Worst-Case Execution Time Analysis for Optimized Code

Jakob Engblom
Dept. Of Computer Systems
P.O.Box 325
S-751 05 Uppsala
Sweden
jakob@docs.uu.se

January 1998

M.Sc. Thesis DoCS 97/94
ASTEC Technical Report 98/01

Worst-Case Execution Time Analysis for Optimized Code

Master's Thesis
DoCS 97/94

Jakob Engblom

Department of Computer Systems
Uppsala University
Uppsala, Sweden

e-mail: `jakob@docs.uu.se`
homepage: `www.docs.uu.se/~jakob`

September 1997

ABSTRACT

In the field of real-time systems, accurate estimates of the worst-case execution time of programs are required for real-time modelling and verification, scheduling analysis, feasibility analysis, and dimensioning of the system hardware.

At present, such estimates are produced by measuring program runs on inputs which are believed to produce long execution times. Unfortunately, such measurements provide no guarantee for finding the worst execution time, thereby producing unsafe systems. To obtain safe estimates (estimates which do not underestimate the worst-case time), we have to use formal static analysis.

Because the market demands high performance, low cost products, we must use optimizing compilers when developing software for real-time systems. An optimizing compiler improves the performance of a program and reduces the size of the code, both important factors to save costs while maintaining performance.

An optimizing compiler makes static analysis of a program more difficult by complicating the relationship between the program source code (where we can obtain knowledge about possible program executions) and the object code (where we can find concrete execution times).

In this thesis, we present (1) a new framework for static timing analysis of optimized programs and (2) *co-transformation*, a new approach to keep track of timing information while performing the various transforming and optimizing steps in a compiler. The co-transformer is based on a close cooperation between compiler and timing analyzer.

We have implemented a prototype co-transformer, and evaluated it using a representative set of compiler optimizations. We show that the idea is feasible, but that much more work is needed to design a datastructure powerful enough to capture all necessary information about program execution.

In addition, the thesis points out some directions for future research in the field of worst-case execution time analysis in particular and the real-time programming field in general.

Contents

Foreword

I believe that it is the mission of all computer science graduates to try to bring the best methods, skills, and tools they learn during their studies with them to industry, to help advance the state of practice.

Over the past few years, I have studied a number of fields of computer science, all which I believe have a lot to offer industry. Among the subjects which I feel I should help bring out into the famous Real World are formal methods, software engineering, programming tools supporting good engineering practice, and better programming languages.

At the same time, I have had a feeling that I should go into research, to help advance the state of the art, and not just go out into industry to advance the state of practice.

In this thesis work, I have found a field of study where I can actually do both: advance the state of the art and have some hope to improve the state of the practice. Execution time analysis applies theoretical computer science to build a software engineering tool. By combining the tool with a compiler, we have the opportunity to introduce it on the market with a higher probability of success than a stand-alone tool would have.

My present goal is to continue this research and development after graduating, and to work with the development of new exciting tools, thereby bridging the gap between industry and academia.

Credits

I am very grateful to my thesis advisors, Hans Hansson and Andreas Ermedahl at Uppsala University, and Anders Berg at IAR Systems, Uppsala, for their help in producing this thesis. Additionally, I thank Peter Altenbernd (visiting Uppsala University on a leave from C-LAB in Paderborn, Germany) and Jan Gustafsson (Mälardalens Högskola, Västerås), for interesting discussions and helpful feedback on my work. Peter has been especially helpful in the writing of this thesis, providing sharp criticism and making the final result much more focussed and to the point.

IAR Systems provided the thesis topic and funding, and Uppsala University provided me with an office and computing resources. I think this is a good example of a successful industry–academic cooperation, which in this case was performed under the auspices of the ASTEC[1] program.

Throwing one away

Finally, as stated in the introduction (coming next), the executable result of this thesis was very throw-away-able `:-)`. Nothing else was to expect however; the time allocated for a Master's Thesis is a little too short to both structure and analyze the problem, and implement something useful. I learned a lot in the process, and I feel optimistic about the continuation of my research. To quote a well-known(?) movie:

> When I started here, all there was was swamp. Other kings said I was daft to build a castle on a swamp, but I built it all the same, just to show 'em. It sank into the swamp. So, I built

[1] ASTEC, Advanced Software TEChnology, is a Swedish national competence center in software development. See the ASTEC homepage at `http://www.docs.uu.se/astec/`

> a second one. That sank into the swamp. So, I built a third one. That burned down, fell over, then sank into the swamp, but the fourth one... stayed up! [CCG+75]

This is the way the world of reseach and engineering works: you have to try something a few times to obtain the understanding which allows you to build better tools, cathedrals, or swamp castles. I hope that I some day will see my fourth swamp castle materialize: the complete timing analysis tool.

Uppsala, January 20, 1998, Jakob Engblom

Chapter 1

Introduction

A real-time system is a system where both the result of a computation and the time at which the result is produced are relevant for the correctness of the system. Real-time systems are usually found as embedded control systems in various real world applications: large and complex like production processes, ABS brakes for cars, or aircraft engine control; or simple like toys and household appliances.

In the real-time systems field, there are many reasons to determine the execution time of a program prior to using it in a system. The reasons include modelling, scheduling and schedule analysis, determining resources needs, feasibility analysis, dimensioning systems, and many more.

The execution time of a program can be defined and measured in many different ways, but there are three common measures in use: the *worst-case execution time* (WCET) the *best-case execution time* (BCET), and the *average execution time*.

Unlike most other fields of computer science, in the field of real-time systems we are interested more in the worst-case performance than in the average case. It is the WCET of a program which is used in analysis. Because the systems are used to control real-world devices, and failure to meet deadlines can have catastrophic effects, we need safe WCET estimates when we construct systems (we can only produce estimates, since an exact computation is infeasible for complexity reasons).

There are two ways to estimate the worst-case execution time of a program: to *measure* it experimentally, or to calculate it by *static analysis*. Measuring an execution time is an unsafe practice, since we cannot know whether we have managed to capture the worst case in our measurements. A static analysis gives safer results, but there are at present no commercial tools available for static timing analysis of real-time programs.

A good tool for static timing analysis offers improvements in product quality and safety for embedded and real-time systems. Development time could be saved as the verification of timing behavior is automated and simplified. This thesis is a step towards such a tool.

The analysis of a program can be carried out either at the source code level or at the (compiled) object code level. Analysis of program behavior is more efficient at the source code level, since this is the most accessible form of program representation. At the object code level, too much information about the program structure has typically been lost. On the other hand, timing analysis depends upon the time taken to execute the object code, which means that we must analyze the object code to obtain necessary information for the timing analysis.

The problem is how to combine the information from the the source code analysis with the timing information from the object code, in order to perform calculations to estimate the WCET of a program. Our approach is to map the information from the source code down to object code, and work on the object code level when estimating execution times.

The mapping problem is created by the compiler, which transforms the source code to the object code. If the compiler is simple, so is the transformation, but a modern *optimizing compiler* can make the relation between source code and object code anything but trivial.[1]

We need to analyze optimized code, since unoptimized programs have different properties compared to optimized code, and we must analyze a system in the shape it will have when we ship it to customers. The shipping version typically includes optimized code.

We have investigated the mapping of source code information down to the object code. Our approach is to transform the information about the program source code in parallel to the compilation process, an approach which we call *co-transformation*. We present the following results:

- We have created a new framework for WCET analysis, dividing the problem into a number of clearly separated components. This helps us analyze the problem of WCET analysis, and allows for research and development to proceed on several different parts of the tool independently. It also allows us to integrate results from several sources into one tool, and to port the tool to new architectures more easily.
- We have designed a novel device called a *co-transformer*, which is a tool that transforms timing information (or other information) in parallel to the transformation of the program code in the compiler. The co-transformer depends upon information from the compiler about how a certain program is transformed, and knowledge about how the compiler implements its transformations. The co-transformer approach can handle a larger set of compiler optimizations than previous approaches, and it allows for more information to be mapped from the source code to the object code, which gives us tighter timing estimates. We have also made the problem clearer by analyzing it in isolation.

In comparison to the present state of the art, we have presented a clear division of the WCET problem, and used this division to attack the problem of performing timing estimates for optimized code. We have localized the problem of handling optimizations to a single component, the mapping of information from the source code to the object code, and have shown how we can perform this mapping using a co-transformer.

This thesis is organized in the following way:

- Chapter 2 presents the problem of timing analysis, and discusses the issues of optimized code and portability of tools.
- Chapter 3 presents the new framework for constructing timing tools.
- Chapter 4 presents the general concept of the co-transformer and the data structures needed by it, Chapter 5 presents our prototype tool, and Chapter 6 presents the results of evaluating our prototype.
- Chapter 7 gives an overview of previous work and related research.
- Chapter 8 presents conclusions and plans for future work.
- A number of appendices complement the thesis. Notably, Appendix C contains a complete co-transformation example for a small program.

[1] This causes problems for source code debuggers as well, where it is difficult to present a coherent picture of the program execution at the source code level when the object code is very differently structured.

Chapter 2

Background: Real-Time Systems and WCET Estimation

A real-time system is a computer system which has demands upon when a result is produced, and not just what the result is. This is often the case in control applications, where a computer is used to control some process —everything from running a factory to controlling a sensitive aircraft engine. If output is not produced within a certain time, the system controlled may fail, with potentially disastrous consequences.

An embedded system is a computer system used as a component in some larger system. It is "a computer which does not look like a computer"[1]. They are used to perform or control some function in the total system, and they usually have some form of real-time constraints, making them real-time systems.

However, there are real-time systems which are not embedded, and embedded systems which are not real-time. An example of a non-embedded real-time system is multimedia, where exact timing is important to assure a smooth playback of sound and video, on a desktop system.

In this thesis, we consider real-time systems, with a focus on embedded real-time systems.

2.1 The correctness of real-time systems

Real-time programs differ from "ordinary" programs in that they have to be correct both in terms of what is delivered and when it is delivered. In [BW97, p. 2] the following definition of the correctness of a real-time system is given:

> The correctness of a real-time system depends not only on the logical result of the computation but also on the time at which the results are produced.

That a real-time program needs to be logically correct is obvious, and not unique to real-time programs. What might be different is the *need* for logical correctness. Real-time systems are often used to control real-world systems, the failure of which can cause physical damage or even injury and loss of human lives. This makes the demands for correctness higher than for ordinary desktop programs. Nobody is hurt when a word processor crashes; somebody could get hurt when a plane does.

Second, a real-time system needs to have correct timing behavior, under all circumstances. Correct timing behavior for a certain program or other piece of code is usually defined as "guaranteed to finish within a certain amount of time". There are various levels of critically: in a hard real-time system, a missed deadline means that the result is useless. In a soft real-time system, a result which is produced after the deadline usually has some value, even though it is less than if the deadline had been met.

[1] As stated by Hans Hansson from Uppsala University on the ARTES-kickoff in Lund, Sweden, August 20, 1997.

When discussing the meeting of deadlines, we are interested in the *worst case execution time* (WCET) of a certain program.[2] The WCET is the longest time the program takes to run to completion, given the worst possible circumstances. If we have a correct estimate of the WCET, no execution of the program will take longer than the WCET.

In some cases, it also of interest to the system designer to know the minimum execution time of a certain program, for example when outputs must be separated in time by a certain minimum time, or to perform scheduling analysis [Alt96b]. This is called the *best case execution time*, BCET.

In the general case, we cannot determine the exact execution time of a program, but only an estimate. The reason is that the number of execution paths of a program is exponential in the size of the program (except for simple programs). More on this below.

The WCET estimate may overestimate the actual WCET, but it must not underestimate it, and the BCET estimate may correspondingly underestimate the actual BCET, but never overestimate it. See Figure 2.1.

Knowing the the WCET of a program is useful in a number of contexts:

- To perform task scheduling analysis. Most approaches to real-time scheduling assumes that the maximum execution times of the programs involved in the scheduling problem are known [HBW94].
- To check that a program meets timing constraints; an example is a feedback system where sensors deliver inputs at a regular pace, and the software has to produce output within a certain time of receiving input.
- To find the most time-consuming parts of a program in order to target optimizations (profiling).
- To compare two different implementations of the same algorithm and select the faster one.

We focus on WCET estimates for the rest of this thesis, since the WCET has more uses than the BCET, and most of the literature in the field deals with WCET. Furthermore, calculating the BCET can usually be done using the same techniques used for calculating the WCET.

2.2 The nature of WCET estimates

There are a number of measures and concepts involved when discussing execution time estimates.

We start with the actual execution times: The *actual* WCET is the longest time the program will ever take to execute. The *actual* BCET is the shortest time the program will ever take to execute. The *average execution time* is somewhere between the actual WCET and the actual BCET. It is usually very hard to determine the exact actual WCET (or BCET) of a program, as this could depend upon inputs received at runtime.

Timing analysis aims to produce *estimates* of the WCET and BCET, using a method giving reasonable results in a reasonable time. A timing estimate *must* be *conservative*. An underestimated WCET is worse than no WCET estimate at all, since we will produce a system which rests on a false assumption, and which we believe is correct, but which certainly *can* fail. Note that execution time estimates are always defined by the assumptions we make about input values, sizes of data, etc.

On the other hand, it is trivial to produce conservative but perfectly useless estimates. A statement like "the program will terminate within the next 5 billion years" is certainly true, but not very useful (dimensioning for a WCET like this would give a tremendous waste of resources). To be useful, the estimate must not only be conservative, but also *tight*. A tighter estimate is closer to the actual value. Untight estimates lead to a waste of resources in hard real-time systems.

In general, producing a tighter estimate requires more computation time, and more complicated algorithms. A less tight estimate can be produced in less time using simpler algorithms.

[2] Some researchers call this this measure "MAXT", for *maximum execution time*.

Figure 2.1 gives a graphical presentation of the relation between actual and estimated times, and the direction in which tightness takes us.

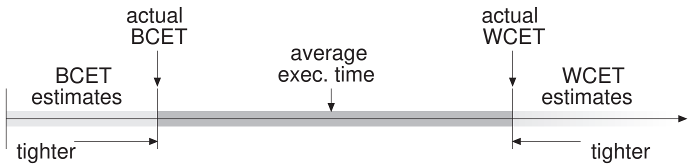

Figure 2.1: Measures and estimates of program execution times.

The *safety* of a timing estimate determines how trustworthy the estimate is. The safety depends upon the method used to obtain the estimate, and the assumptions used in the estimation.

The state of the practice in WCET analysis today is *measuring* the run time of a program, running it with "really bad" input (or a set of typical inputs). Typically, we then multiply by some safety factor, and hope that the real worst case lies inside our estimate. However, there are no guarantees that we find the worst input, and measurements can only produce statistical evidence of the probable WCET and BCET, but never complete security (except for trivial examples). See Section 7.1 (page 55) for a short description of various measuring methods.

Obtaining a safe theoretical maximum requires formal analysis of a program, by hand or by machine. The goal is to find the longest executable path through the program, including loops and function calls. This is known as *static analysis*, since the program is not executed.

Regardless of the method (measurement or static analysis), we must always be very careful to state the assumptions behind our estimates. In the case of static analysis, we need to put bounds on various values in order to enable the analysis to terminate at all. The quality of these bounds must be questioned[3]. We must realize, distasteful as it may seem to the theoretical computer scientist, that our painstakingly produced WCET estimates may be exceeded, given that some assumption is violated. The assumptions are not usually derivable from theory, but must be produced by humans working with the system. The generation of assumptions is more of an engineering issue than pure computer science.

2.3 Complexity of static WCET estimation

Calculating the WCET of a program statically is in the general case undecidable. It is a case of the halting problem; if we can determine a time bound within which an arbitrary program will terminate, we have determined that it *will* terminate, thus solving the halting problem.

It *is*, however, decidable whether a program will terminate *within a given time-budget*. The algorithm is simple: just run the program with all possible inputs and check that a result is produced within the given time limit; if the time limit is exceeded we abort the execution and note that the program did not fulfill its specification. The set of all inputs is enumerable, since the time budget also limits the size of the input: no computer can read an infinite amount of input in finite time[4] (assuming that none of the inputs are infinite in itself). The size of the set is likely to be quite large, since the number of possible values of input is $2^{(bits\ of\ input\ read)}$.

In practice, the termination of a program and the termination of the analysis of the program is assured by giving bounds for all loops, or a time budget for the entire program. Which limits are provided depends upon the WCET tool, and the methods used in the analysis.

[3] If input happens to exceed the assumed bounds, what will happen? One extreme approach, used in the MARS system, is to throw a run-time error. A more common approach is to let the system run, and hope that not all subsystems have exceeded their WCETs all at once.

[4] Thanks to Professor Bengt Jonsson, Department of Computer Systems (DoCS), Uppsala University for the simple argument.

2.4 WCET estimation problems

Finding the worst-case execution time statically implicitly means that we identify the longest possible execution of the program under consideration. As we are only producing an estimate, it is certainly possible that the path we find will never be executed, or that it cannot be executed due to the fact that certain input values or combinations of input values will never occur. Our goal is, however, to come as close as possible to the actual WCET, and this requires us to look for the sources of untightness.

For this discussion, we introduce the concept of a basic block graph (also known as a program flow graph). This is a graph which represents the structure of an object-code program. Each node in the graph is called a basic block, and is a sequence of instructions which are executed together. The only entry to a basic block is at the beginning, and the only exit is at the end. Typically, labels and jumps in the object code determine the extent of the basic blocks. The basic blocks can be considered to contain assembly language instructions.

Looking at the low-level representation of a program as a basic block graph populated by information about the program flow, we realize that there are two main causes for untightness in estimating WCET: *loops* and *infeasible paths*.

On a larger scale, *Function calls* also cause problems.

The execution time of a program is obviously dependent upon the hardware the program runs on: processor speed, memory system characteristics, etc. This is dealt with as a separate issue.

2.4.1 Loop bounds

Loops are at the heart of any realistic program; a program without any loop is trivial. For most programs, most of the execution time is spent in loops.

If the number of iterations of a loop is overestimated, the resulting WCET estimate will be untight by a factor proportional to the overestimation. We want our loop bounds to be as close as possible to the actual worst case.

It is difficult to produce good estimates of loop bounds in the general case, since they can depend on input data unknown to a static analysis.

2.4.2 Loop iteration dependences

Another loop-related problems is that if a loop body contains a conditional branch, knowing too little about the dependence between the loop iteration and the path taken inside the loop can lead to grave overestimation.

A typical example is a read loop where a time-consuming buffer refill process is invoked only once per one thousand iterations. If we do not know how often the refill is invoked, we must assume it happens every time through the loop, leading to a grave overestimation of the time required to execute the loop.

A similar problem is posed by dependences in iteration counts between inner and outer loops in a loop nest. Consider the following program:

```
for i=1 to n loop
  for j=i to n loop
    a[i,j] = f(a[i,j]);
  end loop
end loop
```

This only affects an upper triangle of the array `a`, and the actual execution count is $\frac{n^2}{2}$. If we cannot capture the dependence, we would end up with an estimate of n^2, since the outer loop executes n times, and the worst case for the inner loop is n executions. This is untight by a factor of two.

2.4.3 Infeasible paths

Another problem is to correctly estimate the actual execution time of a non-looping sequence of statements. This may include branches in the flow-graph, and the longest executable path between two points must be found. The problem here is that some paths are not *logically* possible, even though they seem possible considering only the *structure* of the program. A short example:

```
if (a==b) then
  A;     // takes 20 time
else
  B;     // takes 5 time

if (a!=b)
  C;     // takes 30 time
else
  D;     // takes 5 time
```

In this case (assuming that A and B does not change the variables `a` and `b`), A and C will never be executed in sequence, and neither will B and D. A→C and B→D are examples of *infeasible paths*.[5] However, looking at the structure of the program without considering the conditions for the branches, A and C may both be assumed to execute in the same execution of the code fragment.

In the example, using infeasible path information gives a maximum execution time of 35, from the path B→C, while not using it gives 50, from the path A→C.

This leads us to define the *structural* WCET of a code fragment:

> The *structural WCET* is the longest time possible through a code fragment using a simple `max` operation to calculate the time taken through a split in the flow graph, i.e. not taking information about path dependences into account. It corresponds to the longest structural path through the code fragment graph.

The structural WCET gives an upper bound on the time taken to execute the code between two points, and is easy to compute (the time complexity is just $O(n)$). Adding more information about the program flow will tighten the WCET until, in the perfect case, the actual WCET is found. Any approach that tightens the WCET beyond the actual WCET is faulty (remember that we *must* make a conservative estimate).

In general, the problem of identifying infeasible paths is exponential and computationally intractable, but we can produce good approximations using heuristics. See Section 7.2.1 (page 56) for previous work in the field of detecting and representing infeasible paths.

2.4.4 Function calls

Function calls must be analyzed in context in order to yield tight execution times. If we assume a single WCET value for a function, and use this to calculate the time for all invocations of that function, we risk huge overestimations.

An example of the problem of overestimation is a dispatcher function where different code is executed depending on the value of a selector argument. The selector will usually be a constant at the call site, and this information must be used if we are to obtain a tight WCET estimate.

In the example below, the time taken to perform a multiplication may be very different from the time taken to perform an addition.

[5] The term *false path* is also used in some literature [Alt96a].

```
function foo(operation,arg1,arg2)
  case operation of
    multiply: multiply(arg1,arg2);
    add:      add(arg1,arg2);
  end case
end function

function call(...)
  ...
  X = foo(add,Y,Z);
  ...
end function
```

Another case is where the bounds of one or more loops inside the function is determined by the arguments to the function. If we know the value (or a possible range of values) of the arguments, a tighter WCET can be calculated compared to assuming the worst case, which is the greatest possible value for the datatype of the argument (like `MAXINT` for a C integer).

A *function instance* is a specific invocation of a function, with known values of the parameters (the values may be sets of values, depending upon the information obtained about the program). We believe that each function instance must be analyzed as a local WCET problem in its own right. See Section 7.2.2 (page 57) for other approaches to the problem of timing functions.

2.4.5 Execution times for basic blocks

A basic block is a compiler term used to denote a piece of code which is executed in its entirety or not at all. This means that a basic block does not contain any entries or exits, except at the beginning or the end. When analyzing a program, the basic block is a natural atomic unit.

In order to calculate execution times, we need to have information about the time taken to execute each basic block. One approach is to give the execution time of each basic block as symbolic constant, in which case we produce a formula describing the execution time of the program as a function containing a large number of unknowns. A more common approach is to generate a local WCET estimate for each basic block, and then use this information to calculate the execution times of the larger structures in which the block is a part. We can either ascribe a single time to each basic block, or a simple (min, max) pair. This depends upon the method used to obtain the low-level times.

The times as such depend upon the CPU used in the system, the memory system, etc. Since data can be placed in memory of different speeds, the time required to execute a single basic block may vary according to which data references are actually performed. Often, this information can be deduced from the object code, since references to different types of memory tend to look different.

The calculation of execution times for basic blocks is made vastly more complex when we want to take the effects of caches and pipelines into account (where we can get global dependences inside the program, forcing us to consider the effect of every piece of the program on every other piece).

For a list of approaches to the problem of obtaining low level timing estimates, see Section 7.4 (page 59).

2.5 Determining the WCET of optimized code

When developing any kind of computer system, and real-time systems in particular, it important to actually work with the final version of the programs involved. This means debugging and timing the optimized version of the code. The alternative —to work with an unoptimized version during development, and then recompile with full optimization before shipping— is not feasible in most cases. We give a list of reasons why we want to work with optimized code:

- The optimized version of a program obviously has different timing characteristics than an unoptimized program. This has several implications:
  - When using timing analysis to reduce the hardware requirements, analyzing unoptimized code gives unnecessary overestimation which makes the hardware over-dimensioned and more expensive. You basically render the optimizations of your compiler economically useless.
  - When debugging a system consisting of several programs, their relative timing behavior is changed when one of them is optimized or changed. Debugging such a system using the unoptimized version and then optimizing may yield new errors because of the changed timing behavior.
  - Scheduling will be wrong: the optimized program may run faster (optimized for speed), in which case we waste processor resources and perhaps forcing us to include fewer features in our products as we cannot guarantee that there is time for all functions.
  - A system which is perfectly schedulable in its (speed) optimized version may be assumed unschedulable when analyzing it using values obtained from unoptimized code.
  - Scheduling will be wrong: the optimized program may run slower (a possible sideeffect of optimizing for space[6]), in which case the analysis is invalid.
  - When an application is pushing the performance envelope it needs the speed given by an optimizer to meet requirements. The alternative conclusion is that the system cannot be built.
  - A speed optimization may speed up the average case, but actually slow down the worst case. In this case, our unoptimized WCET estimates are actually wrong.
- The amount of memory available in the target system may be so small that a program must be optimized (for space) to fit into memory. The program simply cannot be run without being optimized.[7]
- For debugging, the necessity of working with the optimized program has been pointed out in several works [AT96, Cop92, Coo92]. The main reasons cited are:
  - Language definition ambiguities and misunderstanding may cause an optimized program to differ from the expected behavior of the source program.
  - Optimizations may mask or reveal bugs compared to unoptimized program because of changed memory layouts, less redundant code, and other subtle effects.
  - You want to debug the code you ship.
  - There may be errors in the compiler which can only be discovered by examining the optimized code.
  - No compiler actually guarantees reasonable treatment of erroneous programs.

Today, timing analysis for an optimized program must be performed using measurements. We cannot use measurements which change the code (by inserting calls to timing functions) unless we intend to ship the program with the (often expensive) instrumentation code still in it, but some techniques (simulators and emulators) can time a program without changing it.

As stated above, we would like to perform static analysis to obtain more confidence in our WCET and BCET estimates. And we need to perform this analysis using the optimized versions of programs.

Most research in the field of static analysis today consciously ignores optimized programs, or assumes that the problem is easy to solve. Work centering on the basic analysis of straight-line code on the assembly-language level avoids the problem, since there is no difference between timing optimized and unoptimized assembler code.

[6] Experience shows that optimizing for space usually also gives a faster program; only really agressive space optimizations give slower programs.

[7] This was pointed out by Anders Berg of IAR Systems, and is a very common problem in real-time systems development.

The problem with analyzing optimized code is to connect high level (source code) information about a program to the optimized target code. This problem is shared by debuggers, compilers, and timing tools.

The relation between the source code and the object code can be quite complex in a modern optimizing compiler. Operations can be reordered, moved, or eliminated, compared to their appearance in the source code. Optimizations like loop unrolling and loop peeling make several copies of a loop body, and may overlap their execution. A variable may exist in several different copies at once. Code may be moved or copied between functions (an example of this is function inlining). See Section 6.1 (page 35) for more on how optimizations can complicate the relation between the source code and the object code.

We believe that this problem (of mapping information from the high level to the low level) is serious, and that it deserves attention as a problem separate from any particular tool. This is the focus of the second part of this thesis, where we present our approach to handling the mapping problem: the co-transformer.

2.6 User interface issues

An issue which has received comparably little attention is the user interface of a timing analysis tool. The technical issues involved in producing any kind of estimate are complex enough to keep researchers busy, and the presentation of results seems a little unnecessary before the results have been generated.

