The CoSTA Transformer: Integrating Optimizing Compilation and WCET Flow Facts Transformation

Adrian Prantl*

Institute of Computer Languages
TU Vienna, Austria
adrian@complang.tuwien.ac.at

Abstract. The determination of the *worst-case execution time* (WCET) of a program is a critical issue for the design of safety-critical real-time systems. Because the exact timing of the program depends on the low-level hardware instructions, tools that automatically calculate an upper bound for the WCET typically operate on the object code level. In order to get tighter WCET estimates, these tools often rely on control flow annotations, also called *flow facts*, which can be provided manually or by a tool. For the programmer, it is more convenient to provide flow facts directly at the source code level. To make this possible, it is necessary to extend the compiler to transform the source-based flow facts alongside the program optimizations. In this paper, we introduce an approach that integrates optimizing compilation and flow fact transformation, called the CoSTA Transformer. Our approach is designed to operate on a very high level of abstraction and thus can easily be adopted to different compiler/target-machine combinations.

1 Motivation

The determination of the *worst-case execution time* (WCET) is a critical issue for the design of safety-critical real-time systems. The calculation of an upper bound for the WCET can be achieved by searching the longest path in the control-flow graph (CFG) of a program. To do this, it is necessary to *annotate* the CFG with flow information such as upper bounds for loop constructs. This information is commonly called *flow facts*. Flow facts can either be calculated automatically by a tool through a static analysis or they can be provided manually. Although a remarkable amount of flow facts can be extracted from the program sources, the programmer may always have additional *domain-specific knowledge* (such as characteristics of the input data) that will result in a tighter WCET bound.

Flow facts can be supplied at different levels of abstraction. From the point of view of the WCET-calculation tool, it is important that the information will be available at the object-code level, which is as close as possible to the program

* This work has been supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract P18925-N13.

that will eventually be executed on the target hardware. For the programmer, it would obviously be more convenient to supply information at the highest level of abstraction possible, ideally directly in the source code. The compiler then has to translate these annotations to the object code level. This implies that the annotations are transformed alongside the optimizations. Such an approach has recently been proposed by Kirner [4]. This way of providing annotations at a high level of abstraction is not yet implemented by state-of-the-art industrial tools, as indicated by a survey on WCET calculation tools [2, 12]. This is likely to change, however, and a first step towards an implementation based on the compiler's low-level intermediate representation was presented by Schulte [11] earlier this year.

As an integral part of the CoSTA project (Compiler Support for Timing Analysis) we are working on integrating the transformation of flow facts with optimizing compilation. For the implementation of our approach, we have decided to work on a very high level of abstraction which allows us to maintain a high degree of flexibility regarding the choice of the compiler infrastructure and target hardware. More details about our overall architecture and the static analysis of flow facts has been presented earlier in a companion paper [8].

2 Implementation Environment

In this section we introduce the two frameworks we are building upon. The first framework we are using is LLNL-ROSE, a source-to-source program transformation framework for C++, which is being developed at the Lawrence Livermore National Laboratories [10]. LLNL-ROSE uses the C++ front end by the Edison Design Group (EDG) and is also written in C++. For the implementation of the program transformations we are using the loop processor of the LLNL-ROSE distribution. It operates on the C++ abstract syntax tree (AST) and can perform a variety of standard loop transformations including loop unrolling, blocking, fusion, interchange and also partial redundancy elimination [1, 6].

The second framework is SATIrE (Static Analysis Tool Integration Engine). It is currently being developed at our group and aims at integrating LLNL-ROSE with program analysis frameworks such as the Program Analysis Generator (PAG) from AbsInt [9, 7].

In the context of CoSTA, it is important to note that SATIrE supports the export and import of the AST provided by LLNL-ROSE in the form of an external term representation that is using the very same syntax as the Prolog programming language. These terms, which represent program fragments, can easily be manipulated by a Prolog program and allow for the construction of transformation tools that can be formulated very efficiently. The external representation in combination with our term manipulation library forms a framework that we call TERMITE. The CoSTA Transformer, in particular the extraction ("unweaving"), transformation and reintegration ("weaving") of the annotations has been implemented using our TERMITE framework.

3 Architecture of the CoSTA Transformer

Viewed as a whole, the CoSTA Transformer accepts C++ sources intermingled with user-specified (or tool-delivered) annotations as input and will generate optimized C++ code containing correctly transformed annotations. Figure 1 represents a schematic overview of the components of the CoSTA Transformer. This section gives an overview of the data flow between the individual components, which conceptually can be considered to consist of 4 phases:

Phase 1 - unweave. In preparation for the subsequent steps, the C++ code has to be stripped off any annotations. This allows us to integrate our annotation transformer without the need for extensive modifications of the optimizer.

