# Facilitating Worst-Case Execution Times Analysis for Optimized Code *

Jakob Engblom[1] Andreas Ermedahl[1] Peter Altenbernd[2]

[1] {jakob,ebbe}@docs.uu.se, Department of Computer Systems (DoCS),
Uppsala University, P.O. Box 325, S-751 05 Uppsala, Sweden, Fax: +46-(0)18-550225
[2] peter@c-lab.de, C-LAB, D-33094 Paderborn, Germany†

## Abstract

*In this paper we present* co-transformation*, a novel approach to the mapping of execution information from the source code of a program to the object code for the purpose of worst-case execution time (*WCET*) analysis. Our approach is designed to handle the problems introduced by optimizing compilers, i.e. that the structure of the object code is very different from the structure of the source code.*

*The co-transformer allows us to keep track of how different compiler transformations, including optimizations, influence the execution time of a program. This allows us to statically calculate the execution time of a program at the object code level, using information about the program execution obtained at the source code level.*

## 1. Introduction

In the real-time systems field, there are many reasons to estimate the execution time of a program prior to executing it, including scheduling and schedulability analysis, determining resource requirements , dimensioning systems, and many more. The most common execution time measure is the *worst-case execution time*, WCET, of a program, i.e. how long a program may take to execute given the worst possible set of inputs and other circumstances.

There are two ways to obtain the WCET of a program: measure execution times for certain inputs, or perform static analysis on the program. Unlike measurements, static analysis can guarantee a safe bound on the worst possible execution time to be obtained.

Unfortunately, the problem of static execution time determination is in the general case too complex to be solved exactly. For non-trivial programs, the number of possible executions are virtually infinite. Thus, we cannot simply enumerate all possible executions and select the longest. Within a reasonable analysis time, we can only produce *estimates* of the WCET.

A WCET estimate must be both *tight* and *conservative* (safe), which means that it should be as close as possible to the actual WCET, while not underestimating it.

In order to achieve a tight estimate, both the program flow (number of iterations in loops, possible and impossible execution paths, etc.) and the execution characteristics of the object code on the target system (the time taken to execute machine instructions, waiting for I/O devices and memory, etc.) must be taken into account. The ideal is knowing exactly how many times an individual object code instruction in the (compiled) program is executed (in the worst case), and how long each occurrence takes to execute.

The analysis of program behavior is easier at the source code level, since this is the most accessible form of program representation (at the object code level, information about the program structure is less obvious). On the other hand, concrete execution times for fragments of a program can only be calculated from the object code. Hence, in order to calculate WCET estimates for a program, a reasonable approach is to have a tool which *maps* the information obtained at the source code level onto the object code.

Optimizing compilers are necessary in embedded systems development in order to make a program small enough or fast enough to be feasible on a given hardware platform. Unfortunately, the mapping from source code to object code is complicated by compiler optimizations, which can change the structure and execution characteristics of a program quite drastically. For example, loops may be unrolled, statements removed, and functions inlined or duplicated. Also, code generation (especially for primitive processors) can introduce new control structures not visible in the program source code.

Most work in the field of WCET analysis have not confronted the full problem of timing optimized programs. Typical approaches have been to limit the optimizations used to those which are easy to handle, or to simply assume

*This work has been supported by ASTEC (Advanced Software TEChnology), www.docs.uu.se/astec, and IAR Systems AB, www.iar.se.

†The work presented in this paper was performed while on a postdoc at DoCS, Uppsala University.

that the problem is easy to solve.

In this paper, we present a novel approach (called *co-transformation*) to the mapping of information about program execution from the source code level to optimized object code, in order to facilitate the combination of powerful source level program analysis with the high resolution execution times from object code analysis. Our approach depends upon information from the compiler about how a certain program is transformed, and knowledge about how the compiler implements its transformations.

The main problem is how to describe the effects of compiler optimizations on the execution information. To this end, we introduce a language called *Optimization Description Language*, *ODL*. It is a specialized formalism to describe how *execution information* is to be transformed when the program code is transformed.

