Università degli Studi di Verona
Université de Bretagne Occidentale

DEPARTMENT OF COMPUTER SCIENCE
Degree in Computer Science

Data flow analysis for cache optimization in real-time scheduling

Candidate:
Efrem Agnilleri
Student ID VR094272

Thesis advisors:
Prof. Fausto Spoto
Prof. Valérie-Anne Nicolas

Academic Year 2014–2015

Abstract

Nowadays we entrust our lives to planes, cars, trains, medical equipment, and many other things that are driven by real-time systems. Most real-time systems have to handle safety-critical applications and many embedded computer systems are real-time systems.

Real-time systems are computer systems that have to perform their actions with accomplishment of timing constraints. A real-time application typically consists of a set of cooperating tasks which are activated at regular intervals or on particular events. A task usually senses the state of the system, performs certain computations and, if necessary, sends commands to change the state of the system. Each task should finish its execution before a certain time, called the deadline.

Scheduling of tasks involves the allocation of processors and time to tasks. Task scheduling can be either preemptive or non-preemptive. In non-preemptive scheduling once a task starts its execution on a processor, it finishes. A preemptive scheduling, on the other hand, allows that the execution of a task on a certain processor can be interrupted and resumed later. For instance, a task can preempt the one in execution because it has higher priority.

Preemptive scheduling offers better schedulability but brings an overhead because of the context switches involved. State of the art scheduling algorithms are not able to compute the context switch time precisely, so they take a safe value which is always overestimated. Because of that, the time required for the context switches may be a significant fraction of the total execution time.

The objective of this dissertation is to contribute to the minimization of the time estimated to perform context switches. More precise estimations imply more efficient task schedulings.

Cheddar is a free real-time scheduling framework used to simulate the scheduling of real-time tasks, and the aim of the thesis is to add the data flow analysis functionality to the Cheddar tool. This kind of analysis gives useful information about the memory usage of a task. For instance, we can analyze a real-time task and discover that if it will be interrupted on a certain time, when restored its possible to not load into memory some variables which will be no more used by the task.

The aim is achieved by studying, and implementing in Ada programming language, static and dynamic analysis, as well as a novel model of analysis, called hybrid analysis.

Sommario

Al giorno d'oggi affidiamo le nostre vite ad aerei, automobili, treni, apparecchiature mediche, e a molte altre strumentazioni che sono controllate da sistemi real-time. La maggioranza di questi sistemi deve gestire applicazioni in cui la sicurezza gioca un ruolo essenziale, e molti sistemi embedded sono sistemi real-time.
I sistemi real-time sono sistemi informatici che devono svolgere i loro compiti entro determinati vincoli temporali. Un'applicazione real-time normalmente è composta da un insieme di task cooperanti, i quali sono attivati ad intervalli regolari o al verificarsi di determinati eventi. Un task di solito rileva lo stato del sistema, esegue alcuni calcoli e, se necessario, invia dei comandi per cambiare lo stato del sistema. Ogni task deve finire la sua esecuzione entro un tempo prestabilito, chiamato deadline.
Lo scheduling dei task si occupa dell'assegnamento dei processori ai singoli task e di stabilire per quanto tempo un task deve rimanere in esecuzione. Lo scheduling può essere preemptive o non preemptive. Nello scheduling non preemptive, se un processore inizia l'esecuzione di un task, quest'ultima viene portata a termine. Al contrario, nello scheduling preemptive è possibile che l'esecuzione di un task venga interrotta e ripresa successivamente. Per esempio, l'esecuzione di un task può essere interrotta perché un altro task, con priorità maggiore, deve essere eseguito.
Lo scheduling preemptive offre una migliore gestione dei task, ma porta con se un overhead a causa dei context switch. Nemmeno i più recenti algoritmi di scheduling possono calcolare precisamente il tempo necessario per compiere un context switch. Normalmente si stima per eccesso questo tempo. A causa di questo, il tempo richiesto per i context switch può essere una considerevole parte del tempo di esecuzione di un task.
L'obbiettivo di questa tesi è quello di contribuire alla minimizzazione del tempo stimato dei context switch. Da stime più precise derivano scheduling dei task più efficienti.
Cheddar è un framework gratuito per lo scheduling real-time, ed è usato per simulare lo scheduling di applicazioni real-time. Lo scopo di questo lavoro è aggiungere una funzionalità di analisi di data flow a Cheddar. Questo tipo di analisi calcolerà informazioni utili riguardo all'uso della memoria di un task. Per esempio si potrà analizzare un task real-time e scoprire che se sarà interrotto in un determinato momento, quando sarà ripristinato sarà possibile non caricare in memoria alcune variabili che non saranno più usate.
L'obiettivo è stato raggiunto studiando e implementando nel linguaggio Ada l'analisi statica e dinamica, oltre che un nuovo modello di analisi chiamato analisi ibrida.

Contents

Preface

This degree thesis was done in Brest, France, at *Université de Bretagne Occidentale*, in the department of Computer Science, during the period January 2015 to May 2015, as a part of the bachelor education in Computer Science at *Università degli Studi di Verona*, Italy. Moreover, it's the outcome of an intensive and incredible foreign experience that I did thanks to the *Erasmus+* european programme.

I would like to thank my french supervisor, Prof. Valérie-Anne Nicolas, for giving me the opportunity for writing this thesis and the overall support during its development. I'm also grateful to my italian supervisor, Prof. Fausto Spoto, that helped me choosing a good foreign destination and for always being available and helpful.
Finally, I would like to thank my family, my friends and all the amazing people that made this abroad period one of the most exiting I ever had so far.

Chapter 1

Introduction

1 Background and context

1.1 Real-time systems

Real-time systems span several domains of computer science. Their application area varies from every day life to very critical systems. A *real-time system* is any system led by a real-time program in which the time at which output is produced is significant. A non-real-time program is correct when the output meets the specifications. The correctness of a real-time program depends on both the correctness of the outputs and their compute time. A real-time program that calculates the correct output after the deadline[1] has passed is incorrect.

Real-time systems may be classified as either hard or soft. A *hard real-time system* is one in which a single missed deadline causes the complete failure of the system. Some circumstances involving hard real-time systems are the airbag inflation after a crash, the landing gear and the fly-by-wire systems of a plane, the over-temperature monitor in nuclear power plants, ECG medical monitor and many others. In *soft real-time system* there is some flexibility in the real-time requirement, and the missing of a deadline produces only an inconvenience but not the failure of the system. For instance, is a soft real-time system the live streaming of a video. Streaming applications require timely delivery of information, but a lag of some TCP packages doesn't compromise the integrity of the system.

