Applying compiler technology to solve generic workflow problems

Paulo Jorge Matos
Departamento de Informática e Comunicações
Escola Superior de Tecnologia e de Gestão
Instituto Politécnico de Bragança
Mail: pmatos@ipb.pt

Pedro Rangel Henriques
Departamento de Informática
Universidade do Minho
Mail: prh@di.uminho.pt

Abstract

Compilers are tools that transform a high level programming languages into assembly or binary code. The essential of the process is done by the interpretation and the code generation steps, but nowadays most compilers have also a strong component of code optimization, that explore as much as possible the potential of the computer architectures to which the compiler must generate the code. These optimizations are based on the information provided by several analysis processes. This paper present some of these code analysis and optimizations, and shows how they can be used to solve problems or improve the quality of solutions used at areas such as industrial engineer and planning.

Keywords: workflow processes, flow graphs, code analysis, code optimization.

INTRODUCTION

The main goal of the compilation is to identify the operations described in some source language (Fortran, Pascal, C/C++, ML, ...), and convert them into a list of assembly or binary instructions (the output code), executable by the chosen processor (Pentium, PowerPc, Sparc, ...). All this must be done without losing the semantics of the source program, which means that the output code must do exactly what is described by the source code.

The evolution of this technological area has increased the gap between the source languages and the output code. Programming languages have evolved, and nowadays they are more abstract, syntactically more powerful, supporting paradigms that are quite different from the one that is used by the assembly and the machine languages. This evolution creates new problems to solve and requires, from the compiler developers, more powerful solutions that, by one side, can solve efficiently these problems and, by the other side, can take advantage of the evolution of the processors architecture.

To achieve a high performance output code the compiler must execute several code optimizations. Typically, each one is done in two steps: one that collects the information

(about the source program), necessary to execute the optimization; and the other that is the code optimization itself. At the context of this paper we will designated the first step by "analysis" and the second step by "optimization".

Compilers typically contain analysis algorithms of distinct kinds, the most common deal with *control* and *data flow* across the program or with *data dependencies* (but these are not the only ones). In this paper we introduce some analysis processes that can be useful to solve problems in different technological areas. The goal is not to solve directly the problems but produce information about them that can lead to a solution or improve the quality of an existent one. For each kind of analysis introduced (in the paper we deal with two: CFA and DFA), we also discuss its importance in the compilers domain (presenting code optimization routines that use the results provided by the analysis), and we illustrate (using a case study) its applicability to problem solving in other technological areas.

Before continue, it is important to relate a compiler with the compiler developer and the program developers, clarifying the role of each actor. Figure 1 shows at the center the compiler itself, surrounded by: the compiler developer (left side), the program developer (upper right corner) and the program users (lower right corner).