The benefits of our approach are that:

- The co-transformer can handle a larger set of compiler optimizations than previous approaches.
- Tighter timing estimates can be obtained, since more information can be mapped from the source code to the object code.
- The compilers we use only require slight modification to extract the information we need.
- The ODL makes it easier to separate the compiler and the timing tool, since we only need to change the ODL descriptions when we change compiler.
- We present a general open framework for WCET analysis, in which different kinds of object code and source code analysis, compilers and target hardware can be integrated. Porting of the tool to new targets is relatively easy, since the dependences on the target system are localized to a single component.

In the next section, we briefly review related work. In Section 3 we present our framework, in Section 4 we describe the co-transformer and transformation descriptions (ODL). Section 5 presents a small example of co-transformation. Section 6 presents our preliminary results, and in Section 7 we conclude and present plans for future work.

# 2. Related Work

In the area of WCET research, the problem of managing optimized code and mapping information from the high level to the low level has received comparably little attention. Most research groups concentrate on other issues, even though all are confronted with the problem in some way.

In [5, 19], the approach is to perform the mapping from the object code to the source code, and perform timing calculations on the source code level. This makes the analysis less exact, since it can be difficult to accurately ascribe an execution time to a source level statement. Also, caches and pipelines cannot be analyzed, since such analysis requires detailed knowledge about the program flow on the object code level. By mapping from source code to object code and performing calculations on the object code level, as in our approach, caches and pipelines can be handled, and tighter execution time estimates are obtained.

A common approach is to use *debug information* to figure out how the source code and object code relates to each other [10, 16]. The disadvantage is that it is quite difficult to map the information needed for WCET analysis, since much information about the program structure and execution is discarded by the compiler when generating debug information. In contrast, the co-transformation approach allows us to maintain the full set of information we need for WCET analysis.

In the debugging field itself [1, 7], there is a similar need to track what happens to a program when it is compiled. However, the information needed by a debugger only concerns the *static* aspects of a program; i.e. the mapping of source statements to object code instructions in order to handle breakpoints. Since information about loop bounds etc. is not needed for debugging, it is not suitable for WCET analysis.

Another approach is to *modify a compiler* to output information about program flow, data addresses, and other information needed for WCET analysis [13, 17, 23]. The idea is similar in concept to co-transformation, but requires extensive modifications of the compiler. Hence, most of the work must be redone when porting a timing tool to other compilers or processors. Our approach allows for easy ports to new compilers, and is independent of target processor. An interesting twist, with some similarities to our ODL, is presented in [4]. There they let a compiler output a sequence of messages detailing how a program is modified, with the purpose of visualizing the compiler's transformation of the program.

In the MARS project, the compiler for the `Modula/R` language transforms execution information about a program at compile time [22]. Execution information is provided as annotations in the source code. The approach is limited to transformations which keep the program well-structured, and which do not make it too difficult to present an understandable view of the program to the user. The basic principle, updating the execution information during compilation, is the same as the co-transformer, but we do not require the program to be well-structured or presentable to a user.

In the CHaRTS project [6], timing constraints on a program are maintained and transformed by the compiler during compilation. The approach is similar to our co-transformer, but the purpose is different: they propagate timing constraints in order to guide the compiler, while we want to propagate execution information to obtain timing estimates.

The purpose of the co-transformer is to transform program execution information. Such information is usually provided manually (using annotations), but there is some promising research on the automatic analysis of programs to obtain loop bounds [9], and discover infeasible paths [2, 14].

On the output end of the co-transformer, we need to perform calculations of execution times given information about the program flow and times for the basic blocks involved. We consider the most promising approach to WCET calculations to be the use of implicit path enumeration techniques (IPET) based on constraint solving; this technique is used by several research groups [15, 18, 20].

# 3. Co-Transformation Context

We have incorporated the co-transformer in a general framework for WCET tools. The framework divides the problem of WCET estimation into distinct components, and makes it easier to combine different tools. Also, dependences on various external parameters are localized, minimizing the effort needed to port the timing analyzer to a new context. Figure 1 gives an overview of the framework.