Embedded real-time systems and context switches

Hard real-time systems are usually *embedded systems* and vice versa. Most of real-time embedded systems are implemented with a single core processor, and most of these systems have multiple real-time program operations running concurrently. In general, many embedded real-time systems have more concurrent tasks than processors, and the result is that each embedded processor may be responsible for a number of related control procedures. Running concurrent operations on a single processor requires the interleaving of instructions from each process. This is managed by a *task scheduler*, which aim is to determine the execution order and the time slice for each concurrent process. Switching the processor from executing the instructions of one process to those of another process is known as a *context switch*. A context switch

[1]A deadline is a given time by which an activity must be completed.

requires the scheduler to save the current state of the processor and restore the state of the process being resumed. The state of a process includes all the registers that the process may be using, the program counter and the state of the memory.
There are many ways to manage the task scheduling: in many scheduling algorithms, a task may be preempted by the scheduler because a higher priority task is now ready or the task has used up its allotment of processor time.

1.2 The Cheddar project

Cheddar is a free real-time scheduling framework developed by a team from the *Lab-STICC*, of *Université de Bretagne Occidentale*. The development of the project started in 2002, motivated by the lack of free, flexible and open scheduling tools. For portability and maintainability reasons it's written in Ada, and the graphical editor is made with GtkAda[2]. Cheddar runs on all main systems such Solaris, Linux, Windows and on every platform supporting GNAT/GtkAda. The framework is designed for checking the temporal behaviour of real-time applications, which most of the time have to comply with temporal constraints like response times, execution rates or deadlines. Detailed information about Cheddar can be found in [8] and [9].

Applications are defined in Cheddar by a set of processors, buffers, shared resources, messages and tasks. Each periodic task t_i represents a different concurrent operation, and it is defined by its deadline D_{t_i}, its period P_{t_i} and its capacity C_{t_i}. The task t_i is woken up every P_{t_i} units of time, then it does his job whose execution is bounded by C_{t_i} units of time. The job has to end before D_{t_i} units of time after the task wake up moment. Starting from a set of tasks, cheddar provides two kind of features: a simulation engine and feasibility tests.

Simulation engine

Scheduling simulation consists in predicting, for each unit of time and for each processor, which task will be executed and on which processor the task will perform its execution. Cheddar is able to simulate the scheduling with most of usual real-time algorithms and provides a set of useful information such worst, best, average case response time and blocking time, number of preemptions, number of context switches and buffer utilization factor.

Feasibility testing

Feasibility tests allow the study of real-time applications, without computing a task scheduling, in the case of single core processors, multi core processors and distributed systems[3]. Cheddar also provides feasibility tests focused on systems which are less studied by the community, like shaded-buffer systems or systems with task priority. Feasibility tests can be applied instead of a scheduling simulation in certain situations, for instance if a scheduling simulation is too long to compute. Cheddar provides feasibility tests based on different properties: processor utilization factor, task response time, buffer utilization factor, probabilistic properties on task deadlines and many others.

[2]GtkAda is an Ada graphical toolkit providing the complete set of Gtk+ widgets using the object-oriented features of this language.

[3]A distributed system is one whose concurrent processes are assigned to different computers connected by a network.

2 Scope and achievements

The main goal of this thesis is to implement data flow analysis that will be integrated into the Cheddar framework. In addition to conventional static and dynamic analysis, a new kind of analysis, called hybrid analysis, is presented and implemented. Data flow analysis allows a precise analysis of useful data to save when performing context switches in preemptive scheduling, which are computationally intensive and expensive in terms of time.
It follows that more precise estimates of memory usage allow to minimize the amount of data that have to be saved, and restored, during context switches. Cheddar will benefit from this work: the less time each context switch takes, the more efficient will be the task scheduling.

The final result of this thesis is a reliable and efficient stand-alone Ada application which can be easily integrated in the Cheddar framework. Each module of the software was tested with more than ten different control flow graphs and optimized for reaching good performances in terms of speed.

3 Relevant technologies

The algorithms proposed in the following chapters were developed in Ada. All the code work was written and compiled on a Linux system, and the used compiler is GNAT[4]. The choice of Ada was driven by the previous work made with Cheddar, which is written in Ada as well.

The ADA language

Ada is a modern high-level programming language designed for large, long-lived and embedded applications, where reliability and efficiency are essential. It was originally developed in 1977 by the U.S. Department of Defense, and was revised and enhanced in the early 1990s. The resulting language, Ada 95, was the first internationally standardized object-oriented language. The name "Ada" was chosen in honor of Augusta Ada Lovelace (1815-1852), a mathematician who is sometimes regarded as the world's first programmer because of her work with Charles Babbage. Ada is seeing significant usage worldwide in high-integrity, safety-critical and high-security domains including commercial and military aircraft avionics, air traffic control, railroad systems, and medical devices.
Ada is a good teaching language for both introductory and advanced computer science courses, and it has been the subject of significant university research especially in the area of real-time technologies.

Instances of systems, in operation or under active development, in which Ada is used at least to a significant degree, are:

- the air traffic management system of many countries in the world, including the European air traffic flow management and the one of China, France, Germany, United Kingdom;
- the metrorail systems of more than 10 cities including Hong Kong, London and Paris;
- French national railways, Channel Tunnel and the TGV French high-speed rails;

[4]GNAT is a free compiler for Ada95, integrated into the GCC compiler system.

- commercial rockets Atlas V, Delta II and Delta IV;
- NASA space shuttle training aircrafts;
- X-35 joint strike fighter;
- banking and financial systems of at least eight large European financial institutions;
- ISS flight software;
- Czech nuclear shutdown system;
- Pratt & Whitney aircraft engines;

4 Overview of the dissertation

This thesis addresses several techniques of data flow analysis and presents an Ada application that implements each algorithm exposed in the dissertation.
In chapter 2 we give the basics of graph theory and some notes about the data structures used in program analysis, which are essential to understand the underlying idea behind the different types of analysis.
Static analysis is presented in chapter 3 as well as the most typical example of static analysis: the reaching definitions analysis. In the chapter are also defined some concepts used in later chapters.
In chapter 4 is shown dynamic analysis that, despite the static one, takes into account the real execution of a program.
Further, a novel approach to data flow analysis is introduced in chapter 5. Hybrid analysis considers executions of part of the code and makes inferences about definitions and uses of variables.
Finally, the conclusions of the work are presented in chapter 6. Some printouts of the application developed in Ada, which implements all the three kind of analysis, will be found in the appendix section.