Nevertheless, user interface aspects must be taken into consideration early in the process when designing a timing analysis tool, so that the tool generates information which can be understood and used by the programmer. There are three main parts in the interface of a timing tool:

- *Supplying information* about the program. The programmer may help the high-level analysis by specifying limits on input values, bounds for loops, and other information.
- The *specification of program sections to time.* The user of the tool needs to specify which portions of the code to calculate execution information for.
- The *presentation of results.* The results of the timing analysis must be presented to the user. This is related to the specification of what to time.

Information about the program

The information supplied about the program is used in the analysis of the program, at the object code or source code level. Information about the source program (input value limits etc.) can be entered as a part of the program code itself (*annotations*), or separately entered into the timing analysis tool. See Section 7.3.1 (page 58) for a number of annotation methods.

Parts of a program to time

The specification of program sections to time depends upon the resolution possible in the timing analyzer. It has a large influence on the design of the timing analyzer, and we will investigate this in some detail.

In the presence of highly optimizing compilers, the problem of linking the instructions in the object code to statements or other entities in the source code is quite difficult. [8]

There are a number of possible units of correspondence between the timing information and the source program, each with its own advantages and disadvantages:

- The user would like to have timing information about individual *source statements*, in order to find the most time-consuming parts of the program and to identify culprits when programs do not fit

[8]The user interface problems faced when constructing a WCET tool is very similar to those of symbolic debuggers (this is noted in [KHR+96]). In [Ger94], Richard Gerber points out the importance of traceability between the source code and the generated assembly code in order to support debugging of real-time programs.

their timing slots. Unfortunately, this mapping is not very easy to obtain, since one source statement may be present in several copies in the final program, or may be spread out and interleaved with other statements.

Restricting the allowed optimizations to those that allow for an easy mapping between source code and object code is not a solution, as timing analysis on a non-final version of a program is rather meaningless (see the introduction to this thesis). We can choose to ship a less optimized version, but this could make the program too large or too slow.

The tool could try to *hide the complexity* from the user by patching the displayed code and timing information to undo the optimizations. This is both very complex and a bit misleading to the user, as (s)he cannot see the code that is actually executed and timed.

- The *basic block level* is not as intuitive as the source statement level, as basic blocks are hard to identify in the program source code. Problems include hidden basic blocks (jump-code for the evaluation of boolean expressions, the C `?:`-construct, Ada bounds checks, etc.). The compiler may also move, remove, and create additional basic blocks during optimizations.

- The *function level* seems attractive: a function should be an entity with quite clear borders both for the compiler and for the user. However, this is not true with modern aggressive optimizers. For example, functionality (instructions and loops) may be moved between functions. Functions may be inlined into other functions, or several specialized copies may be created.[9]

  Nevertheless, the time taken to execute *a certain function call* in the program can be meaningfully defined both for the user and the timing analysis tool.

- The time taken to execute *an entire program* is appropriate if the program is one in a set of separate tasks. Such programs are invoked by a scheduler external to the program, and the interesting time is their end-to-end execution time (which is needed for scheduling the tasks). Since looping is taken care of in the scheduler, a program will run to termination each time it is invoked. There is no need to loop infinitely inside the program.

  On the other hand, in the case that the program runs alone in the target system (without any scheduler), the program will typically run in an infinite loop, reacting to external events and controlling its environment. The WCET of such a program is trivial: it never terminates, and useless: we do not know whether the main loop runs each iteration fast enough. One plausible way to handle this is to report the WCET of the program as two parts: one for the initialization section and one for the main loop body.

- In many cases, the interesting measure is the time taken between an input and its corresponding output. This is timing on an *event basis*, and is used in many formal treatments of real-time systems, where a distinction is made between externally visible events and internal state transformations.

  Events are usually coded as a write to or read from some hardware register (identified to a C compiler by the keyword `volatile`) or as a function call, and should be quite easy to identify automatically.

- Another approach is to insert start and stop *markers* into the code, and report the times between the markers. This is similar to the event based approach. From a compiler perspective, however, the difference can be quite large, since the markers have no meaning in the program, they do not restrict optimizations in the same way that output and input operations do.[10]

  The placement of markers must be restricted to sensible combinations: every path from a start marker must pass the corresponding stop marker, and every path to the stop marker must pass the corresponding start marker.

[9] For more on specializing functions for certain input, see [CP+91], or any text on partial evaluation.
[10] This approach is taken in [Bör95]; see section Section 7.3.1 (page 58).

Presentation of results

The presentation of the results of the timing analysis should make it easy to draw conclusions and identify timing culprits in the code. It is the main tool by which the programmer tries to understand her program.

If we specify the parts of the program we want to time before invoking the timing analyzer, the presentation will follow the same partitioning of the program. It might be possible to allow a programmer to zoom in and out on her program, but this requires that the timing analyzer generates information about each part, either on the fly or as extra information in one long analysis pass.

The presentation of timing results could be integrated into a comprehensive program examination tool. Other information of interest for a such a tool is data about how the program was optimized, stack use limits, the layout of data and code in memory, and the results of various semantic and syntactical analysis, etc.

2.7 Porting a timing analysis tool

A problem with the real-time systems market is that there is a huge number of different processors and technologies available for building systems. A designer does not want to be constrained in her choice of components by the tools available, and often other factors are far more important than the software tools available (like cost or power consumption).

This means that an effective tool must be available for a number of different system components, and that it must be easy to retarget the tool.

Timing analysis operates on such a low level that all parts of the target system affects the result. Some examples of issues which have to be dealt with are:

- The *processor* is very important: how long time instructions take to execute, pipeline effects, etc.
- The *memory system* is an equally important factor: the presence or absence of caches, wait-states, and memories with different speeds can have a dramatic impact on the execution time of a program.
- *I/O systems* affect the performance of a program by introducing latencies for certain operations.
- The *operating system* used affects the time taken to perform system calls, interrupt latencies, and similar factors.
- The *library functions* used in a program also affect the time taken to execute it.

We believe that memory and processor factors must be taken into account even by the most basic tool. This is where most research, including this thesis, is focused today.

Libraries can be handled by analyzing them together with the program (by including their source code in the entire system being analyzed). An alternative is to pre-calculate timing information for libraries, and store this in a separate file provided with the library —including a large library in the analysis of a `hello world` program is certainly very inefficient.

We can only analyze sequences of code which run uninterrupted. For cache and pipeline analysis, the possibility of an interrupt makes the analysis impossible or forces us to assume an unrealistically pessimistic worst case (for example, constant cache misses, since we can be interrupted at any time). This can be handled in the schedulability analysis[11], where we have information about other tasks in the system.

I/O operations must have bounded execution times, which can be included in the analysis when I/O is performed. If the system cannot give such guarantees, it can hardly be considered a suitable real-time system.

[11] See for example [Alt96b].

Still, there are many functions and algorithms in a timing analyzer which are *not* dependent upon the target machine. The architecture presented in Chapter 3 strives to isolate the issues related to the target system to a single component. Isolating the target dependences to a single component allows the rest of the timing analysis tool to be retargeted without change.

Chapter 3

A New WCET Analysis Framework

This chapter presents our general framework for performing WCET analysis. Our goal is to provide a clear division of the problem into subproblems which can be treated in isolation. The advantage of such a division is that it is easier to analyze the various subproblems in isolation, and that it facilitates the combination of results from different research groups into one tool. It also makes a good model for the implementation of a maintainable and retargetable tool.

We assume that we obtain information about the execution behavior of a program from the source code, and that we use this information to guide our calculations at the object code level.

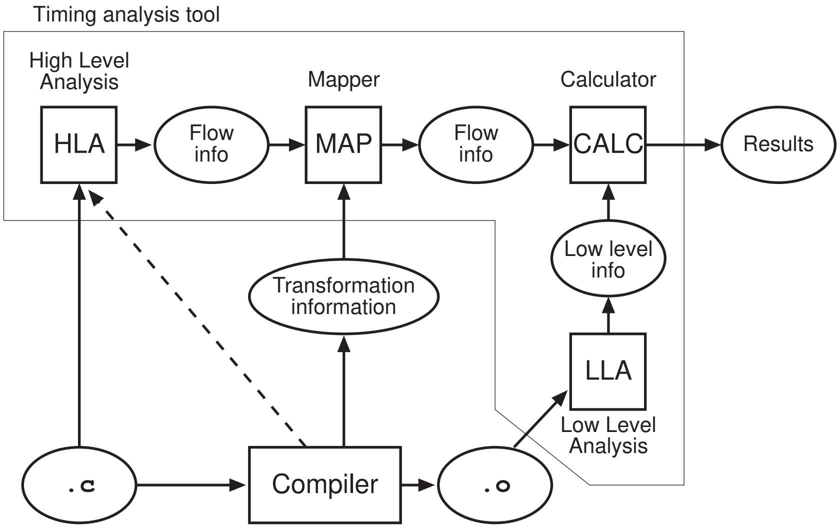

Figure 3.1: Components of a WCET tool.

In general, the problem of timing a program can be broken down into four distinct parts (the code names refer to Figure 3.1):

- The *high-level analysis* (HLA) analyzes the source program and derives information about the program flow.
- The *low-level analysis* (LLA) analyzes straight-line segments of object code (basic blocks), taking memory system, processor, and other target system characteristics into account. For a system with caches and pipelines, the analysis provides information used by the calculator. It generates raw data used by the calculator to calculate the execution time of a program.

- The *mapper* (MAP) maps the information from the HLA to the low level where the LLA is performed. The information generated corresponds structurally to that generated by the LLA. Our approach to the mapper is the co-transformer described in Chapter 4.[1]
- The *calculator* (CALC) takes the mapped program flow information and the information from the LLA and performs the calculations necessary to produce an execution time estimate. Pipelines and caches are handled in the calculator, using raw data provided by the LLA.

There are a number of data structures involved in the timing analysis:

- The *flow information* is generated by the HLA. It contains information about how the program can execute, such as information about loop bounds, infeasible paths, and function calls. It is transformed by the mapper into a form useful to the calculator. More information about the data structure as we envision it can be found in Section 4.3 (page 24).
- The *low-level timing information* contains information about the execution of the basic blocks in the optimized program. It is derived from the program object code by the LLA, and used by the calculator in its calculations. See Section 3.2 (page 17) for more on the format of this information.
- The *timing result* is the answer to be presented to the user. It contains information about the WCET (and BCET) for the parts of the program requested by the user of the tool (which could be another program, like a schedulability analyzer).
- The *transformation information* contains information about how the program has been transformed by the compiler during the compilation and optimization process. This is a central part of our co-transformer. See Chapter 4.
- The *source program* (`.c` in Figure 3.1) is the program being compiled and analyzed. It is the input to the HLA and the compiler.
- The *object code* (`.o` in Figure 3.1) is the output of the compiler for the source program. It is an executable binary for the target system.

In the following sections, we investigate each component in detail.

3.1 High level analysis (HLA)

The high level analysis, HLA, is responsible for deriving information about the flow of the program from the program text (and maybe other information provided by the programmer).

It is hard to efficiently derive information about the program flow from the object code, since most of the information embedded in the source program is lost during compilation. For example, variables are difficult to identify since they may now occupy memory or registers or both, or even several different registers at the different times in the same calculation. Loops may move or be unrolled, functions can be inlined into other functions, and the code generated for switch statements can get quite complex, containing jump tables or other clever implementation devices which are hard to analyze without information about how they were created.

There are two basic approaches to HLA:

- Let the programmer *annotate* the source program with information about loop bounds and executable paths.
- *Automatically derive* information through semantic analysis of the program text.

[1] When we talk about a *mapper*, we mean mappers in general. The *co-transformer* is a particular instance of a mapper, and when we discuss the co-transformer we use that term instead.

The most common approach in the static WCET field is to let the user annotate the code of the program with extra information about the program flow, using comments, special language constructs, or even completely new programming languages with built-in annotations. A list of approaches using annotations can be found in Section 7.3.1 (page 58).

Using annotation has the weakness of assuming that the programmer knows how her program will behave in all circumstances. We believe this to be a little unrealistic; the information should at least be checked in some way, probably using manual proofs or inspections by other programmers.

The advantage of using annotations is that they are easy to implement, since no extra analysis is required.

The automatic derivation of information about possible program executions is also known as semantic program analysis. It is not widely used in the WCET field, probably because it is quite difficult.

The main problem with automatic analysis is that most languages have complex or incomplete semantics. There are two ways of getting around this problem:

- Restrict the analysis to language constructions with simple and unambiguous semantics, and forbid all other constructions. The drawback is that programming in a subset of a language may be more difficult or lead to cumbersome formulations, and makes reuse of existing programs difficult. How much a language can be restricted before it is discarded as useless by practioners is an area which deserves further investigation. It should be noted that some commercial high-integrity systems are developed using restricted versions of languages, like SPARK Ada[2].

- Allow all constructions, and do a *best effort* attempt at deducing information. If the program analyzed is too complex, the programmer can be asked to intervene to either simplify her program, or provide information to help the analyzer. The complexity limit is given by the program analyzed and not by the language. If we assume that human programmers have a tendency to write simple programs, this is a workable approach. A possible disadvantage with this approach is that we might end up with a larger number of unanalyzable programs, since the programmer is not forced to make her programs amenable to analysis.

  This is similar to the way compilers handle program analysis: if enough information can be gained to optimize a program, do optimize, but if not, compile the program anyway, but with worse code.

Note that annotations can be used to aid automatic analysis by providing bounds on input data, and by specifying other information about the program which might not be obvious from the code (like identifying code for handling exceptional situations, or code that should not be analyzed).

Another twist on the analysis–annotation relation is to use the analysis to verify the annotations, in order to educate the programmer about her program, and to check that the programmer has made correct assumptions. This is potentially a very powerful program development tool.

Finally, information gained by the compiler about the program could be used in the HLA. One could consider the information obtained in the compilation process as a starting point for a high level analysis. To our knowledge, this approach has not been used until today.

Note that the HLA is hardware-independent. It only depends upon the source language used (and the encoding of annotations, if they are used), but does not depend upon the actual hardware or specific compiler used (unless the compiler implements some non-standard extensions to the language).

A discussion of previous work in the area can be found in Section 7.3.2 (page 59), and an idea about the cooperation between the compiler and the HLA in Section 8.2.3 (page 68).

3.2 Low level analysis (LLA)

The low level analysis analyzes the basic blocks in the object code to obtain execution time information for each block. This information is used by the calculator to obtain a final execution time estimate, and

[2]The SPARK Ada language was designed in York, and is described in [Mar94, O'N94].

contains information about low level effects such as execution times and memory usage.

If we do not model caches and pipelines, the obvious output from the LLA is a constant execution time or a (min, max) pair of execution times for each basic block. The times can be obtained simply by summing the cycles taken to execute each instruction in the basic block. However, this only works for quite simple processors.

Things get slightly more complex when we have different types of memory with different speeds present in the system. In this case, we need to know to which address a memory reference is made in order to know how long it takes to execute. For programs without pointers, the problem is easy to solve within the LLA, since all memory references are to known variables, and we know where the variables are located after the program is linked (unless we use dynamic linking). For programs with pointers, it becomes as hard as the handling of caches (see below).

When moving to more complex processors, we get many more difficult problems: *prefetch buffers* make the instruction fetch time variable, *pipelines* overlap the execution of a number of instructions, and the execution time of an instruction may vary due to data dependences and surrounding instructions. Memory access may use *caches*, in which case the time required to fetch instructions and data depends on whether the cache were hit or not.

In this case, the execution time for a certain basic blocks depends upon which other basic blocks have been executed before. To resolve these dependences, we need information from the HLA, which is only available in the calculator module. The solution is to let the LLA provide the calculator with information which can be used to determine the execution times of basic blocks when the HLA information is added. For pipeline analysis, this data would represent the pipeline behavior of the basic block, and for caches information about instruction and data accesses inside the basic block would be needed.

The most important property of the LLA is that it takes care of all target processor and memory system dependences. Supporting a new processor requires the construction of a new LLA, which means that it is important that the LLA is loosely coupled to the other components of the timing analyzer. The target dependences involve some factors external to the processor, most notably the memory system. It would be advantageous if the LLA could be parameterized to account for different target systems using the same processor, perhaps by using some kind of plug-in structure.

Constructing a good LLA requires intimate knowledge of the processor, which may take time to obtain and encode. However, similar information is needed in order to construct a simulator for the processor, which means that for a company already building simulators, building a low-level analyzer should be quite simple. The optimal solution would be to generate the LLA, the simulator, and the compiler code generator from a single processor description; as far as we know, this has not been done.

An overview of methods used for LLA is given in Section 7.4 (page 59).

3.3 High level to low level mapping (MAP)

The HLA outputs information about the program based on the structure of the source code (loops, conditionals, function calls, etc.). The LLA outputs information about the program based upon the structure of the final optimized target code (execution times and basic blocks). These structures are very different, and the information from the high level needs to be transformed so that it follows the structure of the low level.

The restructuring of the program code from high to low level is performed by the compiler, and the relation between the high and the low level becomes more complex as the compiler gets smarter and optimizes the program better.

The work of the mapper is to transform the high level information about the program into a form compatible with the low level structure, maintaining as much information as possible —the more information maintained, the better the final timing estimates. The mapper requires information from the compiler about how the program has changed; it is the compiler which creates the problem, and the cooperation of the compiler is needed to solve it. We do not think that it is reasonable to try to figure out what happened to the code just by looking at the object code.

We have implemented a mapper using *co-transformations*. The concept is explained in detail in Chapter 4, but the general idea is to transform the high level information about the program in a way corresponding to the way the code of the program is transformed during compilation.

See Section 7.6 (page 62) for more on other approaches to the mapping problem.

3.4 Calculation of WCET estimates (CALC)

Once we have the program flow information from the HLA, mapped to the same structure as the output from the LLA, and information about the execution of each basic block from the LLA, we can start calculating the WCET of a program.

Looking at the program as a function call graph, the calculation order of the calculator is bottom-up: before we can calculate the execution time of a function we need to know the execution time for all function calls performed in the function.[3]

It is possible to calculate the execution time for each call graph node in isolation, using just the information provided by its immediate successors. Healy et al ([HWH95]) has shown that the timing, cache, and pipeline information from the immediate successors in a call graph is sufficient to perform timing analysis.

If we perform timing analysis on systems containing caches and pipelines, the effects of those must be taken into account in the calculator. The LLA provides raw data, but it is the the calculator which uses the information from the high level to perform the actual cache calculations. Taking caches into consideration, the pipeline behavior becomes dependent upon the cache behavior [HWH95]. Calculating execution times when caches are involved include iteration (it is usually some variant of fixed point iteration); the iteration may be localized to a separate component (a cache simulator), as done in [WMH+97], or integrated into the general calculator [OS97].

There are several different methods of calculation used in the literature on WCET estimation. Each method has its own way of representing the program and performing the calculations. For an overview of calculation methods used in the literature, see Section 7.5 (page 61).

3.5 Compiler connection

In this section, we will investigate the connection between the compiler and the timing tool in detail. We will concentrate on the needs of a co-transforming mapper; the information provided by the compiler will vary with the method of mapping used.

An overview of the connection between the compiler and the timing analyzer is given by figure 3.2. Note that the compiler is assumed to compile C, although this is of no importance to the discussion here.

The compiler is divided into the following parts:

- The *front end* (FE) performs syntactical and semantical analysis on the source program. The output is information used by the optimizer to optimize the program, and some intermediate representation of the program.

- The *optimizer* (OPT) is the second compiler stage. It takes the intermediate information from the front end and optimizes it, generating a new intermediate code.

[3]Note that it is very likely that the calculator and the mapper have different calculation orders: the mapper should perform operations corresponding to compiler transformations in the same order as they were performed in the compiler, while the calculator works bottom-up in a call graph. This means that the complete result of the mapper must be presented to the calculator before calculations can begin.

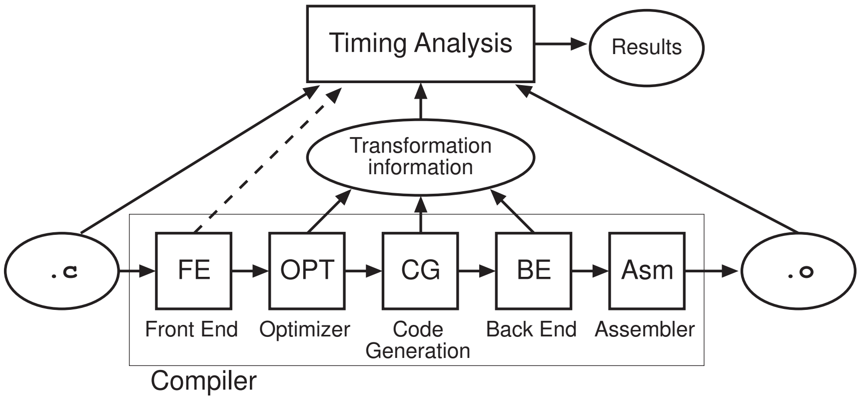

Figure 3.2: The connection between a WCET tool and the compiler stages.

- The *code generator* (CG) takes the intermediate code from the optimizer and generates another intermediate representation very close to the target code.
- The *back end* (BE) performs final simple optimizations and clean-ups on the code generated by the code generator. This includes peephole optimization and code compaction[4]. The back end consider the program as a flat sequence of instructions or even simple bytes.
- The *assembler* (ASM) converts the intermediate form used by the back end to the binary format of the target processor. In the case that a program consists of several source files, a linking stage is also necessary to produce a single object code file.

Every part of the compiler except the assembler emits information to be used in timing analysis. The timing analyzer also uses the program source code and the final object code.

The program source code is used by the HLA (see Section 3.1 (page 16)). As stated in the discussion on high level analysis above, information from the compiler front end may be used in the HLA part of the timing analyzer (this is indicated by a dashed line in Figure 3.2).

The optimizer, code generator, and back end all transform the program, and generate information about how they modified the program (the *Info* data structure in Figure 3.2). This information is used by the mapper.

Finally, the optimized target code generated by the assembler is used by the LLA, see Section 3.2 (page 17).

Note that the mapper depends heavily on the compiler, and that the compiler needs to be retargeted for each new target processor supported. However, if the compiler can be ported to new platforms easily, the mapper should be portable with a similar effort. If we have a general compiler which contains a number of different transformations for code generation and optimizations, and each specific target uses some subset of these, constructing a mapper which can handle the entire set of transformations allows the mapper to handle a compiler for any target.

[4]A typical example of code compaction techniques which change the program structure is replacing strings of code with calls to shared subroutines. This technique is known as *external pointer macros* [MG95, Section 3.5].

Chapter 4

The Co-Transformation Approach

Having presented our general framework for WCET analysis, we now focus on a specific component: the mapper, implemented using co-transformation techniques.[1]

The problem is how to unify the *dynamic* information about the program flow gained during the analysis of the source code (HLA), with the *static* information in the optimized object code. We have decided that the best solution is to map down from the source code to the object code, since we want to analyze the program on the source code level but calculate execution times on the object code level in order to use the best information available for each task (see Chapter 3). Conceptually, the execution information is transformed just like the compiler transforms the program code from the source code to the object code.

In the compiler, a transformation is defined by a change to the static structure of the program code. The fundamental problem addressed by the co-transformer is how to perform changes to the dynamic information about the program flow so that it remains in correspondence with the static program structure. For every compiler transformation we need a corresponding co-transformation.

We have implemented a simple co-transformer to validate the concept. See Chapter 5 for more details on our prototype.

4.1 Architecture of a co-transformer

Figure 4.1 gives an overview of the architecture of our simple co-transformer.

The co-transformer takes the following inputs:

- **Co-transformation definitions** —for a given compiler, there is a certain set of transformations which it may use to optimize a program. Co-transformations for these transformations are defined in a file, and used by the co-transformer.

- **Transformation trace** —for a specific source program, a specific compiler performs a certain set of transformations in a certain order. The transformation trace lists the transformations performed on the static program structure (each transformation has a corresponding co-transformation). This information is invariant over program invocations with different data[2]. How this information is obtained is described below in Section 4.2.

[1] A small note about terminology: in this chapter we will consistently write about the *co-transformer*, which is a specific kind of mapper. Since we are not discussing mappers in general, we do not use the more general term. In previous chapters we discussed the mapper in general, and used the word *mapper*.

[2] We could imagine optimizing a program using information about input values given by the programmer. This relates to the discussion of a merged HLA and compiler given in Section 8.2.3 (page 68). However, in all normal cases, the trace will be static.

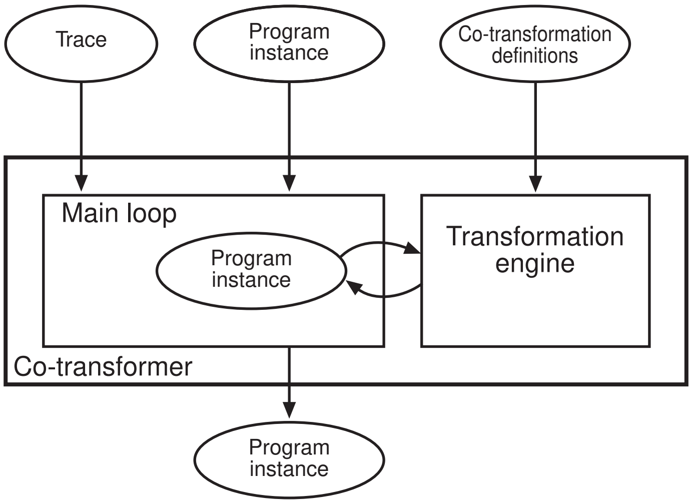

Figure 4.1: Architecture of our co-transformer.

- **Program instance** —given a specific source program, the high level analyzer (HLA) produces information about the program execution. This is called a *program instance*, and contains information about how the program may execute. The format of the program instance representation is discussed below in Section 4.3.

  Several execution scenarios can be tried using the same trace and co-transformation definitions, but with different program instances.

We believe that this is a natural division of the input data to a mapper in general and a co-transformer in particular.

Comparing the division above to Figure 3.1, the program instance corresponds to the *flow info*, and the transformation trace together with the co-transformation definitions correspond to the transformation information.

The co-transformer program is divided internally into two parts:

- The main loop uses the *transformation engine* to perform transformations on the program instance. It applies the co-transformations defined in the *co-transformation definitions* to the program instance, and returns the updated program instance.

- The *main loop* loops over the transformation trace. It maintains the program instance as its working data structure, and uses the transformation engine to transform it. The trace and the program instance are used as inputs to the main loop.

In every step of the main loop (the processing of one transformation from the trace), several calls may be made to the transformation engine, since there can be several parts of the program instance affected by each change to the static structure of the program (if a certain function is used twice in the program instance, both instances of the function must be co-transformed when the function is transformed).

When the co-transformer is finished (the entire trace has been processed), the output is a new program instance, with a structure suitable for unification with the output from the LLA.

Note that the transformation engine can be implemented very efficiently if we use the co-transformation definitions as an implementation language and compile a transformation engine from them. For each release of a compiler, a corresponding transformation engine can automatically be created.

4.2 Obtaining information about transformations

In order to co-transform the program instance, we need information about how the compiler transforms the program. This means that we must know the compiler, since we need to know precisely how the compiler can transform a program — a co-transformer is by necessity bound to a certain compiler.