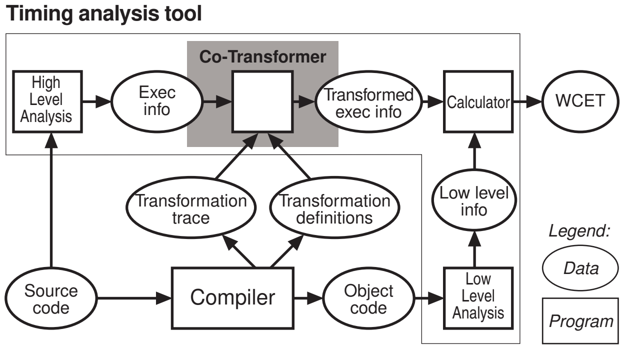



Figure 1. Components of a WCET tool.

Short explanations of the components and the data flows between them are given below:

- The *high-level analysis* (HLA) analyzes the source program and derives information about possible program executions. The extracted *execution information* contains information about loop bounds, infeasible paths, function calls, etc. Examples of HLA approaches are found in [9, 11].
- The *low-level analysis* (LLA) analyzes the object code created by the compiler. It analyzes straight-line segments of object code (*basic blocks*), and determine their low-level execution characteristics. All target hardware dependences (processor, memory system, etc.) are localized in the LLA. Examples of LLA can be found in [12, 15]. The *low-level information* generated contains information about the execution of the basic blocks in the optimized program, and it may include data on cache and pipeline effects for single blocks.
- The *co-transformer* transforms the program execution information generated by the HLA so that it agrees with the structure of the low level information. It uses information about how the individual compiler transformations are carried out (the *transformation definitions*, written in ODL), and a *trace* of how the program has been transformed.
- The *calculator* takes the program flow information and the low level information and performs the calculations necessary to produce an execution time estimate. The effects of pipelines and caches are actually handled in the calculator, using the low-level information. The calculator is a portable general component. Examples of calculation methods are found in [2, 15, 18, 21, 23].

In the rest of this paper we focus on the co-transformer. A more detailed description than the one below can be found in [8].

# 4. Co-Transformation

The co-transformer helps execution time analysis by transforming information about the execution of a program in parallel with the transformation of the code in a compiler.

In the next subsection, we will discuss how program execution information can be represented. Then we present the specific representation we use, followed by a presentation of the transformation trace and the Optimization Description Language, ODL. ODL is used to specify the transformations performed on the datastructure representing program executions. Finally, we discuss how the co-transformer relates to the compiler.

## 4.1. Representing Program Executions

In order to obtain tight execution time estimates, the co-transformer needs a datastructure which captures as much detail as possible about the program execution, and which can be transformed without losing (too much) information in the process.

For example, in the case of a loop, for the compiler it is only relevant that there is a loop, and changes may be performed to make the loop go faster or the code smaller. For timing analysis, on the other hand, it is important to know (exactly) how many times the loop is iterated. When the compiler changes the loop, the execution information must be changed too. This is what the co-transformer does.

To demonstrate that it is not entirely trivial to describe the relation between the *static* structure of the program as it is compiled and the *dynamic* information about the execution, we give two illustrative examples:

**Representing loops** To demonstrate the potential problem of a weak loop representation, consider the example given in Figure 2. The figure presents a nested loop, where the inner loop is performed five times for each iteration of the outer loop (the leftmost diagram in Figure 2 (a)). The compiler wants to move the grey block out of the inner loop (assume it is loop-invariant). We expect the final result to be that the grey block will be executed 10 times, as shown to the right in Figure 2 (a).