Chapter 2

Preliminaries

1 Notes on graph theory

Conceptually, a *graph* is formed by *vertices*, or *nodes*, and *edges* connecting the vertices. Graphs can be used to visualize related data, show knowledge in a graphical way and represent many discrete structures. Graph theory provides algorithms to solve multiple kind of problems, such the finding of the shortest path from one vertex to another vertex. Below a more formal definition of graph is provided.

Definition 2.1 (Graph)**.** A *graph* G consists of a finite nonempty set V, together with a symmetric binary relation δ on V such that $v \not\delta\, v \quad \forall v \in V$. According to [2], we can define a graph in a different but equivalent way: a grapth G as an ordered pair $G = (V, E)$ where $V \neq \emptyset$ and E is a set of subsets of V having cardinality 2. The elements of V are called *vertices* of the graph, and the elements of E are called *edges* of the graph.

Therefore, the two edges (v_1, v_2) and (v_2, v_1), with $v_1, v_2 \in V$, are the same. In other words, the pair is not ordered.

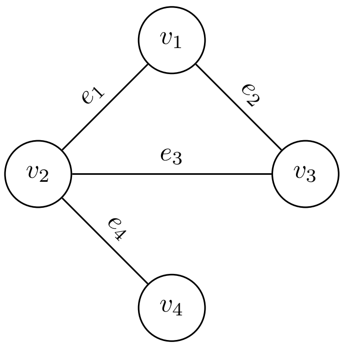

Figure 1: Graph example

It's possible to represent *directed edges*[1] just removing the symmetric property from the binary relation of the graph. What we obtain is called *directed graph*.

[1]An edge represented by an ordered pair.

Definition 2.2 (Directed graph)**.** A *directed graph* D, also called *digraph*, is a finite nonempty set V with a binary relation E on V. Therefore $E \subseteq V \times V$. Likewise to the previous definition, the elements of V are called *vertices* of the directed graph, and the elements of E are called *edges* of the directed graph.

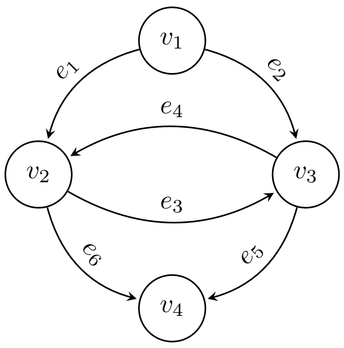

Figure 2: Directed graph example

Given a graph $G = (V, E)$, a *walk* or *path* is a sequence $v_0, e_1, v_1, \ldots, v_k$, where $v_i \in V$ and $e_i \in E$, such that for $1 \leq i \leq k$, the edge e_i has endpoints v_{i-1} and v_i. The *length of a walk* is its number of edges.

In the following sections, we will represent a program as a directed graph.

2 Program representation

Analyses are performed on programs, which are given as a *sequence of instructions*. Instructions are either machine instructions or more generally minimal statements in the language the analysis works on. The given sequence of instructions is split into *basic blocks*, which are the basics of analysis.

In this discussions we will refer to a particular instruction by using its line number.

2.1 Basic blocks

A *basic block* is a maximal sequence of consecutive statements of a program with a single entry point, a single exit point and no internal branches. The first statement of a basic block is the *leader* of the basic block. Under these circumstances, whenever a leader is executed, the remaining statements of the basic block are necessarily executed, in order, exactly once.

Finding basic blocks

Giving the code of a program as input, the procedure for finding basic blocks is divided in two parts:

1. identify the leaders in the code. Leaders are instructions which come under any of the following 3 categories:
   - the first instruction of the program is a leader;
   - the target of a conditional or an unconditional jump instruction is a leader;

- the instruction that immediately follows a conditional or an unconditional jump instruction is a leader.

2. for each leader, the set of all following instructions until and not including the next leader is the basic block corresponding to the considered leader.

Example 2.3 (Build basic blocks)**.** Starting from the following C function, which calculates Fibonacci numbers, we will identify basic blocks.

```
int fib(int n) {
  int a = 0;
  int b = 1;
  int c, i;
  if(n <= 1)
    return n;
  for (i = 2; i <= n; i++) {
    c = a + b;
    a = b;
    b = c;
  }
  return b;
}
```

In order to better point out the jumps, in this example we will rewrite the function using the three-address code. In a real case, this step is not needed and can be substituited by replacing complex control structures with simpler statements.
We can apply the first step of the algorithm and recognise the leaders. Each leader is underlined.

```
 1  read n
 2  a := 0
 3  b := 1
 4  if n <= 1 goto 13
 5  i := 2
 6  if i <= n goto 8
 7  return b
 8  c := a + b
 9  a := b
10  b := c
11  i := i + 1
12  goto 6
13  return n
```

Then, we can create the basic blocks as indicated in the second step of the algorithm.

b_1

```
1  read n
2  a := 0
3  b := 1
4  if n <= 1 goto 13
```

b_2

```
5  i := 2
```

b_3

```
6  if i <= n goto 8
```

b_4

```
7  return b
```

b_5

```
8  c := a + b
9  a := b
10 b := c
11 i := i + 1
12 goto 6
```

b_6

```
13 return n
```

2.2 Control flow graphs

Most static and dynamic analyses work along the control flow of a program. The concept of control flow has to be approximated by a data structure for further analysis. A *control flow graph*, or *CFG*, is a directed graph abstracting the control flow behaviour of a program.

Definition 2.4 (Control flow graph)**.** A *control flow graph* is a directed graph that represents all the paths that might be traversed through a program during its execution. The nodes of the directed graph represent basic blocks and edges represent possible transfer of control flow from one basic block to another. From now on we will indicate as $B = \{b_1, b_2, \ldots, b_n\}$ the set of all basic blocks of a control flow graph.

Given two basic blocks b_1 and b_2, there is a directed edge from b_1 to b_2 if b_2 can immediately follow b_1 in some execution sequence. That is if either:

- there is a branch from last statement in b_1 to the leader of b_2;
- b_2 immediately follows b_1, and b_1 does not end with an unconditional branch.

We say that b_1 is a *predecessor* of b_2, and b_2 is a *successor* of b_1. The basic blocks to which control may transfer after reaching the end of a basic block b_i are called block's *successors* and they are indicated as $Succ\,(b_i)$, while the basic blocks from which control may have come when entering a basic block are called block's *predecessors* and they are indicated as $Pred\,(b_i)$.

