Technical Report

Department of Computer Science and Engineering

Mälardalen University, Sweden

Theories for Estimating Execution Times in Soft Real-Time Systems Applied in the Telecommunication Domain

Jouni Axelsson
Department of Computer Science and Engineering
Mälardalen University

Johan Erikson
Department of Computer Science and Engineering
Mälardalen University

October 25, 2001

Abstract

Real-time systems are normally divided into two classes: Hard and Soft. The difference lies in how critical it is to meet the systems timing constraints. In a hard real-time system, every deadline **must** be met. If not, the consequences to the environment is above an accepted level. On account of this, the *Worst Case Execution Time* is in focus for a number of proposed approaches.

A soft real-time system, on the other hand, is characterized by the fact that a missed deadline do not cause system failure and do not cause damage to the environment. This is most likaly the reason why timing constraints in soft real-time systems are handled by different ad hoc solutions. (E.g. *Do not write more than k lines of code in one block.*) But even if a missed deadline is not directly dangerous, the consequences may indirectly be serious.

In this report, we propose a more systematic approach to handle execution times in soft real-time systems. A combination of graphical representation of the system together with information on loops, gives the developers a picture of the execution time which enable identification of critical areas and parts that use much time relative other parts of the system. The graphical representation is annotated with estimated execution times. Loop detection is based on DJ graphs and processes the control flow graph of the program which, in our case, is extracted from the back end of the compiler.

The theories is applied in the telecommunication domain where the targeted system (the AXE telecommunication system from Ericsson) has about 10 milion lines of code spread over about 1000 executable units, continously developed during 25 years.

Acknowledgments

Parts of this research project was carried out as a Masters Thesis project and was written during the period spring to autumn 2000, partly at *Ericsson Research and Development AB* and partly at *Mälardalen University*, Västerås, Sweden. It is part of a research project on execution time analysis and system understanding in **Soft Real-Time Systems**.

The authors would like to thank *Janet Wennersten* and *Anton Massoud* at Ericsson Research and Development AB, and *Peter Funk* at Mälardalen University for supervising this work.

Contents

List of Figures

List of Tables

Part I

Context

Chapter 1

Introduction

1.1 Background

Programming Language for Exchanges, PLEX, is the programming language used by Ericsson in their AXE system, a system that has been classified as a soft real-time system [AGG99]. A soft real-time system is characterized by the fact that a missed deadline do not cause system failure and do not cause damage to the environment. This is most likaly the reason why timing constraints in soft real-time systems are handled by different ad hoc solutions. (E.g. *Do not write more than k lines of code in one block.*) But even if a missed deadline is not directly dangerous, the consequences may indirectly be serious. (Imagine a failing telephone system in an emergency situation.)

1.2 Aim

The main task in this report is to explore possible methods for estimating execution times in soft real-time systems (e.g. the AXE system) and propose a more systematic approach than the ad hoc methods mostly used today.
A secondary task is to extract the data needed as input input to GRETA[1], and annotate the graphical representation with estimated execution times.

[1]The graphical prototype developed by Arnström et. al. [AGG99]

1.3 Related work

- [AGG99] made an analysis of PLEX and the AXE system, a visual tool to represent the structure of PLEX and also briefly discussed execution time calculations for PLEX. This thesis discusses some theories and possible ways to go, while we show a applicable approach. Also [KO00] has looked at PLEX. Here the approach was towards register allocation.

- The E-CARES research project, [MH01], deals with code abstraction and reverse engineering and explores the same target system as we (i.e. the AXE switching center). The E-CARES project started at the same time (1999) as the [AGG99] master thesis project and there were some contacts between the projects at this time. The similarity between our projects is the possibility to explore the system graphically. The main difference is that the E-CARES project do not discuss execution times, their focus is on system understanding.

- A lot of work has been done in the area of loops in graphical representations. In this work, the DJ graph approach developed in [SGL96] is one of the fundamental parts in the prototype tool. ([SGL96] is a generalization of Tarjan's algorithm for handling flow graph reducibility [Tar74].)

1.4 Limitations

We restrict our results to systems/programs where a control flow graph is available or can be constructed. This is because the loop detection phase is based on the DJ graph (see section 4.1 for an explenation and section 11.7 for an example) of the given program and the input for constructing the DJ graph is the control flow graph.
Much of the work has been focused on loop detection since these constructs have a major impact on the total runningt time of the program. We have not, however, tried to find the number of iterations for a given loop. (We explain why in section 11.1.1.)
Regarding execution times, some generalizations and overestimations have been made. For example, we assume that the execution time for a single statement is one time unit.

1.5 Summary of contributions

Execution times and time restrictions in soft real-time systems are today mostly handled by different ad hoc solutions. In this report we show a more systematic approach to estimate execution times. This, together with the graphical environment developed in the [AGG99] master thesis, give developers a picture of the execution characteristics of the analyzed program which in turn enables identification of critical areas and parts that use much time relative other parts of the program and system.

1.6 Structure of this report

This report is organized in three main parts:

General background theory for Worst Case Execution Time estimation Chapter 3 - 5 covers theory in general. In chapter 3 we briefly discuss some aspects of worst case execution time estimation. Chapters 4 and 5 are introductory to general loop and basic block theory, which are very common in compiler theory.

Background analysis of PLEX and the compiler The chapters in the second part relates the theory described in part I to PLEX and also give a description of the PLEX compiler and those steps performed by the compiler that are relevant for this thesis. Some basic properties of PLEX are discussed in chapters 6 to 8 and 10. In chapter 9, the structure of the (PLEX) compiler and some of the tasks of the back-end are presented.

WCET Estimation and implementation is a description of our contribution to the analysis of PLEX programs. Chapter 11 is a presentation of our work while chapter 12 is an evaluation of our approach.

The report ends with a summary and a description of (possible) future work.

To get a short introduction and understanding of our work, we recommend the reader to read chapters 11 and 14.

The uninitiated reader should read part II to part IV to get a good picture of our work. Part II is an introduction to the theories in real-time, worst case execution time calculation (WCET) and basic compiler theory.

For the initiated reader who has knowledge of compiler theory, chapters 6 to 10 can be of interest. Here the basic theories in PLEX are discussed. Also chapter 3 can be interesting for those not initiated in real-time and worst case execution time calculations.

Chapter 2

Problem definition

Calculating the *Worst Case Execution Time*, WCET, of a (real-time) program is a field of intensive research. The research, however, is mainly focused on hard real-time systems where calculating a correct WCET is critical to the systems correctness. The methods developed in this field could be applied on soft systems (e.g. AXE) as well, but with soft real-time systems the demands are different. In soft real-time programs, it is often sufficient to get a picture (or an estimation) of the run time of the program and individual jobs[1] than calculating a safe WCET. Estimating the run time of a program enables a better control of different series of jobs that each are servicing different requests at the same time (which is the case in a system like AXE). Also, identification of the possible loops is an important issue for the analyses since one job should not take a disproportionate amount of time. Even if it may be theoretically impossible to give an exact execution time for a loop, the run time for one iteration is useful information. What methods should be used if we want to provide developers with a picture of the program behavior?

The AXE10 software system comprises approximately 10 milion lines of code spread over circa 1000 executable units.[MH01]. To get an overall picture of the entire system, graphical representation is a possible approach. [AGG99] has developed a tool for representing the blocks in the AXE system graphically. But to be able to do this, some basic information is needed; The signals sent and received in a block. This information makes it possible to relate the blocks to each other. This is done by extracting information

[1] A job is defined in section 7.1

from the control-flow graph which is generated by the compiler (see 11.3).

Part II

General background theory for Worst Case Execution Time estimation

Chapter 3

Foundations for Worst Case Execution Time estimation

This section will treat some background on execution time analysis in general and discuss some of the related questions.

3.1 The importance of Execution Time Analysis

For a *Real-Time* system, functional correctness is equally important as temporal correctness. What this means is that not only is the correctness of the system dependent on correct computations. Equally important is that the answers are delivered within a predictable time. In other words, *predictability* is one of the key words in a real-time system[1].

To achieve a predictable system, execution time analysis is one of the fundamental parts. The simple reason for this is that if we can not tell for how long a piece of code will execute, we can not guarantee that the answer will be delivered before the deadline[2].

There are different kinds of *real-time* systems, namely *hard* and *soft*. What it means by a *hard real-time system* is that the deadlines can not be violated. This may lead to catastrophic consequences while the deadlines in a *soft real-time system* can be violated on a few occasions without endangering the system or the surroundings.

[1] which is the case with AXE/PLEX [AGG99]

[2] The deadline of an answer is the specified time that the answer must be delivered within. If an answer is delivered after its deadline, we have no use for that answer. This is thoroughly described in several books, one example is [But97]

Some of the more important factors that have an impact on the execution time, are [Gus00]:

- The input data.
- The behavior of the program (as defined by the program source code).
- The compiler (the mapping from the source code to the object code).
- The processor, memory and other hardware.

The ultimate goal in execution time analysis, is to find out exactly how long it takes to execute a piece of code in all different cases. This however is very difficult, if not impossible, due to a number of reasons (e.g. hardware that is very hard to predict how they behave (pipes, caches and out-of-order exertion etc) and the fact that a program can consist of n lines of code and the execution paths are x^n, where n can be several million). One approach is to run the program and see how long it takes for different input. The problem here is to find the appropriate input (relevant to the program) and to know if all possible paths have been found.

The approach to calculate the worst execution time (WCET) is a safe but not perfect way to find the WCET. The calculations are conservative[3]. Figure 3.1 shows the difference between the actual times and the calculated. The different times are: $WCET_C$ - Calculated WCET, $WCET_A$ - Actual WCET, $BCET_C$ - Calculated Best Case Execution Time, $BCET_A$ - Actual BCET, AVET - Average Execution Time

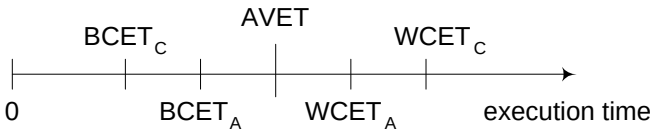

Figure 3.1: *Different times in WCET-calculations.*

[3]We do not guarantee too much. Sometimes the WCET-calculation shows that it is going to miss its deadline when in fact it is not.

3.2 Decisions on Execution Time Analysis

Execution time analysis is usually divided into two parts: *high-level analysis* and *low-level analysis*[4]. In a general way, we can say that low-level analysis means counting the number of clock cycles for each assembly instruction and taking into account how the hardware acts (i.e. pipes, caches), while high-level analysis deals with things like finding all the loops in a given program and to estimate the maximum number of iterations in the loops. So, the first decision that has to be made is if the analysis should work on a high- or low-level.

Another decision that has to be made concerns the code level. Should the analysis work on source code level, on intermediate level or on assembly level? Each level has its own advantages (and disadvantages). Working on assembly (or object) code level has the advantage that the analysis is working with the final[5] code. The disadvantage is that it could be difficult to understand what the programmers intention is and to give feedback to the programmer. Working on an intermediate level has the advantage that the analysis could be independent of the source code (at least up to some point) and that some optimizations are performed on the code. The disadvantage is that this approach demands a close co-operation with the compiler and that many compilers, in an optimization attempt, can change the structure and execution characteristics of the program by something called code motion[6]. Finally, working on the source code level has the advantage that much of the program information is explicit on this level and it is also easier to give feedback to the programmer. The obvious disadvantage is that parts of the program may never be executed or that parts of the program may be syntactically wrong[7].

A third point concern loop detection. One way is to use syntactic structures (e.g. **for, while**, etc.), another to use flow-graphs. Using flow-graphs is a more general method since it can detect loops in programs that use GOTO statements.

---

[4]This division is also described in [AGG99]

[5]Final in the meaning that optimizations are performed and the current code is the one that will really execute.

[6]Code motion is a technique to move code from inside a loop to outside the loop in an attempt to minimize the execution time of the loop.

[7]The above advantages and disadvantages is mainly from [Gus00]

Chapter 4

Loops and SCC:s

In static program analysis, and calculation of the worst case execution time, the detection of loops is an important topic. The reason for this is that the total running time of the program, as well as other characteristics, is mainly affected by loops (and the number of iterations in the loop). But what is a loop?

In data processing, a basic concept in computer programs. A part of a program is repeated, iterated, a number of time until a logical expression[1] *changes its value.* [NE93]

This is, however, only a brief/general description of what a loop is and in this chapter, we will study the loop concept in more detail (since detection of loops is one of our primary goals). But before we do that, we need to give some definitions.

4.1 Definitions and Notations

- *Dominators* A node x of a flow-graph is said to *dominate* a node y, x *dom* y, if every path from the initial node of the flow-graph to y goes through x. (By this definition, every node dominates itself and the entry of a loop dominates all nodes in the loop[2]). [ASU86]

- *Strict dominator* Node x strictly dominates y, denoted by x *stdom* y iff[3] x *dom* y and $x \neq y$. [SGL96]

[1] True or False

[2] i.e. if the loop is a natural loop, see 4.3

[3] if and only if

- *Immediate dominators* A node y is said to immediately dominate another node x, denoted by $y = idom(x)$, if y *stdom* x and if there is no other node z such that y *stdom* z *stdom* x. [ASU86, SGL96]

- *Back edge* A back edge is an edge whose head dominate its tail. (If $a \rightarrow b$ is an edge, b is the head and a is the tail.) [ASU86]

The dominance relation is reflexive and transitive and can be represented by a tree, called the *dominator tree*.

- *DJ Graph The DJ graph of a flow-graph combines the flow-graph and its dominator tree in a single representation*[SGL96].

  The vertices in the DJ graph are the same as in the flow-graph and the edges in the DJ graph are of two types: D and J edges. D edges are dominator tree edges (i.e. they point out the dominance relation among the nodes/vertices) and J edges are edges in the flow-graph (e.g. $x \rightarrow y$) which fulfills the condition $x \neq idom(y)$. (J edge is an abbreviation for join edge.) Constructing a DJ graph of a flow-graph is very straightforward:

  First compute the dominance relation (and build the dominance tree) of the flow-graph. Algorithms for computing the dominance relation in a given graph is given in for instance [ASU86](algorithm 10.16, pps 670-671) and [App98]. Then, for each J (join) edge $x \rightarrow y$ in the flow-graph, insert the edge $x \rightarrow y$ in the dominator tree.

  (An alternative approach for constructing the DJ graph of a flow-graph is by inserting the D edges into the flow-graph.) When the DJ graph is constructed, the J edges are determined as one of two kinds: back J (BJ) edges and cross J (CJ) edges. A BJ edge ($x \rightarrow y$) is a J edge which fulfills the condition y **dom** x. If not, the J edge is an CJ edge.

- A subgraph of a graph G is a graph H whose vertices[4] and edges are all in G. [GY99]

- A subgraph H is said to *span* a graph G if $V_H = V_G$[5]. [GY99]

- A *spanning tree* of a graph is a spanning subgraph that is a tree. [GY99]

[4] vertices = nodes

[5] The notation V_G is used for the vertex-set of the graph G

- *Depth-First Ordering* The depth-first ordering is created by starting at the initial node and searching the entire graph, trying to visit nodes as far away from the initial node as quickly as possible (*depth first*). [ASU86]

- *The depth-first spanning tree* The depth-first spanning tree S of a graph G is the spanning tree of G with root(S) = initNode(G). [GY99]

The depth-first spanning tree (of a flow-graph) can be derived by performing a depth-first search on the flow-graph. This search classifies each edge $x \rightarrow y$ in the flow-graph as one of four types [SGL96]:

- *sp-back* edge, if $y = x$ or y is an ancestor of x in the spanning tree.

- *sp-tree* edge, if x is the parent of y in the spanning tree.

- *sp-forward* edge, if x is an ancestor but not the parent of y in the spanning tree.

- *sp-cross* edge, if x and y have no ancestor-descendant relationship.

Besides the given definitions, the term *reducibility* is used frequently in the literature (when discussing flow-graphs and loops). We devote section 4.5 to this concept.

4.2 Strongly Connected Components - SCC's

The most general looping structure of a flow-graph is a *strongly connected component*, which is a subgraph $G_s = (N_s, E_s)$[6] such that every node in N_s is reachable from every other node by a path that includes only edges in E_s [Muc97].

4.3 Natural Loops

A natural loop has two essential properties [ASU86]:

- The loop must have a single entry point, called the "header". This entry point dominates all nodes in the loop, or it would not be the sole entry to the loop.

[6] N=the nodes in G, E=the edges in G

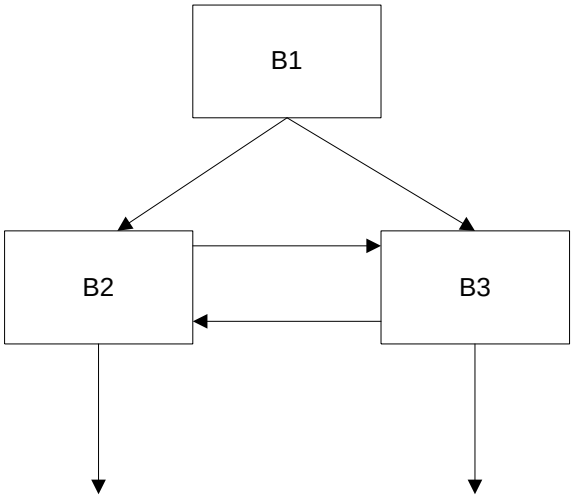

Figure 4.1: *An example of a multi-entry loop. The picture is from [Muc97]*

- There must be at least one way to iterate the loop, i.e., at least one path back to the header.

Another way of saying this is that given a back edge $m \rightarrow n$, the *natural loop* of $m \rightarrow n$ is the subgraph consisting of the set of nodes containing n and all the nodes from which m can be reached in the flow-graph without passing through n and the edge set connecting all the nodes in its node set. Node n is the *loop header*.