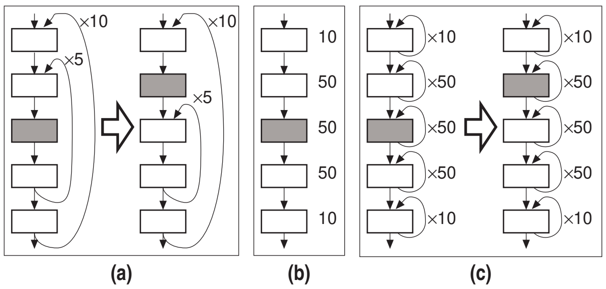



Figure 2. Loop representation problem.

If the datastructure used only represents the actual execution count for each basic block in the program, as shown in Figure 2 (b), it is impossible to update the execution information when the block is moved out of the loop. The problem is that while the representation *does* capture the program execution as it stands, it *does not* capture the fact that blocks are members of loops, and that they derive their execution counts from the loops. We cannot deduce the original loops from the information in Figure 2 (b).

Instead, the tool must make the safe assumption shown in Figure 2 (c): each block is assumed to be a loop of its own. Since the information about the original loop structure is lost, we cannot reduce the execution count of the block when we move it. The final result is shown on the right in Figure 2 (c).

Our representation does explicitly represent loops, and performs like in Figure 2 (a).

**Representing function calls** The treatment of function calls is a good example of the difference in precision different descriptions can give.

The simplest approach is to ascribe a single execution time to each function in the program (using a local worst-case analysis): consider the case that a function contains a loop whose iteration count depends on some parameter to the function. If the greatest value for this parameter anywhere in the program is 2000, but all other calls only use the value 20, the execution time for all calls will have to consider the loop as executing 2000 times, even though is usually only executes $\frac{1}{100}$ as many times.

A better but more expensive approach is to build the complete *worst-case call graph* for the program: every possible call to every function in the program is represented on its own. Note that the creation of such a call graph requires an analysis of the dynamics of the program. In this graph, '2000' would only occur once, with '20' used in all other calls to the function. The volume of data and analysis time needed would be higher, but the result better.

## 4.2. Our Program Execution Representation

In our prototype tool (see [8]), a program execution is represented by a worst-case call graph where each node is a specific function call (called *function instance*). Each function instance contains a complete copy of the basic block structure of the function, with execution information specific to the particular call to the function. A similar scheme is used in [3], where a program is represented as a call graph of loops.

Inside each function, we concentrate on the relation between basic blocks and loops. The following information is kept for each basic block:

- Execution count
- Loop scope
- Iteration count for loops

The *execution count* of each basic block is relative to the iteration count of the nearest surrounding loop: if the loop runs 27 iterations, the maximum execution count is 27. The execution count of a block is used when calculating the WCET of a program.

The *loop scope* is the name of the nearest surrounding loop (or the function itself, for blocks and loops on the top level of a function).

We represent a loop by a separate basic block, whose *iteration count* gives the number of times the loop body is executed. Just like for ordinary blocks, the execution count gives the number of times the loop is executed relative to the iterations of the surrounding loop.

In Figures 3 and 5, there are two examples of function graphs annotated with this information.

The execution information is linked to the code by the names of functions and basic blocks: a certain transformation is applied to certain basic blocks in a certain function, and all instances of the function in the call graph are transformed in the same way. The compiler supplies the names of all basic blocks and loops.

### 4.3. Transformation Trace

The *trace* specifies which transformations have been performed and where they have been applied to. It is generated by the compiler, and the compiler needs to be modified to emit the traces.

The co-transformer applies these transformations to the function instances in the same order as the compiler transforms the code of the program. Because the execution information may contain several instances of the same function, each transformation in the trace may cause several co-transformations.

### 4.4. Defining Transformations in ODL

In order to perform co-transformations, the transformations need to be specified. Instead of coding the co-transformations in an ordinary programming language, we have defined a language specifically designed for describing (co-)transformations. This language is called *Optimization Description Language*, *ODL*. The main advantages of the ODL are that:

- Transformations are easier to describe in a special-purpose language than in an ordinary programming language.
- The transformation definitions are easier to change. The co-transformer program does not need to be changed to adapt to a different compiler.
- The issues involved are much clearer when they cannot be worked around by patches (which could be easily done in a general programming language), and when they are not embedded in a larger program.
- The transformation engine, and even the compiler optimization engine, could be generated from the ODL source.