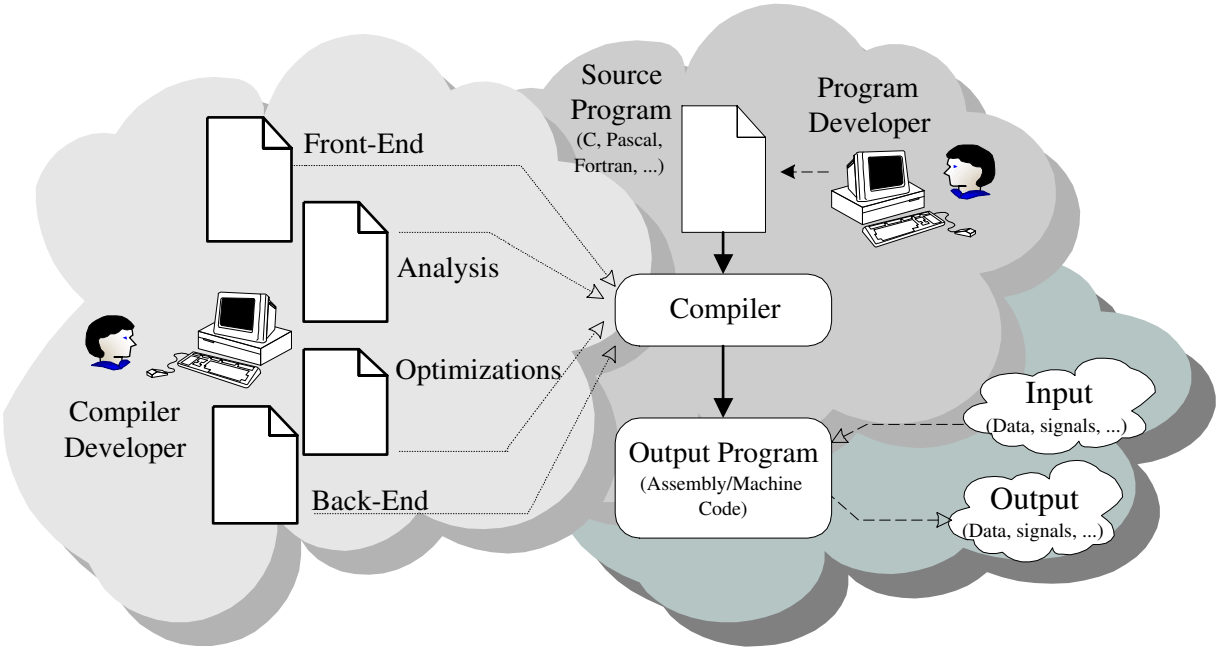


Figure 1: Compiling, developing and execution processes.

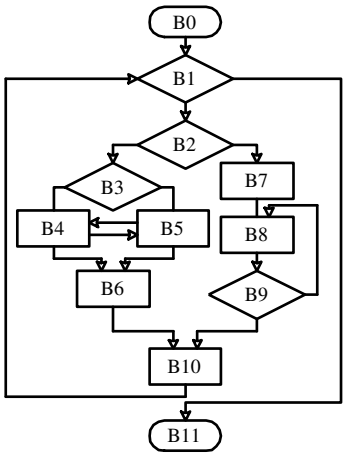
Figure 1 also shows the principal components of the compiler: the front-end, analysis and optimization routines, and back-end. All compilers must execute the tasks included at the front and the back-end components. The first is responsible by the interpretation of the source language and includes the lexical, syntactic and semantic analysis. The last one is responsible by the output code generation and, typically, includes the register allocation and the instruction selection. The analysis and optimizations components are fundamental to improve the quality of the output code. Together they form the middle-level of the compiler.

In the next two sections we explore two distinct kinds of analysis: the Control Flow Analysis (CFA) and the Data Flow Analysis (DFA). The last section presents the conclusions and describes the future work.

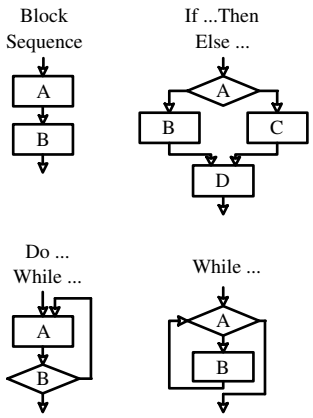
It is important to notice that the examples used along the paper are intentionally small, very simple and do not require special knowledge about compilation or even about programming languages. The objective is to make easy the understanding of the subject, but this does not mean that the real live examples are like these or that the code optimizations are limited to these transformations.

CONTROL FLOW ANALYSIS

The compiler submits the textual representation of the source program to several transformations until obtain the output code. At the middle-level of the compilation process the code representation contains, implicitly or explicitly, a huge graph, which is designated by Control Flow Graph (CFG), and represents the control flow structure of the program with all possible execution paths. Figure 2(a) shows an example of a CFG.



(a) An example of a CFG.



(b) Examples of regions that can be recognized.

Figure 2: The CFG and some examples of regions.

The CFA is very important for many optimizations and even to the code generation routines, for example, it allows to identify the vertices of the CFG that form a cyclic structure or a conditional structure. Notice that this can be quite complex since a control flow structure may contain inside others control flow structures.

The compiler get the desire information about the structure of the source program, building the Control Flow Tree (CFT), which is a tree where the root represents the full program; each intermediate node represents a control flow structure; and the leafs are the original vertices of the CFG (see figure 4).

The process is done by searching the CFG for predefined control flow structures, designated by "regions". When one is found, it is necessary to replace the vertices that belong to that region by a single node. The process ends when the full program is reduced to a single node. The CFT is built while the reduction process is done, by establish a relation between each inserted vertex and the vertices that are replaced.

Figure 2(b) shows the regions that were defined to reduce the CFG of the figure 2(a). The reduction process is illustrated at the figure 3 and figure 4 shows the final CFT.

At the figure 3, it is possible to observe that the reduction process can isolate parts of the graph that do not correspond to any of the predefined regions. These parts are