One node is distinguished as *initial node*. It is the basic block whose leader is the first statement of the program and it has no predecessors. An artificial single entry node can be created if there are multiple entries. The creation is trivial: just add an edge from the artificial entry node to every original entry node.

Another special node is the *exit node*. It is the basic block whose last statement is the last instruction of the program and it has no successors. If there are multiple exit nodes, an artificial exit node can be created. The approach to create this node is the same as the previous case. For simplicity we assume an unique entry node and an unique exit node in later discussions.

Example 2.5 (Build control flow graph)**.** Based on the example 2.3, an instance of control flow graph is provided below.

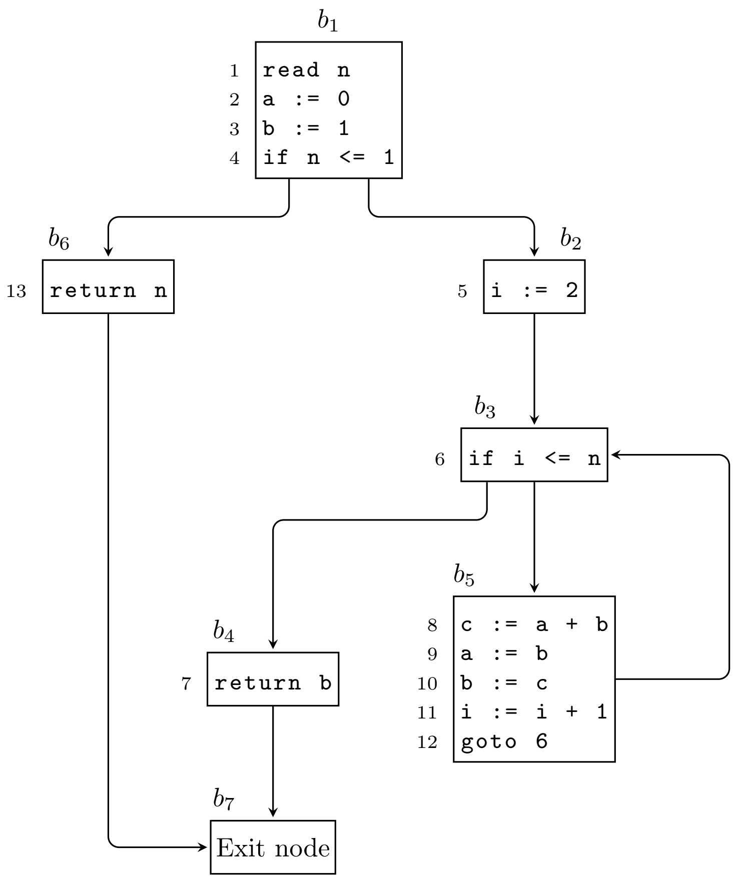

Figure 3: Control flow graph example

Chapter 3

Static analysis

Static analysis is a technique of program analysis used for discovering useful information about the behaviour of programs without executing them. It computes information about the control and data flows for each program point in the program being analyzed. This information is a pessimistic approximation of the properties of the run-time behaviour of the program during each possible execution. This technique of program analysis was originally designed in the context of optimization performed by compilers and nowadays this remains the most common application. It's also used for reliability and correctness tests.
We will use the static analysis as the base for other forms of data flow analysis, presented in the next chapters. There are lots of possible static analysis, some of them are:

- *reaching definition analysis*: determines which definitions may reach a given point in the code;
- *liveness analysis*: calculates for each program point the variables that may be potentially read before their next write;
- *definite assignment analysis*: ensures that a variable or location is always assigned to before it is used;
- *available expression analysis*: determines for each point in the program the set of expressions that need not be recomputed;
- *constant propagation analysis*: evaluates constant expressions at compile time and simultaneously removes dead code.

For our purpose we will focus on the reaching definition analysis.

1 Reaching definitions

A *definition* is a statement where a variable initializes or changes its value.

Definition 3.1 (Reaching definition)**.** A definition d of a variable var reaches a point p if there exists a path from the point d to the point p such that var is not redefined along that path. In that case, d is a *reaching definition* for the point p.

A definition d is an ordered pair $d = (n, a)$ where n represents the name of the defined variable, and a represents the line number containing the definition statement.

Definition 3.2 (Generated definitions)**.** The *generated definitions* $Gen\left(b_i\right)$ is the set of definitions created in basic block b_i.

Definition 3.3 (Killed definitions)**.** The *killed definitions* $Kill\left(b_i\right)$ is the set of definitions modified in basic block b_i.

The reaching definitions analysis computes two sets for each node b_i: the $In\left(b_i\right)$ set, which contains all the definitions valid when the execution path enters the node b_i, and the $Out\left(b_i\right)$, which contains all the definitions valid when the execution path ends the node b_i.

The data flow equations which define the required analysis are:

$$In\left(b_i\right) = \bigcup_{b_p \in Pred\left(b_i\right)} Out\left(b_p\right) \tag{3.1}$$

$$Out\left(b_i\right) = Gen\left(b_i\right) \cup \left(In\left(b_i\right) - Kill\left(b_i\right)\right) \tag{3.2}$$

In other words, the set of reaching definitions valid at the entry of a basic block b_i are all of the reaching definitions from the predecessors of b_i. $Pred\left(b_i\right)$ consists of all of the basic blocks that come immediately before b_i in the control flow graph. The reaching definitions coming out of b_i are all reaching definitions of its predecessors minus those reaching definitions whose variable is killed by b_i plus any new definitions generated within b_i.

Reaching definitions analysis will be used later for connecting definitions to their uses.

2 Algorithms

The most common way of solving the data-flow equations is by using an iterative algorithm. It starts with an init stage that approximates the $In\left(b_i\right)$ and $Out\left(b_i\right)$ sets of each basic block. From these, the $In\left(b_i\right)$ and the $Out\left(b_i\right)$ sets are updated by applying the transfer functions 3.1 and 3.2. The latest step is repeated until we reach a fixpoint: the situation in which the $In\left(b_i\right)$ sets (and the $Out\left(b_i\right)$ sets in consequence) do not change.

2.1 The round-robin algorithm

A basic algorithm for solving reaching definitions equations is the *round-robin iterative algorithm*:

```
for all nodes $b_i \in B$ do
    $In\left(b_i\right) = \emptyset$
    $Out\left(b_i\right) = Gen\left(b_i\right)$
end for
while In-sets are still changing do
    for all nodes $b_i \in B$ do
        $In\left(b_i\right) = \bigcup_{b_p \in Pred\left(b_i\right)} Out\left(b_p\right)$
        $Out\left(b_i\right) = Gen\left(b_i\right) \cup \left(In\left(b_i\right) - Kill\left(b_i\right)\right)$
    end for
end while
```

The second for loop can be optimized by not executing the loop for the initial node, which has no predecessors and consequently does not vary after the init stage.

