440 cycles	1154 cycles	19.9 %	4
Synth. example 2	1140 cycles	819 cycles	28.2 %	5
avionic 1	1480 cycles	1420 cycles	4.1 %	1
avionic 2	3178 cycles	3050 cycles	4.0 %	8
zlib	6706 cycles	5242 cycles	21.8 %	2

Table 3. Results for several test programs

Program	Instructions	Basic Blocks	Size [Bytes]	Type
Synth. example 1	44	13	912	Mach-O
Synth. example 2	38	13	792	Mach-O
avionic 1	764	40	26232192	ELF
avionic 2	523	14	433472	ELF
zlib	163	40	1700	Mach-O

Table 4. Sizes of the test programs

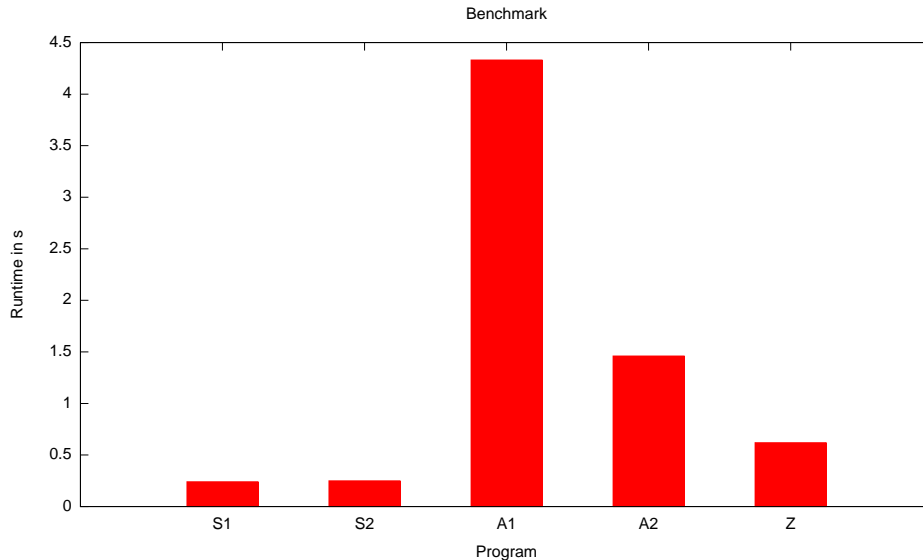


Figure 5. Runtime of the flow constraint analysis for several test programs

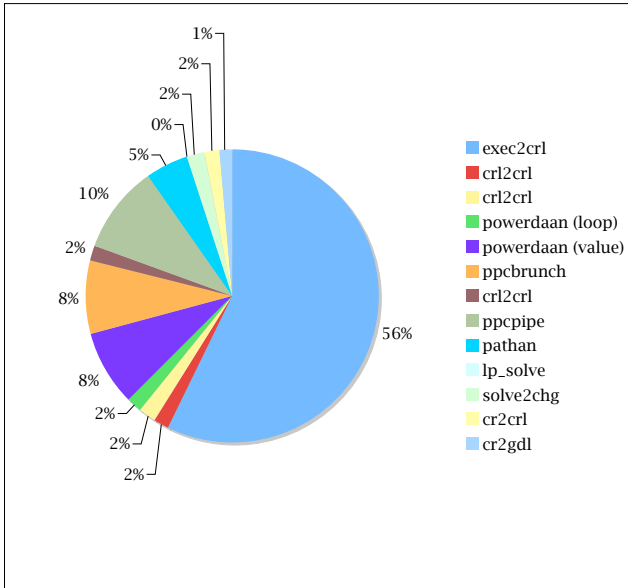


Figure 6. Overall runtime of the WCET analysis for avionic 2 broken down into subprograms

- [2] Esterel Technologies. Scade suite – the standard for the development of safety-critical embedded software in the avionics industry.
- [3] F. Martin. Pag - an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
- [4] Omega Project Team. The omega project: Frameworks and algorithms for the analysis and transformation of scientific programs. 2007.
- [5] M. Schlickling. Generisches slicing auf maschinencode. Master's thesis, Universität des Saarlandes, Saarbrücken, 2005.