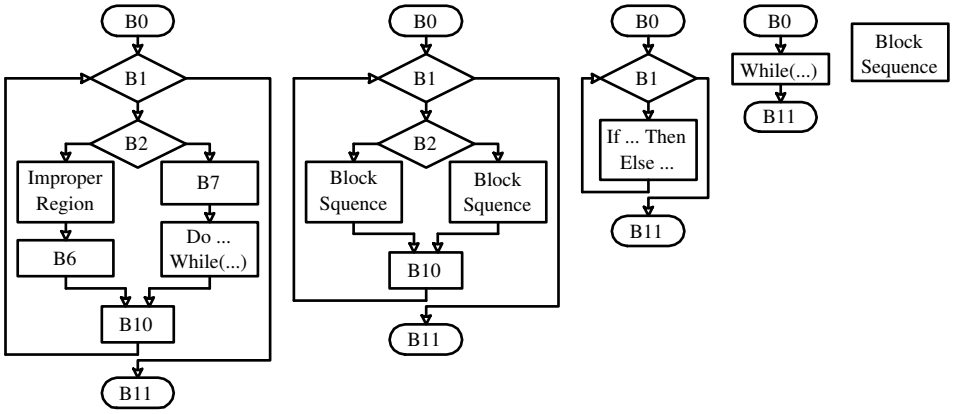


Figure 3: The reduction process of the graph of the figure 2(a).

classified has *Improper* or *Proper* regions, depends if they contain or not cyclic control flow structures.

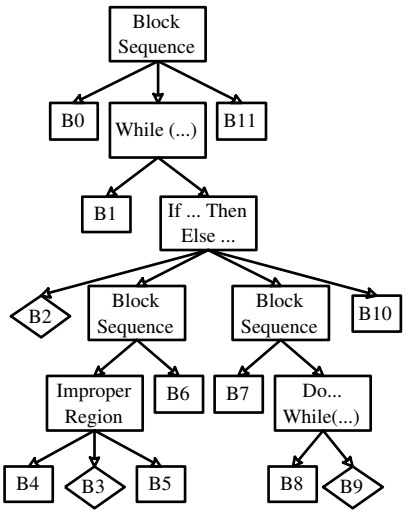


Figure 4: The CFT correspondent to the CFG of figure 2(a).

The type of solution proposed here to compute the CFT is designated by *Structural Analysis* (Sharir, 1980), and is a very versatile solution that can deal with distinct types of regions. The only required condition is that the regions must be accessed over a single vertex, designated by the *entry vertex* of the region.

It is not our intention to explore in this paper the details about *Structural Analysis*. But the main part of the algorithm is shown in figure 5. It is possible to find more details and explanations in Muchnick (1997), or in Matos (1999).

CFA optimizations

There are many code optimizations, code generation tasks and even other forms of the analysis (like the DFA) that use the results provided by the CFA. For example, using the CFT is possible: to linearize the graph to obtain the sequence of operations, which is fundamental for code generation; to determine the regions that can be computed concurrently; or to choose the regions that can be the target of specific optimizations, such as the loop optimizations.

```
Procedure StructuralAnalysis( g :CFG) : CFT
reduction : boolean
rtype : RegionType
n, m, entry : Vertex
interval, ReachUnder, NonEntries : Set<Vertices>
    entry := getRoot(g)
    Repeat
        reduction := false
        NonEntries := ∅
        While |SetVertices(g)| > 1 & !reduction do
            n := pickOneVertice(g)
            rtype := AcyclicRegionType( g, n, interval)
            If rtype ≠ nil then
                Reduce(g, n, rtype, interval)
                reduction := true
            else
                ReachUnder := {n}
                For each m ∈ SetVertices(g) do
                    If Type(m)=BasicNode & Path(g,n,m) & PathBack(g,m,n) then
                      ReachUnder ∪= {m}
                rtype := CyclicRegionType(g,n,ReachUnder)
                If rtype ≠ nil then
                    SetVertices(g) ∪= NonEntries
                    Reduce(g, n, rtype, ReachUnder)
                    reduction := true
                else
                    SetVertices(g) -= {n}
                    NonEntries ∪= {n}
    Until !reduction
    return head(SetVertices(g))
End
```

Figure 5: The Structural Analysis algorithmic.

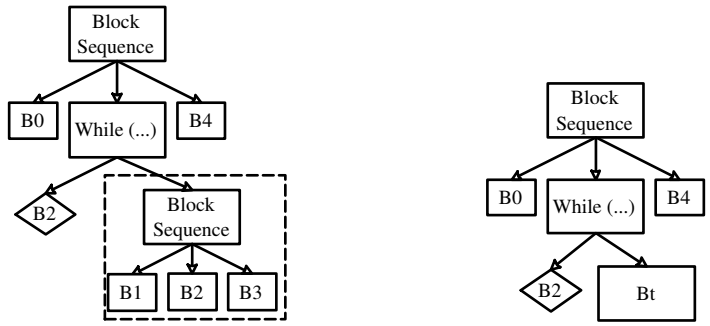
Probably the simplest example is the elimination of the jump operations between vertices. Notice that if we have a sequence of vertices with only one antecessor, then it is possible to join the operations contained in each one into a single vertex. Figure 6(a) shows part of a CFT where this situation is easily detected and figure 6(b) shows the result obtained after the optimization.