Example 3.4 (Execution round-robin algorithm)**.** In the following example we simulate the execution of the algorithm on the control flow graph shown in figure 7.

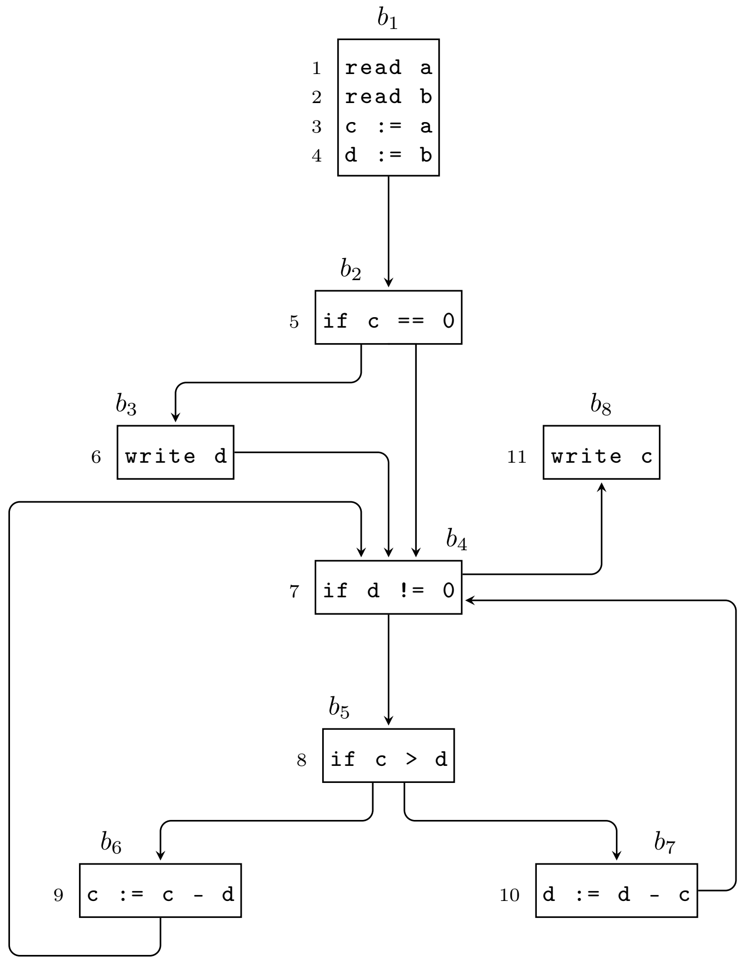

Figure 4: Control flow graph

The first cycle of the algorithm initializes, for each basic block, the $In\left(b_i\right)$ set with an empty set and the $Out\left(b_i\right)$ set with its generated definitions. After this step, this is the state of the sets:

Block	$In\left(b_i\right)$	$Out\left(b_i\right)$
b_1	$\emptyset$	$\left\{(a,1),(b,2),(c,3),(d,4)\right\}$
b_2	$\emptyset$	$\emptyset$
b_3	$\emptyset$	$\emptyset$
b_4	$\emptyset$	$\emptyset$
b_5	$\emptyset$	$\emptyset$
b_6	$\emptyset$	$\left\{(c,9)\right\}$
b_7	$\emptyset$	$\left\{(d,10)\right\}$
b_8	$\emptyset$	$\emptyset$

Now we iterate the second cycle until a fixpoint is reached. After two iterations[1] we have these final sets:

[1]Iteration #2 have the same $In\left(b_i\right)$ and $Out\left(b_i\right)$ sets of iteration #1.

Block	Initialization		Iteration #1		Iteration #2	
	$In\,(b_i)$	$Out\,(b_i)$	$In\,(b_i)$	$Out\,(b_i)$	$In\,(b_i)$	$Out\,(b_i)$
b_1	$\emptyset$	$\{(a,1), (b,2), (c,3), (d,4)\}$	$\emptyset$	$\{(a,1), (b,2), (c,3), (d,4)\}$	-	-
b_2	$\emptyset$	$\emptyset$	$\{(a,1), (b,2), (c,3), (d,4)\}$	$\{(a,1), (b,2), (c,3), (d,4)\}$	-	-
b_3	$\emptyset$	$\emptyset$	$\{(a,1), (b,2), (c,3), (d,4)\}$	$\{(a,1), (b,2), (c,3), (d,4)\}$	-	-
b_4	$\emptyset$	$\emptyset$	$\{(a,1), (b,2), (c,3), (d,4), (c,9), (d,10)\}$	$\{(a,1), (b,2), (c,3), (d,4), (c,9), (d,10)\}$	-	-
b_5	$\emptyset$	$\emptyset$	$\{(a,1), (b,2), (c,3), (d,4), (c,9), (d,10)\}$	$\{(a,1), (b,2), (c,3), (d,4), (c,9), (d,10)\}$	-	-
b_6	$\emptyset$	$\{(c,9)\}$	$\{(a,1), (b,2), (c,3), (d,4), (c,9), (d,10)\}$	$\{(a,1), (b,2), (d,4), (c,9), (d,10)\}$	-	-
b_7	$\emptyset$	$\{(d,10)\}$	$\{(a,1), (b,2), (c,3), (d,4), (c,9), (d,10)\}$	$\{(a,1), (b,2), (c,3), (c,9), (d,10)\}$	-	-
b_8	$\emptyset$	$\emptyset$	$\{(a,1), (b,2), (c,3), (d,4), (c,9), (d,10)\}$	$\{(a,1), (b,2), (c,3), (d,4), (c,9), (d,10)\}$	-	-

The order in which nodes are taken out from the set B depends on the implementation of the algorithm. In the example above we can notice that the nodes were visited in the order $b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4 \rightarrow b_5 \rightarrow b_6 \rightarrow b_7 \rightarrow b_8$, and fixpoint was reached after only two iterations. Intuitively this algorithm reaches its highest efficiency when all predecessors of a basic block have been processed before the basic block itself, since then the iteration will use the latest information. For instance, the execution of the same algorithm where the nodes are visited in the reverse postorder $b_8 \rightarrow b_7 \rightarrow b_6 \rightarrow b_5 \rightarrow b_4 \rightarrow b_3 \rightarrow b_2 \rightarrow b_1$ takes 4 iterations. Thus, we can assert that the efficiency of the algorithm is influenced by the order at which local nodes are visited.