Many components of a compiler will change the structure of a program during compilation (see Section 3.5 (page 19)). The compilation process takes us from an abstract program representation to a detailed basic block graph populated with assembler instructions. In the same way the information from the HLA is at first bound to a high-level view of the program structure, and at the end it is bound to a basic block graph identical to the one produced by the compiler and used by the LLA (or some other suitable structure).

We need to know the exact sequence of operations performed on the program, and precisely which parts of the program they affect. We call the sequence of operations a *trace* of the optimization process. After every step in the sequence, the internal program representation in the compiler and the representation of the timing information in the co-transformer must be in correspondence.

There are several possible interfacing models between the co-transformer and the compiler:

a. The simplest approach is to have the compiler emit a trace of all transformations it has performed on a program, and then mimic the effect on the execution information in a separate co-transformer program. This separates the timing tool and compiler; the only change necessary to the compiler is the output of a trace. Programming the co-transformer requires knowledge about the implementation of the compiler. This is shown in Figure 4.2(a).

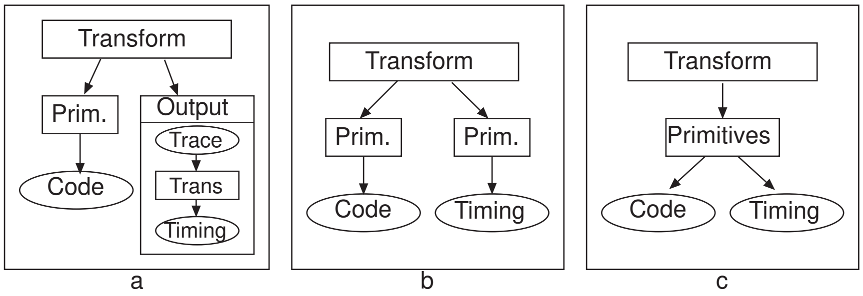

Figure 4.2: Variations on the compiler–co-transformer interface.

b. A better way is to make the co-transformation a part of the compiler. As soon as the compiler updates the code, the execution information (timing information) is also updated. Every transformation will have to include code to manipulate both the program and the execution information. This approach is shown in Figure 4.2(b)

c. The ideal way to structure a compiler would be to let the code-generating and optimizing transformations use a single set of program manipulation primitives, which would manipulate all information about the program. The transformations would contain no knowledge about the types of information being transformed. The co-transform would be subsumed into the transform. This is shown in figure 4.2(c). Managing new kinds of data requires updating the primitives, but does not require changing the transformations. Implementing this approach is very difficult, as we have to find a very good set of primitives, sufficient to capture any program transformation.

For our prototype, we have used approach (a), since it is the easiest to implement and test (we can load test data from files and create test data files without modifying a compiler). In the future, we hope to be able to use approach (b), given that we find a compiler to work with. A lot of research will be needed in order to enable approach (c).

4.3 Program instance representation

The program instance representation is central to the co-transformer. It is a data structure which needs to unify the inherently static data provided by the compiler (the code) with the dynamic information from the HLA about the program execution: loop bounds, infeasible paths, execution bounds, etc..

In the following discussion about the representation of the program instance, we assume that the structure being manipulated by the compiler is a basic block flow graph representation of the program (typically dividing into one flow graph per function). This is not strictly necessary —we can imagine other structures— but we use this representation to simplify and concretize the discussion. Any structure may be used: it is simply a matter of defining corresponding co-transformations. Even "phase changes" in the compiler could be handled.[3]

The dynamic aspect of a program (its execution) can be easily visualized as a *call graph*, where the nodes are *function instances* (a specific call to a function), and the successors to a node the function instances it may call. Note that we are talking about potential calls here: as we do not execute the program, we do not know if a certain function call will actually be executed in an actual execution of the program.

A function instance is a rather vague concept by design. The precise definition depends upon the high level analyzer, and there are many possible representations. The idea is that each potential function invocation encountered in the high level analysis of a program should be represented. In the best case, we may know the exact value for every parameter, or we may have to consider a range or set of values for each parameter. We could also represent the effect of a certain set of parameter values upon the execution of the function, by considering how the function executes given the parameter values (this approach is used in our prototype, where we consider each function instance as a flow graph with (maximum) execution count for all basic blocks).

The graph must be the *worst-case* call graph: the largest possible call graph for the program. Note that it is by no means necessarily the execution of the program which performs the greatest number of function calls which takes the longest time to execute, but we must investigate all possible execution scenarios, and therefore it is necessary to have the complete worst-case call graph. In many cases, much of the graph will never need to be investigated.

In fact, having only function instances in the call graph is not sufficient. The call graph should also contain loop instances, information about specific loop iterations, infeasible path information, etc. One can consider loops as calling their bodies, outer loops as calling inner loops, and functions as calling loops in their bodies, as well as functions calling other functions. At the bottommost level, the basic blocks of the program are called.

Figure 4.3 gives an impression of how a simple call graph may look. Note that there are many instances of the same function, that loops are considered part of the graph, and that several nodes can call a node, if they do it with the same parameters. The parameters are abstracted to a single number in order to make the figure more readable.

The information in the call graph is to be used to overcome the difficulties in tight execution time analysis listed in Section 2.4 (page 6). Thus the call graph includes:

- Information about loop iteration bounds.
- Information about loop iteration dependences.
- Information about infeasible paths.
- Information about function instances (much of this information will likely be provided in the form of loop bounds and infeasible paths).

Obviously, the amount of information required to describe the execution of a program can get very large.

[3]By a "phase change", we mean that the compiler changes its representation of a program between different components of the compiler.

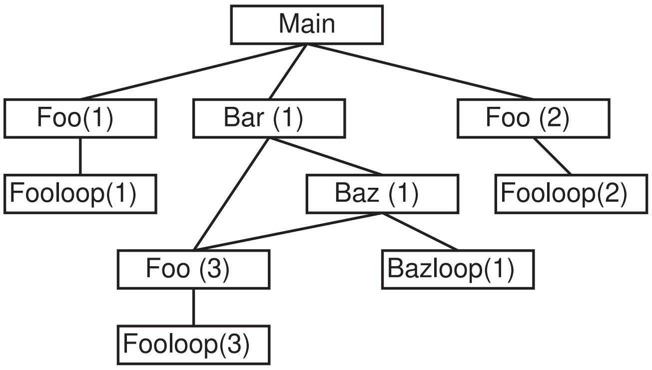

Figure 4.3: A simple call graph, containing both functions and loops.

In the extreme, we represent every instruction instance potentially executed by the program. In order to avoid a data explosion, we must allow the precision of the representation to vary across the graph.

For example, for a loop with a fixed number of iterations, there is no need to represent each iteration as a node of its own. It can be comfortably represented using a single node. These kinds of aggregations must be applied to all parts of the call graph, in order to keep its size manageable.

Another way to compress the program instance would be to group similar function instances into a single node (not just identical as presented above). The definition of "similar" would vary according to the representation of the functions and the high-level analysis.

4.3.1 Static and dynamic information

As stated above, the call graph structure is rooted in the static structure of the program code. We are describing the dynamics of the program as an overlay on the static program.

The relationship between the static program and the dynamic program call graph can be handled in two ways:

1. Include the basic blocks into the call graph, creating a copy for each time each basic block is called (a *basic block instance*). This would make intuitive sense, since we already represent function instances and loop instances in the call graph.

2. Keep the basic block structure separate, with links into the places in the graph where various blocks are used. The links are necessary to locate the parts of the call graph affected by a change to the static structure. There will be some kind of place holders inside the graph providing information about how a certain instance of a basic block relates to the call graph entities which call it.

   In the case that the only information about a basic block outside the call graph is its name, this approach degenerates into the first.

We use the concept of basic block instances in our prototype, where the information about each basic block contains its identity and information about how its execution relates to the loops surrounding it. We perform changes to the instances of a certain function in terms of (named) basic blocks being moved around.

4.3.2 Properties of the call graph

For the call graph used when representing program instances it is possible and desirable to merge identical function instances (calls with the same arguments) into one node. The graph is not cyclic, however:

A cycle in the call graph indicates a non-terminating recursive computation: it can only be generated if a function instance calls itself with the same arguments (possibly through a series of other function calls). Since we assume that the execution of a function is determined only by its arguments, this means that the function will keep on calling itself indefinitely. This is a sufficient precondition for a non-terminating recursion; it is not necessary, however: the absence of cycles in a function call graph does not guarantee termination of a program (since a node could contain some non-terminating computation in itself).

A nice property is that iteration can be represented in the same way as recursion: by assigning each loop iteration to a single node which calls the next loop iteration node, we get the same structure as a function calling itself with different arguments.

4.3.3 Global data and function instances

Global data disturbs the harmonious picture of the program instance as a call graph, since it invalidates the assumption that the execution of a function depends only on its arguments, which is a necessary prerequisite for the timing analysis as sketched so far. We cannot simply forbid global data, since many realistic programs will contain shared data areas and global variables used to track the state of the system.

A possible solution to this problem is to assume that global data is only acted on, and that it does not affect the control flow of the program. Obviously, this solution limits the set of programs analyzable by a timing analyzer. We do not believe that the limitation is too great, however. Looking at the iteration count of loops, the only time where the value of data has an effect are in cases such as linked lists and (at analysis time unknown) strings with zero termination. These kinds of dynamic data structures are not very likely to be present in real-time programs, where predictability is a very important property.

In the case that the number of elements in a global data structure varies, but the counter is kept in a separate variable, the value of this variable should be deduced from the program text, just like any other variable.

A more elaborate solution is to consider the entire state of the program including global data as a part of the arguments to and results from a function. This does make the HLA much more complex.

We have not decided on any particular solution to this problem, as this belongs in the field of high-level analysis, which is not considered in this thesis.

Chapter 5

Prototype Implementation

We have designed and implemented a prototype co-transformer according to the theory presented in Chapter 4. The implementation shows that the concept works in practice, and we have tested it on some typical compiler transformations (see Chapter 6).

In this chapter we describe the implementation of our prototype tool and its input languages.

5.1 System design

Pragmatics caused us to divide the co-transformer into four separate programs. Since we used a powerful tool for the generation of parsers (ELI[1]) which generated C-code, and used another implementation language (Erlang[2]), we implemented the parsers as three separate programs. The output from the parsers are intermediate files containing Erlang terms easily read by the Erlang system.

Figure 5.1 gives an overview of how the four programs in the prototype are connected.

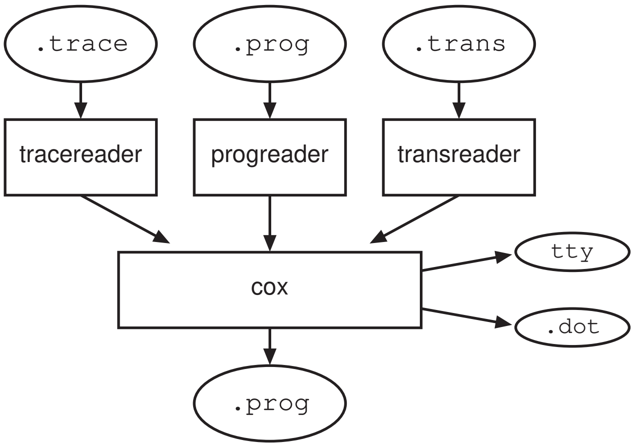

Figure 5.1: Components of the prototype co-transformer.

- The `tracereader` reads the `.trace` files containing transformation traces. It outputs an intermediate file (not shown) which is read by the main program (`cox`).

[1] See [GHL+92].
[2] See [AWVW96].

- The `progreader` reads the `.prog` files containing program instances. `.prog` files are used both as input and output. It outputs an intermediate file (not shown) which is read by the main program (`cox`).
- The `transreader` reads the `.trans` files which describe the co-transformations. This format is far more complex than the others, since it needs to describe very complex operations. We have designed a language called *Optimization Description Language*, ODL, for use in the `.trans` files. It outputs an intermediate file (not shown) which is read by the main program (`cox`).
- The main program is called `cox`, which is a short for "CO-Xformer". Note that in addition to the `.prog` output (containing a new program instance), `cox` sends diagnostic outputs to the screen (`tty` in the figure), and to a `.dot` file.[3]

The main program follows the structure presented in Section 4.1 (page 21) and Figure 4.1.

Most of the work involved in designing the prototype went into designing the ODL used for describing the co-transformations. We decided from the outset that we didn't want to code transformations into ordinary C or Erlang code, but to define a language to express transformations instead. We did this for the following reasons:

- Designing an ODL made the design issues clearer. We had to confront the question of which concepts and operations were needed, and which semantics they should have. Had we implemented the co-transformations in a general-purpose programming language, it would have been all too easy to implement quick kludgey fixes for problems which we now had to confront and solve in a structured way (or realize that they could not be solved).
- Having a stand-alone ODL, it easier to port the co-transformer to new compilers, since the code for the co-transformations is separated from the other, compiler-independent, parts of the co-transformer.
- Assuming that we have a decent ODL, we open up for the exciting ability to generate both the compiler and the co-transformer from the same source (or to generate an integrated compiler and co-transformer).

The ODL concept as such is new. We have found no other attempts in the literature to define a formalism for expressing compiler transformations. Designing an ODL is a good way to reach the understanding necessary to formalize compiler transformations and pave the way for the design of a small set of primitives to describe all transformations.

5.2 Optimization description language (ODL)

The Optimization Description Language (ODL) is used to describe how we co-transform information about the program execution given that the compiler has performed some transformation on the program code. Note that we use both the words "transformation" and "co-transformation" in the text below to refer to transformations defined in ODL.

We need a co-transformation for each possible compiler transformation, and the purpose of the co-transformation is to:

1. Update the static program structure in the program instance so that the program code and the program instance remain in correspondence.
2. Update the execution information about the program in such a way that the it is consistent with the state of the program code and possible program executions after a transformation.

[3]The `.dot` format is the input to the graph drawing tool Dot [Dot97, KN], and is used to visualize the function graphs for debugging and development purposes.

To meet goal 1, we need to restructure the flow graph of the program in the same way as the compiler, giving corresponding names to basic blocks.

Meeting goal 2 is the main reason for having an ODL. We want to describe how to co-transform information about the program execution from the high level to the optimized target code, and in order to do this we must figure out how to transform the execution information with as little information loss as possible.

5.2.1 Data model

The ODL language is built on the following data model:

The data being operated on is a set of function instances. Each function instance is represented separately. There is no direct connection (in the sense of graph edges) between instances of the same function, but we can find all the instances of a certain function.

Each function instance is represented by an annotated graph of basic blocks, which describe both the structure of the function (its static structure) and the specific execution characteristics associated with this instance (the dynamic view). Each basic block has two attributes:

- The **name** of the basic block is a unique identifier which serves to identify the node when performing co-transformations.
- The **data** contained in the block is a record, with a number of named components. The names of the components are given in the ODL program; all basic blocks have the same structure.

Note that there is no data shared by two or more basic blocks. All data is local to a basic block.

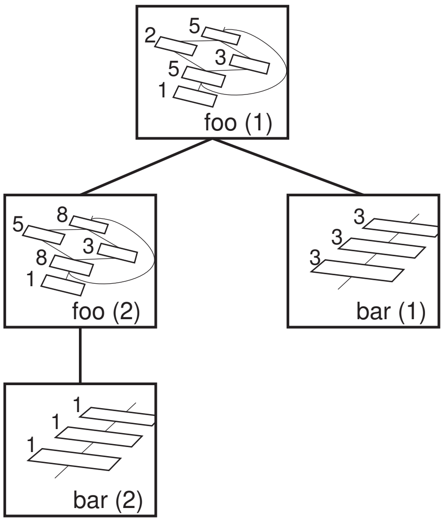

Figure 5.2: Example of a call graph.

Figure 5.2 shows a sample program instance. The nodes are function instances, and inside each function instance there is a basic block graph annotated with numbers. The numbers are execution counts for the basic blocks. To simplify the picture, the names of the basic blocks are not shown.

5.2.2 Graph concepts used in ODL

The ODL program defines operations over the basic block graphs, and the co-transformer uses directed, cyclic graphs in many places in the definition and execution of ODL programs. We use a number of graph concepts:

- The most important concept is the *closed graph fragment.* A fragment of a graph is closed if no paths leave or enter the fragment except through its top and bottom node. Formally, the top node must dominate[4] and the bottom node postdominate all the nodes in the fragment. A graph fragment is defined by giving its top and bottom nodes. Edges from the bottom node back into the fragment are part of the graph fragment, except for one special case: an edge from the bottom node to the top node. Such as edge is treated differently in different circumstances, as it can be considered part of the graph fragment or a part of the surrounding graph. It is referred to as a *loop edge.*

- In many cases, we need to treat groups of nodes as a single node. Typical examples include loop bodies or if-branches which are copied or moved as a whole, regardless of their internal structure. To facilitate this, we use the concept of a *compound node.* A compound node is a closed graph fragment defined by giving its top and bottom nodes, treated as a unit by ODL operations.

  A compound node exists as a small graph on its own, with named nodes containing data inside. They pose a problem when they are copied: new names must be supplied for all the nodes inside the compound in order to avoid name collisions. For this purpose, a special datastructure called a *compound node component name translation table* is introduced. It is a list of $(oldname, newname)$ pairs.

- When updating a graph to account for changes to its structure, we need to *replace one graph fragment by another graph fragment.* We require that both the replaced and the replacing graph fragments are closed. The fragment replaced is simply deleted from the graph, and the edges going into and out of the fragment are tracked. The new graph fragment is copied into the graph, and finally connected at the top and bottom to the same nodes that the replaced fragment connected to. Loop edges count as internal nodes in the respective fragments. This means that in order to have a loop edge in the output graph, it must be present in the replacing graph fragment. The presence of a loop edge in the replaced graph fragment has no effect.

5.2.3 ODL programs

An ODL program consists of two parts:

- The *global declarations.*
- The *transformations.*

Global declarations

The global declarations determine the structure of the data upon which the ODL operates. We give the following information in the global declarations:

- The program name and the version of ODL used to write the program. They are presently ignored by the parsers, but included for future expansion.
- The names and order of the data components in the basic blocks in the program instance.
- Functions which are used to define the transformations. To simplify the implementation of the prototype, we include an Erlang source file and allow the use of the functions in it.

[4] A node N in the call graph *dominates* another node M if all paths from the start node in the graph to M goes through N. A node N *postdominates* another node M if all paths from M to the end node passes through N. See [Bla94, p. 582] for a formal definition.

Transformation declarations

The main body of the ODL program is the list of transformation declarations. Each transformation consists of four parts (see Figure 5.3 for an overview):

- The **header** gives the name of the transformation and its parameters.
- The **in-pattern** matches the areas of the function instance graph to be affected by a transformation. The matching results in data variables in the in-pattern being bound to the data values kept in the matched basic blocks. The areas matched by the in-pattern will be replaced by the out-pattern.
- The **out-pattern** replaces the in-pattern in the function instance graph. Data values for the basic blocks can be copied from the in-pattern or calculated using function calls. To calculate data values, new variables are declared in the out-pattern and given values by the transform functions.
- The **transform functions** define how out-pattern data values are generated from in-pattern data and transformation parameters. It is a list of function calls, where the functions defined in the global declarations may be used.

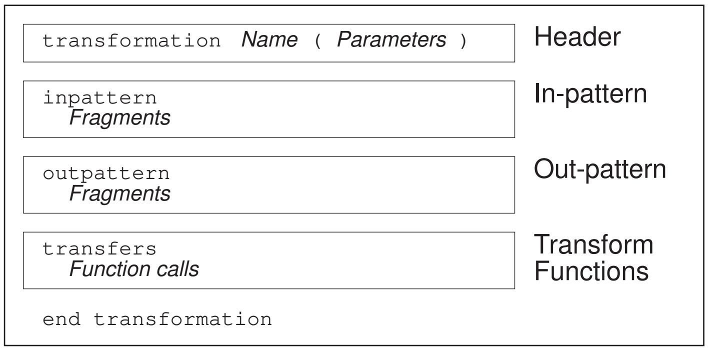

Figure 5.3: Sections of a transformation definition.

In the following, we describe each section of a transformation definition in more detail. Note that the complete syntax for ODL and some further remarks about the semantics can be found in Appendix A.

Note that we use the percent sign (%) to indicate comments, just like in Erlang and Prolog. The comments run until the end of the line.

Header The header gives the name of the transformation (to be used when calling the transformation), and its parameters. The parameters are used to:

- Name the function which a transformation is applied to.
- Position the nodes in the in-pattern in the function instance graph.
- Give names to nodes introduced in the out-pattern.
- Provide arguments for function calls.

The type of each argument is declared by prefixing it with a type identifier: `section` introduces a function name argument, `var` introduces a simple variable, and `tt` introduces a name translation table (for compound nodes). Simple variables can hold any value (as long as the value can be given in the transformation trace), but typically hold names of basic blocks or parameters used for function calls in the transform section.

As an example, we use the loop peel optimization. The effect of loop peeling is to take a loop and place a copy of the loop body before the loop.

We begin by giving the header for the loop peel transformation:

```
transformation LoopPeel
  (section S, var LoopBegin, var LoopEnd, var Loop, tt NewNames)
```

The words `transformation`, `section`, `var`, and `tt` are keywords.

In-pattern The in-pattern is a set of graph fragments. Each fragment has a name; the out-pattern fragment with the same name will replace the in-pattern fragment in the function graph. Each in-pattern fragment contains nodes and compound nodes. Nodes match single basic blocks, while compound nodes may match several. Using two or more graph fragments, we handle transformations affecting two unconnected parts of a function. There must be at least one node in each in-pattern fragment, and there may be any number.

No structure is given in the in-pattern: nodes are simply declared by giving their names (actually simple parameters from the transformation header), and a data variable. The data variable is declared by occurring (no need for previous declarations), and will pick up the data held in the node.[5] If the data from a certain basic block is not needed, a "don't care" symbol may be used (like in Erlang and Prolog).

A compound node is given a name which represents all of the basic blocks within the compound, its data variable similarly contains the data from all the basic blocks. It is defined by giving its top and bottom nodes.

An example in-pattern (from the `LoopPeel` transformation) looks like this:

```
inpattern
  fragment loop(S,LoopBegin,LoopEnd)
    cnode(loopbody,LoopBegin,LoopEnd,LoopBodyData);
  end fragment

  fragment repr(S,Loop,Loop)
    node(Loop,LoopData);
  end fragment
```

The words `fragment`, `node` and `cnode` are keywords. We declare two fragments (`loop` and `repr`): one containing a single compound node, and the other a single simple node. The arguments to the fragment declarations give the function (`S`) and the first and last node affected. The variables `LoopBodyData` and `LoopData` are created and given the values of the data in the respective nodes.

Out-pattern The out-pattern replaces the in-pattern in the flow graph, fragment by fragment.

The nodes (and compound nodes) in the out-pattern fragments are given names just like the in-pattern nodes. If a node name occurs in the in-pattern as well as in the out-pattern, it means that the node identity (name) will be preserved in the out-pattern. If a copy is made of a compound node in the out-pattern, names must be provided for each node in the copy by means of a name translation table. This table is given as a parameter to the transformation. The special symbol `id` can be used to indicate that the names are not to be changed.

Unlike the in-pattern, the out-pattern specifies the structure of the graph fragment. For each node, a list of successors are given. Compound nodes are treated as single nodes (they are given logical names for this purpose).

Edges to the same node are handled in a special way: if a node is present in its own successor list, a loop will always be created from the node to itself. If it is not present, no loop is created, and any old loop is removed. Finally, if the name is given enclosed in parentheses "`(Self)`", a loop will be created if one is present in the input data.

To copy data from an in-pattern node without change, the name of the in-pattern data variable is given, prefixed by an equals sign (`=`). Giving a variable without an equals sign declares a new

[5] This matching model is based on that of ML (and not on that of Prolog and Erlang): no variable may occur more than once, as the matching is only a way to obtain program information and not a control structure.

variable which is given a value in the transforms section of the transformation. No variable may be declared more than once, but several out-nodes can copy data from the same in-node.

If an out-pattern fragment is left empty, the fragment will be removed from the graph, and *all* nodes following the fragment are connected to *all* nodes preceding it.

We continue our example by giving the out-pattern for the loop peeling transformation:

```
outpattern
  fragment loop(looppeeled,loopmain)
    cnode(looppeeled,loopbody,NewNames,NewOut,[loopmain]);
    cnode(loopmain,loopbody,id,LoopBodyOut,[(loopmain)]);
  end fragment

  fragment repr(Loop,Loop)
    node(Loop,LoopDataOut,[]);
  end fragment
```

The keywords are the same as in the in-pattern. Note that the `loop` fragment is replaced by a fragment containing two copies of the loop body (which is represented by a compound node). The copy is given node names from the `NewNames` variable, while the original node does not change the names of its node names, which is indicated by the special code `id`. Also note that the `loopmain` compound node has a loop to itself if it had one previously ("`(loopmain)`" in the successor list). All the variables `NewOut`, `LoopBodyOut`, and `LoopDataOut` are data variables which are to be given values in the transforms section.

Transform functions The transforms describe how values for variables introduced in the out-pattern are computed from the data variables of the in-pattern and the parameters given to the transformation.

Values are created per record component; we must give each data variable a value for each of its components. The values can either be copied (using assignment syntax), or computed using function calls. The functions used must be defined in the function declarations section of the ODL program.

A special syntax is used to define functions over values for compound nodes. We use a source value from a compound node with the same structure as the destination value, and apply a function to each node in the source and assign the result to the corresponding node in the destination value. A compound function is introduced using the `forall` syntax.

We finish our example transformation by giving the transfer functions:

```
transfers
  %% update the loop iteration count
  LoopDataOut.iterations = sub(LoopData.iterations,1);
  LoopDataOut.scope = LoopData.scope;
  LoopDataOut.execcount = LoopData.execcount;

  %% update the execcounts of the loop body
  LoopBodyOut.scope = LoopBodyData.scope;
  LoopBodyOut.iterations = LoopBodyData.iterations;
  LoopBodyOut.execcount = forall( X1:LoopBodyData |
                           capsub(X1.execcount,LoopData.iterations,1));

  %% update the peeled code (only exec once)
  NewOut.scope = forall( X2:LoopBodyData | copy(LoopData.scope));
  NewOut.iterations = forall( X3:LoopBodyData | copy(1));
  NewOut.execcount = forall( X4:LoopBodyData |
    fraction(X4.execcount,LoopData.iterations,LoopData.execcount));
```

Note how each component of every *x*`Out` variable is given a value. Also note the use of both function calls (`sub`), direct assignments between components of the same type, and function calls for compound nodes (`forall`...). The `copy` function is used to get around the fact that we cannot directly assign a constant value to a component. It is defined in the included Erlang function file (see Figure C.4 on page 88).

5.3 Prototype limitations

The main limitation in our prototype is the data representation used: it only handles a very simple call graph model, far from the general structure described in Section 4.3 (page 24). The simplified call graph used contains the following information:

- A call graph containing function instances. Since we do not perform optimizations across function boundaries or calculations, this graph is actually not needed.
- Each function instance in the call graph is represented by an annotated copy of the function flow graph. The basic unit is the basic block, and the graph contains data for each basic block and how they are connected.