CFA case study

Our case study is a generic industrial process, represented in figure 7. Elements like Res_i are resources; elements like B_i are buffers; and the other elements denote various processes (**W**eld, **Pr**e or **Po**s-Assembly, **A**ssembly, **T**est, **R**epair and **P**ack). The regions that we want to recognize are represented at figure 8.

Figure 9 shows the correspondent CFT, obtained by applying the CFA to the process graph. This one has several applications, for example:

- With a top-down, left-right, traversal of the CFT, it is possible to obtain the topological order of the CFG;
- It is easy to identify subsets of tasks that can be executed concurrently, by searching



(a) The CFT before the control flow optimization.

(b) The CFT after the control flow optimization.

Figure 6: An example of a control flow optimization.

for *Region 1* nodes at the CFT. Each descendent is one of the concurrent set of tasks;

- As we will see in the next section, the CFT is very important to analyze the flow of the data (product, information, control parameters, etc) along the CFG.

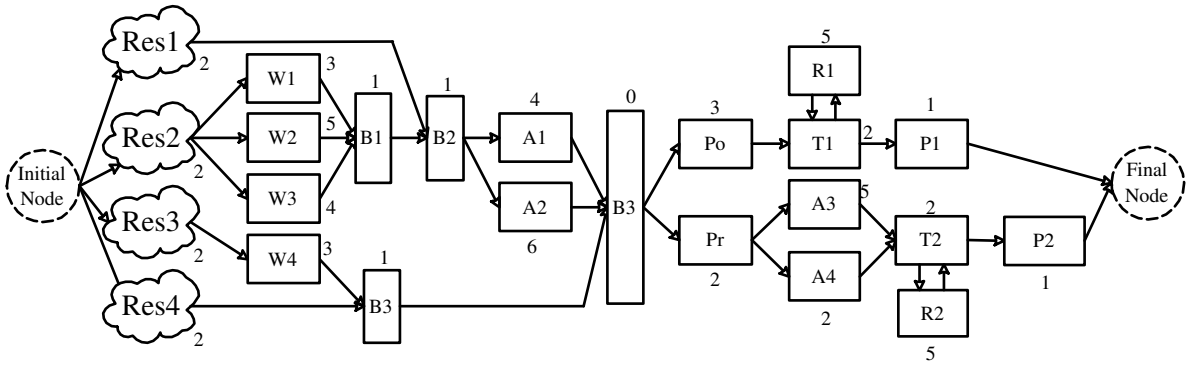


Figure 7: A generic industrial process.

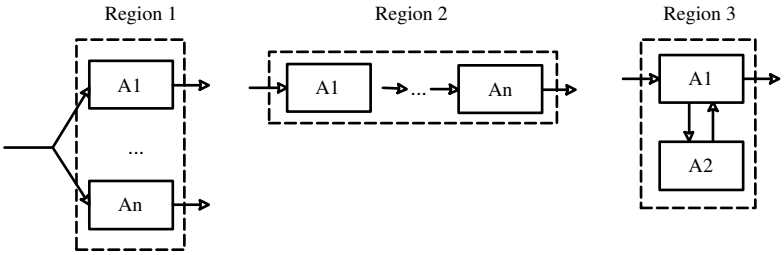


Figure 8: The regions defined to be recognized.

DATA FLOW ANALYSIS

The Data Flow Analysis (Kam and Ullman, 1976; Muchnick, 1997; Nielson *et al.*, 1999) is responsible for collecting information that change along the execution paths of the graph (flow sensitive information); that information is used later by the optimization routines to minimize the code length or make its execution faster. To explain how this is done