ODL is based on pattern-matching: an in-pattern is matched with the basic block graph of a function, and replaced by an out-pattern. Each pattern may consist of several graph fragments, in order to allow for transformations which affect disjoint parts of a function.

The patterns are expressed in terms of nodes, where a node can correspond to one or more basic blocks in the program flow graph. Groups of basic blocks are treated as a unit by using the concept of *compound nodes* (`cnodes`). Compound nodes are needed since transformations often involve large groups of basic blocks which are transformed as a unit. Typical examples are loop bodies and switch-statement cases.

The *structural change* to the program is coded into the patterns (the out-pattern shows how the code corresponding to the in-pattern looks after the transformations).

The patterns are positioned in the program code by giving function and basic block names. The names for the blocks in the out-pattern can be either the same names as in the in-pattern, or new names (if new blocks are introduced into the code).

The *execution information* is transformed by binding data variables to nodes in the in-pattern and out-pattern. The values of the out-pattern variables are calculated from the values of the variables in the in-pattern. The functions used in these calculations capture the transformation of execution information.

An example of ODL code can be found in Figure 4.

### 4.5. Relation to the Compiler

Writing the co-transformation definitions requires access to the definition of the compiler transformations. The compiler must also be modified slightly to emit the transformation trace. However, the compiler does not need to know anything about the structure or semantics of the execution information.

The co-transformer only depends upon the compiler, not upon the target system. We can have one compiler in different versions for different targets, and the co-transformer does not need to change when the compiler and the tool are ported to another target. Provided that the compiler consists of a certain set of possible transformations, $T$, and that a certain subset of these ($T_t \subseteq T$) are used for any specific target. Then, if the co-transformer handles a set of transformations $C$, where $C \supseteq T$, the co-transformer and the transformation definitions does not change when we implement the compiler for a certain target, since $T_t \subseteq T \subseteq C$.

The co-transformer is independent of the compiler, since it is enough to give new co-transformation definitions in order to support another compiler.

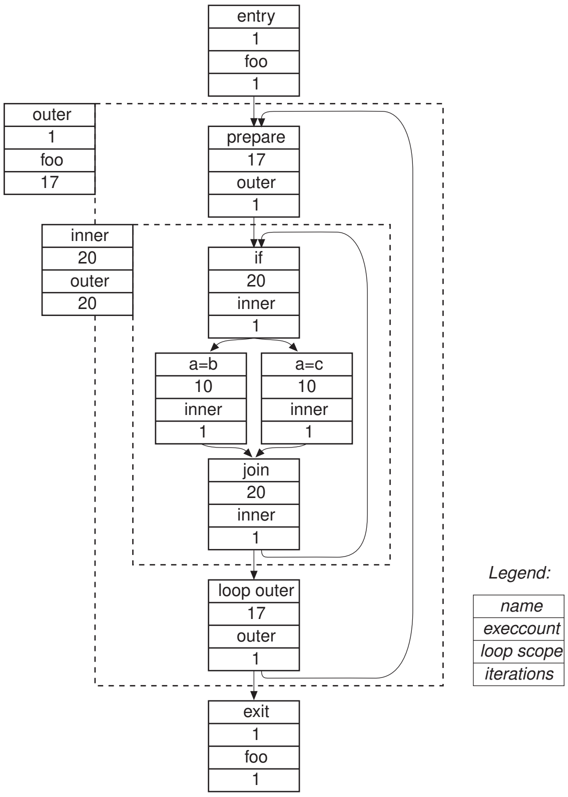
## 5. Example

We have implemented a prototype co-transformer, which is described in more detail in [8]. In the following, we give a simple example to show how co-transformation works.

The example given is a loop optimization called *loop collapsing*, where a nested loop is transformed to a single loop. This is typical for operations over large matrices. The storage layout (the matrix is stored in consecutive memory locations) is exploited in order to make the code more efficient.