4.4 Other types of Loops

Although other types of loops, *non-natural loops*[7], seldom occur in practice, they do exist which means that they must not be forgotten in the discussion. One example of such a construct is the *multi-entry loop* in fig 4.1.

In section 11.6 we show that we can handle these kinds of loops. In optimization theory these constructs has the property that they make their flow-graphs irreducible (see 4.5).

[7]Strongly connected components that are not natural loops. (Our definition)

4.5 Reducibility

Reducibility is a very important property of a flow-graph. The term *reducible* results from several kinds of transformations that can be applied to flow-graphs that collapse subgraphs into single nodes and reduce the flow-graph successively to simpler graphs. A flow-graph is considered to be reducible if applying a sequence of such transformations ultimately reduces it to a single node.

Formally, a flow-graph G = (N,E) is reducible iff E can be partitioned into disjoint sets E_F, the *forward* edge set, and E_B, the back edge set, such that (N,E_F) forms a DAG (Directed Acyclic Graph) in which each node can be reached from the entry node, and the edges in E_B are all back edges as defined in 4.1.

Another way of saying this is that if a flow-graph is reducible, then all the loops in it are natural loops characterized by their back edges and vice versa. It follows from this definition that in a reducible flow-graph there are no jumps into the middle of loops - each loop is entered only through its header [Muc97].

To partition the edges into the two subsets *forward* and *backward*, one can perform a depth-first search on the flow-graph, which will classify the edges into sp-back, sp-tree, sp-forward and sp-cross edges as mentioned in 4.1.
But, the depth-first ordering has one important property: It is ***not*** unique [Muc97]! Does this matter, then? The answer, actually, depends on the loop. If the loop is reducible, no matter what the depth-first ordering is, the same set of sp-back edges will be found and, for a reducible graph, all sp-back edges are back edges. This property allows for a straightforward way to determine the body of a reducible loop since the entry node of a reducible loop is unique (i.e. it dominates every node in the loop)[8].

If, on the other hand, the flow-graph contains irreducible loops, different depth-first search orderings **can** result in different sets of sp-back edges. This means that the body of an irreducible loop can not be found in the same way as the body of a reducible loop. The reason for this is that an irreducible loop contains more than one entry node and that an entry node of an irreducible

[8]An algorithm for constructing the natural loop of a back edge is given in [ASU86](Algorithm 10.1, page 604)

loop does not dominate *every* node in the loop.[9]

[9]According to this, the multi-entry loop in fig 4.1 is an example of an irreducible loop.

Chapter 5

Basic Block

To use basic blocks is a common technique in compiler theory. The code is broken down into basic blocks in the back-end of the compiler. These are used to build a flow-graph which in turn is used to analyze the control flow and also the data flow. The control flow analysis is used for optimization and the data flow analysis is used for assigning variables to registers.

The basic blocks are the nodes in a flow-graph, which "explains" the program structure. First we give a formal definition of a flow-graph [Wol92]: *A flow graph is the tuple* $G = \langle V,\ E,\ Entry,\ Exit \rangle$*, where* V *is the set of vertices corresponding to basic blocks, (alternatively, the vertices in* V *may correspond to statements or operations),* E *is a set of directed edges* $E = \{(m,n) | m,n \in V$ *and* m *is a flow predecessor of* $n\}$*,* $Entry \in V$ *and* $Exit \in V$ *are distinguished vertices. We will write* $m \rightarrow n$ *to mean there is an edge* $(m,n) \in E$*. A path in a flow-graph from vertex* m *to vertex* n *is a length* k *sequence of vertices* $(v_0, v_1, v_2, \ldots, v_k)$*, with* $v_i \in V\ \forall i, 0 \leq i \leq k$ *and* $v_0 = m$ *and* $v_k = n$*, such that* $v_0 \rightarrow v_1, v_1 \rightarrow v_v, \ldots, v_{k-1} \rightarrow v_k$*. By convention, we say there is always a lenght-0 path from a vertex to itself.* G *is connected with a path from Entry to any vertex* $n \in V$ *in* G *and with a path from any vertex* $n \in V$ *to Exit in* G*.*

The definition of basic blocks is [Muc97]:

(Formally) *"A maximal sequence of instructions that can be entered only at the first of them and exited only from the last of them. Thus, the first instruction in a basic block may be the entry point of the routine, a target of a branch, or an instruction immediately following a branch or a return. Such*

instructions are called leaders. To determine the basic blocks that compose a routine, we first identify all the leaders and then, for each leader, include in its basic block all the instructions from the leader to the next leader or the end of the routine, in sequence."

(Informally) *"A straight-line sequence of code that can be entered only at the beginning and exited only at the end."*

To visualize what *leaders* and *enders* are we give a general example below:

- Leader
  A general example in C:

```
x = 2;          /* first statement -> LEADER*/
y = x + 1;

if(z < x) {
   y = y + 1;   /* statement following IF -> LEADER*/
   x = x - 1;
}     /* end if */

z = 0;          /* statement following IF -> LEADER*/
m = z + x;
```

- Ender
  A general example in C:

```
x = 2;
y = x + 1;

if(z < x) {     /* conditional jump -> ENDER */
   y = y + 1;
   x = x - 1;   /* last statement before the
                   next basic block -> ENDER */
}     /* end if */

z = 0;
m = z + x; /* last statement -> ENDER */
```

For each *leader*, its basic block consists of the *leader* and all statements up to and including the *ender* or the end of the program.

When the flow-graph is constructed (from the basic blocks), the relations *predecessor* and *successor* are introduced. A *predecessor* is the basic block that is preceding the current, and the *successor* is the basic block following the current basic block. A branch node has more than one *successor* while a join node has more than one *predecessor*. Or formally (we write the edge from a to b as $a \rightarrow b$):

$$G = \{< N, E > \mid N \textit{ is a set of nodes}, E \textit{ is a set of edges such that } E \subseteq N \times N\}$$

$$Succ(b) = \{n \in N \mid \exists e \in E \textit{ such that } e = b \rightarrow n\}$$

$$Pred(b) = \{n \in N \mid \exists e \in E \textit{ such that } e = n \rightarrow b\}$$

To create a graph which can be analyzed for the whole block, two extra nodes are introduced into the graph. These are the INITIAL and TERMINAL[1] nodes.

The purpose of the INITIAL node is to represent a single point of entry into the graph. Similarly the purpose of the TERMINAL node is to get a single point of exit from the graph[2].

5.1 Example

For better understanding the flow-graph, an example is given:

Consider the following C-code which computes, for a given $m \geq 0$, the m^{th} Fibonacci number.

[1] In compiler literature mostly called the ENTER and EXIT node. Here (in PLEX) the ENTER and EXIT node have a different meaning, e.g. the ENTER/EXIT statement.

[2] this is further discussed in the Basic Block in PLEX section 10

```
unsigned int fib(m) /* m is a unsigned int */
{
   unsigned int f0=0, f1=1, f2, i;
   if(m <= 1) {
     return m;
   } else {
     for(i=2; i<=m, i++) {
        f2=f0+f1;
        f0=f1;
        f1=f2;
     }
   }
   return f2;
}
```

From this code we get a graph (fig 5.1).

If we convert this into a flow-graph, where the nodes constitutes of basic blocks, with additional entry (INITIAL) and exit (TERMINAL) nodes (to get a graph with only one entry and exit), we get a flow-graph (shown in figure 5.2).

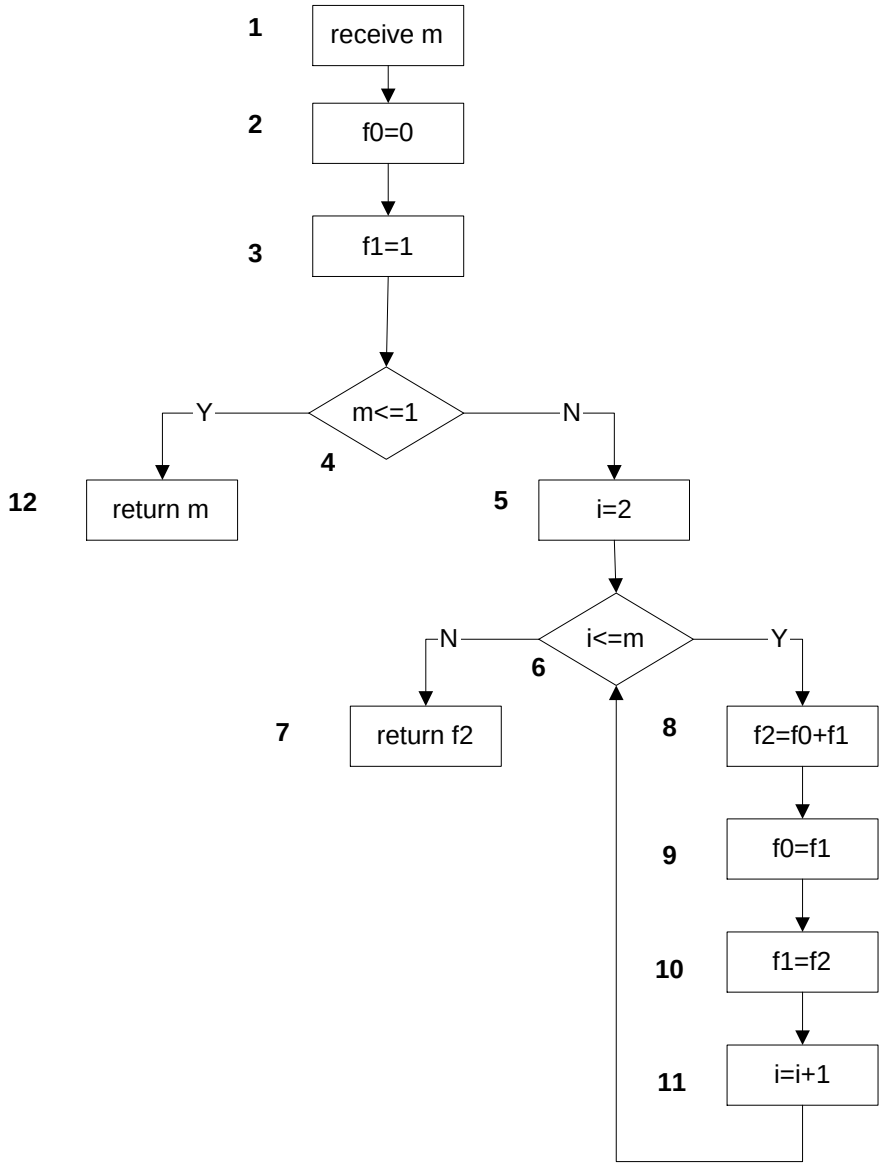

Figure 5.1: *The graph for the Fibonacci code.*

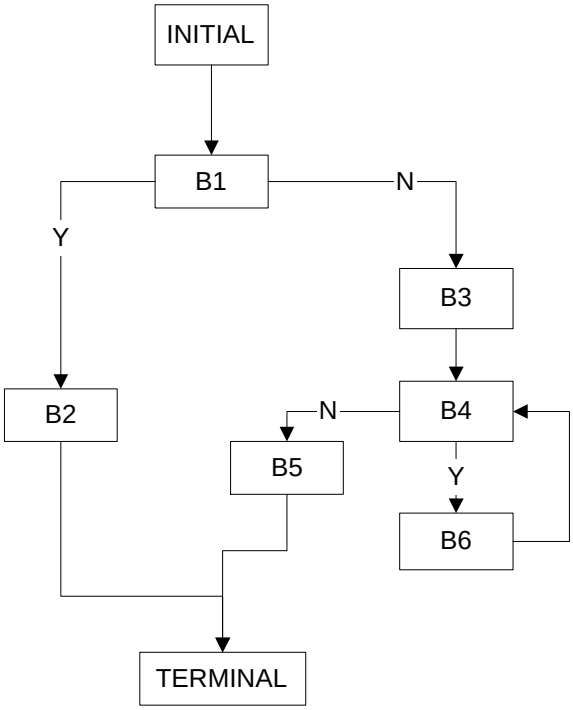

Figure 5.2: *The flow-graph for fig 5.1*

Part III

Background analysis of PLEX and the compiler

Chapter 6

Signals in PLEX

6.1 Signals and Block Interaction

Signals are very important in PLEX because these make up the back-bone of the entire system. All communication is handled by signals, everything is more or less dependent on signals. A job (an execution from start till end) is invoked by a signal (see figure 6.1). A job may run in only one block or even in only one code-item although it usually traverses lots of different pieces of code in different blocks. Transitions from one block to another is handled by signals of different kinds such as e.g. direct unique, direct multiple, combined forward and backward (see figure 6.1).

The signal concept in PLEX/AXE is described in [AGG99]. We will recall that the different function blocks in AXE communicate by the means of signals and that the signals could be of different types:

Signal Type		**Direct**	**Buffered**
Single	unique	***X***	***X***
	multiple	***X***	***X***
Combined	unique	***X***	
	multiple	***X***	

Table 6.1: *The different types of signals in PLEX. Marked boxes in the table indicates a legal combination, while unmarked boxes indicates illegal ones*

Unique / Multiple signals: A unique signal (figure 6.2) can only be received by one particular block, while multiple signals can be received

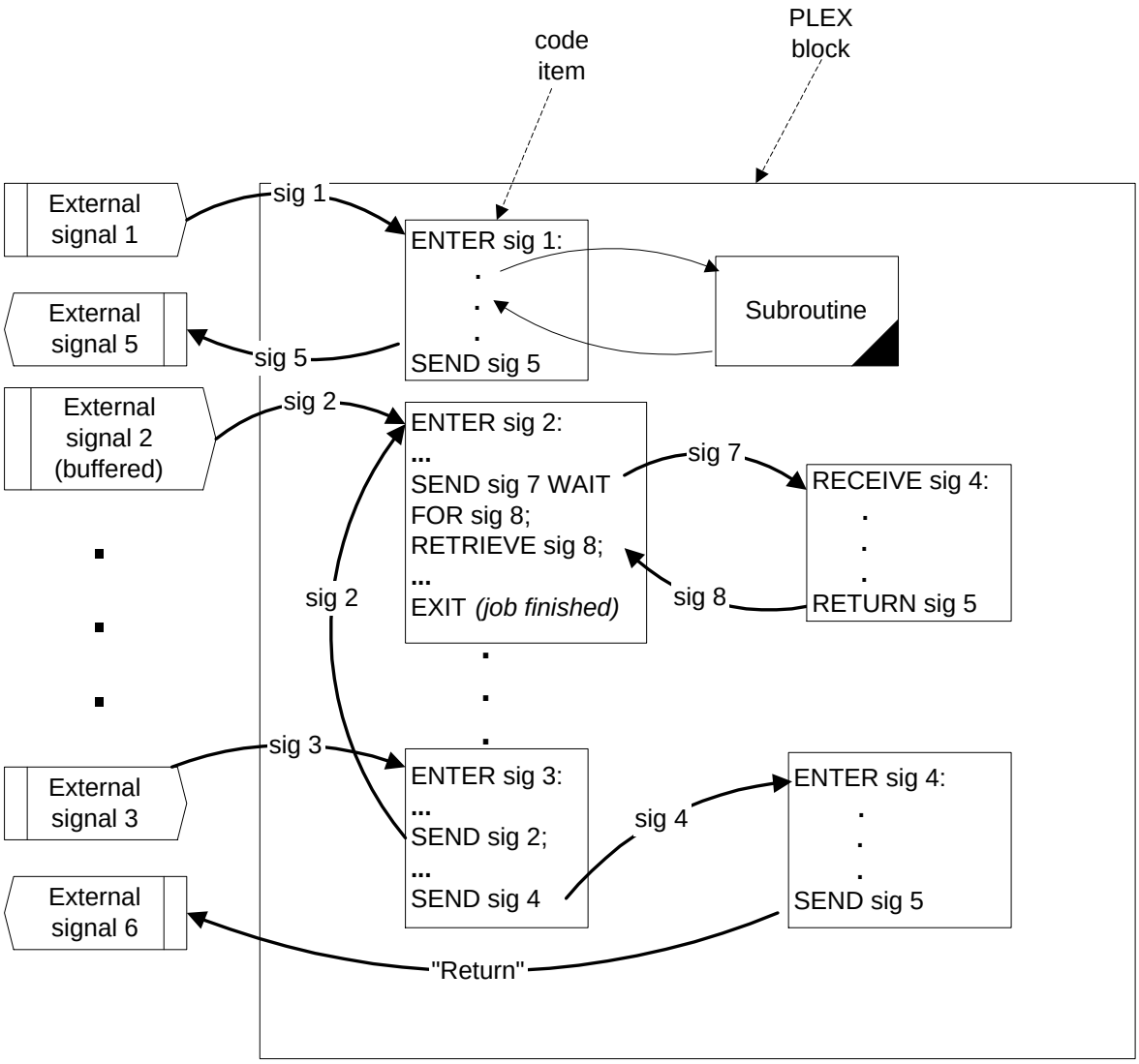

Figure 6.1: *The execution of a job. Some different kinds of communications within and between blocks. Signals 1,3,4,5,6 could be either direct or buffered, signal 2 is buffered (execution continues after SEND) and signals 7 and 8 are combined. The jumps to and from the subroutine are not signals but jumps.*

by several blocks. (However, it is not possible to send a multiple signal to more than one block simultaneously, only to choose the block to which the signal will be sent. This is done with REFERENCE in the signal sending statement.)