Note that there are no loop nodes or other special nodes in the call graph; just function instances. Loops are represented using a kludge (free-floating basic blocks; see Section 6.2.1 (page 37)), with quite ugly semantics. The static structure of the code is represented in each function instance, since all instances of a certain function have the same flow graph structure.

The only transformations which can be handled in this framework are those which do not cross function boundaries.

5.4 Notes about the implementation

As stated above, we used the ELI system to generate the parsers. ELI is a very powerful system for generating parsers, far superior to a combination of `lex` and `yacc`, the standard UNIX tools for parser generation.

Our estimate is that the use of ELI saved us about a week of initial implementation time (including the time required to learn to use the system) and made adjustments to the parsers easier, compared to a `lex-yacc` combination. The generation of a text-to-text transformer can take as little as hours using ELI; it took only two hours to implement the `transreader` program!

We used Erlang for the main program because its very easy and fast to code in.[6] Erlang is a functional language developed at Ericsson, and intended for parallel applications; we only used it for serial code, mostly because it has a large set of very useful libraries, including graph manipulations.

Most of the core code of `cox` is graph manipulation and loops over variable length lists. Loops over lists are very natural to formulate using higher-order functions (`map`, `foldl`, etc).

The drawback of using Erlang is that the system is quite slow. For the toy programs we tried it on, we never had problems with the execution time, but we do not expect the prototype to scale very well.

[6] Ericsson estimates that it is possible to code about five times faster in Erlang than in C.

Chapter 6

Testing and Evaluation of the Prototype

We have tested the prototype described in Chapter 5 by attempting to implement a number of transformations in ODL, and then running them on sample programs using our prototype tool.

The purpose was to investigate the power of the ODL, in order to learn how it could be improved. We modelled a number of typical compiler optimizations, maintaining most of the execution information. In some cases, however, the limits of the ODL turned out to be quite severe.

Code and pseudo-code are used in several instances below to describe operations and code structure. We have used C's operator syntax, since C has a larger set of operators than most other imperative languages, but simpler pseudo-code to describe program fragments including loops and branches. The transformations are (obviously) written in ODL.

The transformation definitions shown in this chapter sometimes use function calls in their transforms. Definitions for the functions used (written in Erlang) are found in the listing in Figure C.4 in Appendix C.

6.1 Choice of transformations

Using our data structures as presented in Chapter 4 and Chapter 5 as guides, we have identified three broad categories of transformations:

- Transformations internal to basic blocks. These transformations only change the composition of the basic blocks, but not the structure of the program graph.
- Function local transformations, which change the structure inside a function.
- Transformations across functions. These transformations affect several functions at once.

In the evaluation of our co-transformer prototype we have chosen only to handle transformations local to functions. The block internal transformations have no effect on the timing information since they do not change the structure or flow of the program, and the transformations across functions cannot be handled in ODL.

In this chapter, we will examine some function level transformations found in the literature, transformations for both optimization and code generation. We have used two catalogs of transformations as our main source of transformations [Nul97, BGS90], and we have not selected transformations based on how easy they are to handle in ODL.

Next, we give an overview of the categories of transformations which we do not handle, and then we give a more detailed presentation of the transformations which we do handle.

6.1.1 Block internal optimizations

This section lists optimizations which affect the internal structure of one or more basic blocks (changing, moving, or deleting instructions within or between blocks). They do not affect the number of times or the order in which basic blocks are executed, and thus have no effect on the mapping of timing information[1]. We present a list of sample optimizations to give an impression of the optimizations we consider irrelevant for co-transformation.

These optimizations are very interesting from a debugger viewpoint since they affect when, where, and in which order values are computed and instructions executed. This is an important difference in the high to low level mapping required by a timing analyzer and that required by a debugger.

CSE [Nul97, ASU86] Common Subexpression Elimination. A subexpression common to two calculations (or more) is broken out and evaluated only once.

Constant Folding [Nul97] Expressions with values which can be computed at compile time are replaced with the (constant) result of the expressions. For example, `a=4+5*6` → `a=34`.

Constant Propagation [Nul97, ASU86] A constant value assigned to a variable is propagated through the flow graph and substituted at the use of the variable.

Expression Simplification [Nul97] Expressions can be simplified to equivalent expressions (which are faster to execute), using mathematical equivalences. For example, `i*0` → `0`, or `a+5*(a+b)` → `6*a+5*b`. The integer div, mod, and mul optimizations described below can be considered special cases of expression simplification.

Hoisting/Loop Invariant Code Motion [Nul97] Moving loop invariant code out of loops, usually to before the loop.

Induction Variable Elimination [Nul97] Some loops contain two or more induction variables (variables which are dependent upon the iteration count) that can be combined into one induction variable.

Instruction Combination [Nul97] Combining several instructions into a single one with the same effect. For example, `a++;a++` → `a+=2`.

Integer Divide, Mod, and Multiply [Nul97] Constant integer divides, modulo operations, and multiplies can be replaced with a sequence of shifts, adds, subtracts, and logical and operations. For example, `x*4` → `x<<2`, or `c%8` → `c&7`.

Sinking Moving code later into the instruction stream.

Printf Optimization [Nul97] Replacing `printf()` calls with more specialized calls for certain common cases.

6.1.2 Transformations across functions

To handle transformations (usually optimizations) across functions we need to handle the movement of code and data between function instances in the program call graph, and to change the structure of the call graph. This is not possible given the data model used in the prototype. A few examples of such transformations are given below:

[1] The reason for moving an instruction is often to move it from a frequently executed block to one executed less often. These changes will affect the results of the LLA, but not the mapped information from the HLA.

Function Inlining [Nul97] Puts a copy of a function inside another function, removing the call overhead.

Loop Pushing [BGS90, Section 6.8.7] A loop is pushed from a calling function to a specialized version of the called function.

Procedure Cloning [BGS90, Section 6.8.6] Specialized versions of a function are used instead of the original function. Different calls are replaced by calls to different versions.

Tail Recursion Optimization [BGS90, Section 6.8.8] Recursion is replaced by iteration. Function calls to the same function are replaced with `goto`s to the beginning of the function (or some other appropriate place).

6.2 Implementing transformations in ODL

We have tested our prototype by implementing a number of typical (function local) optimizations and code generation transformations in ODL.

First we will introduce the structure and semantics of the data in the basic blocks, and then we will examine the transformation implementations in detail. The optimizing transformations are handled before the code generation transformations.

6.2.1 Data in basic blocks

In ODL (as described in Section 5.2.1 (page 29)), data is bound to basic blocks. We have concentrated on handling the execution behavior of basic blocks relative to the loops in the program. The following information is kept for each basic block:

- Execution count (`execcount`).
- Loop scope (`scope`).
- Iteration count for loops (`iterations`).

The *execution count* of each basic block is to be considered relative to the iteration count of the nearest surrounding loop: if the loop runs 27 iterations, the maximum execution count is 27; an alternative would be to represent the execution count as a fraction, with maximum $\frac{1}{1}$ (this would make it somewhat simpler to implement certain operations on loops). The *loop scope* is the name of the nearest surrounding loop (or the function, for blocks and loops on the top level of a function).

Note that the execution count is a maximum value. In no case will the program execute the basic block more times than the execution count, but it may execute it less. The sums of execution counts of the successors to a conditional may sum up to more than the execution count of the conditional itself.

We represent a loop by a free-floating basic block, whose iteration count gives the number of times the loop body is executed for each execution of the loop. The execution count gives the number of times the loop is executed relative to the iterations of the surrounding loop.[2]

For an ordinary basic block, the iteration count is set to 1. For a loop representative, it represents the maximum number of iterations of the loop body for every passing of the loop. An inner loop will iterate its `iterations` value number of times each time through the outer loop. The execution count of an inner loop representative determines how many times the inner loop is executed relative to the iteration count of the outer loop.

[2]We are aware that this way of representing loops is a *kludge* which introduces some pseudo-global data; we will later see that it has some negative effects on the expressivity of our ODL. However, it would have introduced more implementation difficulties to have global data in the data model, and we choose to use a simple data model in order to get our prototype finished.

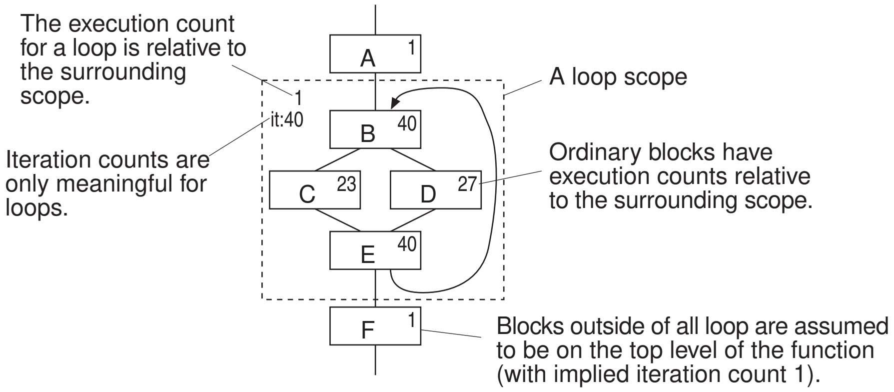

Figure 6.1: An example illustrating our data model.

Figure 6.1 illustrates our data model. The graphical notation shown is used later in this chapter. Note that we do not write out the iteration counts for ordinary basic blocks, and that the loop scope is shown as a dashed rectangle rather than as a free-floating block.

6.2.2 Implementations of co-transformations

The table below summarizes the success we had in implementing transformations in ODL. There are four levels of success:

1. **Complete** —We captured the transformation and we do not lose any execution information in the transformation.

2. **Partial** —We captured the transformation, but some information is lost in the transformation, or the transformation is only applicable in some special cases.

3. **Partial failure** —Some problem made the implementation of the transformation impossible or pointless. We did not write ODL code for the transformation.

4. **Complete failure** —We need completely new concepts to handle the transformation.

In the notes column in the table below, we note the main problem when partial successes are reported. The "embedded loops" problem refers to the fact that no loops may be nested inside the loops handled explicitly. "Input data too weak" refers to the fact that we do not have enough knowledge to handle the transformation fully. "Surrounding loops" means that loops from the last to the first block involved in a transformation are lost.

Transformation	Success?	Notes
Block merging	partial	Surrounding loops.
Branch elimination	complete failure	
Dead code elimination	complete	
If optimization	partial	Surrounding loops.
Loop preheader creation	complete	
Loop collapsing	partial	Embedded loops.
Loop distribution	partial	Embedded loops.
Loop fusion	partial	Embedded loops.
Loop interchange	partial	Embedded loops.
Loop peeling	partial	Input data too weak.
Loop unrolling	partial failure	Input data too weak.
Loop unswitching	partial	Input data too weak.
Long ifs	partial	Input data too weak.
Switches	complete failure	
Loop introduction	complete failure	

In the following, we will investigate each transformation more closely, and describe how we came to the conclusion about the success listed above. The ODL code for some transformations will be shown, but in most cases we refer to Appendix B.

Block merging

Block merging means merging a number of basic blocks (which are always executed in sequence) into one block. [Nul97]

We handle the merge of two blocks (since larger groups of blocks can be merged in steps). Merging two blocks requires that the first block is the only predecessor to the second block, and the that the only successor to the first block is the second block. The execution data for both blocks should be the same before the merge (otherwise the transformation would not be valid), so the final block simply copies the execution data from the first of the blocks.

The "surrounding loops" problem enters in the following way (see Section 6.3.1 (page 48)):

- The in-pattern for the transformation contains a single fragment, which matches two blocks (nodes).
- The (single) out-pattern fragment contains a single node.
- If the out-pattern fragment does not contain a loop around the node, there will be no loop in the resulting graph. If there is a loop in the out-pattern, we will always get a loop in the result.

The problem is that we cannot define the out-pattern in such a way that there is a loop around the single out-pattern node if there was one around the two in-pattern nodes. A separate transformation has to be defined for each case, and the compiler must tell the co-transformer which to use, which is slightly more complex than handling the loops inside the co-transformation.

The ODL code can be found in Appendix B, as Transformation B.1 (page 78).

Branch elimination

A branch elimination optimization shortcuts a jump (conditional or not) to an unconditional jump, replacing it by a single jump to the destination of the unconditional jump [Nul97, ASU86].

We cannot handle the optimization in ODL, since we cannot simply change the endpoint of edges in the flow graph, instead we must pattern-match the entire set of basic blocks involved, and there is nothing forcing the set of basic blocks involved in a series of unconditional jumps to be a closed graph fragment.

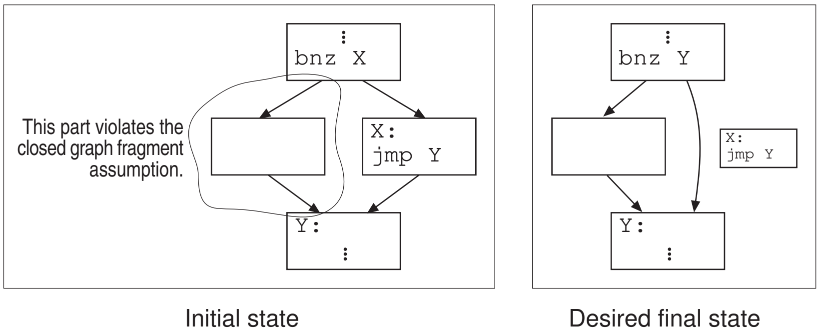

Figure 6.2: Illustrating the problem with branch elimination.

Figure 6.2 shows an example of the problem. We cannot pattern-match only the basic blocks involved in the branch sequence, but must match the entire set of blocks.[3].

This is a fundamental problem with ODL. See Section 6.3.4 (page 51).

Dead code elimination

Removing a basic block (which is dead) from a flow graph is very simple. Just match the node in the in-pattern, and leave the corresponding out-pattern fragment empty. The code for dead code elimination is given in Appendix B, as Transformation B.2 (page 79).

If optimization

If the outcome of the condition of an if-statements is known at compile time, the if can be eliminated and replaced by straight-line code [Nul97]. This is a more sophisticated variety of dead code elimination, eliminating not only the dead code but also the conditional leading up to it.

Removing one alternative branch from an if-statement is quite simple. As we do not model the content of basic blocks, we need only care about cutting away the blocks in the removed branch and not about removing the if-statement, some labels, and gotos from the code.

The execution counts for all involved entities are preserved: if the optimization was legal, then the execution count of the removed branch should be zero[4] and the other the same as the execution count for the if-statement. No loop scope change is performed for any block.

However, we have the same problem here as with block merging: if the first and last blocks involved (the block ending with the if, and the block where the branches join) are the first and last blocks of a loop, the loop is lost. See Section 6.3.1 (page 48) for more on this topic.

The code is given in Transformation 6.1. Note how two compound nodes are used to pick up the if-branches, and how only one of them remain in the out-pattern.

Loop preheader creation

A loop preheader is a basic block executed once before every loop execution [ASU86]. It is used to hold loop invariant code that have been moved out of the loop together with loop initialization code created

[3]In this particular example, it might seem simple, but in the general case there might be other edges incident on the blocks affected.

[4]If we have performed some information-losing transformations before the if-optimization, the execution count may be positive. This is problematic, since the timing information and the compiler information are out of synch. We really would like to avoid situations like this.

Transformation 6.1 If-optimizing transformation.

```
%-----------------------------------------------------------
% ARGUMENTS:
%   S: the section in which to operate
%   BlockIf: the block containing the if statement.
%   KeepBegin: the first block in the if branch to keep.
%   KeepEnd: the last block in the if branch to keep.
%   DeleteBegin: the first block in the if branch to delete.
%   DeleteEnd: as for KeepEnd.
%   BlockJoin: the block at which the if joins again.
%-----------------------------------------------------------
  transformation IfOpt
    (section S,var BlockIf, var KeepBegin, var KeepEnd,
          var DeleteBegin, var DeleteEnd, var BlockJoin)
    inpattern
      fragment one(S,BlockIf,BlockJoin)
        node(BlockIf,IfData);                        %% if block
        cnode(keep,KeepBegin,KeepEnd,KeepData);      %% branch to be kept
        cnode(del,DeleteBegin,DeleteEnd,_DeleteData); %% branch to be removed
        node(BlockJoin,JoinData);                    %% block which rejoins the execution
      end fragment

    outpattern
      fragment one(BlockIf,BlockJoin)                %% Note: one cnode removed
        node(BlockIf,=IfData,[keepbl]);              %%   (the removed above)
        cnode(keepbl,keep,id,=KeepData,[BlockJoin]);
        node(BlockJoin,=JoinData,[]);
      end fragment

    transfers
      %% no transfers needed
  end transformation
```

by the compiler.

Creating a preheader means inserting a new basic block before the first basic block in a loop (outside the loop scope). The preheader has the same scope and `execcount` as the loop itself, since every loop execution (one loop execution is the execution of all iterations of a loop) will be preceded by the execution of the preheader.

The ODL code is shown in Transformation 6.2: a compound node in the in-pattern picks up the loop body (bounds provided by the call). A new basic block is inserted before the loop body in the out-pattern (the name of the node is given by the call), and the scope and execution count information for the preheader is copied from that for the entire loop (represented by the block named LScope).

Loop collapsing

Loop collapsing means that a double-nested loop is transformed to a single-nested loop. Typically used for array operations on multi-dimensional arrays where every element is to be processed in some way, and the elements are stored in consecutive memory locations.[Nul97][BGS90, Section 6.3.4]

No work can be performed in the outer loop, since the code in the outer loop is removed (this is part of the definition of the optimization).

Loop collapsing can only be handled in the case that there are no nested loops inside the inner loop (this is an ODL problem, see Section 6.3.2 (page 48)).

The execution counts for the basic blocks in the inner loop must be scaled to be consistent with the new iteration count of the combined loop by multiplying them by the iteration count of the outer loop, as defines below, (where i=iterations, e=execution count).

$$e_{new} = \left(\frac{e_{old}}{i_{old}}\right) i_{new} = \left(\frac{e_{old}}{i_{inner}}\right) i_{inner} \cdot i_{outer} = e_{old} \cdot i_{outer}$$

Figure 6.3 shows the effect of the transformation. The numbers after "it:" are iteration counts, and the other numbers are execution counts. Note the change of the executions counts. The dotted lines are

Transformation 6.2 Loop preheader creation transformation.

```
%------------------------------------------------------------
% ARGUMENTS:
%   LoopBegin: name of first basic block in loop
%   LoopEnd: name of last basic block in loop
%   Preheader: name of preheader basic block to create
%   LScope: name of the loop to create a preheader for
%------------------------------------------------------------
  transformation CreatePreheader
    (section S, var LoopBegin, var LoopEnd, var Preheader, var LScope)

  inpattern
    fragment one(S,LoopBegin,LoopEnd)
      cnode(loop,LoopBegin,LoopEnd,LoopData);  %% the loop body
    end fragment

    fragment lscope(S, LScope, LScope)
      node(LScope,LScopeData);
    end fragment

  outpattern
    fragment one(Preheader,theloop)
      node(Preheader,PHData,[theloop]);        %% preheader block created
      cnode(theloop,loop,id,=LoopData,[]);     %% loop body unchanged
    end fragment

    fragment lscope(LScope, LScope)
      node(LScope,=LScopeData,[]);             %% loop scope unchanged
    end fragment

  transfers
    PHData.execcount = LScopeData.execcount;
    PHData.scope = LScopeData.scope;
    PHData.iterations = copy(1);       %% work-around for lack of  direct assignment
  end transformation
```

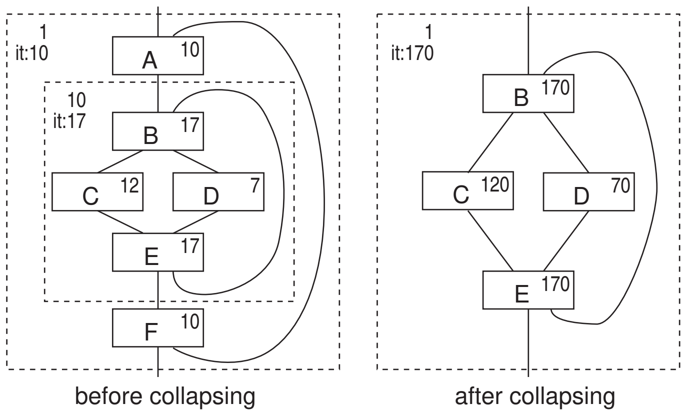

Figure 6.3: Collapse of a loop nest. Note the execution counts.

loop scopes. The ODL code can be found in Appendix B, as Transformation B.3 (page 80).

Loop distribution

Loop distribution means splitting a loop in two parts, each with the same iteration count as the original loop. This is done to reduce register pressure and reduce memory bandwidth needs.[BGS90, Section 6.2.7]

The implementation is given in Transformation 6.3 (page 43). Note that the only change for the basic blocks in the loop is that their scopes are updated. Otherwise, everything is copied. The two new loop representatives are put in the same scope and with the same execution count and iteration count as the old loop representative (and connected, which is not semantically meaningful but necessary in order to get a connected fragment; see Section 6.3.2 (page 48) for more on the loop representation problem).

We assume that the loop can be cleanly split (i.e. that both loop parts are closed fragments) and that none of the parts contain a loop inside (this would give us scoping problems; see Section 6.3.2 (page 48)).

Transformation 6.3 Loop distribution transformation.

```
%----------------------------------------------------------
% ARGUMENTS:
%   S: the section in which to operate
%   Begin1: the beginning of the first part of the loop.
%   End1: the end.
%   Begin2: the beginning of the second part of the loop.
%   End2: the end.
%   LoopOld: the old  loop.
%   LoopNew1: the name of the first new loop.
%   LoopNew2: the name of the second new loop.
%----------------------------------------------------------
  transformation LoopDistribution
    (section S, var Begin1, var End1, var Begin2,var End2,
      var LoopOld, var LoopNew1, var LoopNew2)

  inpattern
    fragment loop(S,Begin1,End2)
      cnode(part1,Begin1,End1,Part1Data);  %% first part of loop
      cnode(part2,Begin2,End2,Part2Data);  %% second part of loop
    end fragment

    fragment repr(S,LoopOld,LoopOld)
      node(LoopOld,LoopOldData);           %% the loop representative
    end fragment

  outpattern
    fragment loop(loop1,loop2)             %% loop parts are separated
      cnode(loop1,part1,id,Part1Out,[loop2,loop1]);  %% first loop
      cnode(loop2,part2,id,Part2Out,[loop2]);        %% second loop
    end fragment

    fragment repr(LoopNew1,LoopNew2)   %% Note the false edge from first to second loop representative
      node(LoopNew1,LoopNew1Out,[LoopNew2]);  %% first loop representative
      node(LoopNew2,LoopNew2Out,[]);          %% second loop representative
    end fragment

  transfers
    %% Update scopes for the loops: the new loop names
    Part1Out.scope = forall(X1:Part1Data | copy(LoopNew1));
    Part2Out.scope = forall(X2:Part2Data | copy(LoopNew2));
    LoopNew1Out.scope = LoopOldData.scope;   %%  both loop representatives in same scope
    LoopNew2Out.scope = LoopOldData.scope;   %%  as the old loop.

    Part1Out.execcount = Part1Data.execcount;   %% execution counts are copied.
    Part2Out.execcount = Part2Data.execcount;
    LoopNew1Out.execcount = LoopOldData.execcount;
    LoopNew2Out.execcount = LoopOldData.execcount;

    Part1Out.iterations = Part1Data.iterations; %% iteration counts are copied
    Part2Out.iterations = Part2Data.iterations;
    LoopNew1Out.iterations = LoopOldData.iterations; %% new loops iterate as
    LoopNew2Out.iterations = LoopOldData.iterations; %% often as old loops

  end transformation
```

Loop fusion

Loop fusion means combining two successive loops with the same iteration counts into one loop; it is the opposite of loop distribution [Nul97][BGS90, Section 6.2.8].

The only change needed is to structurally fuse the loops and unify them into one scope. No execution counts need to change. In ODL, none of the loops fused may contain inner loops, since we cannot access their loop representatives to change their scope values; see Section 6.3.2 (page 48).

The ODL code can be found in appendix B, as Transformation B.4 (page 81).

Loop interchange

Loop interchange means that the nesting of loops in a loop nest is inverted: the inner loop is swapped with the outer loop, maintaining the same code in the inner loop. The purpose is to improve the locality of memory references.[BGS90, Section 6.2.1]

To perform a loop interchange, we swap the inner and outer loop iterations for a loop nest. The loops keep their original names, and the basic blocks are not moved between the loops, which means that the structure and scope information can be kept unchanged.

The iteration counts of the inner and outer loops are switched. This leads to a need to update the execution counts of the blocks in the outer and inner loop, as well the execution count of the inner loop representative (relative to the outer loop iterations). The execution count of the outer loop remains the same.

The execution counts for all basic blocks involved are scaled according to the following formula (e stands for `execcount` and i for `iterations`):

$$e_{new} = \left(\frac{e_{old}}{i_{old}}\right) i_{new}$$

For the blocks in the inner loop, the i_{old} is the iteration count of the inner loop, and i_{new} is the iteration count of the outer loop. For the outer loop, the values are swapped.

Note that the resulting values do not have to be integers in the present implementation. See Section 6.3.6 (page 51).

The ODL code can be found in appendix B, as Transformation B.5 (page 82).

Loop peeling

Loop peeling means inserting a small number of copies of a loop body before the loop. This is used to take care of odd operations in the first few iterations, and to prepare the loop for software pipelining and other optimizations.[BGS90, Section 6.3.5]

Loop peeling is handled by putting a copy the loop body before the loop with the same execution count as the loop itself, and then subtracting one from the execution counts inside the loop.

Unfortunately, this does not work well for loop bodies containing conditionals. The execution data (in its present form) cannot tell which of the loop branches was taken in the iteration we moved out, and the only safe approximation is to keep all execution counts unchanged, except for blocks executed on every iteration, where we reduce the execution counts by one. In the peeled copy of the loop body, all execution counts must be set to one, as we do not know which way conditionals go.

The result of the operation is a slight loss of information. The problem is discussed in detail in Section 6.3.3 (page 49).

The ODL code can be found in appendix B, as Transformation B.6 (page 83).

Loop unrolling

Loop unrolling means replacing the body of a loop with several copies of the loop body, and reducing the execution count of the loop with the same factor. The goal is to reduce the loop overhead for jumping back to the loop header, and allow overlapping execution of iterations.[Nul97][BGS90, Section 6.3.1]

Just like loop peeling, loop unrolling cannot be handled tightly if the loop body contains conditional branches. We do not know which way the conditional(s) go in each loop iteration. The only safe assumption is that no execution count of a block in the loop may be greater than the unrolled loop execution count.

For this reason, we chose not to implement loop unrolling.

See Section 6.3.3 (page 49) for a detailed discussion.