2.2 The worklist algorithm

The *worklist iterative algorithm* improves on the round-robin algorithm by focusing the iteration on regions in the graph where information is changing. Simple data-structure modifications to the previous algorithm can make it more efficient:

```
$Worklist = \emptyset$
for all nodes $b_i \in B$ do
  $In(b_i) = \emptyset$
  $Out(b_i) = Gen(b_i)$
  Add $b_i$ to $Worklist$
end for
while $Worklist \neq \emptyset$ do
  Remove a node $b_i$ from $Worklist$
  $In(b_i) = \bigcup_{b_p \in Pred(b_i)} Out(b_p)$
  $Out(b_i) = Gen(b_i) \cup (In(b_i) - Kill(b_i))$
  if the new $In(b_i) \neq$ old $In(b_i)$ then
    Add $Succ(b_i)$ to $Worklist$, uniquely
  end if
end while
```

The algorithm begins by initializing the sets for each node and constructing an initial worklist. It then repeats the process of removing a node from the worklist and updating its In and Out sets according to the data flow equations. If the update changes the $In(b_i)$ set of the basic block, then all of the basic blocks that depend on the changed information are added to the worklist In other words any of its successors that are not already on the worklist are added to the worklist. In this way, the algorithm avoids recomputing the equations of a basic block where none of the facts have changed.

3 Def-use associations

A *use of a variable* is a reference to the variable, either in a predicate or a computation.

Reaching definitions analysis can be used for constructing *def-use associations* which connect definitions to their uses, procuring more precise information on actual values of variables. Static def-use information has been shown to be useful not only for optimizing and parallelizing compilers but also for testing and maintenance of programs. It's also commonly used in debugging: with def-use chains it's possible to prevent uses of variables without having defined them before. Later in the discussion, we will use def-use associations to test if a basic block contains minimal information.

Definition 3.5 (Variable uses)**.** The *variable uses* set, indicated by $Use(b_i)$, is the set whose elements are all the variable uses available in a basic block b_i. A variable use is indicated by an ordered pair $u = (n, g)$ where n represents the name of the used variable, and g represents the point in the program where the the variable is used.

Example 3.6 (Variable uses of a basic block)**.** Below is represented the basic block b_1 of the example shown in fugure 7.

b_1

```
1 read a
2 read b
3 c := a
4 d := b
```

The uses related to this node are:

$$Use\,(b_1) = \{(a, 3)\,, (b, 4)\} \tag{3.3}$$

Definition 3.7 (Def-use associations)**.** A *def-use association* for variable n is an ordered tuple $du = (n, d, u)$ where n is the name of the variable, d is the point where variable n is defined and u is the point where variable n is used, and is valid if there is a path with no redefinitions from d to u. The set containing all the def-uses associations of a basic block b_i is indicated by $DefUse\,(b_i)$.

3.1 The def-use algorithm

The *def-use algorithm* computes the $DefUse\,(b_i)$ for a given basic block b_i of the control flow graph. This local algorithm is based on the result of the reaching definitions algorithm explained before.

```
for all uses $u_j = (n_j, g_j) \in Use\,(b_i)$ do
  $b_{t_i}$ = the sub-block containing all the statements located before point $g_j$
  if $Gen\,(b_{t_i}) \neq \emptyset$ and $Gen\,(b_{t_i})$ contains at least a definition for variable $n_j$ then
    Select $d_m = (n_m, a_m) \in Gen\,(b_{t_i})$ closest to point $g_j$ having $n_m = n_j$
    Add $du_j = (n_j, a_m, g_j)$ to $DefUse\,(b_i)$
  else
    for all definitions $d_k = (n_k, a_k) \in In\,(b_i)$ having $n_k = n_j$ do
      Add $du_j = (n_j, a_k, g_j)$ to $DefUse\,(b_i)$
    end for
  end if
end for
```

For every use $u_j = (n_j, g_j)$ of a node, the algorithm checks if there is a definition for the variable n_j. At first, the definition is searched inside the block: if there are more than one definitions in the node itself, the one situated before the point g_j and in the nearest point is chosen, then the $DefUse\,(b_i)$ set is updated with the new def-use element. If no definitions are present inside the block and before point g_j, the search continues in the $In\,(b_i)$ set. For each definition in that set, defined for the variable n_j, a def-use element is added to the $DefUse\,(b_i)$ set. This is computed for each node of the control flow graph.

Example 3.8 (Def-use associations of a basic block)**.** In this example is presented the computation of the $DefUse\,(b_i)$ set for just one node. In a realistic situation, the computation must be extended at every node in the control flow graph.

Consider the control flow graph shown in figure 7 of the previous example, and consider the computation of the $DefUse\,(b_5)$ for basic block b_5. We already have the sets $In\,(b_i)$, $Out\,(b_i)$ from the previous steps of the static analysis, and the $Use\,(b_i)$ can be easily computed with a simple local algorithm.

Block	$In\,(b_i)$	$Out\,(b_i)$	$Use\,(b_i)$
b_1	$\emptyset$	$\{(a,1),\ (b,2),\ (c,3),\ (d,4)\}$	$\{(a,3),\ (b,4)\}$
b_2	$\{(a,1),\ (b,2),\ (c,3),\ (d,4)\}$	$\{(a,1),\ (b,2),\ (c,3),\ (d,4)\}$	$\{(c,5)\}$
b_3	$\{(a,1),\ (b,2),\ (c,3),\ (d,4)\}$	$\{(a,1),\ (b,2),\ (c,3),\ (d,4)\}$	$\{(d,6)\}$
b_4	$\{(a,1),\ (b,2),\ (c,3),\ (d,4),\ (c,9),\ (d,10)\}$	$\{(a,1),\ (b,2),\ (c,3),\ (d,4),\ (c,9),\ (d,10)\}$	$\{(d,7)\}$
b_5	$\{(a,1),\ (b,2),\ (c,3),\ (d,4),\ (c,9),\ (d,10)\}$	$\{(a,1),\ (b,2),\ (c,3),\ (d,4),\ (c,9),\ (d,10)\}$	$\{(c,8),\ (d,8)\}$
b_6	$\{(a,1),\ (b,2),\ (c,3),\ (d,4),\ (c,9),\ (d,10)\}$	$\{(a,1),\ (b,2),\ (d,4),\ (c,9),\ (d,10)\}$	$\{(c,9),\ (d,9)\}$
b_7	$\{(a,1),\ (b,2),\ (c,3),\ (d,4),\ (c,9),\ (d,10)\}$	$\{(a,1),\ (b,2),\ (c,3),\ (c,9),\ (d,10)\}$	$\{(d,10),\ (c,10)\}$
b_8	$\{(a,1),\ (b,2),\ (c,3),\ (d,4),\ (c,9),\ (d,10)\}$	$\{(a,1),\ (b,2),\ (c,3),\ (d,4),\ (c,9),\ (d,10)\}$	$\{(c,11)\}$