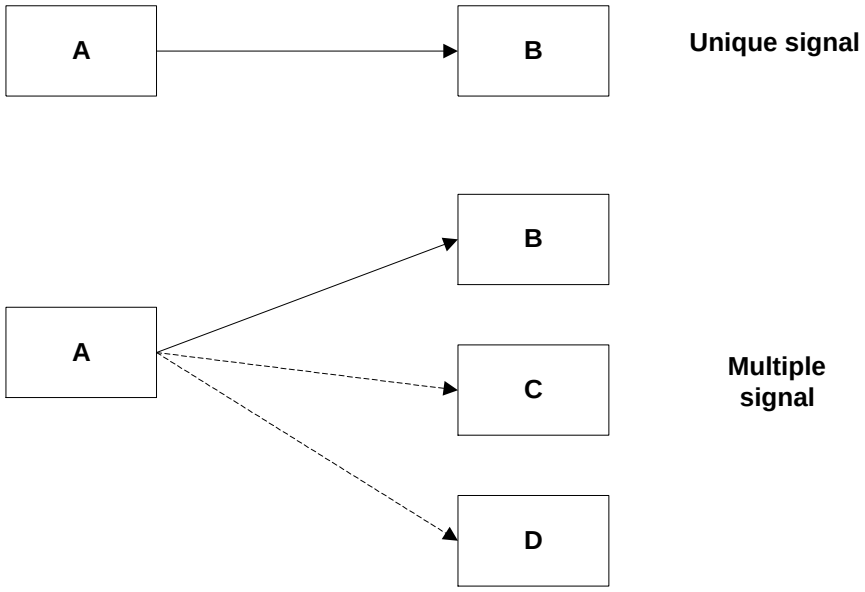

Figure 6.2: *Unique and multiple signal.*

Direct / Buffered signals: Direct signals are sent immediately to another block while buffered signals (figure 6.3) are queued in a job buffer[1]. The meaning of this is that by a direct signal, the programmer maintains control over the execution sequence, while with buffered signals, the control is returned to the operating system.

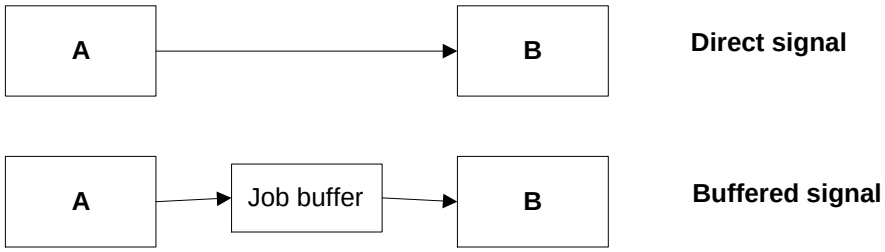

Figure 6.3: *Buffered and direct signal.*

Single / Combined signals: The difference between these two types is that a combined signal (figure 6.4) demands an immediate answer, while single signals do not require such feedback. For this reason,

[1]Actually, there are different kinds of job buffers.

combined signals can never be buffered (as shown in table 6.1).

(The signal descriptions from [AB98]).

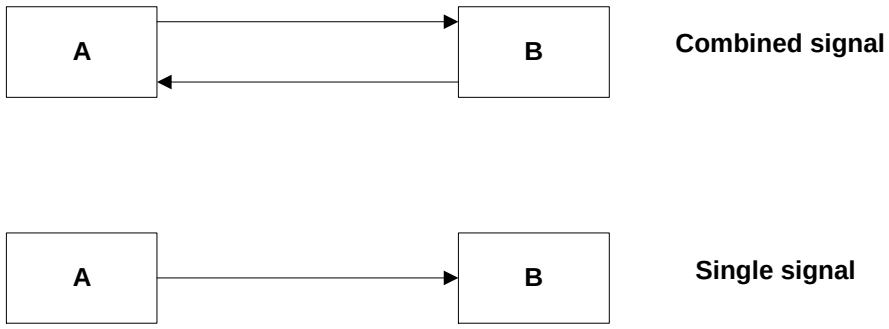

Figure 6.4: *Combined and single signal.*

Besides these types of signals, there are some constructs of signals that have an impact on the path of execution.

- A buffered signal can be made direct (implicitly) by adding the keyword **HURRY** to the signal sending statement. This means that the signal is sent as a 'direct buffered signal' which can be seen as a temporarily direct signal that is usually buffered.
- **LOCAL** signals are always direct and sent within the same block. It is very similar to a **GOTO** statement. A **LOCAL** signal can be received by a PLEX sector or by an ASA[2] sector[3].
  LOCAL PLEX signals are sent in the same manner as 'usual' signals, i.e. with **SEND** and **RECEIVE** while LOCAL signals to ASA sectors are sent with **TRANSFER** and **ENTRANCE**.

  If the local signal is sent to another PLEX sector, the receiving sector is linked as a successor to the senders basic block (as if it were a GOTO).

There are also different kinds of signals in respect to where they are used. Some signals are used between functions in central processor units [4].

[2]The assembler code used.
[3]signals sent to ASA sectors will not be handled in this thesis.
[4]called CP-CP signals

Others are sent from regional processor units to the central processor units[5] or between regional processor units[6].

CP-RP signals and RP-CP signals enables concurrency and are explicit. AXE contains many regional processors so RP-RP signals may result in concurrency if they are different regional processors.

[5]RP-CP/CP-RP signals, these will not be handled in this thesis.
[6]RP-RP signals, these are very rare and will not be handled here.

Chapter 7

Code item

7.1 Code-Items and Jobs

A *code-item* starts with a signal receiving statement and ends with an EXIT statement. (The code that is executed between these two statements belongs to the code-item.) Code-items are sometimes referred to as *subprogram*. A *PLEX source code file* consists of one or several code-items. A skeleton of a PLEX source code file is given in table 7.1. Here, the code-items are:

CI_1 = ENTER SIGNAL1; ...SEND SIGNAL2; EXIT;,
CI_2 = ENTER SIGNAL3; ...SEND SIGNAL4; EXIT;,
CI_3 = ENTER SIGNAL5; ...SEND SIGNAL6; EXIT; and
CI_4 = ENTER SIGNAL7; ...EXIT;

The PLEX source code file constitutes a SPI (Source Program Information). The SPI, together with a SS (Signal Survey), which is a description of the signals sent and received, forms a PLEX unit. (Actually, a PLEX unit also consist of a parameter list. However, the parameter file is added after the control flow has been analyzed in the back end of the compiler. See fig 9.1 in section 9.1.) It is the PLEX unit (i.e. the SPI (Source Program Information) and the SS (Signal Survey)) that is analyzed in the compiler.

A *Job* is defined in the following way: A job is a continuous sequence of statements executed in the processor. A job begins with an ENTER statement for a buffered signal (or with a COMMAND statement) and ends with an EXIT. A job may send zero or more buffered signals. So, a job consists of one **or** several code-items.

```
PROGRAM; PLEX;
                        ENTER SIGNAL1;
                        ...
                        SEND SIGNAL2;
                        EXIT;

                        ENTER SIGNAL3;
                        ...
                        SEND SIGNAL4;
                        EXIT;
                        CUSELESS = 0;
                        ENTER SIGNAL5;
                        ...
                        SEND SIGNAL6;
                        EXIT;
                        ENTER SIGNAL7;
                        ...
                        EXIT;
                        ...
END PROGRAM;
```

Table 7.1: *A PLEX source code file consists of code-items.*The row 'CUSELESS = 0;' will never be executed because it is not inside an ENTER-EXIT block.

Chapter 8

Loops and SCC's in PLEX

In section 4 we discussed loops and similar structures in general, but what about loops in PLEX? We have found three different ways to construct a loop in PLEX. By iterations, by jump statements and by signals. (All three will be discussed below.)

8.1 Iterations

PLEX offers three different statements for iteration; **ON**, **FOR ALL** and **FOR FIRST**. All of them behave like ordinary FOR-loops in other programming languages. In PLEX, iterations are used for scanning files or indexed variables between given start and stop values. The ON-statement is the only one that can scan from the highest to lowest, or from the lowest to the highest value. Both FOR-statements scan from the highest to the lowest value. The difference between the two FOR-statements is that in FOR ALL it is possible to state a condition under which the DO-part is executed, where in the FOR FIRST it is mandatory to give the condition. Another difference is that the FOR FIRST-statement is left as soon as the condition is met. A comparison between the three statements is shown in table 8.1.

8.2 Jump statements

The second approach for constructing loops in PLEX is by using jump statements[1]. There are two types of jump statements:

[1]The jump have to be made backwards, otherwise there is no loop.

Criterion	**ON**	**FOR ALL**	**FOR FIRST**
Ascending or descending order	Yes	Always descending	
Several statements in action part	Yes	No, except in statement blocks, IF, CASE and loop statements	
Condition in iteration statements	NO	Possible	Always
Iteration ends after matching condition once and handling one individual	Not applicable	No	Yes
Iteration variable/pointer after loop	Undefined	Undefined	Defined if matching individual/condition
Generates high-speed loop	No	Possible	Possible

Table 8.1: *PLEX-C iteration statements - a comparison.*

- Unconditional jump statements
- Conditional jump statements

The unconditional jump statement always perform a jump to the indicated program label, where the conditional jump statement first check if the given condition is met (and, if so, performs the jump).

Since no code expansion is performed on jump statements, we show their syntax below (with the unconditional jump statement specified first):

- $\left\{ \begin{array}{l} \textbf{GO TO} \\ \textbf{GOTO} \end{array} \right\}$ *label*;
- **IF** [**NOT**] *condition* [**PROCEED ELSE**] $\left\{ \begin{array}{l} \textbf{GO TO} \\ \textbf{GOTO} \end{array} \right\}$ *label*;

It should also be noted that there is another, more flexible, conditional IF statement which can be used to construct loops. (As can be seen in its syntax, this depends on its action part.)

IF [[**NOT**] *condition* **THEN** *sequence of statements*]$^{1...}$ **ELSIF**
[**ELSE** *sequence of statements*] **FI**;

8.3 Signals and Block Interaction

The signal concept in PLEX/AXE is described in section 6, here we only state that since the different function blocks in AXE communicate by the means of signals, signals may be used to create loops both between function blocks and internally within a function block.

Chapter 9

The Compiler

In section 3.2 we mentioned that when doing static program analysis (which estimation of the execution time are a part of) a number of questions has to be answered and we recall that one of these concern the code level. Then, in section 11.1.1, we will show that working with the intermediate code was a natural choice. This, together with the following quotation [Gus00]:
Analysis on the intermediate level obviously demands a close cooperation between the compiler and the $WCET_C$ tool.
demands a description of the PLEX-C compiler together with those sub parts that are of interest for us.

9.1 The internal structure of the PLEX-C compiler

The compiler consists of a separate front-end (FE), a back-end (BE) and an object code handler (MOT). These are divided into sub programs. The input to the compiler is PLEX-code and the output is a text-file that is loaded into the APZ[1]. A number of intermediate representations are used during the compilation (see fig. 9.1). Here the different parts of the compiler and some of the intermediate representations are explained briefly.

9.1.1 Front End

The input to the front-end is the PLEX code (which can be written directly by the programmer or compiled from HL-PLEX[2]), a signal survey (a list of

[1] the control part including the central and regional processors.
[2] High Level PLEX

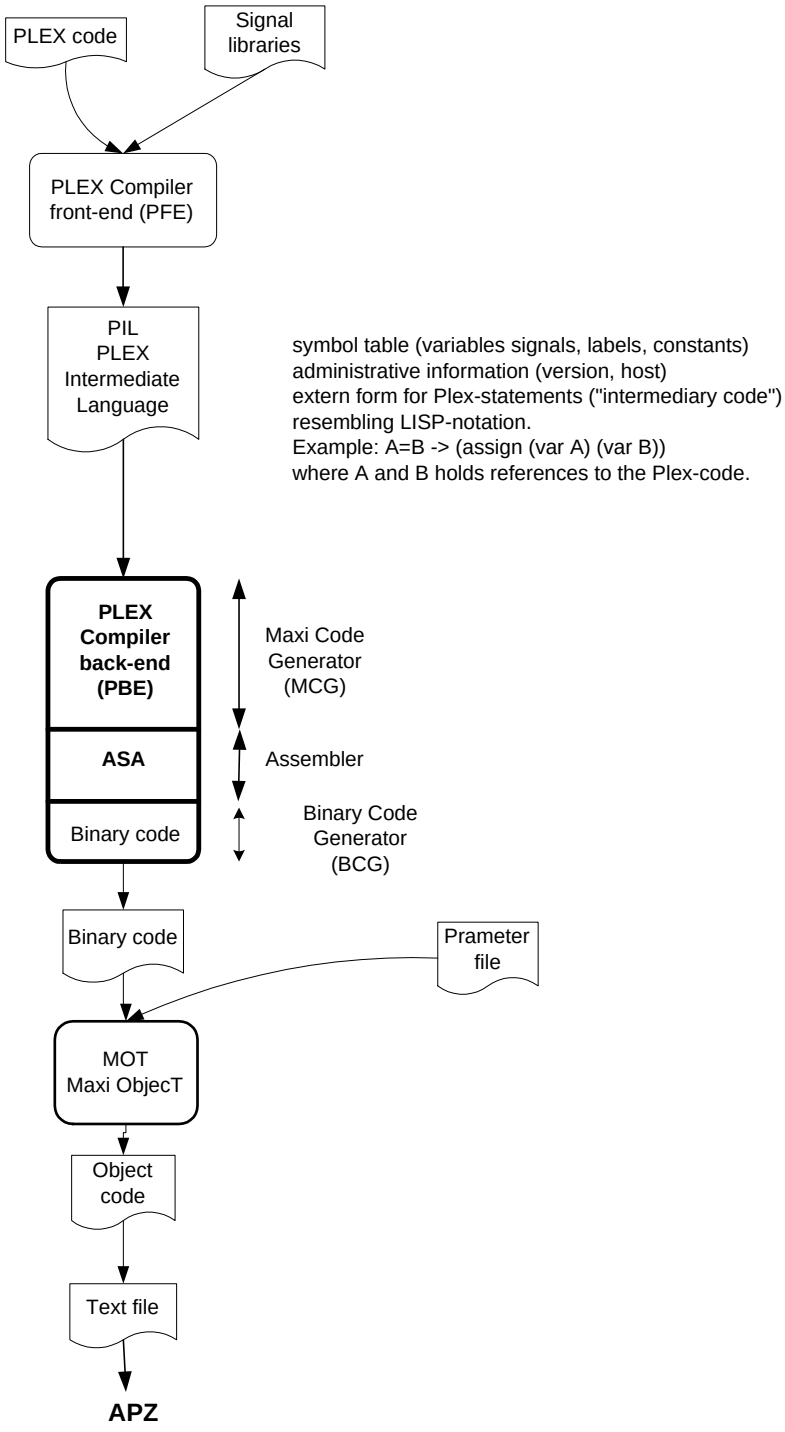

Figure 9.1: *The structure of the compiler. Input: PLEX code (written by the programmer or compiled HL-PLEX code)*

all signals a unit sends and receives in alphabetical order) and a parameter file. The output is a so called PIL-file (PLEX Intermediate Language). The structure of this file resembles LISP-syntax (for an example, see table 9.1).

The PIL-file holds:

- a symbol-table (holding information about e.g. variables, signals, labels and constants)
- administrative information (e.g. version, host)
- an intermediate code on extern form for PLEX-statements.

PLEX	*PIL - extern format*	*comment*
X = A + B	`(ASS` `  (ID X + + +)` `  (PLUS` `  (ID A <24 17> + +)` `  (ID B <24 21> + +)` `  <24 19> + +) + <25 7> +)`	*where A, B and X are references to the PLEX-code*
IF A = B	`(IF` `  (COND_LIST` `  (COND` `  (EQ` `  (ID A <13 14> + +)` `  (ID B <13 29> + +)` `  <13 27> + +))))`	*the numbers within '<>' are references to the source code position*

Table 9.1: *Examples of the PIL format and the matching source code (PLEX-C).*

9.1.2 Back End

The back end of the compiler takes the PIL-file as input. The structure of the back end system is a chain of different processes or "modules" (see fig. 9.2). The back end consists of two subsystems, the MCG (MAXI Code Generator) and the BCG (Binary Code Generator). The output from the MCG is input to the BCG, which converts ASA-code to binary code which is then handled by the MOT (Maxi ObjecT)[3] before being loaded down to the APZ.

The Maxi Code Generator – Module description

Here we shortly describe the different modules in the back end to show what the system looks like today. Some changes will be made in the work with our system. Changes will be made to the back-end (see 11.5). To get an understanding about why we do the changes we will do, it is good to know the structure of the system.

Each module is performed in different passes.

CX (Code generator eXecutive) The module supervises processing of the other modules within subsystem MGG. CX initiates the different modules in the MCG, therefore it can be seen as the “main” method for the MCG.

CI (Code generator Initialization) part of the MAXI CODE GENERATOR (MCG). The module reads the external form (PIL) of the program and block document. This module also initializes the symbol table with connected code attributes.

CE (Code generator Expansion module) The modules main task is to convert the statements of PLEX program sectors in PIL internal format to SPIL (Simple PIL) internal format. This is done for statement blocks, structured statements and expressions (see chapter 9.2).