Flow graphs are used to illustrate and explain the example. Each node in the graph contains the following information: name of the basic block, execution count, loop scope name, and iteration count. Loops are shown as dashed lines around the blocks in their bodies, with the special blocks representing the loops attached to the side.

Figure 3 shows the execution information before the transformation, and the corresponding C program.




```
for(i=0;i<17;i++)
  for (j=0;j<20;j++)
    if(j<10)
      a[j][i] = b[j][i] * 2;
    else
      a[j][i] = c[j][i] * 2;
```

Figure 3. Example program before transformation.

Figure 4 shows the ODL code for the loop collapsing transformation. The in-pattern describes the situation before the optimization. The first fragment (called `loops`) matches the loop bodies. It contains three compound nodes (`cnode`): `lo1` and `lo2` are the parts of the outer loop outside the inner loop, and `li` is the body of the inner loop. We pick up execution information from the inner loop in the variable `LI1Data`. The second fragment matches the block representing the outer loop, and the third the block representing the inner loop.

The out-pattern shows how the code will look after the transformation. The `loops` fragment now only contains the inner loop body. The compound node called `body` copies the content of the `li` compound node from the in-pattern; the compound nodes `lo1` and `lo2` are discarded. The execution information for the new loop is represented by the variable `LIOut`.

```
%------------------------------------------------------------
% ARGUMENTS:
%   S: the function to be transformed
%   LO1begin: the beginning of the first part of the outer loop.
%   LO1end: the end.
%   LO2begin: the beginning of the second part of the outer loop.
%   LO2end: the end.
%   LO: the outer loop representative block.
%   LIbegin: the beginning of the inner loop body.
%   LIend: the end.
%   LI: the inner loop representative.
%   LN: the name of the replacement loop
%------------------------------------------------------------
  transformation LoopCollapse
    (section S, var LO1begin, var LO1end, var LO2begin, var LO2end,
               var LO, var LIbegin, var LIend, var LI, var LN)

  %%------------------------------------------------------------
  %% Inpattern: how does the code look before transformation?
  %%------------------------------------------------------------
    inpattern
      fragment loops(S,LO1begin,LO2end)  % pick up pattern
        cnode(lo1,LO1begin,LO1end,_);    % outer loop part 1
        cnode(lo2,LO2begin,LO2end,_);    % outer loop part 1
        cnode(li,LIbegin,LIend,LI1Data); % inner loop
      end fragment

      fragment reprLO(S,LO,LO)  % the outer loop representative
        node(LO,LOData);
      end fragment

      fragment reprLI(S,LI,LI)  % the inner loop representative
        node(LI,LIData);
      end fragment

  %%-------------------------------------------------------------
  %% Outpattern: how does the code look after transformation
  %%-------------------------------------------------------------
    outpattern
      fragment loops(body,body) % the new loop body
        cnode(body,li,id,LIOut,[]);
      end fragment

      fragment reprLO(_,_)      % one loop is removed
      end fragment

      fragment reprLI(LN,LN)    % only this loop remains
        node(LN,LNOut,[]);
      end fragment

  %%-------------------------------------------------------------
  %% Transfers: how do we update the execution information?
  %%-------------------------------------------------------------
    transfers
      LIOut.scope = forall(X2:LI1Data | copy(LN));
      LNOut.scope = LOData.scope;

      LIOut.execcount =
        forall(X5:LI1Data | mul(X5.execcount,LOData.iterations));
      LNOut.execcount = LOData.execcount;

      LIOut.iterations = forall(X8:LI1Data | copy(1));
      LNOut.iterations = mul(LOData.iterations,LIData.iterations);
  end transformation
```

Figure 4. Code for loop collapse transformation

Since the outer loop is removed, the fragment `reprLO` is empty in the out-pattern. The inner loop remains, but we need to recalculate its iteration count, which is contained within the `LNOut` variable. We also change the name of the loop to the value of the variable `LN`.