Phase 2 - optimize. In the next step, the LLNL-ROSE loop processor is invoked to perform a variety of optimizing program transformations on the stripped C++ sources. The loop processor has to be modified for our approach to generate a trace listing the transformations that were performed on the original sources. This optimization trace will then be read by the annotation transformer, which additionally takes the annotations that were stripped off the original C++ sources as input.

Phase 3 - transform. The annotations will then be transformed according to a body of rules that specify how to update annotations for the transformations found in the optimization trace. These rules need to be specified only once for each program transformation that is implemented in the optimizer. The rules are generally very short, but might become more complex if the optimization duplicates basic blocks in the program. An example of both a simple and a more complex rule is displayed in Figure 3.

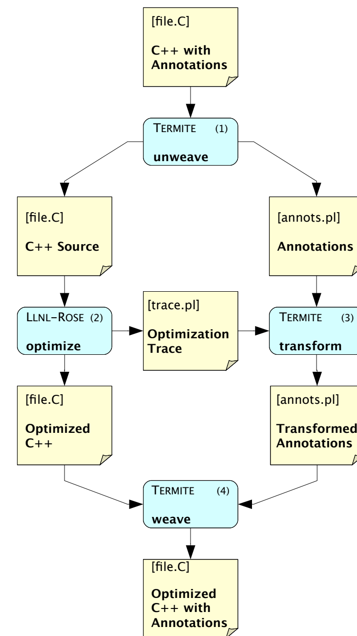

Fig. 1. The data flow between the components of the CoSTA Transformer

Phase 4 - weave. After the annotations have been transformed, the only task left is to insert the annotations back into the optimized C++ sources - this is done in the final step.

As shown in the diagram, the actual transformation of annotations is kept separate from the optimization of the program. This decision allows for a very concise and clean implementation of the annotation transformation; additionally it also allows us to easily link the annotation transformer to other compiler frameworks. The compiler that compiles the output of the CoSTA Transformer has to be restricted to program transformations that preserve the (annotated) control flow.

3.1 Classification of Annotations

Up to this point we have discussed the high-level view of our approach. We will now show how the actual transformation of flow annotations works. In the context of the CoSTA Transformer, we define three types of *annotations* that are inspired by the definition of WCETC, but with a slightly altered syntax to resemble Prolog terms more closely [3]:

- *Markers* are labels for the very basic blocks they are defined in.
- *Restrictions* are inequalities that are used to describe the ratios between the execution frequencies of markers. In contrast to markers, the location of a restriction declaration in the source code is not significant. Restrictions are expressions that are composed of arithmetic and Boolean operators that can contain markers as operands.
- *Loop Bounds* are a special class of restrictions. They can be interpreted as a shorthand for the equation $m_{this} = b_{loop} \cdot m_{parent_scope}$. Loop Bounds thus express the ratio between the basic block they are defined in and the surrounding parent scope.

These annotations represent the lowest common denominator of current annotation languages, and were thus selected to illustrate our approach. However, we are also researching different types of annotation languages [5] and plan to extend our tools in the future to reflect this. A typical WCET-calculation tool would use these annotations to formulate an integer linear programming (ILP) problem that can then be solved efficiently by existing third-party tools.

The optimizations that are performed by the LLNL-ROSE Loop Processor impose different requirements on the CoSTA Transformer. In general, we can distinguish three kinds of transformations, which always/never/sometimes modify the control flow graph (CFG) and the annotated information. If a program transformation alters the CFG, for example by fusing two loops of identical iteration space into a single new loop, it is necessary to update the *location* of the annotations to the newly created loop. If a loop is unrolled by a factor that is a integer divisor of the iteration count, the locations of the basic blocks in the CFG will not necessarily change, but the *information* described by the contained annotations has to be adjusted accordingly.

Affects	Loop Unrolling	Loop Blocking	Loop Fusion
Control flow graph	maybe	yes	yes
Annotation location	no	yes	yes
Annotation information	yes	yes	no

3.2 Transformation of Annotations

How to update the location of the annotations can generally be induced from the optimization trace that has to be provided by the program transformation component. The optimization trace consists of statements that summarize which basic block has been affected by which kind of optimization. The annotation $unrolled(M_{LoopBody}, 2)$ means that the loop body $M_{LoopBody}$ has been unrolled

Original user-annotated program

```
int* f(int* a)
{
        int i;
#pragma wcet_marker(m_func)
        for (i = 0; i < 48; i += 1) {
#pragma wcet_loopbound(48)
#pragma wcet_marker(m_for)
          if (test(a[i])) {
#pragma wcet_marker(m_if)
            // Domain-specific knowledge
#pragma wcet_restriction(m_if =< m_for/4)

            a[i]++;
          }
        }
        return a;
}
```