All structured and nested statements are removed except special FOR statements. The statement blocks are inserted in-line or linked at the activation point. The structured statements (IF, CASE, ON, FOR) are for instance expanded to a sequence of simple statements (assignment,

[3]see section 9.1.3

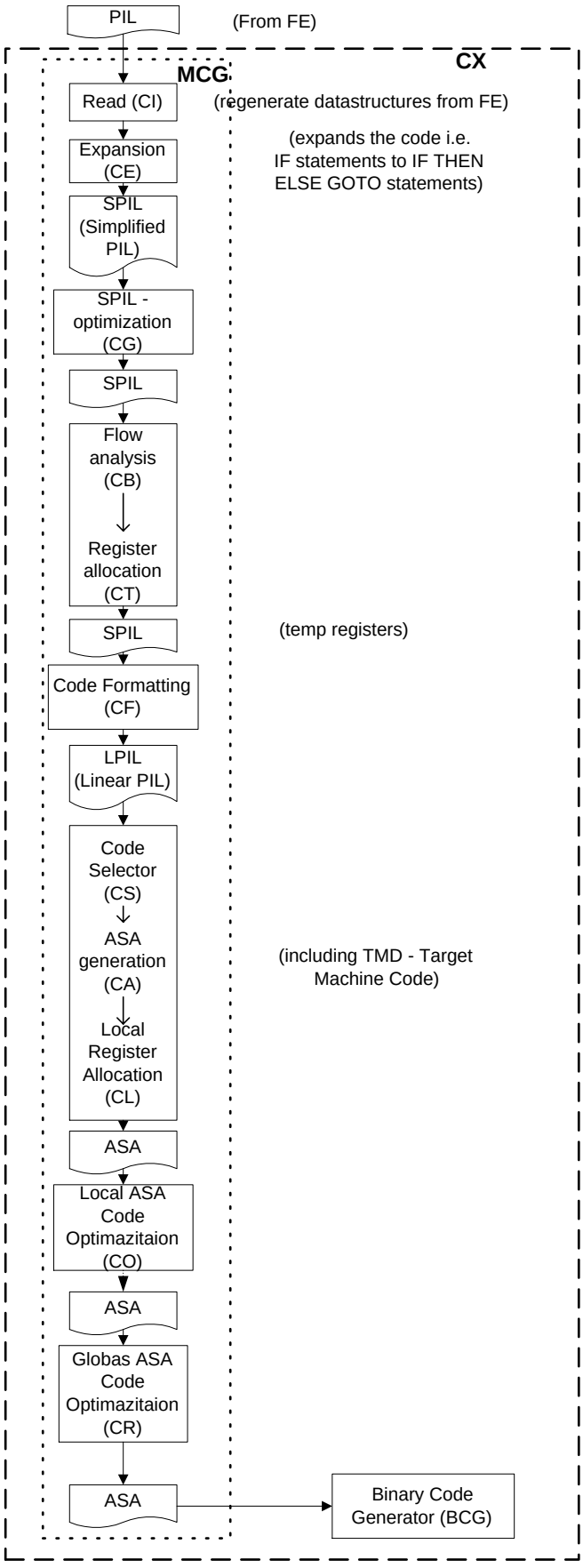

Figure 9.2: *The structure of the back end. Input: PIL*

GOTO, simple IF). Some other statements (JOBTABLE, CONVERT and Forlopp-statements) are expanded to a sequence of simple PLEX statements. The signals used in I/O-statements and Forlopp statements are introduced in the symbol table. The I/O-statements are expanded to signal statements. Some Forlopp statements are expanded to signal statements. The called ASA sectors are expanded to in-line code. Symbolic names in expressions are changed to their numerical values. Strings and string symbols are changed to generated variables. Global number symbols and expressions with global number symbols are changed to generated variables.

CG (Code Generator optimization module) Here simple and local (peephole) optimizations are made on SPIL (see chapter 9.3). Bad code generated in CE is removed and some code is replaced by more efficient code. The optimizations lead to fewer basic blocks (in CB) and the connections between them (predecessors/successors) are reduced.

CB (Code generator Basic block analyzer) This module constitutes the part in the MCG that performs the global flow analysis and invokes the Temporary Variable Allocator (the CT module) in order to obtain information on in which register each temporary variable resides in each domain in the PLEX program. The domains represent the PLEX program without any dead code.

Both global data flow analysis, using use-definition chaining and definition-use chaining, and control flow analysis, analyzing the basic blocks and creating a depth-first ordering, are performed here.

The CB module is invoked by the Code Generator Executive (CX module). Its input data is the SPIL code.

CT (Code generator Temporary variable allocator) The part of the MCG that performs the register allocation. The CT is invoked by the CB.

CF (Code Formatting) The SPIL format, which is a textual representation of a tree structure, is converted into LPIL (Linear PIL) format. The LPIL is a linear representation of a tree structure in prefix order.

CS (Code Selector) The main task of this module is to define the sequence of ASA assembler code to be generated by other modules. The

target machine description is a part of the specification module for a specified target machine.

CA (Code generator ASA intermediate form) ASA PIL format is generated. The module is called when some other module need to generate ASA PIL statement modes or statement lists.

CL (Code generator Local register allocator) Performs the local register allocation for indexes, pointers, expressions etc.

CO (Code generator local ASA code Optimization) Performs peephole optimizations on the ASA code.

CR (Code Generator Global ASA Optimizer) The global ASA optimizer module is divided into three major parts, basic block partitioning, propagation of the content of the registers all over the flow-graph and finally the optimization of the code using the produced information about the registers.

The Binary Code Generator

The main purpose of the Binary Code Generator subsystem is to generate binary code from the assembled internal records. The generated code may contain gaps for signal number, signal group number and global number symbols.

Furthermore the following functions are performed:

- The variables are allocated to base addresses.
- Label addresses and local signals are handled.
- Code-attributes for signal sending and signal distribution tables are created.
- Handles initial data.
- The binary code is assembled.
- The output file is created.

9.1.3 Maxi ObjecT

The main function of the Maxi ObjecT subsystem is to assemble load information. This information consists of data for program, reference and data memories in APZ and signal information.

The subsystem uses code generation result, MARSHAL information or signal survey, parameter list and BAT document as inputs. The purpose of this subsystem is the creation of a load file for an APZ together with list files describing initial data and signal information.

9.2 Code Expansion

The first things that are done in the back-end of the compiler, is to read the input (PIL format), which is done in the CI (see 9.1.2) module, and then to expand it to SPIL format, in the CE (see 9.1.2) module. As this is done, some PLEX constructs in the code are expanded (transformed). The constructs that are handled are ([LAB98]):

- EXPRESSIONS - the expressions in the statements are checked. Some global symbols and symbol values are replaced by their numerical values[4]. Some other variables and string symbols are also handled.
- LABELS - new labels are created from the old.
- Statement blocks - a decision is made for every statement block whether it is going to be linked or expanded in-line (copied).
- The following constructs are expanded to a list of statements:
  - **IF**
  - **FOR**
  - **ON**
  - BRANCH ON
  - CASE
  - SEND
  - ENTER
  - **DO Statements**

[4] this is done for some processors

- ASA Sectors
- I/O Statements
- JOBTABLE
- CONVERT

All the constructs above have an impact on the control flow. The constructs marked in bold text are the once that are interesting to us. These are especially interesting because they can be used to create implicit loops.

PLEX has three iteration, or loop, constructs: FOR FIRST, FOR ALL and ON (as was also mentioned in 8.1). These are expanded to IF-statements in most cases [5]. A loop can also include a DO statement which is a form of a subroutine.

Below the different expansions, that we are interested in, are described.

9.2.1 The IF statement

The IF statements are transformed into GOTO-statements as follows:

IF *conditional-expression* **THEN**
    statement-list
[**ELSE IF** *conditional-expression* **THEN**
    statement-list] ∗
[**ELSE**
    statement-list]
FI;

are expanded into:

[5] some IF, FOR and ON constructs are not expanded. See 9.2.1, 9.2.2 and 9.2.3

IF *conditional-expression_1* **GOTO** ⟨*label_1*⟩
IF *conditional-expression_2* **GOTO** ⟨*label_2*⟩
...
GOTO ⟨*label_k*⟩ `/* The ELSE label */`
⟨*label_1*⟩) *statement-list_1*;
GOTO ⟨*end_label*⟩;
⟨*label_2*⟩ *statement-list_2*;
GOTO ⟨*end_label*⟩;
...
⟨*label_k*⟩ *statement-list_k*; `/* The END clause */`
⟨*end_label*⟩;)

where the angle brackets ([]) denote conditional syntax and the asterisk (*) denotes that the construct can occur zero times or more. The ⟨*label_x*⟩ is a created label.

The IF-statement is however **not** expanded if it is on the form:

IF *conditional-expression* **GOTO** ⟨*label*⟩;

9.2.2 The FOR statement

This PLEX construct is actually two types of FOR statements, FOR FIRST and FOR ALL. Both statements are loops that scans from the highest to the lowest value (FROM - UNTIL). The difference is that the FOR FIRST loop ends after finding the first value that makes the condition[6] true while the FOR ALL loop goes through all values affecting those values that make the condition true. Note that the WHERE part is mandatory for FOR FIRST. The construct:

[6] <conditional-expr> is true OR the <variable> IS CHANGED TO <field-expr>.

FOR { **FIRST** / **ALL** } *control-entity*

FROM *start-expr*

[**UNTIL** *stop-expr*]

[**WHERE** { *variable* **IS CHANGED TO** *field-expr* / *condidional-expr* } { **GOTO** ⟨*label*⟩ / **DO** *statement* }]

is transformed according to the (generated) code layout below. The FOR statement is expanded as a loop with IF statements, except for two special cases.

Type 1

FOR { **ALL** / **FIRST** } *s* **FROM** *expr1* [**UNTIL** *expr2*]

{ **DO** *statement* / **GOTO** *label* };

Is replaced by the equivalent code:

var = *expr2*;
s = *expr1* + *1*;
⟨*loop*⟩)
IF *s* =! *var* **GOTO** ⟨*exit*⟩; or
IF *s* =! *cv* **GOTO** ⟨*exit*⟩;
s = *s* - *1*;
{ **DO** *statement* / **GOTO** ⟨*label*⟩ };
GOTO ⟨*loop*⟩; (not generated for **FIRST** case)
⟨*exit*⟩)
s = *s* - *1*;

Where '<var>' is a generated temporary variable and 'cv' is a constant value.

Type 2

FOR { **ALL** / **FIRST** } *s* **FROM** *expr1* [**UNTIL** *expr2*]
WHERE *lexpr relop rexpr*
{ **DO** *statement* / **GOTO** ⟨*label*⟩ };

Is replaced by the equivalent code:

var = *expr2*;
s = *expr1* + *1*;
⟨*loop*⟩)
IF *s* =! *var* **GOTO** ⟨*exit*⟩; or
IF *s* =! *cv* **GOTO** ⟨*exit*⟩;
s = *s* - *1*;
IF *not* (*lexpr relop rexpr*) **GOTO** ⟨*loop*⟩;
{ **DO** *statement* / **GOTO** ⟨*label*⟩ };
GOTO ⟨*loop*⟩; or (**ALL** case)
GOTO ⟨*done*⟩; (**FIRST** case)
⟨*exit*⟩)
s = *var* - *1*; or
s = H'FFFF; (if constant stop value)
⟨*done*⟩) (**FIRST** case)

Where '<var>' is a generated temporary variable and 'cv' is a constant value,

Type 3

FOR { **ALL** / **FIRST** } *s* **FROM** *expr1* [**UNTIL** *expr2*]

WHERE *v* **IS CHANGED TO** *cexpr*

{ **DO** *statement* / **GOTO** ⟨*label*⟩ };

Is replaced by the equivalent code:

var = *expr2*;
s = *expr1* + *1*;
⟨*loop*⟩)
IF *s* =*!* *var* **GOTO** ⟨*exit*⟩; or
IF *s* =*!* *cv* **GOTO** ⟨*exit*⟩;
s = *s* - *1*;
IF *v* = *cexpr* **GOTO** ⟨*loop*⟩;
v = *v* - *1*;
IF *v* =! *cexpr* **GOTO** ⟨*loop*⟩;

{ **DO** *statement* / **GOTO** ⟨*label*⟩ };

GOTO ⟨*loop*⟩; or (**ALL** case)
GOTO ⟨*done*⟩; (**FIRST** case)
⟨*exit*⟩)
s = *var* - *1*; or
s = H'FFFF; (if constant stop value)
⟨*done*⟩) (**FIRST** case)

Where '<var>' is a generated temporary variable and 'cv' is a constant value. The two special cases described below will not be expanded.

Special loop constructs

There are three kinds of FOR loops that are not expanded or tampered with at all. These are constructs that under certain conditions generate special

high-speed ASA instructions. In these cases, the FOR statement remains as it is after the code expansion. They are handled, the ASA code is generated, in the CS[7] module.

9.2.3 The ON statement

The ON statement can scan from either the highest value to the lowest or from the lowest value to the highest. Pointers are used as iteration variables when scanning files and field variables are used when scanning indexed variables. The construct:

ON *variable* **FROM** *field-expression_1*
{ **UPTO** / **DOWNTO** } *field-expression_2*
DO *statement-list*
NO;

is expanded in the following cases:

1. The loop variable is a temporary variable ascending through an interval (i.e. UPTO some expression). The new code is:

temp-var = *field-expression_2*;
variable = *filed-expression_1*;
IF (*variable* > *temp-var*) **GOTO** ⟨*exit*⟩;
⟨*loop*⟩)
statement-list;
IF (*variable* <= *temp-var*) **GOTO** ⟨*loop*⟩;
⟨*exit*⟩)

   where '<temp-var>' is a generated temporary variable.

2. The loop variable is a temporary variable descending through an interval (i.e. DOWNTO some expression). The new code is the same as in 1 except that the signs in the IF statements are changed:

[7] Code Selection, see 9.2 and 9.1.2.

'**IF** (*variable* > *temp-var*) **GOTO** ⟨*exit*⟩;'
⇒
'**IF** (*variable* < *temp-var*) **GOTO** ⟨*exit*⟩;'

and

'**IF** (*variable* <= *temp-var*) **GOTO** ⟨*loop*⟩;'
⇒
'**IF** (*variable* >= *temp-var*) **GOTO** ⟨*loop*⟩;'

3. The loop variable is a stored variable ascending through an interval (i.e. UPTO some expression). The new code is:

```
    temp-var = field-expression_2;
    variable = filed-expression_1;
    IF (variable > temp-var) GOTO ⟨exit⟩;
⟨loop⟩)
        statement-list;
        variable = variable + 1, ON CARRY GOTO ⟨exit⟩;
        IF (variable <= temp-var) GOTO ⟨loop⟩;
⟨exit⟩)
```

where '<temp-var>' is a generated temporary variable.

4. The loop variable is a temporary variable descending through an interval (i.e. DOWNTO some expression). The new code is the same as in 3 except that the addition is changed to a subtraction:

'*variable* = *variable* + *1*, **ON CARRY GOTO** ⟨*exit*⟩;'
⇒
'*variable* = *variable* - *1*, **ON CARRY GOTO** ⟨*exit*⟩;'

- *note 1* The ending IF statement is represented as an IF statement with code attributes. This statement is both an IF statement and an assignment.

- *note 2* If <field-expression_2> is a constant expression its value will be used directly instead of using '<temp-var>'. If both <field-expression_1> and <field- expression2> are constant expressions their values will be used directly instead of using <variable> and '<temp-var>'.

9.2.4 DO Statements

The DO statement is equivalent to subroutines (without input and output parameters) in other languages. There are two kinds of DO constructs (statement block and assembly language sector). We are only interested in the statement block construct. [8]

A DO statement block can be either linked or in-lined:

Linked No expansion is needed.

In-line For each place where the statement block code should be inserted, produce a new instance of the code containing unique labels.

If the DO statement is linked, it may be linked to multiple basic blocks. In these cases, the true execution paths *can be / are* lost. Figure 9.3 shows such a situation. The actual execution paths are $A \rightarrow C \rightarrow D$ and $B \rightarrow C \rightarrow E$. This information can not be seen in the flow-graph. In the analysis we do not know which paths are valid and which are not (i.e. $A \rightarrow C \rightarrow E$ and $B \rightarrow C \rightarrow D$). The problem is further discussed in section 13.

9.3 Optimization

Some statements are optimized in the Code Generator Optimization Module (CG). This is done to remove bad code generated by the CE[9]. The code improvement made by this module on SPIL reduce the number of basic blocks and the connections between them (predecessors and successors) which decrease the amount of the host computer memory needed and makes the execution time faster.

Optimizing on SPIL instead of ASA makes the optimization module machine independent and improve the generated code.

[8] Because we do not handle assembly code constructs.
[9] Code Generator Expansion module

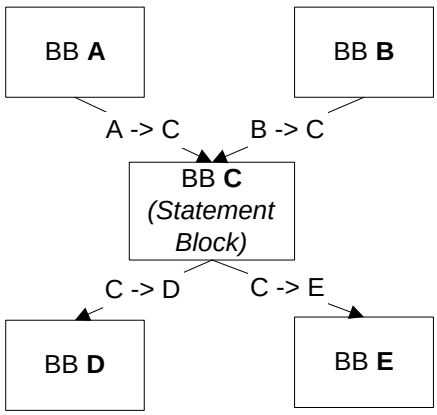

Figure 9.3: *A DO Statement block with multiple entries (from other Basic Blocks (BB)) and multiple exits (to other BB). The notation $A \rightarrow C$ indicates that the edge is a path from A to C.*

The statements that are optimized in the CG are:

- labels
- GOTO-statements
- IF-statements

When performing the optimization, the (SPIL) code is traversed and every statement is examined if it is one of the statement-kinds above. This is done repeatedly until no more optimizations can be done (one optimization can lead to additional improvements).
If the statement is an IF-statement derived from an ON-statement or from any other kind of SPIL-statement except the IF or GOTO-statements, no optimization is done.

9.3.1 Labels

The optimization done here, is mainly meant to remove unnecessary (superfluous) labels. The three ways to optimize labels are shown below:

- Unreferenced labels.

```
(0) <statement>
(1) <label> ')'
```

The label <label> is never referred to, which makes it unnecessary. Remove the <label>.