Loop unswitching

Loop unswitching means taking a loop invariant conditional out of a loop and creating a copy of the loop in each branch of the conditional, with the code from the corresponding branch of the original conditional. A loop invariant if-statement within a loop is changed to an if-statement containing two loops.[Nul97][BGS90, Section 6.1.4]

Both new loops will have the same iteration count as the original loop. We must allow other code to be present in the loop body before and after the invariant conditional. This code needs to be copied into both copies of the loop.

Because of the way the data is represented, both branches of the loop invariant conditional should have the same execution count before the operation: the loop iteration count. The reason for this is given in Section 6.3.3 (page 49).

The scopes of the loops and of the new if and join blocks are the same as for the original loop, and so is the execution count relative to the surrounding scope. The blocks inside the loops need to update their scope, but since the new loops iterate as many times as the original loop there is no need to change the execution counts.

In the detailed discussion about loop dependence issues in Section 6.3.3 (page 49), we use loop unswitching as an example. The ODL code can be found in appendix B as Transformation B.7 (page 84).

Long ifs

When compiling code using values greater than the natural size for the machine, certain operations will be implemented as sequences of operations because the computer must divide the large operation into several smaller. This may cause the structure of the code to change, and therefore we investigate how we can express these transformations in ODL.

As an example, an equality check would require two comparisons (hi and lo word) in the case of a 16-bit processor and a 32-bit value. The *before* graph in Figure 6.4 shows the probable high level flow chart for the following code:

```
if (a==0)    // a is a long value (32 bit), the int is only 16 bit
  A;
else
  B;
end if
```

The generated code is shown in the *after* graph in Figure 6.4. During code generation, the transformation from *before* to *after* will be made.

The transformation for a long if is easy to express in ODL. The only limitation is that the new if-node is given the same execution characteristics as the original if-node, since we do not know how many of the comparisons that will need to execute only the first (partial) `if`.

To handle this case tightly, we would need detailed information about possible values could occur for the deciding variable, and their respective frequencies of appearance. We would need to know how often the low word of the variable `a` is zero.

The transformation is given in Transformation 6.4 (page 46).

Switches

If we assume that switches are represented naively as basic blocks with several successors in the HLA, we need to transform the code in order to mirror the actual decision structure used. There are several

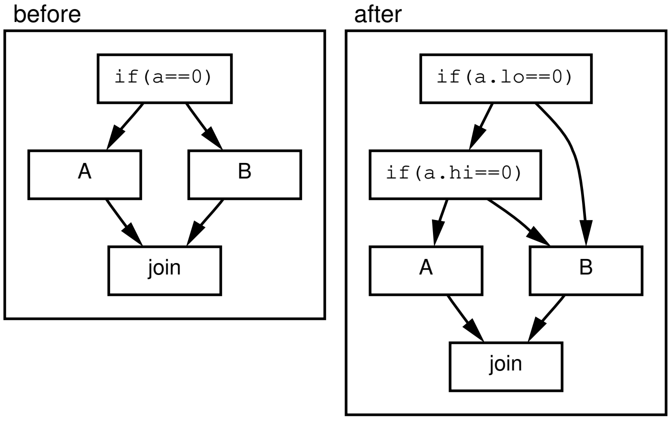

Figure 6.4: A long comparison broken down to two short comparisons.

Transformation 6.4 Long if generation transformation.

```
%------------------------------------------------------------
%   Transformation 2-piece if
%------------------------------------------------------------
% ARGUMENTS:
%   S: the section in which to operate
%   If: the name of the block containing the large if.
%   ABegin: first block in A branch.
%   AEnd: end.
%   BBegin: first block in B branch.
%   BEnd: end.
%   Join: the join block.
%   NewIf: the name of the new block introduced.
%------------------------------------------------------------
  transformation TwoPieceIf
    (section S, var If, var ABegin, var AEnd,
     var BBegin, var BEnd, var Join, var NewIf)

  inpattern
    fragment one(S, If,Join)
      node(If,IfData);
      cnode(a, ABegin,AEnd,AData);
      cnode(b, BBegin,BEnd,BData);
      node(Join,JoinData);
    end fragment

  outpattern
    fragment one(If,Join)
      node(If,=IfData,[NewIf,bbranch]);       %% the old if node
      node(NewIf,=IfData,[abranch,bbranch]);  %% the new partial if node
      cnode(abranch,a,id,=AData,[Join]);      %% same bodies as before
      cnode(bbranch,b,id,=BData,[Join]);
      node(Join,=JoinData,[]);
    end fragment

  transfers
  %% no transfers are needed
  end transformation
```

ways to implement switches, like lists of comparisons, binary trees, and jump tables. Each will need its own transformation(s).

Code generation for switch statements is a complex topic. In [ASU86, Section 8.5], a number of techniques are given: sequences of conditional branches, sequential search jump tables, hashed jump tables, and direct-indexed jump tables. Obviously, each such technique must have its own corresponding co-transform.

There are two problems: ODL cannot handle variable size structures (see Section 6.3.7 (page 52)), and the handling of edges (see Section 6.3.5 (page 51)). The following example demonstrates the problems.

Assume that we are generating code for the following simple C fragment:

```
switch (a)
{
  case 0: A;
          break;
  case 1: B;
          break;
  case 2: C;
          break;
}
```

The initial naive graph and the final code can be seen in Figure 6.5.

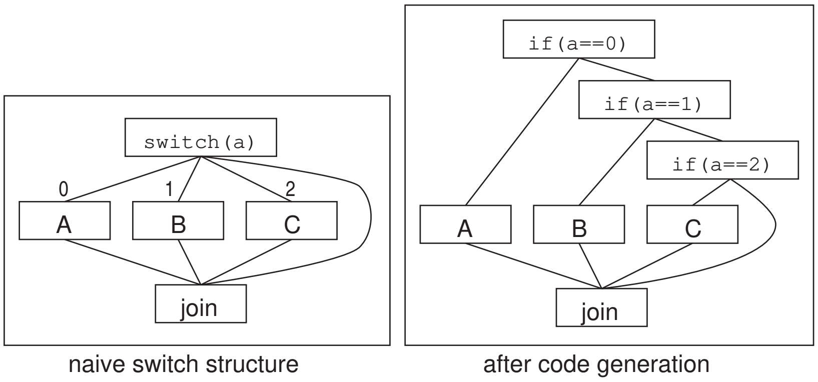

Figure 6.5: Switch statement handling.

We have to reduce the multi-successor `switch` node to a number of two-successor `if`-nodes. If we do it in one step, we need to define a transformation for a variable size structure, which is not possible in ODL (except by defining a transformation for each possible switch statement, which is clearly infeasible).

Alternatively, we can perform the change in a series of step. Each step of the process will take a `switch` node and replace it by an `if` node and a residual `switch`. The problem here is that we would need to change the source of edges leaving a graph fragment, which we cannot do: looking at Figure 6.6, the bold edges in the naive switch structure will be moved to the residual `switch` node when the first `if` is created. The *after* state in Figure 6.6 shows that three edges have changed their origin to the new `switch(a)` node.

Loop introduction

Loops may be inserted into (on the source level) straight-line code to perform data copies between arrays or records, or to initialize arrays, etc.

Introducing loops into the program flow graph corresponding to those generated for certain source language constructs seems quite easy. We simply take a basic block in the graph (the one containing the construction requiring a loop to implement), and split it into three: the part before the loop, the loop, and the part after. The loop basic block is given a back edge to itself.

Unfortunately, our present data structure cannot handle the introduction of new loops, since we cannot create a new unconnected loop representative block without already having matched one. See Section 6.3.2 (page 48). This is a prime example on a weakness in our data representation and in ODL.

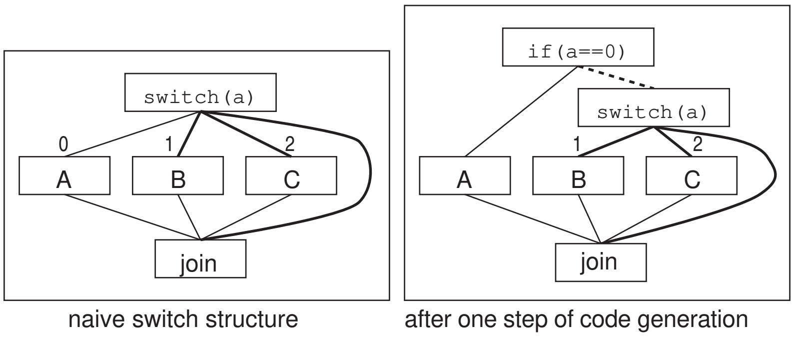

Figure 6.6: Switch statement code generation: the first step.

6.3 Evaluation of ODL

The evaluation of the prototype is really an evaluation of ODL and its data model. The attempts given above at implementing various transformations using ODL clearly indicate that we can indeed implement transformations in a language like ODL, but that we need to reconsider the data model we use.

In this section, we analyze some problems in detail, in order to find their causes and identify what is needed to avoid the problem in future designs. Consider this a list of what we have learned and not a list of reasons why co-transformation is a bad approach. The problems relate to our ODL and, in particular, to its data model, and not to co-transformation in general.

6.3.1 Loops and fragments

All loop edges to be produced in the output of transformation must be explicit in the out-pattern. This makes it impossible to copy a back edge looping back over an entire in-pattern fragment (for single nodes there is a special syntax). The only way to handle this presently is to define different versions of the transformation, one with and one without the encompassing back edge. This affects block merging and if-optimization.

6.3.2 Loop representation problems

The problem with the loop representation used in the prototype is mainly that loops must be explicitly pattern-matched in the in-pattern. We cannot first match some part of a loop, pick up the name of the surrounding loop, and then get at its information. We must get the names of the loops to be manipulated from the transformation arguments.

A second problem with the representation of loops as free-floating basic blocks is that the every loop must be in-pattern matched in a fragment of its own, but that if a new loop is introduced, an artificial connection between the the new loop and another loop is needed as we cannot introduce new fragments in the out-pattern. An example can be found in the section on loop distribution (see page 42). New loops cannot be introduced (unless we matched some loop in the in-pattern, as done in the loop distribution transformation) since we cannot add a free-floating fragment or basic block in the out-pattern.

A third problem is that loops must be named, even though they probably have no names inside the compiler. When chunks of a program containing loops are copied, new loops are created which must then be named. A better representation should avoid this by binding the loops to the basic blocks involved and not giving the loop an explicit name. Note that the basic blocks must always be named, however, since they are the means of communicating positions between the compiler and the co-transformer.

6.3.3 Loop dependences cannot be represented

A problem which appears when performing even quite simple operations on loops is that we do not know which iterations of a loop corresponds to a certain execution path inside the loop. This problem is both a static problem, in that certain dependences cannot be represented, and a dynamic, in that we lose information about the execution when performing a transformation.

Static weakness

Given a set of nested loops, we must assume that for every iteration of the outer loop, the inner loop makes the same (maximized) number of iterations. This leads to large untightnesses in some circumstances.

One example of static weakness (when the execution count of an inner loop depends upon the execution count of an outer loop) is found in Section 2.4.2 (page 6).

Another example of the static weakness is introduced by loop invariant branches inside the loop. The following program will illustrate the problem:

```
c=8;
for x=1 to 10 loop
  for i=1 to 20 loop
    A;
    if (x<c) then
      B;
    else
      C;
    end if
    D;
  end loop
end loop
```

The values of `x` and `c` are not changed by the loop, so the outcome of the `if` is loop invariant. Assume that this is part of an outer loop, where `x` is depends upon the iteration of the outer loop.

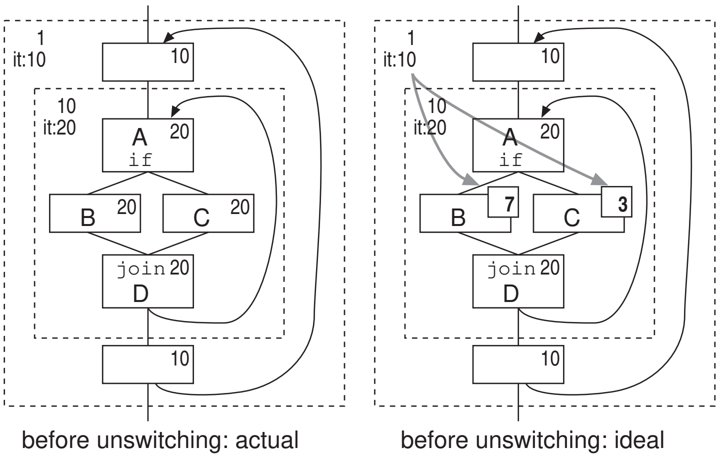

Figure 6.7: An example loop containing a loop invariant if. Illustrating the static weakness.

If branch A is taken even *once* in all the iterations of the outer loop, it will be recorded as having an `execcount` equal to the number of iterations of the inner loop, since that is the maximum number of times that it is executed relative to the inner loop. It will seem that A and B are equally likely to be executed each time through the inner loop. Figure 6.7 (actual) shows the actual state of the execution count figures for the loop above.

The ideal is that we can represent the dependence between the outer and inner loops. This is shown in 6.7 (ideal), where we show how the execution counts of the branches in the inner loop relate to the outer loop.

Dynamic weakness

The second problem with the loop iteration representation is that it is very hard to keep tightness when manipulating loops. The following example quite well illustrates the difficulties encountered when manipulating the present data structure:

```
for i=0 to 19 loop
  if even(i) then
    B;
  else
    C;
  end if
end loop
```

Obviously, the present execution is 10 times `A` and 10 times `B`. When we unroll the loop once, things get more complex: the correct way to restructure the loop body is to make it into straight line code (see Figure 6.8, where the zeros in the execution counts in the picture "after unrolling: ideal" identify the branches never taken).

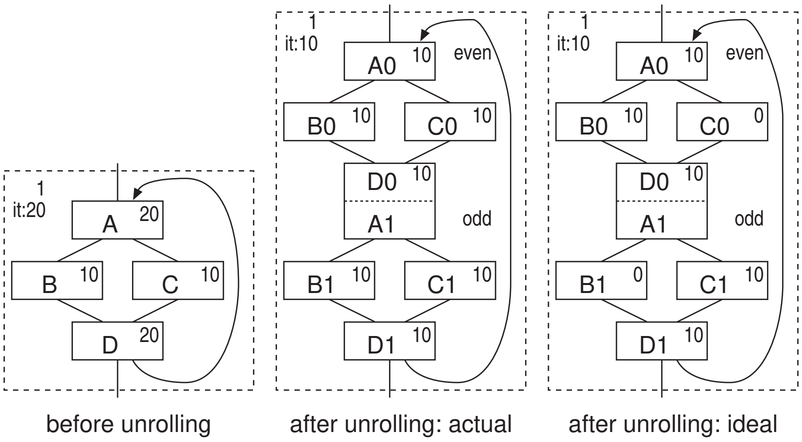

Figure 6.8: Insufficient information when unrolling loops.

Unfortunately, this cannot be captured by our data structure. Because we do not know which iterations correspond to which executions, we have to make the safe and conservative assumption that both `B` and `C` may be executed a maximum of 10 times *in each copy* of the loop body, since we do not know about the even/odd-dependence[5]. This is shown as "after unrolling: actual" in Figure 6.8. The dynamical weakness is clearly seen when trying to implement transformations like loop unrolling and loop peeling.

In the example with the loop invariant if statement above, if we unswitch the loop we end up with grave overestimations of the `execcounts` of the two resulting loops (both will estimated to execute on every iteration of the outer loop), while the actual execution will share the iterations of the outer loop

[5]In the usual case, the compiler would then optimize away the unnecessary conditionals and branch code, leading to a correct end result, but the information in the timing data structures would not let us infer this possibility.

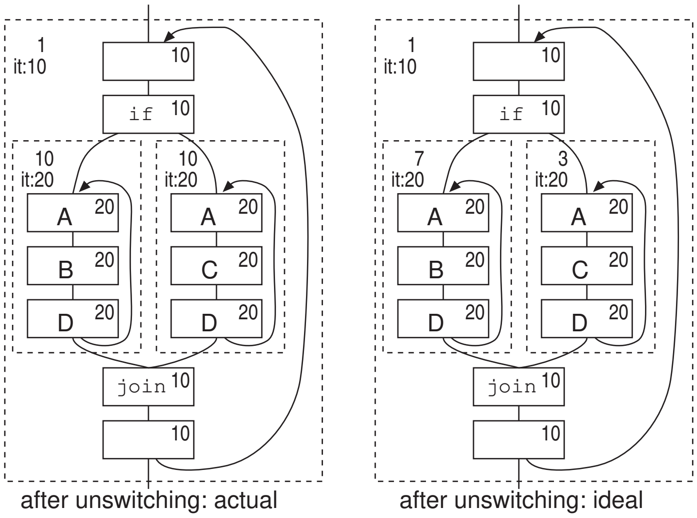

Figure 6.9: After loop unswitching for the example in Figure 6.7.

between them. Figure 6.9 shows the actual state (the overestimation) and the ideal state (where we have converted the loop dependence into execution counts).

What is needed is a way to represent dependences between the path taken inside the loop and the iteration count value of the outer loop. When a loop is unrolled, the iteration count values belonging to each copy of the body could be collected and attached to that part of the loop. See Section 6.5 (page 52).

6.3.4 Edges are not first class entities

In certain cases, compiler transformations simply redirect jumps to new targets (a typical case is branch elimination). This cannot be handled in ODL, as we cannot single out a certain edge and change it.

This indicates that it could be worthwhile to make edges first class entities within the language, allowing them to be changed on their own, without affecting or mentioning any basic blocks.

A related problem occurs when trying to handle code generation for switch statements. Here we would like to redirect the edges reaching out of a certain graph fragment.

6.3.5 Edges and fragments

The present semantics of ODL requires all graph fragments and compound nodes to be closed, i.e. having a unique top node where all edges from outside enters, and a unique bottom node where all edges leave. This restriction makes it impossible to handle certain situations where we would need to have edges leave a fragment from several of the nodes inside. One example of such a transformation is switch code generation.

6.3.6 Execution count representation

As stated above in Section 6.2.1 (page 37), the intended semantics of the `execcount` field is that it is the number of executions relative to the iteration count of the enclosing loop. Intuitively, this should be an integer.

Some transformations generate non-integer execution counts. This does make sense in that it maintains the most information, but it does not make sense in that you cannot execute a basic block for a non-integral number of times. We need to round to the next greater integer when it is time to calculate execution times.

6.3.7 Variable size structures

There are some cases where we have a very regular structure in the code of a program, but the size of the structure varies. A typical example is switch statements (see Section 6.2.2 (page 45)).

Typically, we need to divide a set of nodes into two or more sets and perform some operation on each member of each set. This requires looping constructions and the ability to work with sets of nodes, both of which are not provided by the present version of the ODL.

6.3.8 The ODL language syntax

The ODL language has some less-than-elegant syntactical constructions.

When programming some transformations for loops, we realized how tortured it is to program with non-nested function calls. It does not limit the usability, as we can define the functions ourselves, but it feels clumsy to have to add new functions even for compositions of transfers functions. One example is the `capsub` function used in loop peeling.

A readability problem is that the in-pattern contains no structural information, and this makes it hard to read a transformation definition and understand what is being transformed. This design choice was made because the information is redundant for a correct application and would have complicated the specification of the in-pattern.

6.4 Execution time of the tool

Our prototype tool is quite primitive and not built for speed, but we have not had any problems with large execution times. We never exceeded half a minute in run time on the small toy examples we tested (this was using Erlang 4.4 on a 150 Mhz Pentium under Linux).

6.5 Conclusions about data structures

We have come to the conclusion that most of the problems discussed in the previous section stem from the weakness of the data structure we used in the prototype. We believe that a major improvement in the capability of the co-transformer requires that we improve the data structure used to represent the execution information about the program.

A future datastructure needs to have the following properties:

- The structure must be all-encompassing and consistent. Loops, branches and function calls should be represented within a single graph, in order to allow for the formulation of program transformations on any level. There should be no limit to the size of the hierarchy (at present we have a flat structure, which is quite inflexible).

- To represent the dependences between loop iteration counts and loop body executions, we must allow each iteration of a loop to be represented by an object of its own. This object can then be moved and attached to different parts of an unrolled loop, or follow pieces of code broken out of the loop. A loop can be considered a collection of iterations.

  Dependences from a surrounding scope into a contained scope (like jumps in an inner loop being decided by values in a surrounding outer loop) can be represented by giving each outer loop iteration its own object showing how the inner loop is executed for that iteration of the outer loop.

- We must support the division of straight line code into several sequential parts. We must not assume that each iteration of a loop uses a single value for the iteration variable; when loops are unrolled or pipelined, several different iterations of the original loop are present in one iteration of the new loop.
- The representation and the manipulating functions must allow several ways of representing the same type of objects: loops should be representable both as single objects and as collections of iterations, in order to tailor the data structures used to the complexities of a certain piece of code. Aggregate data must co-exist with detailed.
- The representation must support bottom-up searches: given a basic block, all enclosing objects (loops, functions, etc) must be found. The compiler should be able to speak its native language, static basic block operations, and the co-transformer should handle the problem of converting this to operations on the dynamic data.
- To handle code generation for branching structures (`switch` and `if`) tightly, we need information about all the possible values for the deciding variables and how often they occur, for each passing of the branch. This need is similar to the need to handle individual loop iterations.
- In the formulation of ODL programs, we need to allow flow graph edges to enter and exit a fragment of code at more places than the top and bottom. We cannot require that every piece of code treated is closed. We must be able to manipulate individual edges on their own.

Research into datastructure having the properties given above is one of the follow-up projects we are planning. A good datastructure for the representation of program execution characteristics is also useful for a high-level analyzer (to represent the result) and for other program investigation tools.

Chapter 7

Previous Work

This chapter on previous work has been organized per theme rather than by research group or paper. This is to make it easy to refer to it from other parts of this thesis, and to show how other work fits into the framework we provide in Chapter 3.

7.1 State of the practice

The state of the practice in timing analysis today is strictly speaking not "previous work". However, we feel that a short overview of the methods *used* today to determine execution times is in order to place our work into context.

The state of the practice in industry today seems to be various forms of measurement and educated guesses. In [EG97b] they have investigated which methods are actually used in practice, and they give the following list of techniques:

- *Manually counting assembler instructions* and calculating the time taken for instruction sequences, using the cycles per instruction tables provided by the processor manufacturers. The downside is that the calculations are rather tedious and it is easy to make errors. Also, the manuals may not always be correct.

- Using *emulators* to monitor and measure program execution. The emulators are only available for certain processors, and they are usually quite expensive. The greatest disadvantage is that the results are only measurements. The advantage is that the measured program is not changed and the measured execution times have quite high resolution.

- Using *oscilloscopes* to monitor the execution frequency of certain pieces of code. This requires inserting "debug print statements" into the program which generates signals that the oscilloscope can capture. Unfortunately, this changes the program.

- Using *logic analyzers* to monitor the data flow to and from a processor, and determine when certain events take place and the time between them. This should be possible to do without changing the program being examined. The disadvantage is that we only measure the program.

- Using processor *simulators* which can return information about the simulated time required to run a program. This is almost the same as a "software emulator", and has the same advantages and disadvantages.

- *Profiling* the program. Manual (or tool based) profiling of the program by inserting *timing calls*. The disadvantage is that the program is altered, and that we only measure the program.

Notably, nobody reported using any software tool except for profilers. This is probably due to the (to our knowledge) lack of any commercial tools for timing analysis.

Several of the timing approaches require that probe code is inserted into the program (known as *instrumenting the program*, or *invasive* methods). This has the disadvantage that it is not the final version of the program which is tested, or (in the case that the measured version is shipped) that the shipping version is equipped with extra code which slows it down and adds complexity (if not much) to the system.

Only one approach, the manual calculations, has any theoretical chance of discovering the true maximum execution time of the program. The other approaches are measurements, and thus lacks in the *safety* quality dimension.

From this we conclude that the need for software development tools supporting execution time analysis is great. In [EG97b], most respondents replied that a software tool to help them would be most useful.

Even a simple static tool produces more reliable figures than measurements, and the time spent should be less (since we do not have to construct test cases and run the program on them).

7.2 Specific WCET estimation problems

In this section we present research dealing with particular sub-problems of obtaining (tighter) timing estimates.

7.2.1 Infeasible paths

The identification of infeasible paths has received comparably little attention. We have found the following research in the specific problem of identifying and representing infeasible paths:

- The easiest solution is to assume structural longest paths in every case. This gives a potentially large overestimation, but is easy to program and the only approach possible in the absence of information about the program paths. It has a low complexity and can run quickly.

- Many approaches to the WCET problem include attempts to *represent* information about infeasible paths, usually obtained from the programmer using manual annotations. See Section 7.3.1 (page 58) for more on annotations. The main disadvantage is that the quality of the annotations can be doubted.

- Kountouris describes an approach applied to the SIGNAL language. The language is very high-level and therefore easier to analyze than simple languages like C or Ada (a SIGNAL program is usually compiled to C in order to be executed). The algorithms described are applied during the compilation of SIGNAL programs, and eliminate the need for annotations. The approach seems exact: all infeasible paths are found. The complexity is manageable thanks to the high level of abstraction in the language [Kou96]. The disadvantage is that the analysis is performed for a language which is not in common use.

- Altenbernd has performed research into heuristically determining infeasible paths in simple C programs, as a part of the CHaRy software architecture at C-LAB in Paderborn [Alt97], where his heuristic is used on non-looping programs automatically generated from a higher-level representation. A heuristic is needed since the general problem of identifying infeasible paths is assumed to be NP-complete, even for the simple programs generated in the CHaRY project [Alt96a]. The disadvantage of the approach is that loops and function calls are difficult to handle.

In this thesis we have not done any work on representing or identifying infeasible paths. Because we aim to handle reasonably complete C programs, we will have to figure out some heuristic approach in the HLA. The representation of infeasible paths in a transformable way is still an open question.

7.2.2 Function calls

There are several approaches in the literature to the problem of calculating and representing function call execution times:

1. The WCET for a function is a *given constant*. The *user* gives the system information about the WCET for each function, obtained in some unspecified way. This approach is taken in [For93]. The WCET is the same for all calls to a function. This has the potential to lead to overestimations and makes it hard to capture the behavior of complex functions. Manual calculations of execution times must also be treated with caution with regard to correctness.

2. The WCET is a constant which is *calculated once from user annotations*. Given (non-contextual) loop bounds and other annotations, a WCET for a function can be calculated. The WCET is the same for all instances. This assumes that a function can only run in one way. The disadvantages are the same as for the previous approach, but the risk of error is smaller, since we calculate the time by program instead of by hand.

3. The WCET for a function is *provided a priori* as a function of the function parameters. The *user* has manually deduced a function giving the execution time of a function as a function of its parameters. The WCET varies according to the instance. The functions could capture most behaviors, but generating such functions is not a trivial task. We doubt the feasibility.

4. The WCET function for a function is *generated from other user information*. This approach is used in [CBW94, Cha94], where the user may supply a number of modes for a function. A mode is a set of input values. Each mode is given its own set of execution constraints, and a symbolic expression for the execution time for a certain function in a certain mode is deduced. Function calls are then analyzed to see which mode they belong to, and the corresponding function used. This approach is quite good, except for the identification of relevant modes which may be hard to do. It is not certain that the program source code captures all timing behavior of the program. If the modes are badly chosen, the timing results can be untight (if there is some really bad case in each set of inputs, for example).