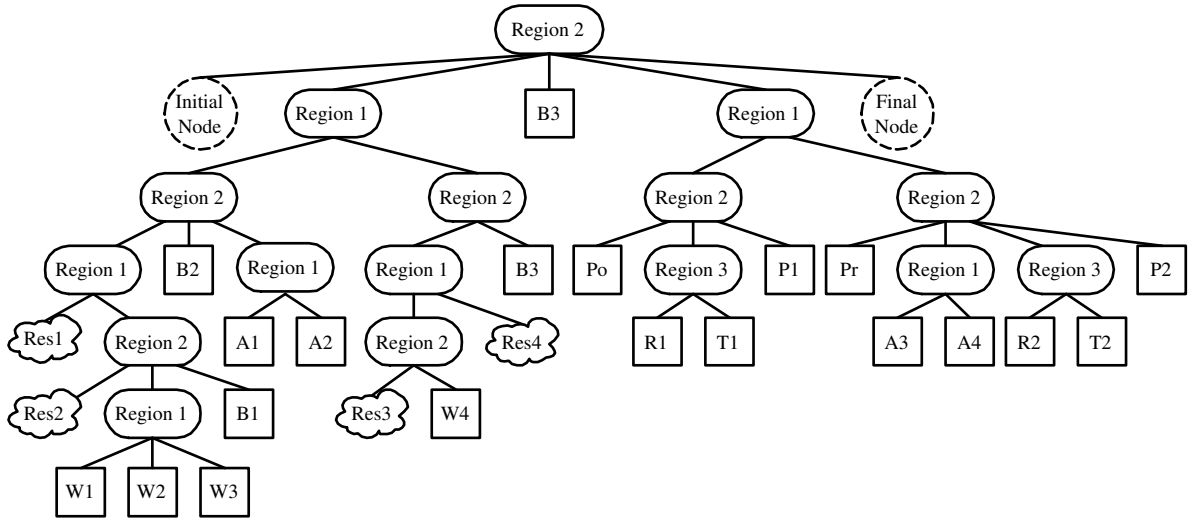


Figure 9: The final CFT of the process of figure 7.

and why this type of analysis is necessary, we will use the *Common Sub-Expressions Elimination* optimization that identifies and removes duplicate expressions of the source program.

Figure 10(a) shows a short fragment of a C program where the same sub-expression $(i+1)$, is computed several times ($op_2, op_3, op_4, op_5, op_7$ *and* op_8).

```
op1  ...
op2  a = i + 1;
op3  if(i+1>0){
op4     i = i + 1;
op5     b = i + 1;
op6  } else
op7     b = 2 * (i+1);
op8  c = (i+1)/2;
op9  ...
```

(a) Code before the optimization.

```
op1  ...
op2  a = i+1;
op3  if(a>0){
op4     i = a;
op5     b = i + 1;
op6  } else
op7     b = 2 * a;
op8  c = (i+1)/2;
op9  ...
```

(b) Code after the optimization.

Figure 10: The Common Sub-Expression Elimination optimization.

Unnecessary computations can be avoided if several occurrences of the same sub-expression are replaced by the variable that holds the value obtained by the evaluation of the first sub-expression (the only one that is not removed). This can be done if we have sure that all replaced sub-expressions, compute the same value as the one that stay, no matter the execution path.

Applying the *Common Sub-Expressions Elimination* strategy to the program of figure 10(a), we obtain the program of figure 10(b). Notice that the occurrence of the sub-expression $i+1$ at the operation op_5 is not replaced, because at this position the value of the variable i was already changed (by the operation op_4). The same happens with its occurrence at op_8, but now because the sub-expression may have different values depending on the actual execution path (it has a direct influence in the value of variable i when it reaches operation op_8). Since the compiler can not change the semantics of the source program,

a sub-expression should only be replaced when it is possible to determine that such operation is safe.

The optimization illustrated in the example above (figure 10) depends of the control flow of the program, which means that the optimization routine must consider the several paths that may occur during the execution of the program. The DFA comes to scene just to solve this problem, feeding the system with information concerned with the computation and propagation of values.

Notice that the DFA is the generic name of this type of analysis, a concrete example is the *Reach Definition Analysis* (RDA) that is used to determine the available sub-expressions at each position of the program. The RDA is used to support the *Common Sub-Expression Elimination*.

Figure 11(a) shows a detailed CFG of the program of figure 10(a). Each expression inside a program block is decomposed, independently from the others, into elementary operations (including operands), and the result of each one is hold by a temporary variable (t_i). The list with all the available sub-expressions, so far obtained, is associated with the start and the end of each block (vertex of the graph). Each element of this list contains a list of temporary variables (one or more) and, between parentheses, the respective sub-expression, according to the following rules:

List_element → List_Temp_Variables '(' Sub-Expression ')'

List_Temp_Variables → Temp_Variable
| List_Temp_Variables ',' Temp_Variable

It is assumed that the start list of the initial block is empty.