- Multiple labels.

```
(0) <statement>
(1) <label-1>
(2) <label-2>
```

Remove all references that are multiple (replace all occurrences of <label-2> with <label-1>.

- Multiple jumps.

```
(0) <statement>
(1) <label>_1
(2) 'goto' <label>_2
```

Replace the multiple jumps ((1)-(2)) with a direct jump to <label>_2.

9.3.2 GOTO-statements

There are three different GOTO-statements that are optimized. One of them is not used in the system today (GOTO - label followed by exit). They are:

- Non-executable GOTO-statements.

```
(0) <statement>
(1) 'goto' <label>_1;
(2) 'goto' <label>
```

Statement (2) can never be executed because it is preceded by another GOTO-statement. The second GOTO-statement (2) is deleted.

- Successive GOTO and label

```
(0) <statement>
(1) 'goto' <label>;
(2) <label>
```

The GOTO-statement is a jump to the next statement in the code (which is unnecessary). The second statement (1) is removed.

- *GOTO - label followed by exit.*

```
(0) <statement>
(1) 'goto' <label>;

(2) <statement-list>

(3) <label-list>
(4) 'exit'
```

The label referenced is directly followed by a label-list[10] and an exit-statement. The GOTO-statement should be removed and replaced by an exit-statement.

This pattern can not be used at the moment in statement blocks because it modifies the return from the statement block by introducing new exits.[AB98]

9.3.3 IF-statements

There are five different IF-statements that can be optimized.

- IF-statements evaluated at compile time.

```
(0) <statement>
(1) 'if' <cond-expr> 'goto' <label>;
```

If the <cond-expr> can be evaluated at compile time, the IF-statement is deleted and replaced by a GOTO-statement (if the <cond-expr> is evaluated as true).

- Successive IF and label.

```
(0) <statement>
(1) 'if' <cond-expr> 'goto' <label>;
(2) <label>
```

[10] a sequence of consecutive labels with empty statements.

The label referenced in (1) is the next statement. In this case the IF-statement has no meaning and can therefore be deleted.

- Successive jumps to the same label.

```
(0) <statement>
(1) 'if' <cond-expr> 'goto' <label>_1;
(2) 'goto' <label>_1
```

Here a jump is made to <label>_1 whether <cond-expr> is evaluated as true or false. The IF-statement has no impact and can therefor be removed.

- Successive jumps with a label-list.

```
(0) <statement>
(1) 'if' <cond-expr> 'goto' <label>_1;
(2) <label-list>
(3) 'goto' <label>_1
```

Here the situation is similar to "successive jumps to the same label" above. The IF-statement does not have any impact and can be removed.

- "Inverted" IF-statement.

```
(0) <statement>
(1) 'if' <cond-expr> 'goto' <label>_1;
(2) 'goto' <label>_2
(3) <label>_1
```

The IF-statement is "inverted", in the sense that the natural way to write the statement would be to simplify the jumps to get only one jump if the <cond-expr> is true.
Invert the operator in <cond-expr>, replace <label>_1 with <label>_2 and delete statement (2).

9.4 Control Flow Analysis in the PLEX compiler

The main task the CB-module performs, is to get information for the register allocation process (CT). The SPIL-code has already been optimized when the CB-module starts.

The control flow analysis, as it is performed today in the CB module, is done in a number of steps [LAB97]:

1. CONTROL flow-graph CONSTRUCTION is done by a single pass over the BBR[11]. Each node in the graph corresponds to one basic block. Information is picked up on the relation between the basic blocks so that the control structure in the PLEX sector can be determined. (An example of a control-flow structure is shown in appendix A.2.)

2. DEAD CODE ELIMINATION is done as the Global Data Flow Analyzer controls the code generation by giving the Code Formatter the statements. By passing over the control flow graph forwardly and then backwardly nodes are found that are not visited at the two traversals. These basic blocks constitute dead code. (See figure 9.4)

   Basic blocks that are not visited in the forward traversal, are not predecessors (because they can not be reached from the INITIAL-node) and are removed from the set of predecessors in each of its successors. The nodes that are not visited in the backward traversal are similarly removed from the set of successors in each of its predecessors. These events generate an error message to the programmer.

3. CONSISTENCY CHECK is done by checking that ENTER statements in the PLEX-program are only reached from the INITIAL basic block. If another basic block than the INITIAL basic block was found an error message will be issued to the user.

4. DESTRUCTIVE STATEMENTS, i.e. statements that destroy all registers, need a special treatment. Usually all SEND ... WAIT statements are re-linked to the TERMINAL-node (as its successor) and all RETRIEVE statements are re-linked to the INITIAL-node (as its predecessor)(see figures 9.5 and 9.6). This is done to aid in the register allocation process, because these statements lead to the "destruction"

[11] Basic Block Representation

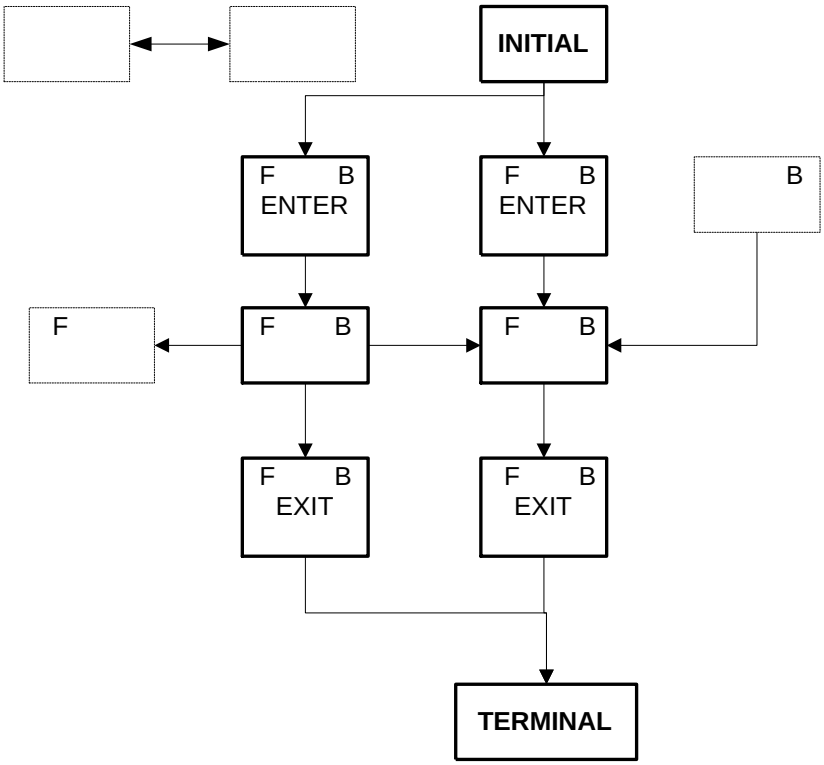

Figure 9.4: *The basic blocks are traversed and marked if visited. F - visited on forward traversal, B - visited on backward traversal. Dashed basic blocks constitute dead basic blocks.*

of all registers.

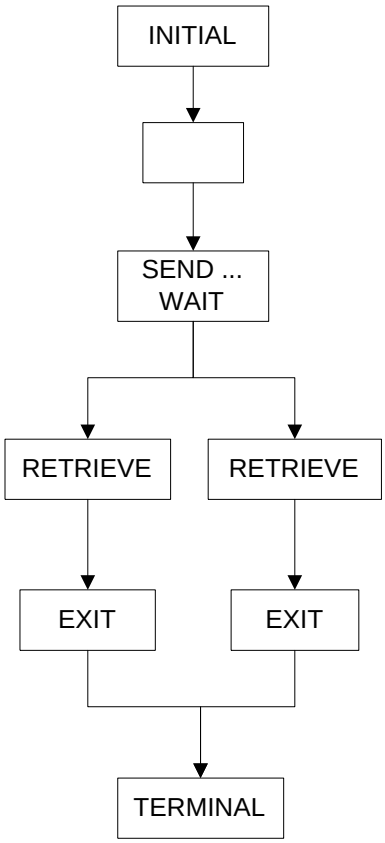

Figure 9.5: *The control flow-graph before the handling of destructive statements.*

5. Finally the DEPTH-FIRST ORDERING is determined for the control flow-graph, i.e. find the ordering between the nodes such that the nodes as far away as possible from the INITIAL node, are traversed as quickly as possible. This ordering is used in the chaining algorithm. It can also be used when detecting loops in the control flow-graph.

   A DEPTH-FIRST SPANNING TREE can be built at the same time as the depth-first ordering (or numbering) is determined. A depth-first spanning tree is a representation of the control flow-graph that directly reflects the depth-first ordering. This tree, together with the depth-first ordering, can be used to determine the retreating edges in

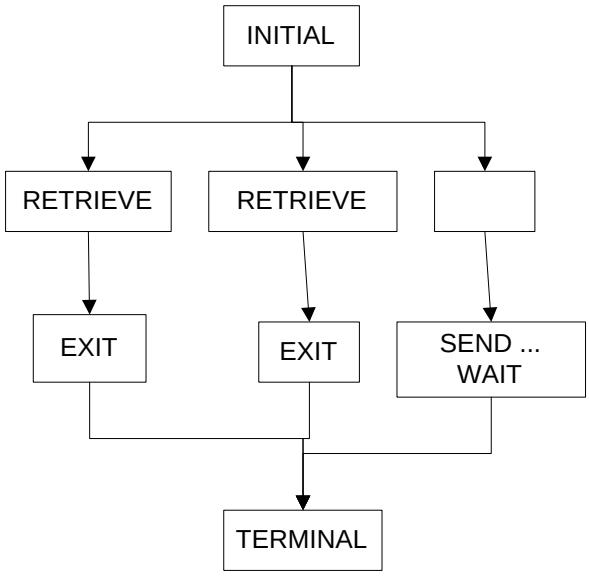

Figure 9.6: *The control graph in 9.5 after the destructive statements have been handled.*

the control flow-graph. This yields a possibility to detect loops in the source program.

After the control flow-graph has been constructed, a third pass is made to discover and eliminate dead code (the basic blocks that are not marked as visited in both the forward and backward traversal). This information is then passed to the code selection process (CS).

Chapter 10

Basic Blocks in PLEX

The method to create the basic blocks and the control flow-graph for PLEX is [LAB97]:

- determine the *leaders*, the first statements of basic blocks. A *leader* is:
  - the first statement in a PLEX program
  - labeled statements
  - statements following IF or assignment with carry
  - ENTER, ENTRANCE, RETRIEVE and RECEIVE statements
  - DO ASA statements where ASA contains local or global entries
  - if a conditional GOTO is an *ender*, the following statement is a *leader*

    The *leader* is put into the BBR (Basic Block Representation) which is a doubly linked list of basic blocks (which also contain a reference to the SPIL statement and the source code position of each statement and various information on the variables).

- determine the *enders*:
  - conditional/unconditional jump (GOTO, IF, BRANCH ON an assignment with carry)
  - DO statement block, FOR statement, SEND statement (local or global and combined), EXIT, TRANSFER, RETURN statements

  - each DO ASA statement where the ASA sector contain local sending statements or EP[1]
  - last statement before the next basic blocks *leader*
  - last statement in SPIL

- for each *leader*, its basic block consists of the *leader* and all statements up to and including the *ender* or the end of the program.

The INITIAL node is created to get a single point of entry into the code (it has no predecessors). It's successors are all basic blocks that has a ENTER or RECEIVE statement as a *leader*.

The TERMINAL nodes purpose is to form a single point of exit for the code. Similarly to the INITIAL node, it has only special nodes connected to it (predecessors). Basic blocks with EXIT and RETURN (returning a combined backward signal) statements as enders are connected to the TERMINAL node.

[1]EXIT statement for ASA code.

Part IV

WCET Estimation and implementation

Chapter 11

Applying the theories - The ESEX prototype

As was said in section 1.2, we had two aims with our work. The main task was to explore possible methods for estimating execution times in soft real-time systems. A secondary task was to extract the data needed as input input to GRETA[1], and annotate the graphical representation with estimated execution times. In this chapter we describe the work that has been done in these areas.

11.1 Execution time estimation

In section 3.2, we listed the decisions that has to be made when one tries to estimate (or calculate) the (worst case) execution time. In this section we repeat the questions and argue on our answers and solutions implemented in the prototype. We also discuss the assumptions and generalizations that have been made.

11.1.1 Questions and Answers

- *High- or low-level analysis ?*
  An early decision was made, that the analysis should be focused on a high level. The reasons for this were several. First, the AXE system runs on a number of different processors and we believed that it was better to study a method which would be relevant for all current **and**

[1]The graphical prototype developed by Arnström et. al. [AGG99]

future kinds of processors. Another reason was that we were more interested in the question *Is it possible to estimate the execution time of a PLEX program* rather than the actual running time of the program.

- *Code level*
  Our first demand on the input format was that it should have a correct syntax. We also wanted to be independent of the source language (if possible) in our method. Another point was that we soon realized that we needed some kind of representation of the program to be analyzed. The decision to work with the intermediate format was taken on the basis of the following aspects: Our supervisors at Ericsson pointed to the fact that a control flow-graph for the compiled program was already available in the back end of the PLEX compiler and that we would save a lot of time by using an existing representation. (By using the intermediate format, we also fulfilled our demands on syntax.) Another important fact is that code motion[2] is **not** allowed in the PLEX compiler. This means that the underlying structure of the program is not changed in the intermediate format.

- *Loop detection*
  In section 4.5, we said that the body of an irreducible loop could not be discovered in the same way as the body of a reducible loop and since our aim was to discover all kind of loops in the control flow-graph, we looked for a method that would handle both reducible and irreducible loops. This is the reason why we chose the method proposed by Sreedhar, Gao and Lee in [SGL96]. The method use a DJ graph to discover the loops and it is a generalization of Tarjans Strongly Connected Component (SCC) algorithm, [Tar74]. The method will be further investigated in section 11.2.

- *Number of iterations*
  The reason why we have not tried to find the number of iterations for a found loop (as we said in 1.4) is that the target system (i.e. the AXE system) consists of approximately 10.000.000 lines of code which makes it an unreasonable task to manual add annotations on loop counts. A

[2]This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a *loop-invariant computation*) and places the expression before the loop. [ASU86]

second reason was that finding the loops was considered sufficient in a first phase. (Possible extensions are discussed in the *Future Work* section 13.)

11.1.2 Assumptions and Generalizations

The following assumptions and generalizations has been made:

- Execution time
  We said in 11.1.1 that the actual running time of the program (i.e. the number of clock cycles) were of less interest for the prototype. For that reason we assume that the running time of a single instruction is equal to one time unit. This means that the running time for a basic block (which is the smallest unit in the examined control flow-graph) is equal to the number of statements in that basic block and the running time of a loop is equal to the number of statements in the loop body multiplied by a constant (which is equal to the number of iterations in the loop).

- Loops
  We have focused on the detection of loops in the control flow-graph and on the execution time of the loop body. The number of iterations is left as an unknown constant (which means that in the output format the execution time is marked as **loop dependent**). We gave our argument for this in the preceding section (11.1.1).

11.2 Detection of loops

Before the chosen loop detection method could be applied, some preparations had to be done. To use the method proposed by Sreedhar, Gao and Lee, [SGL96], we had to build the DJ graph from the given control flow-graph. To build the DJ graph, we needed the dominance tree (or the dominance relation). To compute the dominance relation, we needed the depth first spanning tree. To build the depth first spanning tree, we finally needed to classify each edge in the control flow-graph. These steps has all been performed and the interested reader is referred to section 11.6 for more information on these parts. We now continue with the loop detection algorithm.

As we have already mentioned, the proposed algorithm is a generalization of Tarjans interval finding algorithm. (These intervals are single-entry and strongly connected subgraphs.) The Tarjan method uses a technique known as *graph reduction* and it works inside-out by processing back edges in decreasing order of their destination nodes' depth first number. (Graph reduction could also be seen as a kind of *state elimination* which is described in [AGG99](page 36). See fig 11.1 and fig 11.2.) The Sreedhar, Gao and

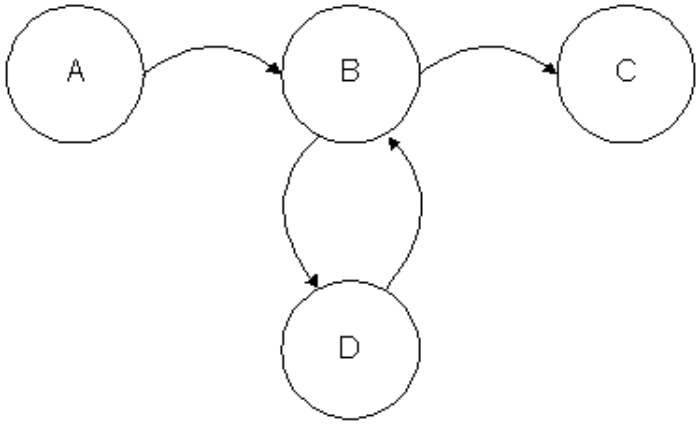

Figure 11.1: *Loop graph - before elimination*

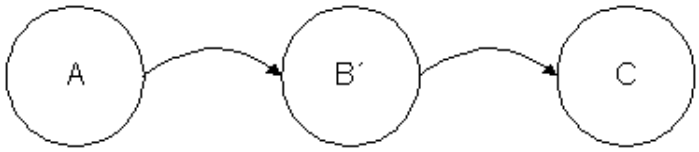

Figure 11.2: *Loop graph - after elimination*

Lee method follows the inside-out approach used by Tarjan but it operates on the DJ graph instead of the flow-graph **and** it uses the *level* information of the DJ graph.[3] To extend the Tarjan algorithm, two lemmas are used[4]:

LEMMA 3.2 *A flow-graph is irreducible if and only if there exist a simple cycle in its DJ graph that does not contain a BJ (Back Join) edge (that*

[3]the meaning of level information is that each node in the dominance tree is associated with a level which tells the distance to the root node (in the dominance tree).