5. Using all information at the call site, and re-analyzing the function for each call instance. The disadvantage is a long execution time for the analysis, and the potentially huge size of the program representation.

We have chosen the last approach, analyzing each function instance separately, as our preferred approach. The systems we aim at analyzing are rather small, perhaps 64 kB programs at most, and we believe that the fact that our development workstations are much faster and equipped with orders of magnitude more memory will make the approach feasible. We would not expect the approach to be applicable to PC-class applications.

Note that in the ideal case, analyzing each function instance and merging identical calls would automatically give a result similar to the fourth item above. The analyzer would deduce (some kind of) function modes.

7.3 High level analysis

Most research projects in the WCET area touch the subject of high level analysis, but only a few make a serious inquiry into it. So far, the field has been dominated by projects using annotations to obtain more information about the program from the programmer. A few researchers have tried to deduce information automatically.

7.3.1 Annotations

The most common approach in the WCET field is to let the user annotate the code of the program with information about program flow. There are a number of different flavors of annotations; the following list gives a brief overview of the approaches found in the literature:

- The MARS project introduces a rich set of annotations [PK89, Vrc93, Vrc94a], allowing for the specification of loop bounds and limits on executions of various parts of the loop body relative to the surrounding loop. The annotations are part of the language used (`Modula/R`), and are processed by the compiler. The MARS system solves the problem of erroneous annotations by checking the limits on execution given in the annotations at run-time, and raising an exception if they are violated. We doubt the real-world desirability of this approach — it would probably be better to keep running and hope for the best. During development, it is the correct approach: fail eagerly in order to catch errors, but not in shipping software.

- The `SPATS` tool adds a set of annotations to SPARK Ada to allow execution time analysis [CBW94, Cha94]. The annotations are used to divide programs into appropriate pieces for analysis, and to give bounds on the input and output data from a function. The annotations are entered inside comments and are processed by stand-alone tools. Program correctness can be proven in parallel with the calculation of the program WCET, using the same tool.

- Forsyth [For92] uses Ada `PRAGMA` statements to enter information about loop bounds, and has modified the York Ada compiler to emit the information from the `PRAGMAS` into the debug information in the object code. The debug information is then used by the timing tool.

- Börjesson [Bör95]uses some special `pragmas` to add a WCET capability to the (commercial) IAR Systems C compiler [IAR97]. Information about loop bounds and infeasible paths can be entered. Infeasible paths are specified by declaring that two blocks of code lie on the same or different paths.

- The `cinderella` tool [LM95, LMW96, Cin97] prompts the user to interactively provide the information required for timing analysis. The information which can be provided are loop bounds and infeasible paths. Every basic block in the program is assigned a variable, and path restrictions are entered as linear constraints over these variables (they use linear constraints to calculate execution times; see below in Section 7.5 (page 61)). The basic block variables are displayed on the source level.

- Park and Shaw [Par93] have created a language called IDL, *Information Description Language*, to allow the programmer to add information about infeasible paths and loop bounds to a program. Park hints at the possibility of using (manual) proof techniques to check the validity of the annotations, but no work has been performed on the validation.

- The annotation information may be provided directly at the assembly code level. This requires the programmer to maintain a mapping between source level and object code loops. Usually, only loop bounds can be provided. This approach is taken in [LBJ$^+$95, AMWH94].

As stated above in Section 3.1 (page 16), all approaches using annotations share the problem of the quality of the annotations. Annotations cannot not be trusted unless supported by some form of validation; humans do make mistakes. Some approaches attempt to mitigate the problem by using run-time checks or recommending manual proofs or code inspections.

We believe that annotations are useful as an aid to automatic analysis, and that they can be used to prototype and develop a timing tool. However, in the long run, we want to automate the analysis as much as possible. This is not a problem dealt with in this thesis, since it only affects the HLA.

7.3.2 Automatic methods

Most of the research in automatic program analysis has been performed outside of the WCET field. This is a classic field in computer science, and there is a vast body of research to build on. Some techniques which appear to be useful are:

- *Data flow analysis* (also known as dependence analysis) is a standard technique used in compilers to deduce information about a program in order to allow for optimization, scheduling and other program transformations. The information gained typically includes which instructions depend on which, and how. Dependences might be carried between loop iterations in modern implementations. In general, compiler data flow analysis is used in too limited contexts to be used for execution time analysis. One area where the results are useful is in memory reference analysis, where we want to figure out the addresses referred in order to model the data and instruction caches of our processor [ASU86, BGS90, HHG+95].
- *Abstract interpretation* is a "a general theory for approximating the semantics of discrete dynamic systems". It works by approximating the semantics of a program language, typically by replacing single values with sets of values [Cou96, Cou81].
- *Symbolic execution* is another approximation approach with a slightly more operational slant. The program is executed using symbolic values for variables instead of concrete values, and the result is symbolic expressions for the values of variables in the program [CR81, Alt97]. Depending upon the accuracy, the approach can be quite time-consuming; it is hard to find a good trade-off between accuracy and performance.

It should be noted that the difference between symbolic execution and abstract interpretation is quite unclear, and it seems useful to blend elements of both techniques. These high-level techniques are useful to obtain knowledge about how the program executes, and we intend to apply them to the HLA of real-time programs.

In the WCET community, some work has been performed on automatically analyzing programs with an eye towards obtaining the information needed by timing analysis tools.

- Ermedahl and Gustafsson [EG97a] analyze a small subset of C using techniques related to abstract interpretation and symbolic execution. The resulting data includes loop bounds for loops, even with quite complex dependences. The present subset is rather small, and they do not allow global data.
- Gustafsson et al [GE97, GPMTB97, RTT97] have analyzed the object-oriented RealTimeTalk language, trying to obtain information useful for WCET analysis for object-oriented programs. The research includes type inference for RealTimeTalk, which should help the WCET analysis by limiting the set of possible functions called by a polymorphic method invocation, and automatic derivation of loop bounds and other information about a program.

We believe that the automatic approach is better than manual annotations, and hope that we some day will be able to integrate a fully automatic analyzer into a WCET estimation tool.

7.4 Low level analysis

The purpose of the low level analysis is to derive the timing information about a single basic block needed to calculate the execution time of an entire program. In the simplest case, this could be a single value (number of cycles), but for more complex processors, the information required can be quite complex. The following approaches have been found in the literature:

- The simplest approach is to simply sum the number of cycles for each instruction in a basic block given the information about the processor found in the manuals. The cycle count can either obtain a single number or a (min, max) pair. The number of cycles can then be multiplied with the cycle time in order to yield a execution times. This requires that the execution time of each instruction can be calculated in isolation, without interference from surrounding instructions or other basic blocks. The approach is applicable to simple processors, which are often found in embedded real-time systems, but gives enormous overestimations for processors employing caches or pipelines.
- A more complex approach called *microanalysis* is described by Harmon, Baker and Whalley [HBW94]. This involves constructing a very detailed model of the processor, and essentially executing each straight segment of code on it. The drawbacks of this approach are that it is very hard to construct a good model, that the model must be reconstructed for every version of every processor, and that the information needed is so detailed that the manufacturers may be reluctant to provide it. The advantage is that very good estimates may be produced. A tool has been created which handles the Motorola 68010 and Intel 80386 processors. The approach only applies in limited ways to cached or pipelined processors.
- Lim et al [LBJ+95] describe an extension of the *timing schema* of Park and Shaw to handle caches and pipelines. They handle pipelines and direct-mapped instruction caches. Possible ways to handle set-associative instruction caches are described, but apparently not implemented. Data caches are handled is a simplistic way, without any attempt to handle addresses not known at compile time. They introduce the idea of a WCTA, a worst-case timing abstraction, which is a concept related to our idea about the connection between the LLA and the calculator.
- Some research groups centered around Florida State University and Florida A&M University [WMH+97, HWH95, AMWH94] have built a powerful low level tool using a static cache simulator. They handle pipeline effects, set-associative and direct-mapped instruction caches, and direct-mapped data caches. The approach is all-encompassing, and they have produced a number of papers on the subject of WCET analysis in general and the handling of caches in particular.
- The CHaRY software architecture [Alt97], uses the advanced Motorola PowerPC 604 processor, and thus need to handle cache and pipeline issues in order to obtain good timing estimates. The code they analyze is rather limited, containing no function calls and no loops. This simplifies the problem enough so that their *Program Timing Analyzer* can handle the effects of pipelining and set-associative instruction and data caches [Sta97].
- The `cinderella` tool from Princeton [LM95, LMW96, Cin97] implements an approach which is powerful enough to handle pipelines, set-associative instruction caches, data caches and unified caches (other approaches have assumed the data and instruction caches to be separated). They allow for set-associative data caches. The tool has been implemented for the Intel i960 processor and the Motorola 68000.

From the work described above, we can infer a difficulty scale for doing low level analysis. In increasing order of difficulty:

1. Single value cycle counting: processors which are perfectly predictable, and where all instructions have a single well-defined execution time. No caches, pipelines, or anything. A single execution time value can be ascribed to each basic block.
2. Variable-time instructions: more advanced processors include instructions which can take a variable amount of time to execute. Typical cases are `mul` and `div` instructions, and branch instructions which take different time to execute depending upon the outcome of the branch. The execution time value is a (min, max) pair for each basic block.
3. Pipelines and similar devices which causes the execution time of an instruction to depend on the surrounding instructions. Execution times cannot be uniquely bound to a single basic block anymore, as the context from previously executed basic blocks are needed to obtain an accurate estimate. More complex information needs to be sent to the calculator.

4. Instruction caching: instruction caches require the modeling of flow on an even greater scale than pipelining. Parts of the program far apart both statically and execution wise can affect the execution times of each other due to cache conflicts. Direct-mapped caches are easier to handle than set-associative caches. The calculator information needs to include caching effects, and is expected to become very complex.

5. Data caching: data caches requires that we analyze and model the data accesses of a program, which is much more difficult than the instruction cache, where all addresses are known and constant. Like for instruction caches, direct-mapped caches are easier to analyze than set-associative. The calculator information will be more complex than the instruction cache information.

Most of the work in the field is centered on managing cache memories and pipelines. We expect our first tools to use the cycle counting technique for simple processors (single or (min, max) pairs), and to adopt one of the cache and pipeline managing approaches later.

7.5 Calculations

There are several different methods of calculation used in the literature on WCET estimation. Note that there are really two different issues involved: one is how to represent the control flow information and execution time information about each basic block, and the other is how to use this information to obtain actual timing estimates (finding the execution time in the worst case). In the list below, we treat them together, since each approach consists of both a representation and a calculation method.

- The simplest approach is the complete and explicit enumeration of all paths through a program. Information about loop iterations is used to produce all possible paths, and information about infeasible paths can be used to remove some paths from this set. Then, a time is calculated for each path. This is not a feasible approach for most programs since the number of paths is exponential in the number of branches encountered (a single if within a loop will double the number of paths for each iteration of the loop). Nobody has actually used this approach, because its exponential complexity. Each if-branch encountered doubles the set of possible executions; an if within a loop produces 2^n paths (where n is the maximum number of iterations of the loop).

- The CHaRy system [Alt97], uses a *k-longest path* search. The k longest paths are enumerated, and then checked for executability in order to decreasing length. This offers a reasonable compromise between performance and precision for the simple code they are examining, but has problems handling loops (which are not present in CHaRy).

- The MARS project at TU of Vienna originally used a *timing tree*, which is an hierarchical structure resembling an abstract syntax tree. This tree is used both for calculations and for presenting the result of the analysis to the user. The tree is evaluated bottom-up, and each node contains information about the timing for a certain statement in the program. The approach requires that the executable program is well-structured[1] [Vrc94b, PS93].

- Park and Shaw [Par93] use *timing schema* to calculate the execution times of a program. A timing schema is a formula which describes how the execution time of a statement is composed from the execution times of its constituent statements. A language called IDL, *Information Description Language*, is used to sharpen the analysis by allowing the programmer to add information about infeasible paths and loop bounds. The final result is a (min, max) pair of times.

  The approach is based on analyzing the structure of the source program, and assumes that reasonable time bounds may be ascribed to each constituent statement. This makes the approach unsuited for optimized code, where the execution time for a certain source statement is hard to determine.

[1] A well-structured program is one where every loop has a unique header. A program written using no `goto`s is always well structured.

- Chapman [Cha94] uses regular expressions to represent program paths and graph rewriting rules to perform calculations. Symbolic values are used to represent the time taken to execute each basic block in the program, which means that the final output is actually a symbolic expression which can be filled with values in order to obtain an actual time. A problem with this approach is that caches and pipelines cannot be taken into account because of the assumption of a single constant time for each basic block.

- *Implicit Path Enumeration*, IPE, is a technique based on linear constraints. Using linear constraints, both control flow information and the information from the LLA are expressed in a single constraint system which can then be solved to maximize or minimize the execution time (to obtain WCET or BCET estimates).

  The advantage of the IPE approach is that we can model any code, even if it is not structured, and that our calculation structure naturally follows from the assembly-language layout of the program. The main disadvantage is that we do not know which execution of a program gives the maximum (or minimum) execution time: this is the trade-off for a reasonable execution time. Furthermore, the complexity of the analysis is very high in the worst case (exponential), even though in practice most problems can be solved quickly because programs produce quite well-behaved constraint systems.

  The IPE approach has been used in a number of research projects [PS95, LM95], and has been extended to handle cache and pipeline analysis [OS97, LMW96]. A good introduction is found in [PS95].

We consider IPE to be the best calculation device for our framework, since it makes it very easy to unify the information obtained from the HLA via the mapper with the LLA information: all information must be structured like the object code for the program.

From a performance viewpoint, we can build on research in the field of constraint satisfaction and linear programming to improve the performance of our tools.

7.6 Mapping

The problem of mapping information between the source code and the final object code is encountered in all fields related to program development. In this section, we investigate previous work on the problem of mapping, with a focus on co-transformation–like approaches. We have singled out three fields for investigation: WCET analysis, compiler internals, and debugging.

7.6.1 WCET research

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. We have found the following ways of mapping information from the high to the low level:

- Use *debug information* in the object code to map the high level analysis information down to the low level code.

  The advantage is that little extra work must be carried out by the compiler, and that changes to the compiler do not affect the timing tool (only changes to the output format are of interest).[2]

  The disadvantage is that it is quite difficult to map the information using just debug information, and that much information about the program structure disappears.

  Debug information is used in the York Software Engineering timing tools [For92], in the SPATS tool [Cha94], and by the `cinderella` tool [Cin97, LMW96].

[2] Using debug information in some sense amounts to avoiding the issue by transforming it into somebody else's problem: the compiler writer has to provide enough information, but nothing is said how she manages to do so. The fundamental problem of making program control flow and structure information survive optimizations still remains.

- Several groups *modify a compiler* to output information about the program flow, data addresses, and other information needed. In our framework, this means using the compiler to perform both the HLA and the mapping. The idea is a step on the way to co-transformation, but it is rather ad hoc and not well documented [AMWH94, HWH95, WMH$^+$97, LBJ$^+$95].

- The timing tool can *co-transform* the timing information: every time the compiler changes the code, an equivalent operation is performed on the timing information.

  This has the advantage that any transform can be handled (given that the compiler or tool writer generates a corresponding co-transform).

  The disadvantage is that a co-transformation must be defined for each transformation that changes the structure of the flow graph. The introduction of such a system is expensive for the compiler writer.

  This approach is used in the `Modula/R` compiler for the MARS project, where a timing tree is processed at the same time as the code [Vrc94b]. Chung and Dietz also describe an approach where timing information is explicitly manipulated by the compiler [CD95].

We believe that the co-transformer approach is the best solution. It makes intuitive sense to send the information about the program in the same direction through the compiler as the code itself. In Chapters 4, 5, and 6, we discuss the concept in more detail and describe our prototype implementation.

7.6.2 Compiler technology

Compilers are involved in all approaches to mapping, since it is the compilers which create the problem. There are some cases where co-transformations are used *internally* in compilers to maintain information during the compilation process.

- A common solution to data flow problems is to simply do some analysis each time information is needed. This avoids the need to track any information at all, but is only feasible for simple analysis techniques.

- In the case that we perform some more expensive analysis, like array dependences or alias analysis, it is advantageous to maintain the information through the compilation, in order to save execution time. In [HHG$^+$95], they discuss the problem of maintaining information about arrays through the compilation process. They also refer to a different solution, where source level information is maintained instead of the result of the source analysis (used in the Multiflow and Cydra-5 compilers).

We believe that the problems faced by compilers and the problems faced by timing analysis tools will converge in the future, as more complex semantical analysis enters into compilers. Presently, the information maintained seems simpler than what we aim for in the timing analysis co-transformer.

7.6.3 Debugging

In the area of symbolic debugging, the problem of debugging optimized code has been studied. Debugging optimized code requires a correspondence between optimized object code and source code, a problem similar to that faced in mapping information for WCET analysis (this is noted in [KHR$^+$96]). We have come to the conclusion that there are some similarities, but also many differences between the needs of debuggers and the needs of timing tools:

- Both debuggers and timing tools need to transform information about a program from the source level to the low level.

- Debuggers are interested only in the *static* structure of the program. The *dynamic* information is only needed for timing analysis.
- Debuggers need detailed information about which code belongs to which source statements, which is relevant to the presentation of timing results, but not to the timing analysis in itself.

In the following, we give a brief overview of some work in the field of debugging optimized code.

There are two distinct subproblems when debugging optimized code: the *code location* problem and the *data value* problem [Zel84]. The data value problem concerns the question where the value of a certain variable can be found at a certain point in the program execution, and whether this value is up-to-date. The code location problem is determining which source statement corresponds to a certain object code instruction, and vice versa. This information is needed by the user interface of debuggers.

In debugging research, the data value problem has attracted the most attention [AT96, p. 28], and in [Coo92, p. 14] it is stated that most problems in debugging are data value problems or can be reduced to them.

For the purpose of WCET analysis, the code location problem is the most relevant. This is a static problem, where correspondences need to be found between the source and object code versions of a program, regardless of the paths taken. The mappings required are summarized in [AT96]: one mapping from object-code instructions to source-level expressions or statements (to show where a program is stopped) , and one mapping from source-level statements to object code (in order to place breakpoints).

- Zellweger completed a PhD thesis about the debugging of optimized code in 1984 [Zel84]. According to Cool [Coo92], Zellweger attacks the problem of mapping source and object code to one another from the execution path perspective.

  She inserts hidden breakpoints into the code, and uses these to determine by which path execution has reached a certain point in the program. This allows her to solve some data value problems precisely, but the program is modified and thus we are back at analyzing non-final code.

  She also mentions the idea that the user may be presented with a modified source program that more closely corresponds to the program being executed.

- Copperman [Cop92] approaches the problem of mapping optimized and unoptimized code by maintaining both an unoptimized and an optimized flow graph for the same program. His main focus is on data value problems, and correspondences between paths. He gives a good overview of the possible ways that optimized and unoptimized code may relate to each other, and describes an algorithm for determining the status of a value at a certain point in the (optimized) object code.

  The main method for static correspondence is to insert special markers into the unoptimized program (marking the end of each basic block), and then see where the markers end up in the optimized version of the program. The compiler needs to maintain the information: a concept similar to co-transformation.

- Cool [Coo92] attacks the problem of debugging optimized VLIW[3] code. The main problem he identifies is that some source program variables will be present in several versions at once. Typically, these problems are caused by techniques that overlap several executions of a loop (such as software pipelining and loop unrolling and scheduling [BGS90]). This problem is also present with modern superscalar architectures.

  His solutions are

  - A user interface that helps explain the changes to the program.
  - More detailed debug information —tracing the *expressions* and not source lines or source statements.

[3] VLIW: Very Long Instruction Word; a type of processor where several concurrent instructions are coded into a single instruction word.

  - He only uses one code map, from object code instructions to the corresponding source level expressions.
  - To handle overlapped execution, the code map contains indications that several instances of a source expression may be in execution at one time.

- Wismüller [Wis93, Wis94] attacks the data value problem in loops.

  He maintains a copy of both the source and object code flow graphs of a program, and maintains a relation called CODE between them. The creation of the mapping CODE is never described in detail, but an example is given in [Wis94]. The mapping has two parts: nodes (instructions) to nodes, and edges to edges.

  To handle loops and loop nests, he uses a technique whereby the graph for a loop in unrolled a few times in order to make it possible to determine which variables are current (available with expected values) and which are not.

- Adl-Tabatabai has written a thesis on the subject of debugging optimized code [AT96], as well as a number of articles in cooperation with Thomas Gross [ATG96, ATG94, ATG92]. The work has resulted in the implementation of a prototype as a part of the `cmcc` optimizing C-compiler.

  His approach to debugging requires *two* code mappings (as described above), one to obtain the source position corresponding to an exception, and one in order to place breakpoints.

  The mapping is created by placing special marker labels in the intermediate code of the compiler, and by modifying the compiler to maintain this information through its optimizations. Markers are placed to mark deleted, inserted, duplicated, and moved code. When the optimizations are done, the tables are created from the markers left in the final code (but not present in the object code). This is similar to the approach of Copperman, and basically a co-transformation.

Debugging optimized code faces many problems similar to those in timing analysis, with a slightly different angle, as the final information to be obtained differs.

The best approached in the field of debugging optimized code seems to use markers in the code in order to build correspondences, and a modified compiler in order to maintain the information. This is basically a co-transformation, even if the information co-transformed is simpler than what we handle.

7.7 User interface issues

The following work in user interfaces for execution time tools have been found in the literature:

- The MARS project uses *timing trees* to relate information about the time taken to execute certain parts of a program to the user (the timing trees are also used to calculate execution times and to maintain timing information during compilation). Times are displayed for individual source statements [PS93].

- The `cinderella` tool has a graphical user interface used to present information about the program to the user, and to prompt for information used in the analysis. Timing information is displayed per function [Cin97].

- Harmon et al has been working on user interfaces for timing tools. They realize that optimized code causes problems, but proposes no solution to the problem. The user can view information about the program on the level of functions, loops, paths, subpaths, and ranges of machine instructions. Pipeline diagrams can be invoked to show detailed information about the execution of the program [KHR+96].

- [Coo92] presents the idea that the user interface should try to explain the changes the program has undergone. This would help the programmer understand his program better, and to see what the compiler has done to it, fostering trust in the compiler as well as making it easier to spot bugs in it.

We believe that some kind of graphical presentation of timing results is necessary; the user interfaces of profiling tools should provide some clues as to what is needed.

We also believe that it is very important that the user is given enough information to understand what the compiler has done to her program, and to explain exactly what is being timed. A tool which works by doing “magic” will not be trusted, which lowers its utility.

Chapter 8

Conclusions and Future Work

In the final chapter of this thesis, we present our conclusions and some ideas for future work in the field of WCET analysis and real-time programming in general.

8.1 Conclusions

This thesis has presented a new architectural framework for static timing analysis of optimized code, and a new approach — *co-transformation* — for mapping information about a program from the source code level to the object code level.

The framework allows reasoning about static timing analysis in terms of five components:

1. The high level analysis is responsible for obtaining information about the program execution.
2. The low level analysis for determining execution times of atomic units of the program (from the object code).
3. The mapper maps the information from the high level analysis to the low level.
4. The calculator calculates the final times given the results from the mapper and the low level analysis.
5. The user interface provides the user with control over the tool, provides the tool with information from the user, and displays the results to the user.

In order to analyze optimized code we need to cooperate with the compiler. The connection to the compiler is one part of our framework. The problem of managing compiler optimizations has been defined to be the problem of mapping program execution information from the source code level to the object code level. This functionality has been localized to the mapping component, where we propose a novel method called co-transformation. A co-transformer transforms program execution information in parallel to the transformations of the program code performed by the compiler.

We have implemented and evaluated a prototype of a co-transformer. This prototype works well in its limited area of application: we are able to run the co-transformer over a trace of optimizations for a program, and the final result is consistent with the execution of the program.

To describe co-transformations corresponding to compiler transformations, we have created a language called ODL (Optimization Description Language). The idea is to formalize program transformations into a language of its own, in order to make it easier to define co-transformations and to extend existing co-transformations to new kinds of data.

For the limited forms of data we have considered, the concepts of ODL are sufficient, but it will not suffice for more complex data. We need to put much more effort into designing a datastructure which is powerful enough to capture all relevant information about a program, and malleable enough to maintain the information through the shifting shapes of the program during the co-transformation process. We also need an extended ODL which can handle more complex data. However, we have to design the data structure first and the language later.

8.2 Future work

During the work on this thesis, a number of related and unrelated research questions have appeared. We give a short summary of the most important and interesting issues below, and hope that we (or some other researchers) will have time to deal with them.

8.2.1 Continuing towards a timing analysis tool

We want to continue our work towards a complete timing tool. This involves creating at least one instance of each of the components in the framework presented in Chapter 3.

The first step should be to create a stand-alone low level analyzer, which can be used to help (and to some extent replace) the measurements used in industry today. A successful low level analyzer would introduce the concept of a timing tool to the practioners in the field, and create a demand for more advanced tools, as well as solving some immediate time-consuming problems.

More advanced tools may be introduced gradually until we have a complete tool on the market. The time perspective for this is probably between five and ten years.

8.2.2 Transformation formalism

The ODL is actually not just an input language to the co-transformer. It is a first attempt at creating a formalism for describing program transformations. If such a formalism could be created, with well-defined semantics and enough power to handle all possible program transformations, the implementation of compilers and co-transformers could be simplified. It would also open the door to formal analysis of compiler transformations, and the possibility to prove the correctness of compiler optimizations. We believe that the development of such a formalism would be a worthwhile project, and that the result probably will be quite different from our present ODL.

8.2.3 Integrating the compiler and the HLA

A very interesting question is the extent to which the compiler can aid us in automatic high level analysis. Optimizing compilers use data- and control flow analysis to discover opportunities for optimizations, and they have a lot of information about the behavior of the program. It would be very helpful if this information could be used by the HLA. This would avoid duplicating work, and would ensure that the HLA reaches conclusions about the program execution which are consistent with what the compiler uses to performs its optimizations. An even more drastic development would be to use the results of the HLA to help the compiler optimize the program. The front-end of the compiler could be merged with the HLA to produce a single unified tool.

8.2.4 Optimizations and real-time programs

An interesting question raised by our investigation in the calculation of worst-case execution times regards the construction of compilers of for real-time systems: *which optimizations are actually worth doing?*

In the case of an ordinary desktop program, average-case execution time is often the main measure of performance. An optimization increasing the time taken to execute an exceptional case while gaining in the average case is a good optimization.

In a real-time system, this is not necessarily true. If execution time estimates are used to guide the scheduling or dimensioning of the system, an optimization increasing the exceptional case can actually make things worse, if the exceptional case happens to be the worst case. As increase in the estimated WCET will force us to dimension our system accordingly, which could increase the cost and decrease the total efficiency. An increase in average speed need not translate into savings or increased system efficiency.