After loop unrolling with factor 2

```
int *f(int *a)
{
  int i;
  for (i = 0; i <= 47; i += 2) {
#pragma wcet_marker(m_f_1_1)
#pragma wcet_loopbound(24)
    if ((test(a[i]))) {
#pragma wcet_marker(m_f_1_1_1)
#pragma wcet_restriction(
  m_f_1_1_1+m_f_1_1_2=<m_f_1_1/2)
      a[i]++;
    }
    if ((test(a[1 + i]))) {
#pragma wcet_marker(m_f_1_1_2)
#pragma wcet_restriction(
  m_f_1_1_1+m_f_1_1_2=<m_f_1_1/2)
      a[1 + i]++;
    }
  }
  return a;
}
```

Fig. 2. Illustrating Example: Flow annotations before and after loop unrolling

by a factor of 2. This implies that the transformation rules can be adopted to consider implementation details of the optimizations.

The update of the annotations is handled by the rule body. Rules can be defined to be parameterized on the optimizations. It is also possible for a rule to generate more than one annotation - this is typically the case for loop unrolling, where copies of annotations have to be inserted in the repeated loop bodies. After the rules have been applied, the resulting annotations are routed through a term replacement system that performs algebraic simplifications to the transformed annotations.

3.3 Illustrating Example

Figure 3 shows an excerpt of the rules we implemented for loop unrolling. The rules in this example are parameterized to take the label of the unrolled loop body and the unroll factor. They are written in Prolog and are applied to each annotation of the original program. The first rule updates the loop bounds according to the unroll factor k, whereas the second rule clones restrictions for each unrolled basic block in the loop body. The program on the right of Figure 2 shows the effect of applying these rules to the program on the left.

```
% loop unrolling
unrolled(M, K, annotation(M, wcet_loopbound(Bound)),
              [annotation(M, wcet_loopbound(New))]) :-
  New is ceiling(Bound/K).

unrolled(M_Loop, K, annotation(M_Annot, wcet_restriction(Term)), NewAnnots) :-
  replace(Term, M_Loop, M_Loop*K, Term1),
  (nested_in(M_Annot, M_Loop) ->
    list_from_to(1, K, Ns),
    maplist(unroll_clone(M_Loop, M_Annot, Term1), Ns, NewAnnots);
    NewAnnot = []).
```

Fig. 3. Example from the transformation rules: loop unrolling

4 Conclusions

In this paper, we have presented a modular and portable approach to integrate the automatic transformation of hand-annotated WCET flow fact information with optimizing compilation. Our implementation is still at an early stage, however, our experiences indicate that this approach is well suited to support the important class of classical loop optimizations. As the next step, we will concentrate on integrating the transformation of annotations (CoSTA Transformer) with our static program analysis component (CoSTA Analyzer [8]).

Acknowledgements. The author would like to thank Markus Schordan for many discussions concerning the Llnl-Rose and SATIrE frameworks, and Jens Knoop for discussions and helpful comments on earlier versions of this paper.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.
2. J. Gustafson. The WCET tool challenge 2006. In *Preliminary Proceedings 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 248 – 249, Paphos, Cyprus, November 2006.
3. R. Kirner. The programming language wcetC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.
4. R. Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2003.
5. R. Kirner, J. Knoop, A. Prantl, M. Schordan, and I. Wenzel. WCET Analysis: The Annotation Language Challenge. In *Proceedings 7th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis*, 2007. To appear.
6. J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion (with retrospective). *Best of PLDI, SIGPLAN Not.*, 39(4):460–472, 2004.
7. F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
8. A. Prantl. Source-to-Source Transformations for WCET Analysis: The CoSTA Approach. In *Proceedings 24th Workshop of "GI-Fachgruppe Programmiersprachen und Rechenkonzepte"*, 2007.
9. M. Schordan. Combining tools and languages for static analysis and optimization of high-level abstractions. In *Proceedings 24th Workshop of "GI-Fachgruppe Programmiersprachen und Rechenkonzepte"*, 2007.
10. M. Schordan and D. J. Quinlan. A source-to-source architecture for user-defined optimizations. In L. Böszörményi and P. Schojer, editors, *JMLC*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer, 2003.
11. D. Schulte. Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler. Master's thesis, Universität Dortmund, 2007.
12. L. Tan and K. Echtle. The WCET tool challenge 2006: External evaluation – draft report. In *Handout at the 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Paphos, Cyprus, November 2006.