[4]Both lemmas as well as the proofs are found on page 653 in [SGL96]

is, the cycle is made of only D (Dominator) edges and CJ (Cross Join) edges). [5]

LEMMA 3.3 *All the entry nodes of an irreducible loop have the same immediate dominator.*

What lemma 3.2 implies is that if a depth-first search is performed on such a DJ graph, one will find at least one sp-back edge which is also a CJ edge. This is a way to *detect* irreducibility.

To find the body of such a loop, the second lemma (3.3) is used in the following way: *What Lemma 3.3 implies is that when we are looking for the body of an irreducible loop, we can look only at nodes that satisfy some level constraint. More explicitly, we can identify irreducible loops with entry nodes at level* $\boldsymbol{i}$ *by performing Tarjans Strongly Connected Component (SCC) algorithm*[6] *on only nodes* $\boldsymbol{x}$ *with* $x.level \geq i$.
The pseudo code for the algorithm, as well as comments on the implementation, is found in section 11.6. There are also a few examples with the code and corresponding control flow-graphs before and after loop-elimination in appendix A.

11.3 Signals

We have not attempted to trace all the signals in the system. However, all signal sending and receiving statements are examined to gather information about the signals. *Direct signals* are made in line in the sense that they are linked in the control flow-graph (if the signal receiving code-item is in the same block). *Buffered signals* are found and executed by the operating system, i.e. they are not linked directly (there is no direct jump from the sending statement to the receiving code-item). *Local signals* are treated as GOTO-statements, this means that the signal is in-lined. One problem is when a local signal is combined. Several basic blocks may send the same local combined signal. The problem that arises is similar to that of linked DO-statement blocks (discussed in 9.2.4), the actual path of execution may be lost.

The information about signals is extracted by performing a pass (traversing) the control flow graph. The information is written to a file in pass after

[5] see section 4.1 for definition on the different kind of edges.
[6] The mentioned algorithm is found in [Tar74]

all transformations have been performed.

11.4 Identification of code-items

Code-items have been described in 7.1. And we recall that the entry of a code-item is a signal receiving statement and we also recall from chapter 5 that the **INITIAL** node is created and inserted in the control flow-graph to connect all basic blocks that has a signal receiving statement as its first statement. (This is to give the control flow-graph a unique entry point.)

To identify the code-items from here is very straight forward: We simply look at each successor to the INITIAL node to identify the first basic block in each code-item and then follow the edges in the graph down to the **TERMINAL** node to find the other basic blocks. (What is actually done is to traverse each code-item to gather the information we need, which is execution time, signal sending/receiving statements, special FOR statements etc.)

11.5 ESEX - The prototype tool

ESEX - EStimating EXecution times in soft real-time systems is the developed prototype tool. The prototype has been adjusted to PLEX, the language used by Ericsson in the AXE system, and it is implemented as an extension of the PLEX-C compiler. (An extension of the back end.)

11.5.1 Implementations made in the back-end

All the work we have done are made in the back-end (see 9.1.2). Some parts have been added and some have been changed or deleted. Most work is performed in the CB-module (see 9.1.2). When analyzing this module we found that after the control flow-graph is created, destructive statements are handled, see 9.4. This means that all SEND / WAIT / RETRIEVE statements (or combined signal sendings) are re-linked to the INITIAL node (see figures 9.5 and 9.6). These statements are handled to be able to use the information in the control flow-graph for register allocation. This however destroys the control flow-graph and information about the paths of execution is lost. Because we are not interested in the register allocation (or data flow

analysis) here, and because these transformations destroy the control flow-graph, we do not make these transformations. The destructive statements handling process in the CB-module is removed.

The depth-first ordering is done in a different manner than in the original back-end. Originally the flow-graph is traversed and the depth-first ordering is done in a bottom up fashion. The depth-first list is then reversed. We have chosen to create the depth-first ordering list in a top down fashion.

First the dominator tree is determined, then the loop elimination process is performed (see section 11.6). When all loops have been collapsed and removed from the control flow-graph, the task of traversing the control flow-graph and gathering information is much simpler. The information or output file is created when traversing the graph.

Everything after the flow analysis is removed in our prototype. Everything performed after the CB-module handles register allocation, ASA-code generation and optimizations on ASA. This is not interesting when doing our analysis, especially considering the high-level approach we have chosen.

11.6 Implementations and pseudo code for loop detection

In section 11.2, we said that some preparing work had to be done before we could apply the algorithm found in [SGL96]. Here, we go through these steps and also give the pseudo code for the loop detection algorithm.

11.6.1 Classification of edges

A classification of the edges in the control flow-graph was necessary to "build" the depth first spanning tree. Although a depth first search is performed in the PLEX compiler, no explicit classification of the edges is performed and the reason for this is that there are no explicit edges! With each node is associated a list of successor nodes. These successor nodes in turn point to a basic block and this basic block is a successor to the first basic block. (i.e. Each basic block know its successors only through the successor item list and there is a unique successor list for every basic block.) (See fig 11.3.) To solve this problem, we did two things: First we added a new field to the basic block representation and then we modified the depth first

search function. The new field added to the basic block representation is a pointer to which an array is allocated. This array contains a number of slots (of predefined size). Every slot in this array then represent a unique edge to a successor. In this way we get explicit knowledge of each edge from a basic block to its successors. The depth first search function was modified to be able to determine the different kinds of spanning tree edges. The ideas we used for this classification is from [CLR90](section 23.3) and uses something called *time stamps*. A nodes *discovery time* is set the first time the node is visited and its *finishing time* is set when there are no unexplored successors left. (For a further description of this concept, we refer to [CLR90].)

11.6.2 Computing the dominance relation

The implementation of the dominance computation is based on the pseudo-code in algorithm 19.9 in [App98]. To perform the computation some additional pointers are added to the basic block representation. (The idom, samedom and semiDominator fields.)

11.6.3 Building the DJ graph

... one can also construct the DJ graph of a flow-graph by appropriately inserting some D^7 edges into the flow-graph, ...,[SGL96]. We use this and insert this edges when we do the dominance computation, see 11.6.2. (Actually, what is done is that the dominance pointers in each node is set to point to the dominator node.)

[7]Dominator tree edges

11.6.4 The loop detection algorithm

```
MainLoop()
{
1:   Perform a depth-first search on the DJ graph and identify sp-back edges;
2:   for(i = NumLevel - 1 downto 0) /* visit nodes in a bottom-up fashion */
3:       IrreducibleLoop = False;
4:       for each node n with n.level = i do
5:           for each incoming edge m → n do
6:               if m → n is both a CJ edge and an sp-back edge then
7:                   IrreducibleLoop = True /* n is in an irreducible loop */
8:               endif
9:               if m → n is a BJ edge then
10:                  Find ReachUnder(n) for all the BJ edges m_1 → n, ..., m_k → n;
11:                  Collapse the loop consisting of nodes in {n} ∪ ReachUnder(n);
12:              endif
13:          endfor
14:      endfor
15:      if(IrreducibleLoop) /* there exists an irreducible loop or loops */
16:          Identify SCCs for the subgraphs induced by nodes at level ≥ i;
17:          Collapse each nontrivial SCC to a single node;
18:      endif
19:  endfor
}
```

Some explanations about the algorithm:

- If the destination node of any sp-back edge does not dominate the source node, *an* entry node of an irreducible loop is found and the flag *IrreducibleLoop* is set to be true (step 7).
- The procedure *ReachUnder(n)* finds all nodes that can reach the source node of a sp-back edge incident on the entry node n, without going through n. The implementation of this function is based on algorithm. 10.1, p.604 in [ASU86].
- In step 11 and step 17 the found loop is collapsed. The question we

asked ourselves at this point was *How do we represent the collapsed loop?*. We solved this by letting the loop entry node (or in case of an irreducible loop, the first discovered entry node) "represent" the collapsed node in the following way: To each loop entry node is an array of basic blocks allocated. The items in this array is the nodes in the loop body. Then, for each successor of a node in the body we make it a successor to the loop entry node. And, in case of an irreducible loop, for those nodes in the body that has a predecessor outside the loop, we redirect this predecessor to point to the loop entry node instead.

- In step 16 we identify the SCC's for the subgraphs induced by nodes at level $\geq i$. What is used is Tarjans algorithm for finding Strongly Connected Components with the level constraint described in 11.2.

11.7 A general example

In this example [SGL96], which assumes the graph in fig 11.4, it is shown how the loop detection algorithm works. When we perform a depth-first search on this graph, we get the node numbered as in fig 11.5. (The corresponding depth-first spanning tree is shown in fig 11.6 and with the remaining edges added we get the graph shown in fig 11.7.) When the dominance relation is computed, we have the dominance tree shown in fig 11.8 which in turn yields the *DJ graph* in fig 11.9. After the DJ graph have been constructed, the loop detection algorithm starts at level 3. At this level, nothing happens. (The only nodes at level three is node number 4 and number 8, and there is no loop between this two nodes.) On level 2, the edge from node 4 to node 3 is identified as a *BJ edge* (i.e. node three dominates node four). As a consequence, the *natural* loop consisting of these two nodes is identified and collapsed. The new graph is shown fig 11.10. The algorithm proceeds and identifies the edge from node 7 to node 3 as *CJ edge* as well as a *sp-back edge*. This sets the flag *IrreducibleLoop* true. After all the nodes at level 2 are processed, the algorithm runs Tarjans SCC algorithm on nodes at level 2 **and** 3. (i.e. on nodes 2, 3, 5, 7, 8 and 9.) Here, the SCC consisting of the nodes 3, 5, 7, 8 and 9 is identified and collapsed (as shown in fig 11.11). At level 1, another sp-back edge is discovered (from node 3 to node 1) and the natural loop (with entry node 1) consisting of node 1, 2 and 3 is found. Collapsing this loop yields the final graph in fig 11.12.

It's easy to see that the complexity regarding execution paths has decreased if we compare the graphs in fig 11.4 and fig 11.12.