The purpose of co-transformation is to maintain execution information, and in this case it is done by recalculating the execution counts of the blocks inside the loop and the iteration count of the collapsed loop.

The iteration count of the combined loop is the product of the iteration counts of the outer and inner loops: ($i$ =

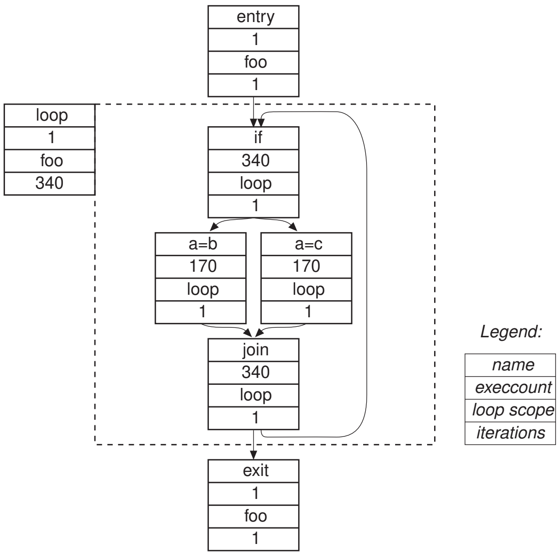
iterations, $e$ = execution count) $i_{inner,new} = i_{inner,before} \cdot i_{outer,before}$. This value is generated on the last line of the code in Figure 4.

The execution counts for the blocks in the resulting loop must be recalculated to be consistent with the new iteration count. Since the execution count was expressed relative to the iteration count of the inner loop, it is multiplied by the iteration count of the outer loop to yield an execution count relative to the new loop: $e_{block,new} = e_{block,before} \cdot i_{outer,before}$. This operation is performed by the second `forall` operation in the code in Figure 4.

To apply the transformation to our example graph, we have the following transformation call in the trace (which gives the actual names to use to locate the patterns in the graph):

```
LoopCollapse(foo,
             prepare,prepare,loop_outer,loop_outer,outer,
             if,join,inner,loop);
```

The final result of the transformation is shown in Figure 5. The code underneath the flow graph gives an impression of how the program would have been transformed at the C level.




```
for(n=0;n<340;n++)
  if(n<170)
    a1[n] = b1[n] * 2;
  else
    a1[n] = c1[n] * 2;
```

Figure 5. Example program after transformation.

The work of the co-transformer is to change the execution counts, loop scopes, and iteration counts to reflect the changes the program is subjected to. As can be seen from the figures, we have correct execution information in the final version of the program, even though we have changed its structure quite dramatically.

Running this example took less than half a minute on a Pentium 150 Mhz under Linux; most of this time was spent parsing the transformation descriptions from file. A larger example (13 transformation, 600 lines of ODL code) takes less than a minute to run.

# 6. Results

In our prototype [8], we used the ODL and datastructure illustrated by the example in the previous section to handle optimizations which have effect inside a single function. We do not consider transformations which just change the composition of basic blocks, since these do not affect the execution counts or looping structure.

Typical transformations handled are loop peeling, loop unswitching, and dead block elimination. The ODL programs were between 10 and 80 lines long.

There were some transformations which we could not handle, such as loop unrolling and branch elimination, mainly because our datastructure was too weak. For instance, to handle loop unrolling sensibly, we would need to represent how the outcome of conditionals depend upon the loop iteration number.

We ignored transformations which perform changes across function boundaries, since our present ODL interpreter only matches patterns within a function. This excludes function inlining and migration of code from one function to another.

# 7. Conclusions and Future Work

The co-transformer and our framework offers the following benefits for finding tight execution time estimates:

- Co-transformation offers the possibility to map information about program execution from source level to object code level, in particular for programs which are heavily optimized.
- Co-transformation handles the problem of combining the static nature of the program code with the dynamic nature of program execution.
- The co-transformer depends only on the compiler, not on the specifics of the target system or the analysis and calculation methods used.
- Only slight changes to the compiler are required to support timing analysis.
- The ODL makes it easy to express compiler transformations, in terms of pattern matching and replacement.
- Our framework provides the possibility to integrate several tools covering small parts of the larger problem of WCET analysis into one tool.