Now we must find the definitions related to every use in the set $Use\,(b_5) = \{(c,8),\ (d,8)\}$. For the first use $(c,8)$ there are no definitions in the statements of node b_5 before line 8, so the definitions contained in the $In\,(b_5)$ set are checked: the two definitions for the variable c are $(c,3)$ and $(c,9)$. Both the def-use associations are added to the set, that becomes $DefUse\,(b_5) = \{(c,3,8),\ (c,9,8)\}$. The computation of the def-use associations for the second use $(d,8)$ is similar. The final set for block b_5 is:

$$DefUse\,(b_5) = \{(c,3,8)\,,(c,9,8)\,,(d,4,8)\,,(d,10,8)\} \tag{3.4}$$

4 Forward def-use associations

For the purpose testing and debugging, the def-use association related to a node b_i works well. However, to optimize the context switch time, we introduce another way to represent the same information. A node b_i and its $Gen\,(b_i)$ set are given. To know where each definition generated by the node b_i is used, we must visit each node $b_k \in Succ^*\,(b_i)$ and, for each node, visit its def-use associations.
$Succ^*\,(b_i)$ represents the transitive closure of $Succ\,(b_i)$, and a simple algorithm to compute this set is provided in section 2 of chapter 5. When a def-use association $du_k = (n_k, d_k, u_k)$ has n_k and d_k such that $(n_k, d_k) \in Gen\,(b_i)$, we know that the node b_k might use the variable n_k defined in node b_i.

With the *forward def-use association* it is possible to see this kind of information at a glance.

Definition 3.9 (Forward def-use associations)**.** A *forward def-use association* $ForwardDefUse\,(b_i)$ is a sequence of n sets, where n is the number of basic blocks in the control flow graph. The set in position k, where $1 \leq k \leq n$, represents a subset of $DefUse\,(b_k)$ whose elements refer to a variable defined in node b_i and used in node b_k.

The idea behind the algorithm for computing this information is to visit the $DefUse$ set of each node and, for each def-use association, copy this association onto the $ForwardDefUse$ data structure of the appropriate node in the proper position.

The example 4.1 of chapter 4 will show the computation of the $ForwardDefUse$ set.

5 Minimal information in basic blocks

Static analysis is safe and pessimistic, because it must consider all the possible behaviours of a program. The $DefUse\,(b_i)$ set of a node b_i might contain multiple entries for a given used variable: if this situation occurs, it means that the valid definition of an used variable at a given point during execution, is known only at run-time.
On the other hand, a basic block b_i is said to have *minimal information* if each variable used into it has exactly one reaching definition, or in other words, if for each variable used in the node we know where the definition is located even before the execution of the program.
When static analysis provides minimal information for a block, we know that this information is the most precise.

Algorithm

This algorithm tests if a block has minimal information or not. The input of the function is a $DefUse$ set, the output is a boolean value.

```
**for** $k \leftarrow 1$ **to** $|DefUse\,(b_i)|$ **do**
  Let $du_k = (n_k, d_k, u_k)$ be the $k$-element of $DefUse\,(b_i)$
  **for** $j \leftarrow (k+1)$ **to** $|DefUse\,(b_i)|$ **do**
    Let $du_j = (n_j, d_j, u_j)$ be the $j$-element of $DefUse\,(b_i)$
    **if** $n_k = n_j$ **and** $u_k = u_j$ **then**
      **return false**
    **end if**
  **end for**
**end for**
**return true**
```

Note that the initial node of a control flow graph have always minimal information.

Chapter 4

Dynamic analysis

In chapter 3 we discussed about static analysis, which allows the testing and evaluation of an application by examining the code without executing the program. It is a conservative analysis because it produces information concerning all the possible paths, in fact because there are many possible executions, the analysis must keep track of multiple different possible states.
Dynamic analysis is able to capture and predict more precisely the behaviour of a program, but despite static analysis, the dynamic one requires the path (or a part of it) done over the control flow graph. That's why this kind of analysis is done at run-time. Because of that, the results of dynamic analysis cannot be generalized to future executions of the program. However, these results can be useful for simulation purposes.

1 The idea

The analysis starts from a control flow graph and a path $b_1, e_1, b_2, e_2, \ldots, b_n$ of nodes and edges representing a particular execution of the program. The idea is to remove all the edges from the control flow graph and add them, one by one, at run-time every time the execution of the program switches from a node to the next in the path. When a switch is done an edge is created, and the previous one is deleted. Having this control graph that still change at run-time, we can apply the static equations for the reaching definition analysis for each node in the path and compute the def-use and the forward def-use sets as well.

2 Algorithm

Given a path, representing a specific execution of the program, and a control flow graph, the algorithm that performs the dynamic analysis is the following:

```
for all nodes $b_i$ in the execution path do
    $In(b_i) = \bigcup_{b_p \in Pred(b_i)} Out(b_p)$
    $Out(b_i) = Gen(b_i) \cup (In(b_i) - Kill(b_i))$
    Compute the def-use algorithm to node $b_i$
    if $b_i$ is not the last node in the execution path then
        Let $b_{i+1}$ be the node after $b_i$ in the execution path
        $Pred(b_{i+1}) = \{b_i\}$
    **end if**
  **end for**
  Compute the forward def-use algorithm
```

Note that after computing the information for a node, the algorithm transmits the information to the following node simply setting itself as unique predecessor of its successor node. In this way once the information of the successor are computed, the equations to calculate the In set takes the info from the only predecessor in the list. There is no need to check if the information in basic blocks are minimal or not, since every node computed by dynamic analysis have minimal information. Every variable used inside a node has only one definition, in fact in case of multiple definitions the newer one overwrites the previous definition for that variable.