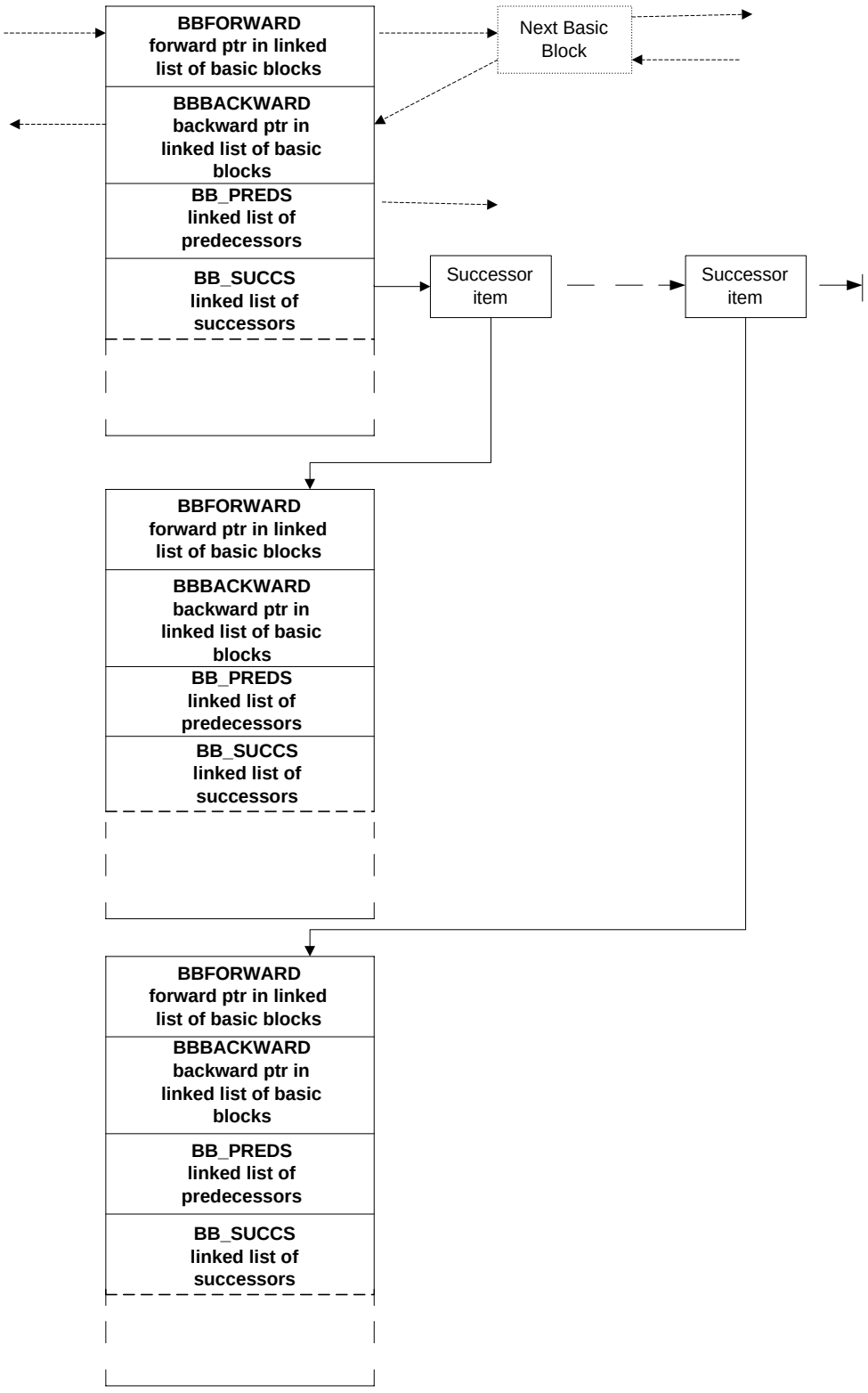

Figure 11.3: *Organization of the successor relationship.*

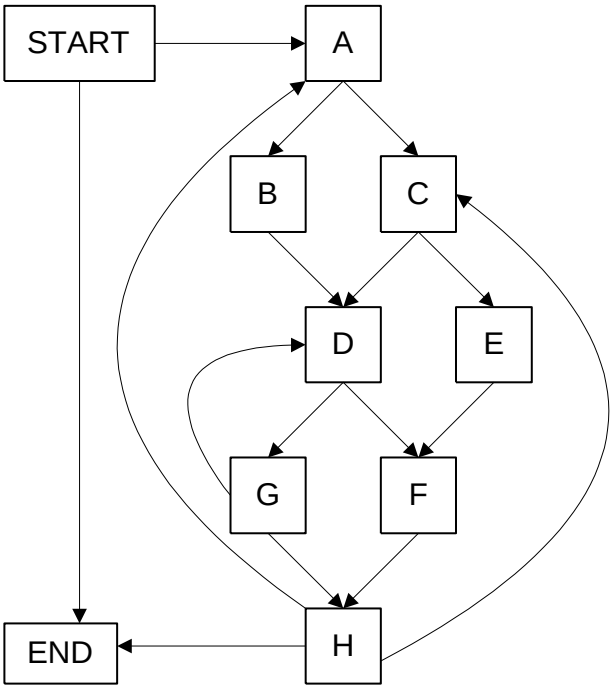

Figure 11.4: *The assumed input graph.*

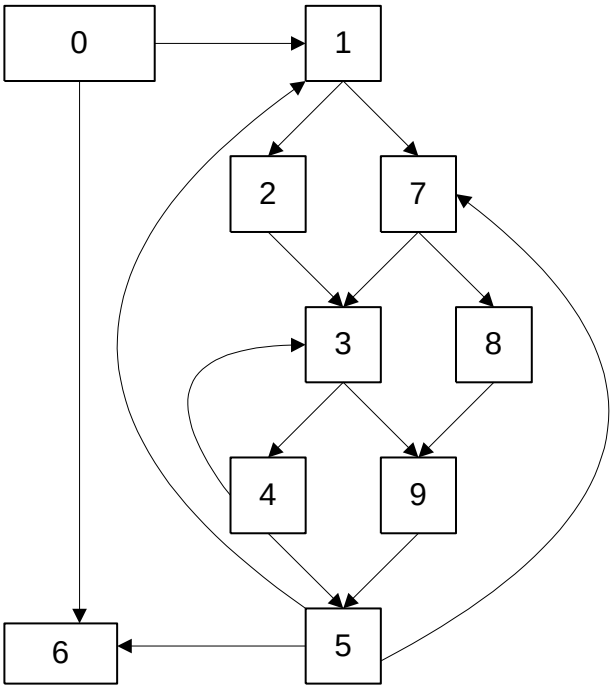

Figure 11.5: *The depth-first numbered input graph.*

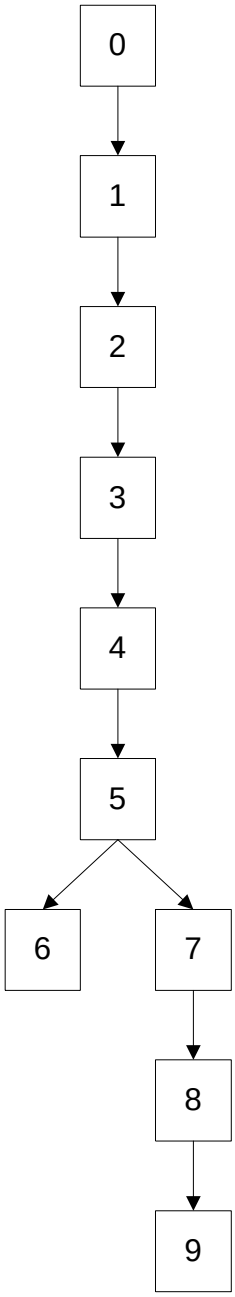

Figure 11.6: *The corresponding depth-first spanning tree.*

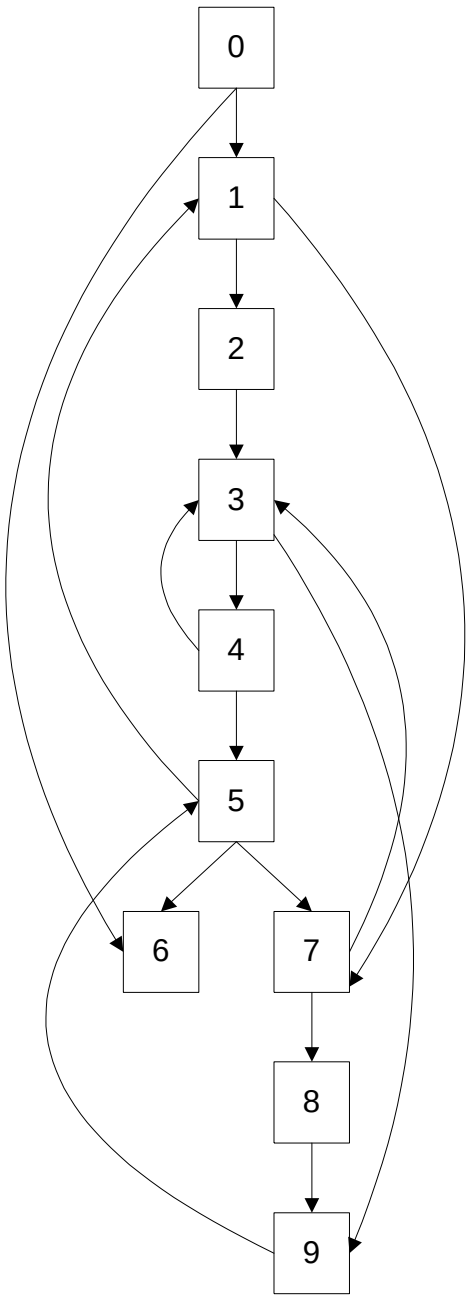

Figure 11.7: *The depth-first spanning tree with the remaining edges added.*

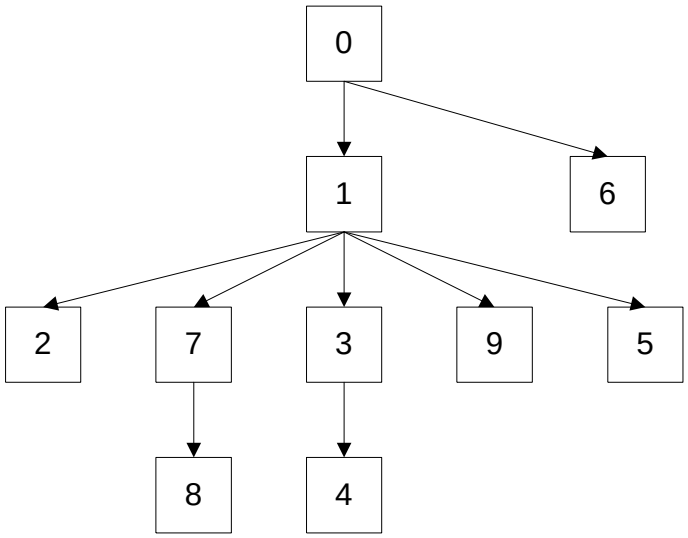

Figure 11.8: *The dominance tree built from the input graph.*

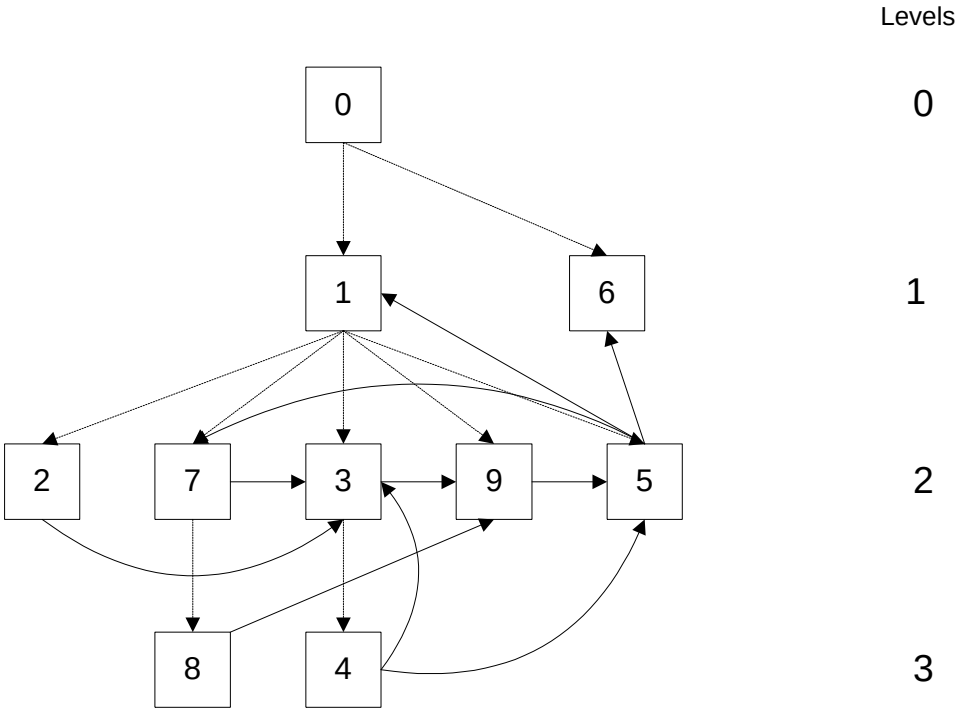

Figure 11.9: *The DJ graph (built from the input graph).*

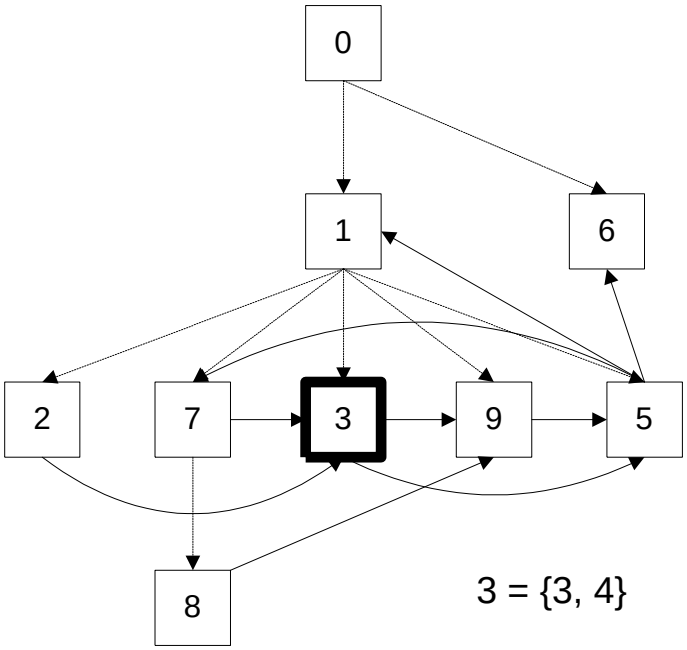

Figure 11.10: *The DJ graph after the first loop is collapsed.*

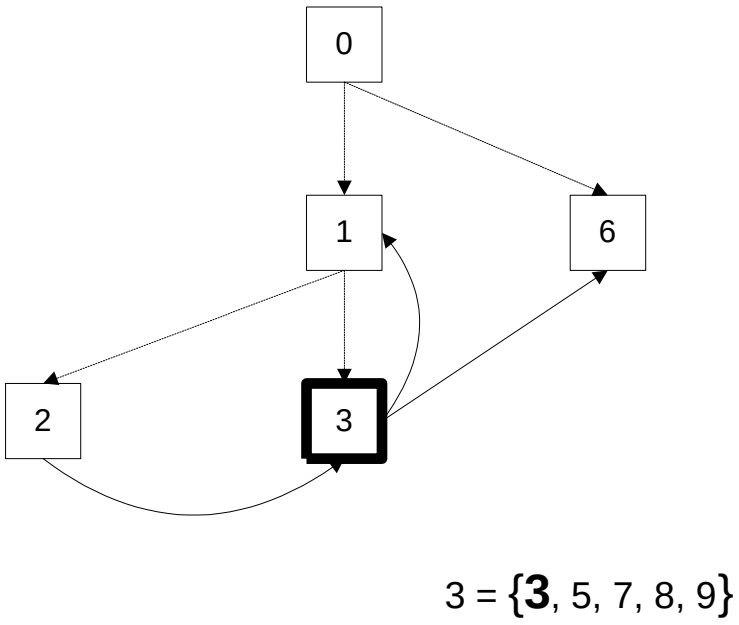

Figure 11.11: *The DJ graph after the nodes at level 2 is processed.*

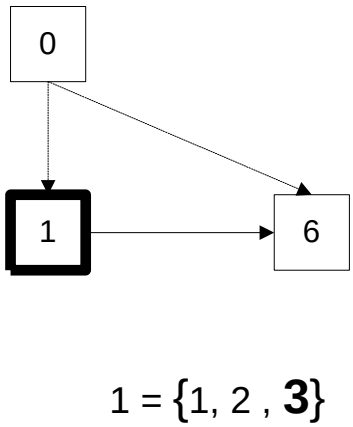

Figure 11.12: *The DJ graph after termination of the algorithm.*

Chapter 12

Evaluation of approach

Is it possible to perform execution time analysis for soft real-time applications and in particular for telecommunications systems using PLEX? In this work we show that execution time can be estimated for systems written in PLEX. Not execution time in absolute measures, but estimation values sufficient for soft real-time systems. The theoretical work in 11 is in critical parts verified by the prototype implementation, showing that the theories hold.

12.1 Example

In the following example the original PLEX code, the control-flow graph before and after loop elimination is shown. The two loops in the example (see fig 12.1) are eliminated as shown in fig 12.2 (the eliminated edges and nodes are marked with dashed lines). The output files for this program (Prolog file for input to the GRETA tool and a textual information file) are found in B.

12.1.1 clear_assign.program

```
DOCUMENT BUFFCOPYUPROGRAM;

DECLARE;


  RECORD combuf4kRec;
    VARIABLE cb4kI COMMUNICATION BUFFER(4098);

  END RECORD;
  POINTER combuf4kRecP(combuf4kRec);

  RECORD combuf8kRec;
    VARIABLE cb8kI COMMUNICATION BUFFER(8192);
  END RECORD;
  POINTER combuf8kRecP(combuf8kRec);

  VARIABLE cb4k COMMUNICATION BUFFER(4098);
```

```
  VARIABLE cb8k COMMUNICATION BUFFER(8192);
  VARIABLE cb64 COMMUNICATION BUFFER(64);
  VARIABLE J, I, tstart, tend;

END DECLARE;

PROGRAM;  PLEX;

  ENTER startcopy WITH
            combuf4KRecP,
            cb4kI,
            combuf8KRecP,
            cb8kI,
            cb4k,
            cb8k,
            tstart,
            tend;

!--------- COMMUNICATION BUFFER IS IN RECORD -------------------------!

FOR ALL J FROM tstart UNTIL tend DO
                cb4k(J) = 0;
          cb4k(tend - 1) = 1;
!--------- COMMUNICATION BUFFER IS NOT IN A RECORD -------------------!
FOR ALL J FROM tstart UNTIL tend DO
                cb4k(J) = cb8k(J);

  SEND startcopyR WITH
            combuf4KRecP,
            cb4kI,
            combuf8KRecP,
            cb8kI,
            cb4k,
            cb8k;

  EXIT;

END PROGRAM;

END DOCUMENT;

ID BUFFCOPYUPROGRAM TYPE DOCUMENT;
CLA 19055;
NUM CAA 107 9999;
REV A;
DAT 99-11-25;
DES EED/U/TG AMB;
RES NOTME;
APP BYME;
END ID;
```

Another example (a self-reference) can be found in A.1.

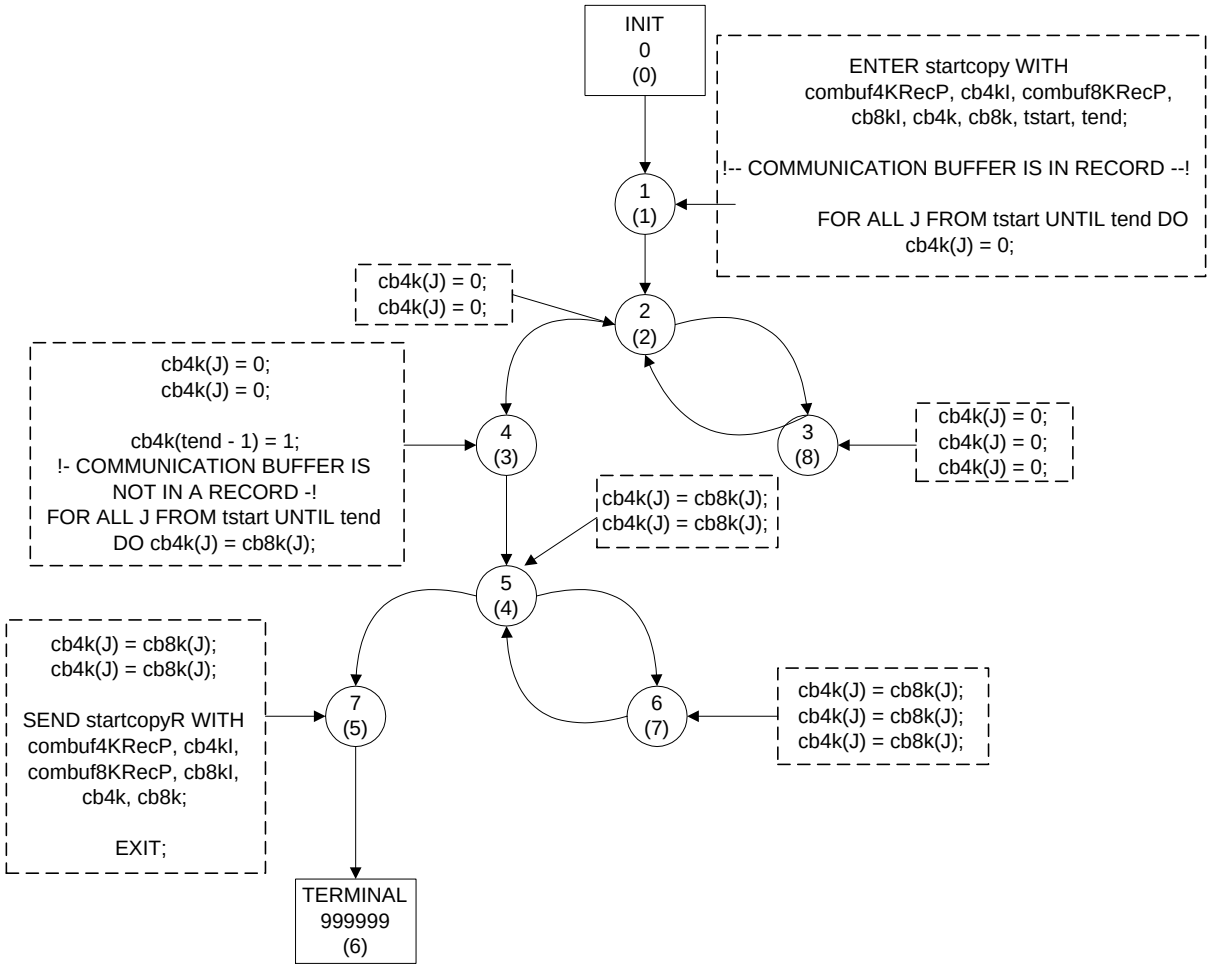

Figure 12.1: *The control flow-graph before loop elimination for program: clear_ assign.program.*

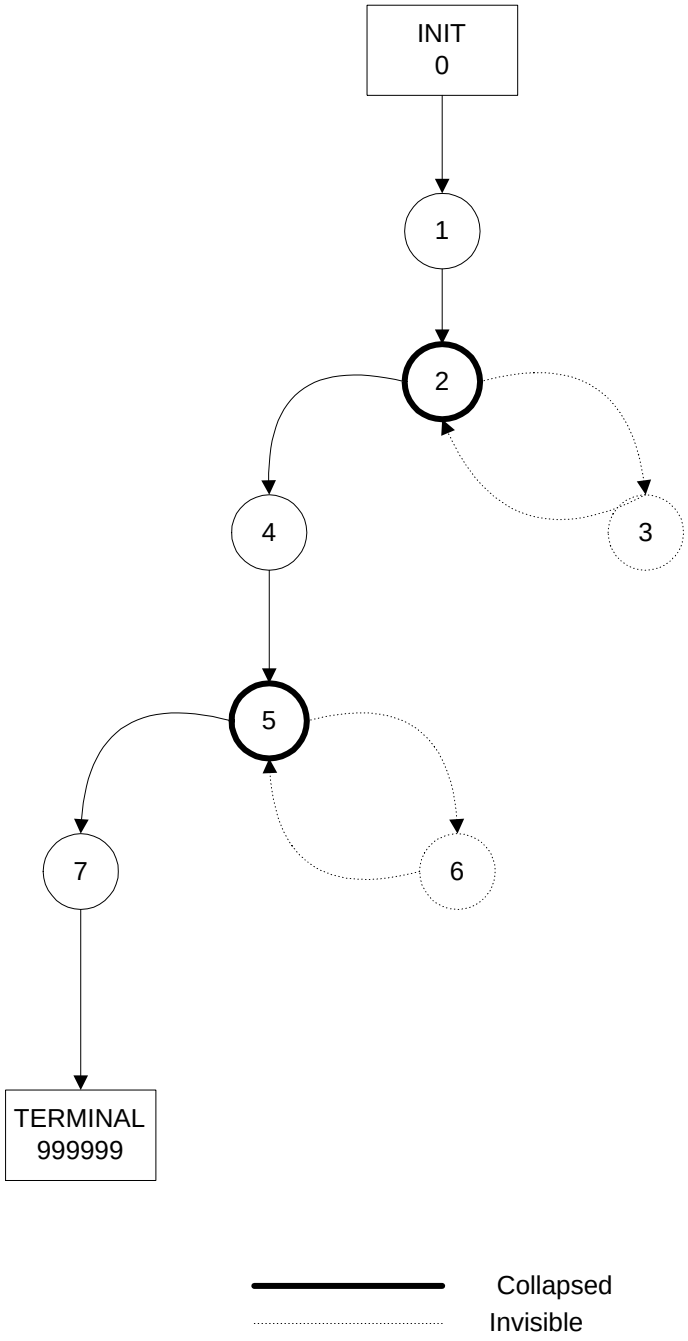

Figure 12.2: *The control flow-graph after loop elimination for program: clear_ assign.program.*