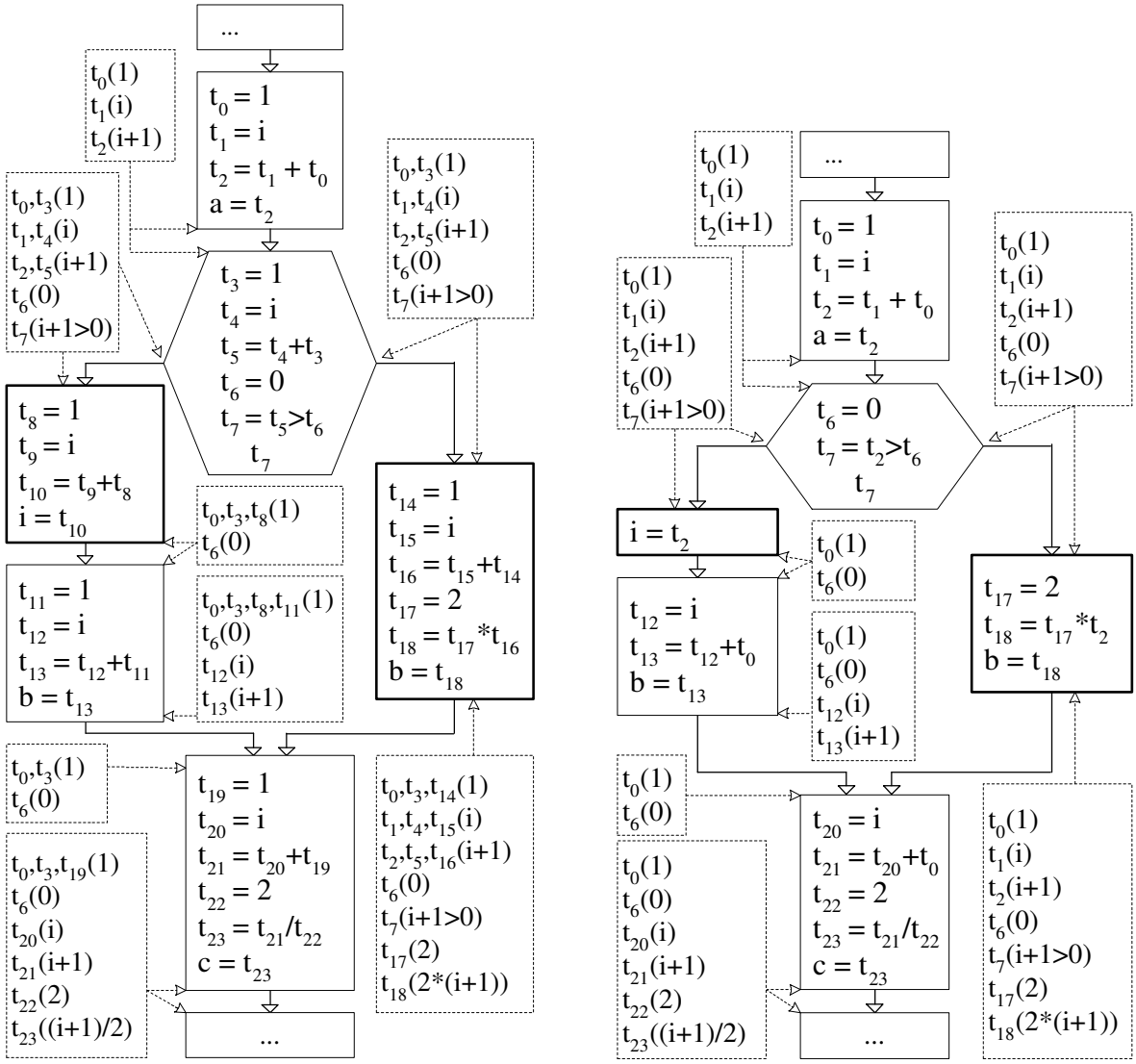
Figure 11(b) shows the effects of the *Common Sub-Expression Elimination* optimization using the information provided by the analysis. For the moment, the important is to observe that in fact this optimization (as consequence of the DFA performed) improved significantly the quality of the output code.

Now we describe in detail how the analysis is done, considering just one node n of the graph. Suppose that the list of available sub-expressions at the entry of n is $L_{entry}(n)$, and at the exit is $L_{exit}(n)$. The goal is to compute $L_{exit}(n)$ based on the values of $L_{entry}(n)$ and on the own contributions of node n. Initially $L_{exit}(n)$ takes the value of the $L_{entry}(n)$. Then it is necessary to analyze, one by one, the sub-expressions of n (starting on the first one), identifying the shape of the sub-expression. If it has the form $t_i = expr$, it is necessary to test if $expr$ already exists in $L_{exit}(n)$; if true, then t_i is appended to the list of temporary variables associated with $expr$; if false, then a new element, of the form $t_i(expr)$, is append to $L_{exit}(n)$. If the sub-expressions has the form $var = t_j$, it is necessary to remove all elements whose sub-expression contains one or more references to var.

Notice that when a variable var_x is defined (a value is assigned to it), its previous value will probably change; as consequence, all sub-expressions that were available at this point of the program and that use the defined variable (var_x), are no longer valid.

The description above can be formulated using equation 1, where the f_{expr} represents the contributions of the sub-expression $expr$.

$$L_{exit}(expr) = f_{expr}(L_{entry}(expr)), \qquad where\ expr \in n \tag{1}$$



(a) CFG before the optimization.

(b) CFG after the optimization.

Figure 11: A CFG with the result of *Reaching Definition Analysis*.