Although our prototype implementation shows promising results, much work is needed in order to produce a tool and data structure powerful enough to handle the full extent of program transformation possible in a modern compiler.

As a first step towards a complete timing tool, we will create a low-level analyzer and calculator. We will also continue our work on high-level analysis of programs [9, 11].

Our long-term goal is to integrate the tool with a compiler targeted to many different platforms, and thus to allow execution time analysis to be carried out for real-time programs on many different systems. We want to break away from tools which are specific to a certain processor architecture or target system. Since the embedded systems marketplace is very diverse, a tool needs to handle this diversity.

We have begun a cooperation with IAR Systems in Uppsala, Sweden, a company producing programming tools for embedded systems, in order to investigate how a timing tool can be integrated into a compiler. Timing analysis can certainly benefit from the knowledge about a program contained in a compiler, and we plan to integrate a co-transformer with the IAR compiler.

# References


[1] A.-R. Adl-Tabatabai. *Source-Level Debugging of Global Optimized Code*. PhD thesis, School of Computer Science, Carnegie Mellon University, June 1996.

[2] P. Altenbernd. On the False Path Problem in Hard Real-Time Programs. In *Proceedings 8th Euromicro Workshop on Real Time Systems*, 1996.

[3] R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, Dec. 1994.

[4] M. R. Boyd and D. B. Whalley. Isolation and analysis of optimization errors. In *PLDI 1993*, volume 28 of *ACM SIGPLAN Notices*, pages 26–35. ACM SIGPLAN, ACM Press, June 1993.

[5] R. Chapman. Worst-case timing analysis via finding longest paths in SPARK Ada basic-path graphs. Technical Report YCS-94-246, Department of Computer Science, York University, Oct. 1994.

[6] T. M. Chung and H. G. Dietz. Language constructs and transformation for hard real-time systems. In *Proceedings of the Second ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, June 1995.

[7] M. Copperman. Debugging optimized code without being misled. Technical Report UCSC-CRL-92-01, University of California, Santa Cruz, May 1992.

[8] J. Engblom. Worst-case execution time analysis for optimized code. Master's thesis, Department of Computer Systems, Uppsala University, Sept. 1997. DoCS MSc Thesis 97/94.

[9] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of Euro-Par 97, LNCS (Lecture Notes in Computer Science) 1300*. Springer Verlag, Aug. 1997.

[10] C. Forsyth. Implementation of the worst-case execution analyser. Technical Report Hard Real-Time Operating System Kernel Study Task 8, Volume E, York Software Engineering Ltd, July 1992.

[11] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. In *Joint Workshop on Parallel and Distributed Real-Time Systems, Geneva, Switzerland*, Apr. 1997.

[12] M. G. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time of code segments. *Real-Time Systems*, pages 159–182, Sept. 1994.

[13] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1995.

[14] A. A. Kountouris. Safe and efficient elimination of infeasible execution paths in WCET estimation. In *Proceedings of RTCSA'96*. IEEE, IEEE Computer Society Press, 1996.

[15] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32th Design Automation Conference*, pages 456–461, 1995.

[16] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modelling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 254–263. IEEE Computer Society Press, Dec. 1996.

[17] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An accurate worst-case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[18] G. Ottosson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proc. SIGPLAN 1997 Workshop on Languages, Compilers and Tools for Real-Time Systems*, June 1997. Also avaible as ASTEC Report 97/01 from the Deparment of Computer Systems, Uppsala University.

[19] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, Mar. 1993.

[20] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, Apr. 1995.

[21] F. Stappert. Predicting pipelining and caching behaviour of hard real-time programs. In *EUROMICRO Workshop on Real-Time Systems*, June 1997.

[22] A. Vrchoticky. Compilation support for fine-grained execution analysis. In *ACM Sigplan Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994.

[23] R. White, F. Müller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 192–202, June 1997.