It would be interesting to investigate the direction of optimization with regards to speed in real-time systems. Perhaps it is better to optimize for the worst case and avoid skewing the execution time profile too much (modern RISC optimizations have a tendency to skew the profile to gain in the average case). The question is complex, especially as we often optimize for space as well as speed, and the relation and priority between these two kinds of optimizations makes it extremely difficult to find a true “optimum”.

8.2.5 Data dependences in real programs

As stated in Section 4.3.2 (page 25), we assume that the global data manipulated by a program does not affect the control flow. This is necessary to allow local evaluation of functions. We have no firm data on how severe a limitation this actually is for real programs. This must be investigated further to establish how (global and non-global) data is used in real-time programs.

8.2.6 Real life programming

The programming habits of real life programmers should guide the research in the area of real-time systems programming, in order to find areas where improvements can be made using either better tools, methods, or both.

Unfortunately, today, we do not know very much about how real systems are developed, or how they look. We believe that it is important to examine the development of real programs, and determine where the most time is spent and where errors enter into the product. It is also of interest to determine the characteristics of real-time programs: are they long or short, how complex are they, which language constructions are used, are there some language features which are never used, etc.

An investigation of this is underway in Uppsala (preliminary results are given in [EG97b]). Ideally, this investigation should be carried out across disciplines (in cooperation between real-time, software engineering, business, and social researchers) and continuously, to obtain data about the development of programming habits and methods.

Appendix A

ODL Grammar and Semantics

This appendix gives a short specification of the input languages used in our prototype co-transformer. There are three languages:

- The `.prog` language is used to describe program instances. It is defined in Section A.3.
- The `.trace` language is used to describe the transformation trace. It is defined in Section A.4.
- The `.trans` language is used to describe co-transformations. It is defined in Section A.5.

A.1 Extended BNF syntax

The grammars are written using extended BNF in order to save space and make the grammar clearer. We use notation derived from that of the ELI system's `.con` language. The following new possibilities are introduced:

- An star ($*$) indicates zero or more repetitions of the preceding grammar symbol.
- A plus ($+$) indicates one or more repetitions of the preceding grammar symbol.
- A double slash ($//$) indicates one or more repetitions of the previous grammar symbol, using the grammar symbol following as the separator in the list. The following grammar fragment defines a *List* containing one or more *Item* separated by commas.

  $$\textit{List} \quad \rightarrow \quad \textit{Item} \; // \, \texttt{,}$$

- Square brackets ([and]) enclose an optional construct. Note that using the double slash and square brackets, a list with zero or more items using separators can be constructed. The typical list enclosed in brackets and containing zero or more items look like this:

  $$\textit{List} \quad \rightarrow \quad \texttt{[} \; [\textit{Item} \; // \, \texttt{,} \;] \texttt{]}$$

A.2 Common lexicals

The following lexical tokens are used in all input languages:

Identifier A sequence of characters taken from the following set, beginning with a letter (upper or lower case). { `a...z`, `A...Z`, `0...9`, `_` }. All other character sequences will have to be escaped using quote syntax.

Atom An is defined by quoting similar to Erlang: all characters between two single quotes (') are considered a single atom. Note that the quotes are not included into the atom. This allows for atoms containing whitespace and special characters to be entered.

Integer A sequence of digits.

DontCare An underscore followed immediately by an Identifier.

String Any sequence of letters (except a quote) between two double quotes is considered as a string.

Comments Comments are introduced by a % (percentage) sign, and continue to the end of the line. This is similar to the comment style in Erlang, Prolog and LaTeX.

The following non-terminals are derived from the basic literals:

Variable	$\rightarrow$	*Identifier*
Constant	$\rightarrow$	*Integer*
	|	*Atom*
	|	*Identifier*
Filename	$\rightarrow$	*String*
VariableField	$\rightarrow$	*Identifier* . *Identifier*

Variable is used when a variable defined elsewhere is used in a grammar production. *VariableField* is used for record indexing in ordinary C/Pascal style.

A.3 Syntax for program instance files

A program instance describes a certain execution scenario for a program, with specific data. Each program instance file names the corresponding program transformation trace file (`.trace`). There is no need to explicitly name a transformation definition file, as it is identified in the transformation trace file. The implication of files to load goes in the following way: `.prog` $\rightarrow$ `.trace` $\rightarrow$ `.trans`.

The name of the trace file given should be complete, including extension.

ODLProgramInstance	$\rightarrow$	`odlprograminstance` *Identifier*
		`odlinstanceversion` *Integer*
		`trace TraceSection`
		`callgraph` *CallGraph*
		`end odlprograminstance`
TraceSection	$\rightarrow$	`include` *Filename* ;

Call graph

The structure of the call graph and the contents of each function instance is given at the same time, in order to make the file easier to read and write. Each call graph node (function instance) contains the following information:

- The *name* of the node is used to identify it, and to name the node as a child of another node. This name varies between instances of the same function.
- The *successors list* gives the names of the children of a node.
- The name of the function in the program of which this node is an instance. Several nodes can be instances of the same function.
- The *instance data* of the node, as defined below. It describes the structure of a single function instance (as a graph of basic blocks).

The function instance data is a complex structure on its own right, consisting of a list of basic blocks. Each basic blocks has the following attributes:

- The *name* of the basic block. Used to build the structure inside the function instance, and to position the in- and out-patterns of the transformations. All instances of a function have the same names for its basic blocks.
- The *data* in the basic block is given as a string. This string is copied to the output of the parser, with the quotes removed. Any kind of Erlang term (except those containing quotes) can be entered in this way. In the prototype, the Erlang term must be a tuple with a number of fields matching the number and order of fields given in the BBFormat declaration in the `.trans` file.
- The *successors* of the basic block, to define the structure inside the function.

CallGraph	→	*CallGraphNode* ∗
CallGraphNode	→	`node` `(` *Identifier* `,` *Succs* `,` *Identifier* `,` `)`
		Data
Succs	→	`[` [*Identifier* // `,`] `]`
FunctionName	→	*Identifier*
Data	→	*BasicBlock* +
BasicBlock	→	`bb` `(` *BBName* `,` *BBSuccs* `,` *BBData* `)`
BBName	→	*Identifier*
BBSuccs	→	`[` [*BBName* // `,`] `]`
BBData	→	*String*

A.4 Trace specification

The `.trace` file is used to specify the transformation trace for a certain run of a compiler on a certain program.

The `transformations` declaration gives the name of the co-transformation file to use with this trace file. The name given should include the extension (usually `.trans`).

The main part of the file gives the details of how the co-transformations defined in the co-transformations file are applied to the program. It is a trace of the transformations applied to the program, in the order they were performed by the compiler.

The parameters to the transformations are read as constants. They may be integers or symbolic atoms, or strings. Strings are copied to the output or the `tracereader` without enclosing quotes. This allows us to enter arbitrary Erlang terms into the trace without having to include a parser for Erlang.[1].

ODLTraceProgram	→	`odltraceprogram` *Identifier*
		`odltraceversion` *Integer*
		`transformations` *TransformationsSection*
		`trace` *TraceSection*
		`end` `odltraceprogram`
TransformationsSection	→	`include` *Filename* `;`
Filename	→	*String*
TraceSection	→	*TransformationCall* +
TransformationCall	→	*Identifier* `(` *TActuals* `)` `;`
TActuals	→	*TActual* +
TActual	→	*Constant*
	|	*String*

[1]This is a typical prototype patch. It can result in very strange parse errors when the Erlang file is read by the main program, and there was some error in one of the copied strings ...

A.5 Co-transformation specification language

The language used to specify the co-transformations is the largest language. The files end in `.trans`. This is the language referred to as ODL in the chapters about the co-transformer prototype.

Note that we (slightly inconsistently) write *transformation* below when we are dicsussing *co*-transformations. There is little risk for confusion, however.

An ODL program consists of a number of sections, each introduced by a special keyword. The following gives the grammar for the top level structure:

ODLTransProgram	→	`odltransprogram` *Identifier*
		`odltransversion` *Integer*
		`bbformats` *BBFormatsSection*
		`functions` *FunctionsSection*
		`transformations` *TransformationsSection*
		`end odltransprogram`

Global declarations

The format of the data in the basic blocks (see Section 5.2 (page 28)) is given in the BBFormat section. In order to allow for the future inclusion of other types of basic blocks, we introduce the default basic block type by the a special keyword `default`.

BBFormatsSection	→	*BBFormats*
BBFormats	→	`default = (` *BBFieldList* `) ;`
BBFieldList	→	*BBField* //`,`
BBField	→	*Identifier*

The filename given in the functions section points to the file containing the definitions of the functions used to implement the transformations. The name includes the extension of the filename (`.erl`, as we use Erlang as our basic language).

FunctionsSection	→	`include` *Filename* `;`

Transformation declarations

The transformations section of the main part of an ODL program. It consists ofa list of transformations. Each transformation is introduced by the keyword `transformation`.

Each transformation has a number of arguments. To distinguish the types of arguments, a C-style type syntax is used. `var` introduces a simple variable (mostly used to name basic blocks), `section` a section determiner (which gives the function a certain transformation is applied to), and `tt` a name translation table (to use with compound nodes).

TransformationsSection	→	*Transformation* +
Transformation	→	`transformation` *TranformationName* `(` *TransformationFormals* `)`
		`inpattern` *InPattern*
		`outpattern` *OutPattern*
		`transfers` *FunctionCalls*
		`end transformation`
TransformationName	→	*Identifier*
TransformationFormals	→	*TFormal* //`,`
TFormal	→	`var` *Identifier*
	|	`tt` *Identifier*
	|	`section` *Identifier*

In-pattern

The in-pattern consists of a list of in-pattern fragments. A fragment consists of a list of node declarations (simple or compound).

The fragment header contains four pieces of information: the name given to the fragment (to be used in the out-pattern), the program section in which the fragment is positioned, the name of the variable containing the name of the first node in the fragment (the variable must defined in the transformation parameters), and the end node of the fragment (also a parameter variable). The start and end nodes are included in the fragment. If they are the same, the fragment represents a single node in the flow graph.

Simple nodes have two arguments: the a variable containing the name of the basic block to match in the program instance, and the name of the data variable to receive the data from the basic block. A wildcard can be used for the data variable if the data will not be needed later.

The compound nodes are more complex, and they are defined using four arguments:

- The name of the node (to be used in the out-pattern to specify compound nodes with the same structure).
- The start node of the compound node.
- The end node of the compound node. If the start and end nodes are the same, the compound node contains only one node.
- The data variable for the compound node. This might be a wildcard. The data variable will contain the data values for all basic blocks included in the compound node. This is called a *compound data variable*.

```
InPattern            →  InPatFragment +
InPatFragment        →  fragment Identifier ( Variable , Variable , Variable )
                        InPatNode +
                        end fragment
InPatNode            →  InPatSimpleNode
                     |  InPatCompoundNode
InPatSimpleNode      →  node ( Variable , InPatDataVariable ) ;
InPatCompoundNode    →  cnode ( Identifier , Variable , Variable , InPatCompoundDataVariable ) ;
InPatDataVariable    →  DontCare
                     |  Variable
```

Out-pattern

Just like the in-pattern, the out-pattern is a list of graph fragments. The out-pattern fragment with the same name as an in-pattern fragment will replace this fragment in the program instance.

The fragment header specifies which fragment in the in-pattern is to be replaced by this graph fragment. The first argument gives the fragment name. The fragment is by necessity located in the same function in the program as the corresponding in-pattern fragment, and there is no need to give a function name.

The first two arguments determines the start and end nodes of the fragment. All edges that were incident on the start node of the corresponding in-pattern fragment are changed to be incident on the start node of the out-pattern fragment. Similarly, all edges leaving the in-pattern end node are changed to leave the out-pattern end node. The arguments are given the special non-terminal *OutPatNodeNameUse* and may be simple variables naming single nodes or logical names for compound nodes (see below for how out-pattern compound nodes are declared).

A fragment consists of a list of node declarations. Simple and compound nodes are declared, just like the in-pattern.

A out-pattern fragment may be empty (as indicated by the $*$ notation in the grammar). In the declaration of an empty fragment, don't-cares are used instead of the names of the start and end nodes, and the fragment body is left empty. When replacing with an empty fragment, all nodes preceding the old start node node are connected to all sucessors to the end node. The empty fragment works like glue, gluing the nodes before and those after together. In the usual case, with one successor or predecessor, the result is very intuitive. With several predecessors and successors, the number of edges is $\#pred \cdot \#succ$, and the result is probably not the expected.

OutPattern	$\rightarrow$	*OutPatFragment* +
OutPatFragment	$\rightarrow$	`fragment` *Identifier* `(` *OutPatNodeNameUse* `,` *OutPatNodeNameUse* `)`
		OutPatNode $*$
		`end fragment`
OutPatNodeNameUse	$\rightarrow$	*OutPatNodeName*
	|	*DontCare*
OutPatNodeName	$\rightarrow$	*Variable*
	|	*CompoundNodeLogicalName*
OutPatNode	$\rightarrow$	*OutPatSimpleNode*
	|	*OutPatCompoundNode*

The out-pattern nodes are rather more complicated than the in-pattern nodes. The simple out-pattern nodes contain a name, a data variable, and a list of successors. The name is a parameters variable; the value of the variable will be used to name the node in the output. No data don't-cares may be given (all nodes in the out-pattern are present in the output, and we cannot have unspecified data in a nodes).

If the data variable is preceded by an equals sign (`=`), it is supposed to appear in the in-pattern and the data is copied from the in-pattern to the out-pattern. If not, a new variable is declared, which is supposed to be given values from the function calls.[2]

The successor list uses the names of the variables naming simple nodes, and the logical names given to compound nodes (see below).

To allow for more flexible treatment of loops from a node to itself, a special syntax is used: if the name of the node itself appears in the successor list, within parentheses, a loop is created in the output only if it existed in the input. If the name is given without parentheses, a loop is always created, and if the name does not appear, any loop which existed would be removed (this is only of interest for compound nodes which can happen to pick up loops from the input program).

The following fields are used to declare a compound node:

- The first field gives the logical name for the compound node. This name is used in the successor lists in the out-pattern. It might be used before it is declared (otherwise some graphs could not be expressed).

  This is needed as we cannot know the name of the top node inside the compound node without peeking into the name translation table for the node (which would be unduly complicated).

- The second field gives the name of a compound node in the in-pattern. The compound node generated is given the same internal structure as this node.

- The third field gives the name of a name translation table which gives names to all the nodes inside the compound node. It can also be the special code `id`, in which case the names are copied from the in-pattern compound node. Not more than one compound node derived from the same in-pattern compound node may use the `id` code (we do not want duplicate node names).

- The fourth field gives the data variable used for the compound node. The same syntax may be used as for simple nodes to copy data from an in-pattern compound node.

[2]This was not the original design: we originally intended the parser to automatically determine whether a variable had been previously declared. Unfortunately, it proved difficult to implement those "pure" pattern matching sematics using the tools and time available to us. So we modified the grammar to save time.

- The fifth field gives the successors to the compound node. The same rules apply as for successors to simple nodes.

OutPatSimpleNode	→	`node` `(` *Variable* `,` *VariableUseDef* `,` *OutPatNodeSuccs* `)` `;`
OutPatCompoundNode	→	`cnode` `(` *Identifier* `,` *Variable* `,` *CompoundNameTranslation* `,` *VariableUseDef* `,` *OutPatNodeSuccs* `)` `;`
CompoundNameTranslation	→	`id`
	|	*Variable*
VariableUseDef	→	`=` *Variable*
	|	*Variable*
OutPatNodeSuccs	→	`[` [*OutPatNodeSuccessor* //`,`]`]`
OutPatNodeSuccessor	→	*OutPatNodeName*
	|	`(` *OutPatNodeName* `)`

Function calls

The function calls section is a list of function invocations, where out-pattern data variables (or rather fields of them) are assigned values.

Simple function calls assign values to a single field of a single named node. The destination is a data variable declared in the out-pattern. The arguments to the function call may be variables (from the transformation arguments), data variables from the in-pattern, or constants.

A special form is needed to handle compound data. As a compound data value really represents a set of simple data variables, an iterator is used to apply the function to each node is the compound node. The output from the function is assigned to the corresponding node in the destination compound node. The iterator is written like this:

```
A.a = forall( X:B | f( .. X.a .. X.b .. ))
```

The intended meaning is that the variable X is successively instantiated with the data for each node in the in-pattern compound data variable B. The values of various fields in X can then be retrieved in the arguments to the function f by using ordinary field subscript syntax. The result computed for each value of X is assigned to the field `a` of the data of the corresponding node in the out-pattern compound data variable A.

There is a shortcut assignment syntax for copying the content of a single data variable field.

FunctionCalls	→	*FunctionCall* ∗
FunctionCall	→	*SimpleFunctionCall*
	|	*CompoundFunctionCall*
SimpleFunctionCall	→	*VariableField* `=` *FunctionName* `(` *FunctionParams* `)` `;`
	|	*VariableField* `=` *VariableField* `;`
CompoundFunctionCall	→	*VariableField* `=` `forall` `(` *Identifier* `:` *Variable* `|` *FunctionName* `(` *FunctionParams* `)` `)` `;`
	|	*VariableField* `=` *VariableField* `;`
FunctionParams	→	[*FunctionParam* //`,`]
FunctionParam	→	*VariableField*
	|	*Variable*
	|	*Constant*

Appendix B

Transformation Implementations in ODL

This appendix contains ODL code for the transformations whose code was not given in Chapter 6.

Block Merge

The ODL code for block merge is given in Transformation B.1 (page 78).

Transformation B.1 Block merging transformation.

```
%-----------------------------------------------------------
%   Transformation BlockMerge
%-----------------------------------------------------------
% ARGUMENTS:
%   S: the section in which to operate
%   BlockA: the name of the first block to merge
%   BlockB: the name of the second block
%   NewName: the name of the merged block
%-----------------------------------------------------------
  transformation BlockMerge
    (section S,var BlockA, var BlockB, var NewName)

    inpattern
      fragment one(S,BlockA,BlockB)
        node(BlockA,BlockAdata);
        node(BlockB,_BlockBdata);   % ignore data
      end fragment

    outpattern
      fragment one(NewName,NewName)
        node(NewName,=BlockAdata,[]);
      end fragment

    transfers
      %% no transfers needed
  end transformation
```

Dead code elimination

The ODL code for the elimination of a dead basic block is given in Transformation B.2 (page 79). Note the use of don't care operators in the data in the in-pattern and the empty out-pattern fragment.

Transformation B.2 Eliminating a dead basic block.

```
%-------------------------------------------------------------
% ARGUMENTS:
%   S: the section in which to operate
%   Block: the name of the block to remove
%-------------------------------------------------------------
  transformation DeadCode (section S,var Block)
    inpattern
      fragment one(S,Block,Block)
        node(Block,_);
      end fragment

    outpattern
      fragment one(_,_)  %% empty fragment
      end fragment

    transfers
      %% no transfers needed
  end transformation
```

Loop collapse

The ODL code for loop collapsing is shown in Transformation B.3 (page 80). Note that the lack of in-pattern structure declarations makes it hard to see what the operation does from the ODL code. This is a clear readability problem with the ODL.

Loop fusion

We have made the following assumptions/limitations implementing the transformation:

- The loop bodied to be fused have no code between them. This means that the priming code for the second loop has to be removed before applying this transformation.
- The last block of each loop body has only two exits: loop to the head of the loop, or exit the loop. No three-way exits are allowed.
- There are no loops inside the loop bodied being fused (this is a general limitation, see Section 6.3.2 (page 48)).

Transformation B.4 (page 81) shows the ODL code for loop fusion.

Loop interchange

The ODL code is given in Transformation B.5 (page 82).

The code is given in Transformation B.6 (page 83). The `capsub` operation used simply makes sure a given value is less than or equal to a ceiling value (minus another value, as we cannot compose functions in ODL; see Section 6.3.8 (page 52)).

Loop unswitching

The transformation code is given in Transformation B.7 (page 84).

Implementation notes: in the out-pattern, we make the one loop into two, and introduce a basic block before and one after the loops in order to hold them together. The created loops both contain a copy of the part of the original loop before and after the branching.

Transformation B.3 Loop collapsing.

```
%-------------------------------------------------------------
% ARGUMENTS:
%   S: the section in which to operate
%   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
      fragment loops(S,LO1begin,LO2end)
        cnode(lo1,LO1begin,LO1end,_);
        cnode(lo2,LO2begin,LO2end,_);
        cnode(li,LIbegin,LIend,LI1Data);
      end fragment

      fragment reprLO(S,LO,LO)
        node(LO,LOData);
      end fragment

      fragment reprLI(S,LI,LI)
        node(LI,LIData);
      end fragment

    outpattern
      fragment loops(body,body)
        cnode(body,li,id,LIOut,[]);
      end fragment

      fragment reprLO(_,_)
      end fragment

      fragment reprLI(LN,LN)
        node(LN,LNOut,[]);
      end fragment

    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
```

Transformation B.4 Loop fusion.

```
%-------------------------------------------------------------
% ARGUMENTS:
%   S: the section in which to operate
%   Begin1: the beginning of loop 1
%   End1: the end.
%   Begin2: the beginning of loop 2
%   End2: the end.
%   LoopOld1: the name of the first loop.
%   LoopOld2: the name of the second loop.
%   LoopNew: the name of the fused loop.
%-------------------------------------------------------------
  transformation LoopFusion
    (section S, var Begin1, var End1, var Begin2,var End2,
      var LoopOld1, var LoopOld2, var LoopNew)

  inpattern
    fragment loop(S,Begin1,End2)
      cnode(part1,Begin1,End1,Part1Data);
      cnode(part2,Begin2,End2,Part2Data);
    end fragment

    fragment repr1(S,LoopOld1,LoopOld1)
      node(LoopOld1,LoopOld1Data);
    end fragment

    fragment repr2(S,LoopOld2,LoopOld2)
      node(LoopOld2,_LoopOld2Data);
    end fragment

  outpattern
    fragment loop(loop1,loop2)
      cnode(loop1,part1,id,Part1Out,[loop2]);
      cnode(loop2,part2,id,Part2Out,[loop1]);
    end fragment

    fragment repr1(LoopNew,LoopNew)
      node(LoopNew,LoopNewOut,[LoopNew]);
    end fragment

    fragment repr2(_,_)
    end fragment

  transfers

    Part1Out.scope = forall(X1:Part1Data | copy(LoopNew));
    Part2Out.scope = forall(X2:Part2Data | copy(LoopNew));
    LoopNewOut.scope = LoopOld1Data.scope;

    Part1Out.execcount = Part1Data.execcount;
    Part2Out.execcount = Part2Data.execcount;
    LoopNewOut.execcount = LoopOld1Data.execcount;

    Part1Out.iterations = Part1Data.iterations;
    Part2Out.iterations = Part2Data.iterations;
    LoopNewOut.iterations = LoopOld1Data.iterations;

  end transformation
```

Transformation B.5 Loop interchange.

```
%-----------------------------------------------------------
% ARGUMENTS:
%   S: the section in which to operate
%   LoopOuter: the name of the outer loop
%   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.
%   LoopInner: the name of the inner loop
%   LIbegin: the beginning of the inner loop body.
%   LIend: the end.
%-----------------------------------------------------------
  transformation LoopInterchange
    (section S, var LoopOuter, var LO1begin, var LO1end, var LO2begin, var LO2end,
                var LoopInner, var LIbegin, var LIend)
    inpattern
      fragment bodies(S,LO1begin,LO2end)
        cnode(lo1,LO1begin,LO1end,LO1Data);
        cnode(lo2,LO2begin,LO2end,LO2Data);
        cnode(li,LIbegin,LIend,LIData);
      end fragment

      fragment outer(S,LoopOuter,LoopOuter)
        node(LoopOuter,LoopOuterData);
      end fragment
      fragment inner(S,LoopInner,LoopInner)
        node(LoopInner,LoopInnerData);
      end fragment

    outpattern
      fragment bodies(lo1body,lo2body)
        cnode(lo1body,lo1,id,LO1Out,[libody]);
        cnode(libody,li,id,LIOut,[lo2body,(libody)]);
        cnode(lo2body,lo2,id,LO2Out,[]);
      end fragment

     fragment outer(LoopOuter,LoopOuter)
        node(LoopOuter,LoopOuterOut,[]);
      end fragment
      fragment inner(LoopInner,LoopInner)
        node(LoopInner,LoopInnerOut,[]);
      end fragment

    transfers
      %% handle the iterations of the loop representatives
      LoopOuterOut.iterations = LoopInnerData.iterations;
      LoopOuterOut.scope = LoopOuterData.scope;
      LoopOuterOut.execcount = LoopOuterData.execcount;

      LoopInnerOut.iterations = LoopOuterData.iterations;
      LoopInnerOut.scope = LoopInnerData.scope;
      LoopInnerOut.execcount = fraction(LoopInnerData.execcount,
                                  LoopOuterData.iterations,LoopInnerData.iterations);

      %% update the execcounts of the loop bodies
      LIOut.execcount = forall(X1:LIData |
        fraction(X1.execcount,LoopInnerData.iterations,LoopOuterData.iterations));
      LO1Out.execcount = forall(X2:LO1Data |
        fraction(X2.execcount,LoopOuterData.iterations,LoopInnerData.iterations));
      LO2Out.execcount = forall(X3:LO2Data |
        fraction(X3.execcount,LoopOuterData.iterations,LoopInnerData.iterations));

      %% copy the scopes and iterations
      LIOut.scope = LIData.scope;
      LO1Out.scope = LO1Data.scope;
      LO2Out.scope = LO2Data.scope;
      LIOut.iterations = LIData.iterations;
      LO1Out.iterations = LO1Data.iterations;
      LO2Out.iterations = LO2Data.iterations;
  end transformation
```

Transformation B.6 Loop peeling to before loop.

```
%-------------------------------------------------------------
% ARGUMENTS:
%   S: the section in which to operate
%   LoopBegin: the beginning of the loop body
%   LoopEnd: the end of the loop body
%   Loop: the scope representative of the loop
%   NewNames: the cncntt renaming table for the copied loop body
%-------------------------------------------------------------
  transformation LoopPeel
    (section S, var LoopBegin, var LoopEnd, var Loop, tt NewNames)

  inpattern
    fragment loop(S,LoopBegin,LoopEnd)
      cnode(loopbody,LoopBegin,LoopEnd,LoopBodyData);
    end fragment

    fragment repr(S,Loop,Loop)
      node(Loop,LoopData);
    end fragment

  outpattern
    fragment loop(looppeeled,loopmain)
      cnode(looppeeled,loopbody,NewNames,NewOut,[loopmain]);
      cnode(loopmain,loopbody,id,LoopBodyOut,[(loopmain)]);
    end fragment

    fragment repr(Loop,Loop)
      node(Loop,LoopDataOut,[]);
    end fragment

  transfers
    %% update the loop iteration count
    LoopDataOut.iterations = sub(LoopData.iterations,1);
    LoopDataOut.scope = LoopData.scope;
    LoopDataOut.execcount = LoopData.execcount;

    %% update the execcounts of the loop body
    LoopBodyOut.scope = LoopBodyData.scope;
    LoopBodyOut.iterations = LoopBodyData.iterations;
    LoopBodyOut.execcount = forall( X1:LoopBodyData |
                              capsub(X1.execcount,LoopData.iterations,1));

    %% update the peeled code (only exec once)
    NewOut.scope = forall( X2:LoopBodyData | copy(LoopData.scope));
    NewOut.iterations = forall( X3:LoopBodyData | copy(1));
    NewOut.execcount = forall( X4:LoopBodyData | copy(1));

  end transformation
```