If we associate the $L_{entry}(n)$ with the L_{entry} of the initial sub-expression of n, and $L_{exit}(n)$ with the L_{exit} of the last sub-expression of n, then it is possible to compute directly $L_{exit}(n)$ based on the $L_{entry}(n)$, using the equation 2, where f_n is the composition of the $f_{expr}()$ of all sub-expressions of n, starting on the last one.

$$\begin{aligned} L_{exit}(n) &= f_n(L_{entry}(n)) \\ & \quad f_n = f_{expr_{last}} \circ \ldots \circ f_{expr_{first}} \end{aligned} \tag{2}$$

Now let us see how the analysis is done for the full graph. If n has only one predecessor, m, then $L_{entry}(n)$ takes the value of $L_{exit}(m)$. If n has more than one predecessor, then it is necessary to join the several L_{exit} of the predecessors of n to compute $L_{entry}(n)$. This corresponds to assign to $L_{entry}(n)$ the elements that are common to all L_{exit} of the predecessor nodes. This can be formulated using the equation 3, where *Flow* represents the edges between nodes. It is important to notice that the edges that belong to *Flow* have not to be the same ones that are used at the CFG; it depends on the direction of

the analysis that can be *forward*, if is done following the same direction of the edges of the CFG, or *backward* if is done in the reverse direction. The symbol $\bigsqcup$ represents the join operation that defines how to combine the lists $L()$ that reach a node.

$$L_{entry}(n) = \bigsqcup\{L_{exit}(n') \mid (n', n) \in Flow\} \tag{3}$$

Equation 3 can be redefined into equation 4 in such way that it is possible to have a non-empty list, $l_{initial}$, at the entry of the initial block of the graph.

$$L_{entry}(n) = \begin{cases} l_{initial} & n \in InitialNodes \\ \bigsqcup\{L_{exit}(n') \mid (n', n) \in Flow\} & otherwise \end{cases} \tag{4}$$

Now that we know how to compute the values for the nodes of the graph, it is necessary to determine the order by which they should be processed. The solution that we proposed here is based on CFA, namely on the Structural Analysis (see the last section). It is enough to traverse the CFT, starting by the root node, and apply the quations 2 and 4 considering the kind of region represented by each node. For example, to a *If ... Then ... Else ...* region, as the one showed at the figure 2(b), the problem is solved applying the equation system 5. It is supposed that $L_{entry}(IfThenElse)$ (the entry value of the region) was already computed and that the goal is to obtain the $L_{exit}(IfThenElse)$.

$$\begin{cases} L_{entry}(A) & = & L_{entry}(IfThenElse) \\ L_{exit}(A) & = & F_A(L_{entry}(A)) \\ L_{entry}(B) & = & L_{exit}(A) \\ L_{exit}(B) & = & F_B(L_{entry}(B)) \\ L_{entry}(C) & = & L_{exit}(A) \\ L_{exit}(C) & = & F_C(L_{entry}(C)) \\ L_{entry}(D) & = & L_{exit}(B) \sqcup L_{exit}(C) \\ L_{exit}(D) & = & F_D(L_{entry}(D)) \\ L_{exit}(IfThenElse) & = & L_{exit}(D) \end{cases} \tag{5}$$

For cycle regions it is necessary to apply the equations more than once. The reason is that the L_{entry} of the entry node may depend of the results of the others nodes of that region.

If the transfer function is monotone, it is guarantee that the successive application of the equations, sooner or later, will result on a stabilized values for all equations. The first time that the equations are applied, the feed-back component that result from other nodes of the region, is despised.

It is quite simple to generalize this DFA solution to solve similar problems (Dwyer, 1995; Knoop and Ruthing, 1994). The algorithmic solution is the same one and the equations 1, 2, 3 and 4 are still valid. Essentially, it is only necessary to redefine the type of data structure that should be associated with the nodes, formally designated by lattice values ; the functions associated with the sub-expression ($f_{expr}(\ )$); the join operator; and to each type of region, the way how the order by equations should be computed.

DFA case study

The goal of our case study is to determine the lowest and the highest time need to assembly a unit of the product to the process of the figure 7. This can be done associating a pair of values (*min*, *max*), to the entry and exit of each node. The *min* holds the lowest time and *max* the highest time. The cost of each task is given by the function $Cost(X)$ that returns the value associated to the element X of the figure 7, for example $Cost(A1) = 6$. The *Initial Node* and the *Final Node* have both cost zero. The entry value of the *Initial Node* is $(0, 0)$. The analysis will follow the same direction of the flow of the process (*forward* direction).