Part V

Conclusions

Chapter 13

Future work

- Currently the estimated execution time for a analyzed program part is the number of lines of code executed in this part. This is unreasonable coarse grained and adding tables with individual figures for different program statements (min, max, average) will increase accuracy considerable when calculating the sums. These tables are also available for different processors. Including them in the prototype is planned.
- Full integration of the ESEX prototype tool (for estimate execution times) with the prototype for graphical abstraction of existing code (i.e. the GRETA prototype tool developed in [AGG99]) is planned, and when completed, field trails with programmers will be performed. This will give a basis for estimation on how big improvement such a tool can bring. If efficiency and quality improvements are large enough, full scale implementation of such tools are considered.
- As we said in 1.4, we do not try to find the number of iterations for a found loop (and we explained why in 11.1.1). This, however, would most likely be valuable information for developers and we see two possible extensions to deal with this.
  - For any code that is added or modified in the system, abstract interpretation (as proposed by [Gus00]) is an interesting and powerful option for certain language constructs (i.e. the ON, FOR ALL and FOR FIRST statements). A combination of the method used in this report and abstract interpretation would probably complement each other and improve the accuracy of time estimations.
  - Data flow analysis is used in [HW99] to detect value dependent

constraints without the need of any manual annotation. These constraints makes it possible to limit the number of times a certain path can be executed in a loop which, in turn, can be used to increase the accuracy in execution time estimations. Since no manual annotations are used, we see this work as a possible extension to our method.

- There are some constructs that have a great impact on the exactness of the execution time estimation. One is the DO statement-block discussed in 9.2.4. The consequence of this construct could be an overestimation of the execution time since, today, we do not separate valid execution paths from those that are not. A solution to this problem would be to always chose the in-line method (see 9.2.4).

- The GRETA tool is prepared for finding connections between the analyzed blocks. This analysis is made on the Prolog-intermediate form that is generated from the compiler (i.e. our prototype tool). The actual connections between the different blocks for a given call scenario are found in the final linking process.
  A possible approach to more accurately trace the execution could be to emulate a running AXE system and hereby collect dynamic information on which code parts is triggered by which signals in a given scenario. (This method is used in the E-CARES project [MH01].)

Chapter 14

Summary

Soft real-time systems has the property that a missed deadline does not have catastrophic consequences. However, there is still a need to get an estimation of the execution time since the aim is to minimize the average execution time, and even if missed deadlines do not cause disaster directly, the consequences may be indirectly serious. In spite of this execution times and time restrictions are today mostly handled by different ad hoc solutions (e.g. guidelines such as *Do not write more than k lines of code in one block*). In this report we show a more systematic approach applied in the telecommunication domain (i.e. on the AXE system).

The methods used is an algorithm developed by [SGL96] and it operates on a DJ graph, which is constructed from the control flow graph for the intermediate format of the compiled program. This, in combination with scanning the source code for some additional looping structures[1], gives us sufficient information to estimate the running time of the input program. This part of the report is (more or less) a general solution.

To be able to represent the (PLEX) program in a graphical way, the signal information in the blocks has to be extracted. This is done by analyzing the signal sending and receiving statements in the program.

To confirm our approach we implemented a prototype integrated with the compiler. The prototype has been used to estimate execution time and detect loops for a number of test-examples of real blocks. It was also used to extract information for the GRETA-tool developed by Arnström, Guillemot and Grosz [AGG99].

[1] i.e. unexpanded FOR-statements

Bibliography

[AB98] Ericsson Telecom AB. *PLEX-C 1*, 1998.

[AGG99] A. Arnstrom, C. Grosz, and A. Guillemot. GRETA: a tool concept for validation and verification of signal based systems (e.g. written in PLEX). Master's thesis, Mälardalen högskola, 1999.

[App98] Andrew W. Appel. *Modern compiler implementation in C.* CAMBRIDGE UNIVERSITY PRESS, 1998.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ulman. *Compilers Principles, Techniques and Tools.* Addison-Welsey Publishing Company, 1986.

[But97] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems : Predictable Scheduling Algorithms and Applications.* The Kluwer International Series in Engineering and Computer sci, 1997.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[Gus00] Jan Gustafsson. *Analysing Execution-Time of Object-Oriented Programs Using Abstract Interpretation.* PhD thesis, Uppsala University, Sweden, 2000.

[GY99] J. Gross and J. Yellen. *Graph Theory and its Applications.* CRC Press, 1999.

[HW99] C. Healy and D. Whalley. Tighter timing prediction by automatic detection and exploitation of value-dependent constraints. In *Fifth IEEE Real-Time Technology and Applications Symposium.* IEEE, 1999.

[KO00] P. Karlsson and S. Ohlsson. Jämförelse av registerallokerigsstrategier för proggrammerinsspråket PLEX. Master's thesis, Mälardalen högskola, 2000.

[LAB97] Ericsson SOFT LAB. *Code Generation Basic Block Analyser Function*, 1997.

[LAB98] Ericsson SOFT LAB. *PIL - SPIL Translation Rules*, 1998.

[MH01] A. Marburger and D. Herzberg. E-CARES Research Project: Understanding Complex Legacy Telecommunication Systems. In *Fifth European Conference on Software Maintenance and Reengineering*, pages 139 – 147, 2001.

[Muc97] S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, 1997.

[NE93] NationalEncyklopedin, Band 12, 1993.

[SGL96] V C Sreedhar, G R Gao, and Y-F Lee. Identifying Loops Using DJ Graphs. *ACM Transactions on Programming Languages and Systems*, 18(6), 1996.

[Tar74] R. E. Tarjan. Testing flow graph reducibility. *J. Comput. Syst. Sci*, (9):355–365, 1974.

[Wol92] Michael Wolfe. Flow graph anomalies: Whatt's in a loop? Technical Report CS/E 92-012, Oregon Graduate Inst. for Science and Technology, Portland, Oreg, 1992.

Index

Part VI

APPENDIX

Appendix A

A.1 fib.program

```
DOCUMENT FIBPROGRAM;

DECLARE;

VARIABLE i 16;
VARIABLE f0 16;
VARIABLE f1 16;
VARIABLE f2 16;
VARIABLE m 16;

END DECLARE;

PROGRAM; PLEX;

RECEIVE sigName WITH
m;
f0=0;
f1=1;
IF m = 1
THEN
RETURN backSigName WITH
m;
ELSE
ON i FROM 2 UPTO m
DO
f2=f0+f1;
f0=f1;
f1=f2;
NO;
FI;
RETURN backSigName WITH
f2;

EXIT;
END PROGRAM;

END DOCUMENT;
```

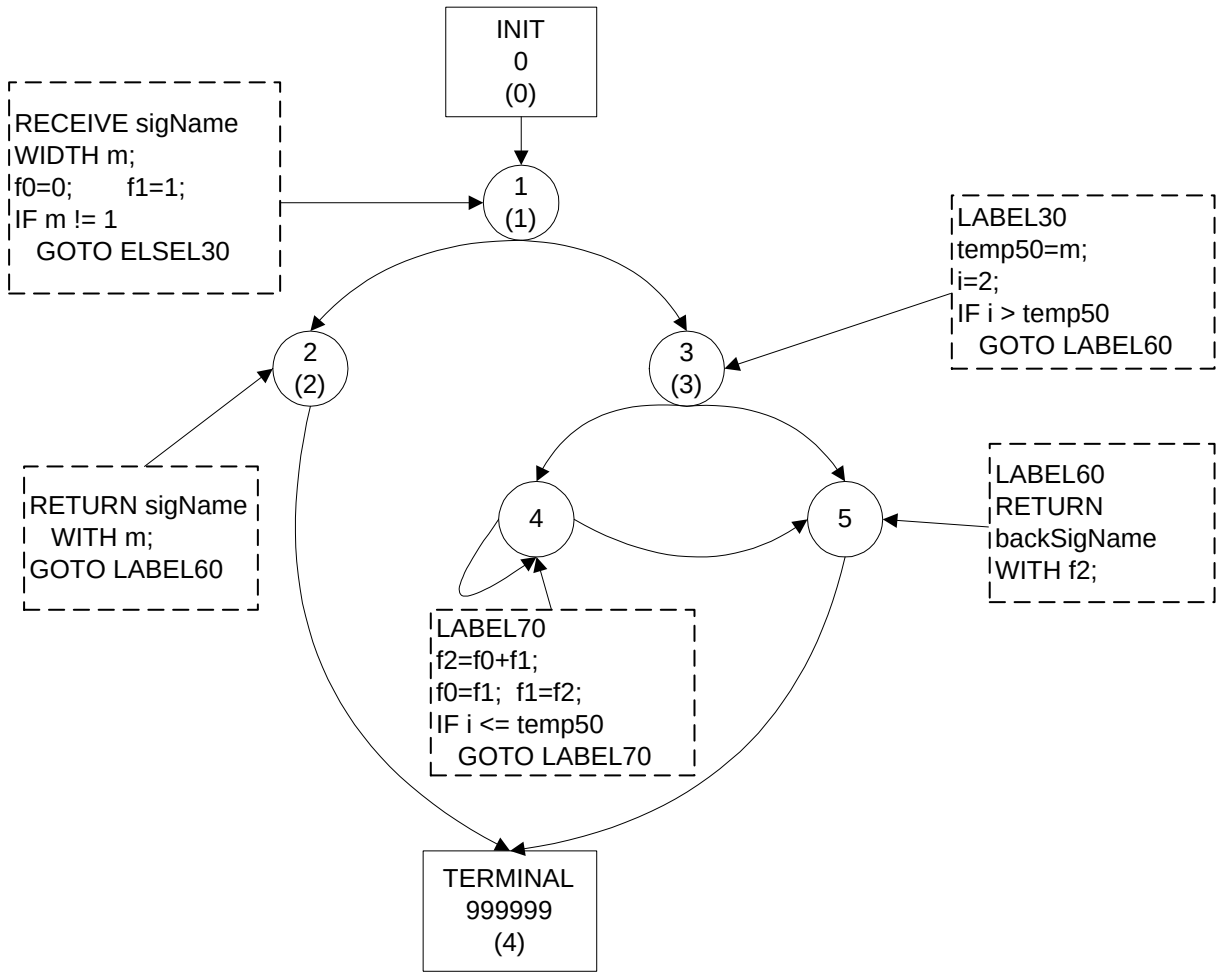

Figure A.1: *The control flow-graph before loop elimination for program: fib.program.*

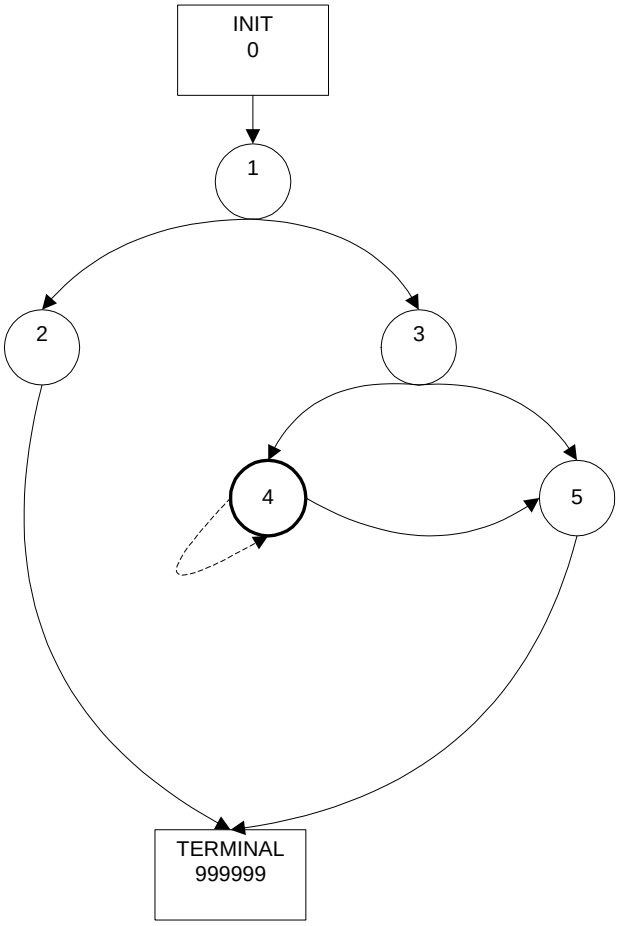

Figure A.2: *The control flow-graph after loop elimination for program: fib.program.*

A.2 test1.program

```
DOCUMENT TESTPROGRAM;

DECLARE;
VARIABLE x 16;
VARIABLE y 16;
VARIABLE z 16 DS;
VARIABLE m 16;
END DECLARE;

PROGRAM; PLEX;
ENTER sigName;
x=2;
y=x+1;
IF z>x THEN
y=y+1;
x=x-1;
FI;

z=0;
m=z+x;
EXIT;
END PROGRAM;

DATA;
z=1;
END DATA;

END DOCUMENT;
```

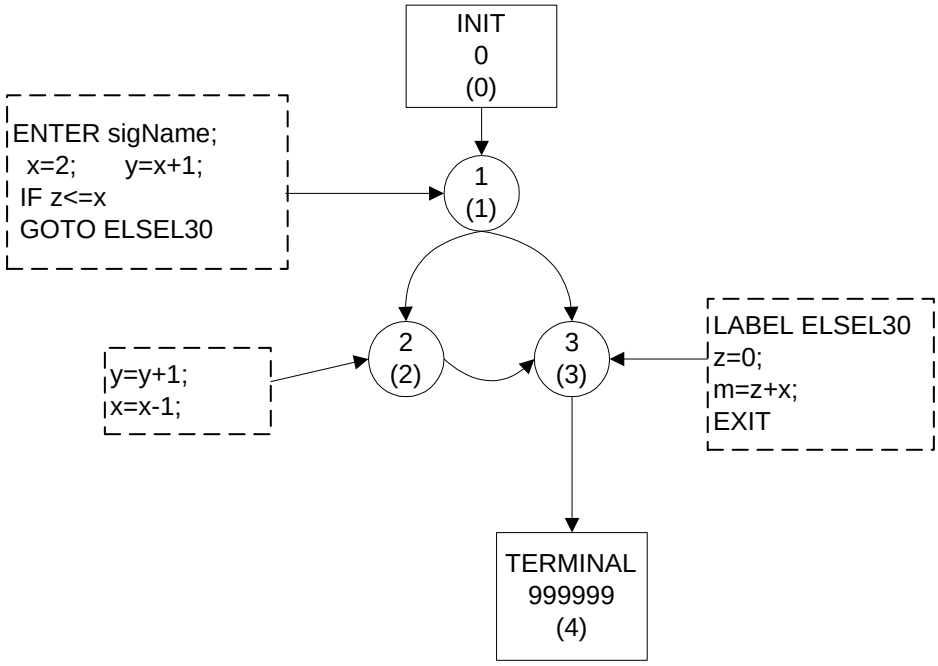

Figure A.3: *The control flow-graph for program: test1.program.*

Appendix B

B.1 Output for the clear_assign.program

B.1.1 Prolog file output

```
% Block information:
block( b-name("BUFFCOPYUPROGRAM"),
pos(X,Y),
opt([])
).
signal( io(in).
s-name("STARTCOPY").
from-blk(X).
to-blk().
type([initial_buffer]).
time(100).
pos(X,Y).
chosen(true).
opt([]).

).


signal( io(out).
s-name("STARTCOPYR").
from-blk(X).
to-blk().
type([initial_buffer]).
time(100).
pos(X,Y).
chosen(true).
opt([]).

).


code-item( b-name("BUFFCOPYUPROGRAM"),
start("STARTCOPY"),
out(),
exit("STARTCOPYR"),
opt([25, 25, LOOPS])
).

 % Nr of code items: 1
```

B.1.2 Information file output

```
Signal info:
Pos: {26, 3}
[STARTCOPY, ENTER  Single
INIT LOC
INIT BUFF

] End Signal

Signal info:
Pos: {45, 3}
[STARTCOPYR, SEND  SPAR
Single
INIT LOC
NO MULT
INIT BUFF

] End Signal



Total number of combined signals sent: 0
Total number of local signals sent: 0


Total number of statements: 0


Loop info:
Loop detected:
nr of statements in loop:(21, 21)
Loop detected:
nr of statements in loop:(9, 9)
Nr of statements for code-item STARTCOPY (min, max): (25, 25)

 % Nr of code items: 1
```