Transformation B.7 Loop unswitching transformation.

```
%---------------------------------------------------
% ARGUMENTS:
%   S: the section in which to operate
%   IfBegin: first in the blocks leading up to the
%            split. (and first block in loop body)
%   IfEnd: end.
%   ABegin: first block in A.
%   AEnd: end.
%   BBegin: first block in B.
%   BEnd: end.
%   TailBegin: the first block after the join.
%   TailEnd: the last.
%   Loop: the scope representative of the loop
%   If: name for new block containing just the branch.
%   Join: name for new block joining after the new loops.
%   BIfNames: new names for If-cnode in loop B.
%   BTailNames: new names for Tail-cnode in loop B.
%   ALoop: name for loop containing A after unswitch.
%   BLoop: name for loop containing B after unswitch.
%----------------------------------------------------
  transformation LoopUnswitching
    (section S, var IfBegin, var IfEnd,
     var ABegin, var AEnd,
     var BBegin, var BEnd,
     var TailBegin, var TailEnd,
     var Loop, var If, var Join,
     tt BIfNames, tt BTailNames,
     var ALoop, var BLoop)

  inpattern
    fragment body(S,IfBegin,TailEnd)
      cnode(if, IfBegin,IfEnd,IfData);
      cnode(a, ABegin,AEnd,AData);
      cnode(b, BBegin,BEnd,BData);
      cnode(tail,TailBegin,TailEnd,TailData);
    end fragment

    fragment repr(S,Loop,Loop)
      node(Loop,LoopData);
    end fragment

  outpattern
    fragment body(If,Join)
      node(If,IfOut,[loopa1,loopb1]);
      cnode(loopa1,if,id,          AOut1,[loopa2]);
      cnode(loopa2,a,id,           AOut2,[loopa3]);
      cnode(loopa3,tail,id,        AOut3,[loopa1,Join]);
      cnode(loopb1,if,BIfNames,    BOut1,[loopb2]);
      cnode(loopb2,b,id,           BOut2, [loopb3]);
      cnode(loopb3,tail,BTailNames,BOut3, [loopb1,Join]);
      node(Join,JoinOut,[]);
    end fragment

    fragment repr(ALoop,BLoop)
      node(ALoop,ALOut,[BLoop]);
      node(BLoop,BLOut,[]);
    end fragment

  transfers
    %% the loop representatives
    ALOut.scope = LoopData.scope;
    BLOut.scope = LoopData.scope;
    ALOut.execcount = LoopData.execcount;
    BLOut.execcount = LoopData.execcount;
    ALOut.iterations = LoopData.iterations;
    BLOut.iterations = LoopData.iterations;

    %% the new blocks
    IfOut.scope =  LoopData.scope;
    IfOut.execcount = LoopData.execcount;
    IfOut.iterations = copy(1);
    JoinOut.scope =  LoopData.scope;
    JoinOut.execcount = LoopData.execcount;
    JoinOut.iterations = copy(1);

    %% the B loop
    AOut1.scope = forall( A1:IfData | copy(ALoop));
    AOut2.scope = forall( A2:AData | copy(ALoop));
    AOut3.scope = forall( A3:TailData | copy(ALoop));
    AOut1.execcount = IfData.execcount;
    AOut2.execcount = AData.execcount;
    AOut3.execcount = TailData.execcount;
    AOut1.iterations = IfData.iterations;
    AOut2.iterations = AData.iterations;
    AOut3.iterations = TailData.iterations;

    %% the B loop
    BOut1.scope = forall( B1:IfData | copy(BLoop));
    BOut2.scope = forall( B2:BData | copy(BLoop));
    BOut3.scope = forall( B3:TailData | copy(BLoop));
    BOut1.execcount = IfData.execcount;
    BOut2.execcount = BData.execcount;
    BOut3.execcount = TailData.execcount;
    BOut1.iterations = IfData.iterations;
    BOut2.iterations = BData.iterations;
    BOut3.iterations = TailData.iterations;
  end transformation
```

Appendix C

A Complete Transformation Example

Using the transformations defined in Chapter 6 and Appendix B, we co-transformed a small example program containing three different function instances (of three functions).

In this appendix, we present the original program, the transformation trace, and the final transformed program in order to show what can be accomplished using our prototype.

Program instance before transformations The listing in Figure C.1 contains a listing of the `.prog` file before being transformed. The `node` declarations declare nodes in the call graph. There are three nodes: `c1`, which is an instance of the function `cesar`, `i1` which is an instance of the function `iffy`, and `h1` which is an instance of the function `hamlet`.

The `bb` declarations declare basic blocks within the function instances. The first argument gives the name of the block (which would be the same in different instances of the same function), the second argument is a list of the successors, and the last argument contains the data. The data is an Erlang list of three elements: the execution count, the loop scope, and the iteration count for the basic block. The list is quoted as it is not parsed by the input reader.

Note the reference to the `.trace` file in the `trace` declaration.

Transformation trace The transformation trace is shown in Figure C.2. Note that the first argument to all transformations is a *function* name, and that the transformation will be applied to *all* instances of that function.

Note the reference to the `.trans` file in the `transformations` declaration.

Transformation definitions As stated above, the transformations used have already been listed elsewhere in this thesis. However, we have not shown a header for an ODL program yet, and this is given in Figure C.3. Note the reference to the Erlang file containing the function used in the transformation definitions, and the declaration of the contents of the basic block data.

Erlang file An Erlang file contains the functions used in the transformations in the `.trans` file. It is given in Figure C.4. The `capsub` function is just a composition of the `cap` and the `sub` operations. This is needed as we cannot compose functions in the transformation definitions.

Final transformed program Figure C.5 shows the state of the program after it has been co-transformed. Note that the automatically generated output is a little less organized than our input files.

Figure C.1 The program instance before co-transformation (file `loops.prog`)

```
odlprograminstance Test1

odlinstanceversion 1

trace
  include "loops.trace";

callgraph

  node(c1,[],cesar)
    bb(block1,[block2],"[1,cesar,1]")
    bb(block2,[block3],"[10,loop1,1]")
    bb(block3,[block2,block4],"[10,loop1,1]")
    bb(block4,[block5],"[1,cesar,1]")
    bb(block5,[block6,block8],"[6,loop3,1]")
    bb(block6,[block7],"[3,loop3,1]")
    bb(block8,[block9,block10],"[3,loop3,1]")
    bb(block9,[block11],"[2,loop3,1]")
    bb(block10,[block11],"[2,loop3,1]")
    bb(block11,[block7],"[3,loop3,1]")
    bb(block7,[block5,block12],"[6,loop3,1]")
    bb(block12,[block13],"[20,loop2,1]")
    bb(block13,[block14,block5],"[20,loop2,1]")
    bb(block14,[],"[1,cesar,1]")

    %% loop representatives
    bb(cesar,[],"[1,main,1]")
    bb(loop1,[],"[1,cesar,10]")
    bb(loop2,[],"[1,cesar,20]")
    bb(loop3,[],"[20,loop2,6]")

  node(i1,[],iffy)
    bb(if1,[if2],"[1,iffy,1]")
    bb(if2,[if3,if4],"[100,loopbig,1]")
    bb(if3,[if5],"[100,loopbig,1]")
    bb(if5,[if7],"[100,loopbig,1]")
    bb(if7,[if9,if10],"[16,loop2,1]")
    bb(if9,[if11],"[12,loop2,1]")
    bb(if10,[if11],"[4,loop2,1]")
    bb(if11,[if12,if7],"[16,loop2,1]")
    bb(if12,[if13],"[100,loopbig,1]")
    bb(if13,[if2,if14],"[100,loopbig,1]")
    bb(if14,[],"[1,iffy,1]")
    bb(if4,[if6],"[0,iffy,1]")
    bb(if6,[if8],"[0,iffy,1]")
    bb(if8,[if12],"[0,iffy,1]")

    %% loop representatives
    bb(iffy,[],"[1,main,1]")
    bb(loop2,[],"[100,loopbig,16]")
    bb(loopbig,[],"[1,iffy,100]")

  node(h1,[],hamlet)
    bb(h1,[h2],   "[1,hamlet,1]")
    bb(h2,[h2,h3],"[7,loopb2,1]")
    bb(h3,[h3,h4],"[10,loopb3,1]")
    bb(h4,[h4,h5],"[10,loopb4,1]")
    bb(h5,[h6],   "[10,loopb56,1]")
    bb(h6,[h5,h7],"[10,loopb56,1]")
    bb(h7,[h8],     "[1,hamlet,1]")
    bb(h8,[h9,h13],"[30,loopu,1]")
    bb(h9,[h10,h11],"[30,loopu,1]")
    bb(h10,[h12],"[20,loopu,1]")
    bb(h11,[h12],"[19,loopu,1]")
    bb(h12,[h15],"[30,loopu,1]")
    bb(h13,[h14],"[30,loopu,1]")
    bb(h14,[h15],"[30,loopu,1]")
    bb(h15,[h16,h17],"[30,loopu,1]")
    bb(h16,[h18],"[25,loopu,1]")
    bb(h17,[h18],"[20,loopu,1]")
    bb(h18,[h19,h8],"[30,loopu,1]")
    bb(h19,[],"[1,hamlet,1]")

    %% loop representatives
    bb(loopb2,[],   "[1,hamlet,7]")
    bb(loopb3,[],   "[1,hamlet,10]")
    bb(loopb4,[],   "[1,hamlet,10]")
    bb(loopb56,[],  "[1,hamlet,10]")
    bb(loopu,[],    "[1,hamlet,30]")

end odlprograminstance
```

Figure C.2 Transformation trace (file `loops.trace`).

```
odltraceprogram Test1
odltraceversion 0

transformations
  include "loops.trans" ;

trace

  IfOpt(iffy,if2,if3,if11,if4,if8,if12);

  DeadCode(cesar,block13);
    %% create preheader to enable loop collapsing
  CreatePreheader(cesar,block5,block7,phloop3,loop3);
  LoopCollapse(cesar,phloop3,phloop3,block12,block12,loop2,block5,block7,loop3,loop57);
  CreatePreheader(cesar,block5,block7,phloop57,loop57);

    %% peel the nest in cesar
  LoopPeel(cesar,block5,block7,loop57,
           %% NOTE: quoted list is a name translation table
           "[{block5,new5},{block6,new6},{block7,new7},
            {block8,new8},{block9,new9},{block10,new10},{block11,new11}]");

    %% rebuilding hamlet
  LoopDistribution(hamlet,h5,h5,h6,h6,loopb56,loopb5,loopb6);
  LoopFusion(hamlet,h4,h4,h5,h5,loopb4,loopb5,loopb45);
  LoopFusion(hamlet,h3,h3,h4,h5,loopb3,loopb45,loopb345);
  BlockMerge(hamlet,h4,h5,h45);
  LoopUnswitching(hamlet,h8,h8,h9,h12,h13,h14,h15,h18,loopu,
                  newif,newjoin,"[{h8,new8}]",
                  "[{h15,new15},{h16,new16},{h17,new17},{h18,new18}]",
                  loopa,loopb);
  TwoPieceIf(hamlet,h9,h10,h10,h11,h11,h12,h9_lo);

    %% interchanging loops!
  LoopInterchange(iffy,loopbig,if2,if5,if12,if13,loop2,if7,if11);

end odltraceprogram
```

Figure C.3 The header of the transformation file (file `loops.trans`).

```
odltransprogram Loops
odltransversion 0

bbformats
  default = ( execcount, scope, iterations ) ;

functions
  include "fix.erl" ;

transformations

  %% see elsewhere

end odltransprogram
```

Figure C.4 The Erlang file (file `fix.erl`).

```
%%% File    : fix.erl
%%% Author  : Jakob Engblom <jakob@Zapata.DoCS.UU.SE>
%%% Purpose : Test function module for cotransformer
%%% Created : 17 Jul 1997 by Jakob Engblom <jakob@Zapata.DoCS.UU.SE>

-module(fix).
-author('jakob@Zapata.DoCS.UU.SE').
-export([
         copy/1,
 mul/2,
 sub/2,
 add/2,
 cap/2,
 capsub/3,
 fraction/3
]).

copy(X) ->
     X.

mul(X,Y) ->
    X*Y.

sub(X,Y) ->
    X-Y.

add(X,Y) ->
    X+Y.

cap(X,Cap) when (X>Cap) ->
    Cap;
cap(X,_) ->
    X.

capsub(X,Cap,Sub) ->
    %% simple workaround for lack of composition in funcalls
    cap(X,Cap-Sub).

fraction(OldExec,OldIter,NewIter) ->
    OldExec * NewIter / OldIter.
```

Figure C.5 The program instance after co-transformation (file `loops.prog.coxed`)

```
odlprograminstance Converted
odlinstanceversion 1

trace
  include "";

callgraph

  node(c1,[],cesar)
    bb(block1,[block2],"[1,cesar,1]")
    bb(block10,[block11],"[40,loop57,1]")
    bb(block11,[block7],"[60,loop57,1]")
    bb(block14,[],"[1,cesar,1]")
    bb(block2,[block3],"[10,loop1,1]")
    bb(block3,[block2,block4],"[10,loop1,1]")
    bb(block4,[phloop57],"[1,cesar,1]")
    bb(block5,[block6,block8],"[119,loop57,1]")
    bb(block6,[block7],"[60,loop57,1]")
    bb(block7,[block14,block5],"[119,loop57,1]")
    bb(block8,[block10,block9],"[60,loop57,1]")
    bb(block9,[block11],"[40,loop57,1]")
    bb(cesar,[],"[1,main,1]")
    bb(loop1,[],"[1,cesar,10]")
    bb(loop57,[],"[1,cesar,119]")
    bb(new10,[new11],"[0.333333,cesar,1]")
    bb(new11,[new7],"[0.500000,cesar,1]")
    bb(new5,[new6,new8],"[1.00000,cesar,1]")
    bb(new6,[new7],"[0.500000,cesar,1]")
    bb(new7,[block5],"[1.00000,cesar,1]")
    bb(new8,[new10,new9],"[0.500000,cesar,1]")
    bb(new9,[new11],"[0.333333,cesar,1]")
    bb(phloop57,[new5],"[1,cesar,1]")

  node(h1,[],hamlet)
    bb(h1,[h2],"[1,hamlet,1]")
    bb(h10,[h12],"[20,loopa,1]")
    bb(h11,[h12],"[19,loopa,1]")
    bb(h12,[h15],"[30,loopa,1]")
    bb(h13,[h14],"[30,loopb,1]")
    bb(h14,[new15],"[30,loopb,1]")
    bb(h15,[h16,h17],"[30,loopa,1]")
    bb(h16,[h18],"[25,loopa,1]")
    bb(h17,[h18],"[20,loopa,1]")
    bb(h18,[h8,newjoin],"[30,loopa,1]")
    bb(h19,[],"[1,hamlet,1]")
    bb(h2,[h2,h3],"[7,loopb2,1]")
    bb(h3,[h45],"[10,loopb345,1]")
    bb(h45,[h3,h6],"[10,loopb345,1]")
    bb(h6,[h6,h7],"[10,loopb6,1]")
    bb(h7,[newif],"[1,hamlet,1]")
    bb(h8,[h9],"[30,loopa,1]")
    bb(h9,[h10,h11],"[30,loopa,1]")
    bb(loopa,[loopb],"[1,hamlet,30]")
    bb(loopb,[],"[1,hamlet,30]")
    bb(loopb2,[],"[1,hamlet,7]")
    bb(loopb345,[loopb345],"[1,hamlet,10]")
    bb(loopb6,[],"[1,hamlet,10]")
    bb(new15,[new16,new17],"[30,loopb,1]")
    bb(new16,[new18],"[25,loopb,1]")
    bb(new17,[new18],"[20,loopb,1]")
    bb(new18,[new8,newjoin],"[30,loopb,1]")
    bb(new8,[h13],"[30,loopb,1]")
    bb(newif,[h8,new8],"[1,hamlet,30]")
    bb(newjoin,[h19],"[1,hamlet,30]")

  node(i1,[],iffy)
    bb(if1,[if2],"[1,iffy,1]")
    bb(if10,[if11],"[25.0000,loop2,1]")
    bb(if11,[if12,if7],"[100.000,loop2,1]")
    bb(if12,[if13],"[16.0000,loopbig,1]")
    bb(if13,[if14],"[16.0000,loopbig,1]")
    bb(if14,[],"[1,iffy,1]")
    bb(if2,[if3],"[16.0000,loopbig,1]")
    bb(if3,[if5],"[16.0000,loopbig,1]")
    bb(if5,[if7],"[16.0000,loopbig,1]")
    bb(if7,[if10,if9],"[100.000,loop2,1]")
    bb(if9,[if11],"[75.0000,loop2,1]")
    bb(iffy,[],"[1,main,1]")
    bb(loop2,[],"[16.0000,loopbig,100]")
    bb(loopbig,[],"[1,iffy,16]")

end odlprograminstance
```

Bibliography

[Alt96a] Peter Altenbernd. On the False Path Problem in Hard Real-Time Programs. In *Proceedings 8th Euromicro Workshop on Real Time Systems*, 1996.

[Alt96b] Peter Altenbernd. *Timing Analysis, Scheduling, and Allocation of Periodic Hard Real-Time Tasks.* PhD thesis, Universität-GH Paderborn, Germany, 1996.

[Alt97] Peter Altenbernd. CHaRy: The C-Lab Hard Real-Time System to Support Mechatronical Design. In *Proceedings IEEE International Symposium and Workshop on Systems Engineering of Computer Based Systems.* IEEE Computer Society Press, 1997.

[AMWH94] Robert Arnold, Frank Müller, David Whalley, and Marion Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, December 1994. Available via WWW from `http://www.informatik.hu-berlin.de/~mueller/publications.html`

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986. Generally known as the "Dragon Book".

[AT96] Ali-Reza Adl-Tabatabai. *Source-Level Debugging of Global Optimized Code.* PhD thesis, School of Computer Science, Carnegie Mellon University, June 1996. Available via ftp from `reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-133.ps`, and as technical report CMU-CS-96-133.

[ATG92] Ali-Reza Adl-Tabatabai and Thomas Gross. Evicted variables and the interaction of global register allocation and symbolic debugging. Technical Report CMU/CS-92-202, Carnegie Mellon University, School of Computer Science, October 1992.

[ATG94] Ali-Reza Adl-Tabatabai and Thomas Gross. Symbolic debugging of globally optimized code: Data value problems and their solutions. Technical Report CMU/CS-94-105, Carnegie Mellon University, School of Computer Science, January 1994.

[ATG96] Ali-Reza Adl-Tabatabai and Thomas Gross. Source-level debugging of scalar optimized code. In *Proceedings of ACM SIGPLAN'96 Conf. on Prog. Language Design and Implementation*, pages 33–43. ACM, May 1996.

[AWVW96] Joe Armstrong, Mike Williams, Robert Virding, and Claes Wikström. *Concurrent Programming in Erlang.* Prentice-Hall, 2 edition, 1996.

[BGS90] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. Technical report, University of California, Computer Science Division, Berkeley, California 94730, 1990.

[Bla94] R. J. Blainey. Instruction scheduling in the TOBEY compiler. *IBM Journal of Research and Development*, 38(5):577–593, September 1994.

[Bör95] Hans Börjesson. Incorporating worst-case execution time in a commercial c-compiler. Master's thesis, Department of Computer Systems, Uppsala University, December 1995. DoCS MSc Thesis 95/69.

[BW97] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 1997.

[CBW94] Roderick Chapman, Alan Burns, and Andy Wellings. Integrated program proof and worst-case timing analysis of SPARK Ada. In *ACM Sigplan Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994. Available on `www.cs.umd.edu/users/pugh/sigplan_realtime_workshop/lct-rts94/`.

[CCG+75] Graham Chapman, John Cleese, Terry Gilliam, Eric Idle, Terry Jones, and Michael Palin. Monthy Python and the Holy Grail. Movie produced by Python (Monty) Pictures, 1975.

[CD95] Tai M. Chung and Hank 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. Available on `www.cs.umd.edu/projects/TimeWare/sigplan95/`.

[Cha94] Roderick 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, October 1994.

[Cin97] WWW homepage for the `cinderella` system, August 1997. Available via WWW at the address: `http://www.ee.princeton.edu/ỹauli/cinderella-3.0/`.

[Coo92] Lyle Edward Cool. Debugging VLIW code after instruction scheduling. Master's thesis, Oregon Graduate Institute, Department of Computer Science, July 1992. Available via ftp from `ftp://cse.ogi.edu/pub/tech-reports/1992/92-TH-009.ps.gz`.

[Cop92] Max Copperman. Debugging optimized code without being misled. Technical Report UCSC-CRL-92-01, University of California, Santa Cruz, May 1992. Available via ftp from `ftp://ftp.cse.ucsc.edu/pub/tr/ucsc-crl-92-01.ps.Z`.

[Cou81] P. Cousot. Semantic foundations of program analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

[Cou96] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.

[CP+91] Alberto Coen-Porisini et al. Software specialization via symbolic execution. *IEEE Transactions on Software Engineering*, 17(9):884–899, September 1991.

[CR81] Lori A. Clarke and Debra J. Richardson. Symbolic evaluation methods for program analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 9, pages 264–300. Prentice-Hall, 1981.

[Dot97] WWW homepage for the graphics visualizations at AT&T Research, August 1997. Available via WWW at the address: `http://www.research.att.com/sw/tools/graphviz/`.

[EG97a] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of Euro-Par 97, LNCS (Lecture Notes in Computer Science) 1300*. Springer Verlag, August 1997.

[EG97b] Andreas Ermedahl and Jan Gustafsson. Realtidsindustrins syn på verktyg för exekveringstidsanalys. Technical Report ASTEC Technical Report 97/06, ASTEC Competence Center, Uppsala University, July 1997.

[For92] Charles 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.

[For93] Charles Forsyth. Using the worst-case execution analyser. Technical Report Hard Real-Time Operating System Kernel Study Task 8, Volume D, York Software Engineering Ltd, July 1993.

[GE97] Jan Gustafsson and Andreas 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*, April 1997.

[Ger94] Richard Gerber. Languages and tools for real-time systems: Problems, solutions and opportunities. Technical Report UMD CS-TR-3362, UMIACS-TR-94-117, Department of Computer Science, University of Maryland, October 1994.

[GHL+92] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, pages 121–131, February 1992.

[GPMTB97] Jan Gustafsson, Kjell Post, Jukka Mäki-Turja, and Ellus Brorson. Benefits of type inference for an object-oriented real-time language. In *Preprints for SNART 97, Konferens om Realtidssystem, Lund, 21-22 augusti 1997*, August 1997.

[HBW94] Marion G. Harmon, T. P. Baker, and David B. Whalley. A retargetable technique for predicting execution time of code segments. *Real-Time Systems*, pages 159–182, September 1994.

[HHG+95] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August. Compiler technology for future microprocessors. *Proceedings of the IEEE*, 83(12):1625–1995, December 1995.

[HWH95] 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*, December 1995.

[IAR97] IAR Systems WWW homepage. WWW-address: `http://www.iar.se`, August 1997.

[KHR+96] L. Ko, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M.G. Harmon. Supporting the specification and analysis of timing constraints. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, June 1996.

[KN] Eleftherios Koutsofios and Stephen C. North. *Drawing Graphs with* dot. AT&T Bell Laboratories, Murray Hill, NJ.

[Kou96] Apostolos A. Kountouris. Safe and efficient elimination of infeasible execution paths in WCET estimation. In *Proceedings of RTCSA'96*. IEEE, IEEE Computer Society Press, 1996.

[LBJ+95] 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. Available using WWW from `http://archi.snu.ac.kr/symin/ets.ps`.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32th Design Automation Conference*, pages 456–461, 1995.

[LMW96] Yau-Tsun Steven Li, Sharad Malik, and Andrew 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, December 1996.

[Mar94] William Marsh. Formal semantics of SPARK: Static sematics. Technical report, Praxis Critical Systems Ltd, `www.praxis-cs.co.uk`, October 1994.

[MG95] Peter Mardwedel and Gert Goosens, editors. *Code Generation for Embedded Processors*, chapter 1. Kluwer Academic Publishers, 1995. Chapter written by Araujo, Devadas, Keutzer, Liao, Malik, Sudarsanam, Tijang, and Wang.

[Nul97] Nullstone Corporation WWW homepage. Available via WWW at the address: `http://www.nullstone.com/htmls/category.htm`, July 1997.

[O'N94] Ian O'Neill. Formal semantics of SPARK: Dynamic sematics. Technical report, Praxis Critical Systems Ltd, `www.praxis-cs.co.uk`, October 1994.

[OS97] 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.

[Par93] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, March 1993.

[PK89] Peter Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(1):159–176, 1989.

[PS93] Peter Puschner and Anton Schedl. A tool for the computation of worst case task execution times. In *Proc. of the 5:th EUROMICRO Workshop on Real-Time Systems*, 1993.

[PS95] Peter Puschner and Anton Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.

[RTT97] RealTimeTalk Homepage at Mälardalens Högskola, Västerås, Sweden. Available via WWW at the address: `http://www.mdh.se/avdelningar/idt/forskning/cus/rtt_projekt/`, August 1997.

[Sta97] Friedhelm Stappert. Predicting pipelining and caching behaviour of hard real-time programs. In *EUROMICRO Workshop on Real-Time Systems*, June 1997.

[Vrc93] Alexander Vrchoticky. Modula/R language definition. Technical Report TU Wien rr-02-92, version 2.0, Dept. for Real-Time Systems, Technical University of Vienna, May 1993.

[Vrc94a] Alexander Vrchoticky. *The Basis for Static Execution Time Prediction.* PhD thesis, Institut für Technische Informatik, Technische Universität Wien, Treitlstraße 3/182.1, A-1040 Wien, Austria, April 1994.

[Vrc94b] Alexander Vrchoticky. Compilation support for fine-grained execution analysis. In *ACM Sigplan Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994. Available on `www.cs.umd.edu/users/pugh/sigplan_realtime_workshop/lct-rts94/`.

[Wis93] Roland Wismüller. Source level debugging of optimized programs using data flow analysis. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 238–240. ACM, May 1993. Available via WWW from `http://wwwbode.informatik.tu-muenchen.de/~wismuell/publications.html`

[Wis94] Roland Wismüller. Debugging of globally optimized programs using data flow analysis. In *Proceedings of the ACM/SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–289. ACM, June 1994. Available via WWW from `http://wwwbode.informatik.tu-muenchen.de/~wismuell/publications.html`

[WMH+97] 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.

[Zel84] P. T. Zellweger. *Interactions between high-level debugging and optimised code.* PhD thesis, Computer Science Division, University of California, Berkeley, 1984. Published as Xerox PARC Technical Report CSL-84-5.