As stated by equation 1, $L_{exit}(n)$ is obtained applying the transfer function of n to the $L_{entry}(n)$. Equation 6, below, defines the transfer function; in that case, its argument is the node itself, and not the value at the entry of the node. So, equation 1 must be rewritten as the equation 7 bellow.

$$f_{transfer}(n) = L_{entry}(n) + (Cost(n), Cost(n)) \tag{6}$$

$$L_{exit}(n) = f_{transfer}(n) \tag{7}$$

L_{entry} is obtained determining the minor of the lowest values of the incoming edges; and the greater of the highest values of the incoming edges. Let $lowest(L)$ and $higest(L)$ return, respectively, the lowest and the highest value of the lattice L. $L_{entry}(n)$, and implicitly the join operation, are defined by the equation 8.

$$L_{entry}(n) = \begin{cases} (0,0) & n = InitialNode \\ \\ (Min(\{lowest(L_{exit}(n'))|(n',n) \in Flow\}), \\ \ Max(\{highest(L_{exit}(n'))|(n',n) \in Flow\})) & otherwise \end{cases} \tag{8}$$

Notice that the lattice concept can be used to represent almost everything, for example: numeric values, text, sets, abstract data structures, objects, etc.

The processes, to which we forecast, that is possible to apply the CFA and specially the DFA, can be characterized as follows:

- The processes are quite big and hardly analyzed by hand;
- The processes can be modeled by a finite flow-graph, where the vertices represent the entities that operate the information under analysis, and the edges describe the information flow between operations;
- The information processed should be flow-sensitive;
- It should be possible to define a finite set of functions that describe how the entities affect the information. These functions are designated by "transfer functions" and should be monotone;
- It should be possible to define the identity function (the one that has not any effect over the information);

- It should be possible to define an algebraic data structure to represent the information that forms a complete lattice (a partial ordered set that contains a *top* and *bottom* value and for which are defined the *meet* and the *join* operators).

CONCLUSIONS AND FUTURE WORK

With this paper, we aimed at showing how it is possible to adapt the *code analysis and optimization techniques*, implemented by most of the compilers, to solve different planning and management problems in areas such as industrial, economical, governmental and information systems.

Analysis such as the CFA and the DFA are quite common in the development of compilers and play an important role in the quality of the output code. We believe that they can be used to solve properly many other problems, and also that these problems are traditionally solved using other solutions that might be interesting for compiler developers.

At this point, some people may ask: why other solutions those are much more complex than those already available and maybe not so efficient? We believe these solutions have some advantages:

- They are quite uniform and is easily adaptable;
- They can deal quite well with abstract information (not only with quantifiers);
- They work independently of the flow-graph topology, which means that a specific analysis for each problem is not necessary, but only one for each class of problems.

The paper focus was CFA and DFA, but we have also in mind the idea of explore other compiler techniques - more elaborated forms of DFA (like the Alias Analysis) and the Dependency Analysis (used for instruction scheduling). We also expect to find out a better set of case studies (we are looking for new ones) to demonstrate the potential of this technology. Then, the next step will be the development of a tool/framework to help the construction of the analysis and optimizations routines (Tjiang and Hennessy, 1992; Tjiang, 1993; Alt and Martin, 1995), which will be done based on a description of some characteristics of the problem - like the specification of the *join* operation or the definition of the *transfer functions* or even the description of region patterns - and not on the description of the concrete problems.

References

Alt, M. and Martin, F. *Generation of Efficient Interprocedural Analyzers with PAG*. In A. Mycroft, editor, *SAS'95, Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 33–50. Springer, 1995.

Dwyer, M. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs*. Ph.D. thesis, Amherst, MA, USA, 1995.

Kam, J. B. and Ullman, J. D. *Global data flow analysis and iterative algorithms*. Journal of the ACM, 21(3), pages 158–171, 1976.

Knoop, J. and Ruthing, O. *Optimal code motion: theory and pratice*. Trans. on Progr. Languages and Systems, 16(4), pages 1117–1155, 1994.

Matos, P. J. *Estudo e desenvolvimento de sistemas de geração de back-ends do processo de compilação*. Master's thesis, Universidade do Minho, Braga, Portugal, 1999.

Muchnick, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

Nielson, F., Nielson, H. and Hankin. *Principles of Program Analysis*. Springer Verlag, 1999. ISBN 3540654100.

Sharir, M. *Structural analysis: A new approach to flow analysis in the optimizing compilers*. Computer Languages, 5(3–4), pages 715–728, 1980.

Tjiang, S. *Automatic Generation of Data–flow Analyzers: A tool for building optimizers*. Ph.D. thesis, Stanford University, Computer Systems, Laboratory, 1993.

Tjiang, S. and Hennessy, J. *Sharlit - a tool for building optimizers*. In *Proceedings on Programming Language Design and Implementation*, pages 82–93. 1992.