Example 4.1 (Dynamic analysis of a control flow graph)**.** In figure 5 we have a control flow graph in which is highlighted the following execution path:

$$path = b_1, e_1, b_2, e_2, b_3, e_3, b_4, e_4, b_5, e_5, b_7, e_6, b_4, e_7, b_5, e_8, b_7, e_9, b_4, e_{10}, b_8 \tag{4.1}$$

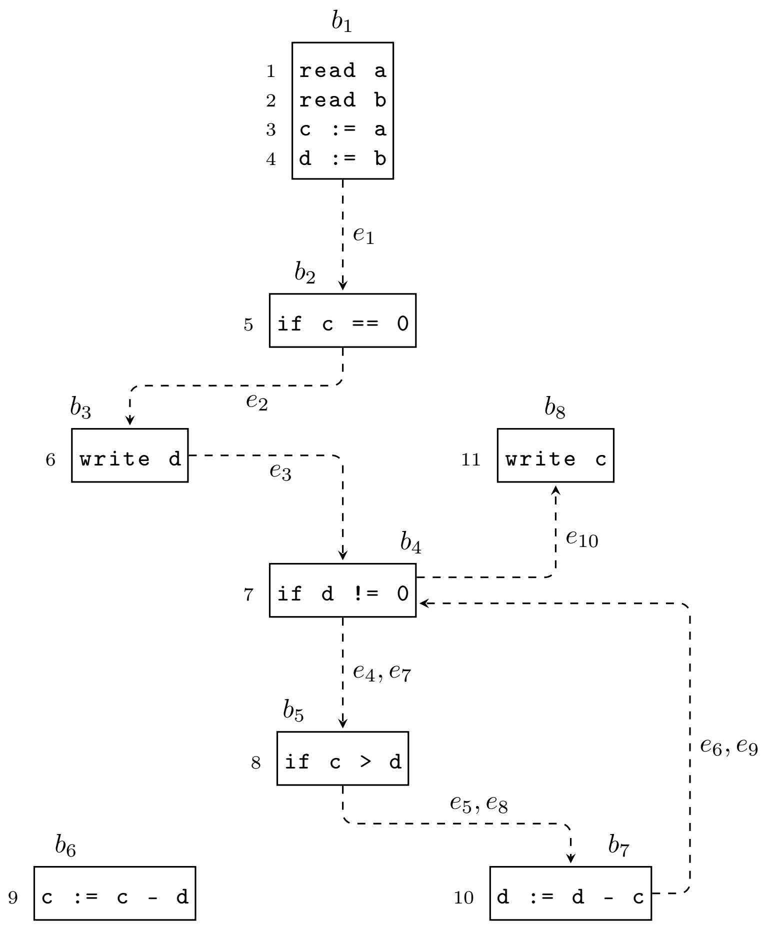

Figure 5: Control flow graph

The algorithm for dynamic analysis is applied: every row in the following table represents, in the order, a visited node.
We can notice that only the nodes belonging to the path are computed in the algorithm. For instance the node b_6, which is not in the path, is not present in the above table. Notice also that node b_4 is evaluated three times, but the information computed on the second time has no differences between the third one. In our implementation of the dynamic analysis we added this kind of control and the program prints only the nodes whose information change during the execution.

Block	$In\,(b_i)$	$Out\,(b_i)$	$DefUse\,(b_i)$	$ForwardDefUse\,(b_i)$
b_1	$\emptyset$	$\{(a,1), (b,2), (c,3), (d,4)\}$	$\{(a,1,3), (b,2,4)\}$	$(\{(a,1,3), (b,2,4)\}, \{(c,3,5)\}, \{(d,4,6)\}, \{(d,4,7)\}, \{(c,3,8), (d,4,8)\}, \{(c,3,9)\}, \{(d,4,10), (c,3,10)\}, \emptyset)$
b_2	$\{(a,1), (b,2), (c,3), (d,4)\}$	$\{(a,1), (b,2), (c,3), (d,4)\}$	$\{(c,3,5)\}$	$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
b_3	$\{(a,1), (b,2), (c,3), (d,4)\}$	$\{(a,1), (b,2), (c,3), (d,4)\}$	$\{(d,4,6)\}$	$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
b_4	$\{(a,1), (b,2), (c,3), (d,4)\}$	$\{(a,1), (b,2), (c,3), (d,4)\}$	$\{(d,4,7)\}$	$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
b_5	$\{(a,1), (b,2), (c,3), (d,4)\}$	$\{(a,1), (b,2), (c,3), (d,4)\}$	$\{(c,3,8), (d,4,8)\}$	$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
b_7	$\{(a,1), (b,2), (c,3), (d,4)\}$	$\{(a,1), (b,2), (c,3), (d,10)\}$	$\{(d,4,10), (c,3,10)\}$	$(\emptyset, \emptyset, \emptyset, \{(d,10,7)\}, \{(d,10,8)\}, \{(d,10,9)\}, \{(d,10,10)\}, \emptyset)$
b_4	$\{(a,1), (b,2), (c,3), (d,10)\}$	$\{(a,1), (b,2), (c,3), (d,10)\}$	$\{(d,10,7)\}$	$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
b_5	$\{(a,1), (b,2), (c,3), (d,10)\}$	$\{(a,1), (b,2), (c,3), (d,10)\}$	$\{(c,3,8), (d,10,8)\}$	$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
b_7	$\{(a,1), (b,2), (c,3), (d,10)\}$	$\{(a,1), (b,2), (c,3), (d,10)\}$	$\{(d,10,10), (c,3,10)\}$	$(\emptyset, \emptyset, \emptyset, \{(d,10,7)\}, \{(d,10,8)\}, \{(d,10,9)\}, \{(d,10,10)\}, \emptyset)$
b_4	$\{(a,1), (b,2), (c,3), (d,10)\}$	$\{(a,1), (b,2), (c,3), (d,10)\}$	$\{(d,10,7)\}$	$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
b_8	$\{(a,1), (b,2), (c,3), (d,10)\}$	$\{(a,1), (b,2), (c,3), (d,10)\}$	$\{(c,3,11)\}$	$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$

Chapter 5

Hybrid analysis

We observed that information computed by static analysis are pessimistic but safe estimates of the behaviour of a program, because it must consider all the possible execution paths through the control flow graph.
Dynamic analysis, on the other hand, computes very precise information, but these data are valid only for a specific execution path of a program, and cannot be generalized for every possible execution. Moreover, dynamic analysis must know the execution path a priori.
In real-time systems, as written in section 1 of chapter 1, the context switch time is critical. Both static and dynamic analysis give some useful information that can be used to optimize the time of context switches. In this chapter is presented a fresh approach to data flow analysis that can be used to reduce that time as well.
Hybrid analysis is based on static analysis, but takes into account the moment when a task is interrupted too (in a preemptive task scheduling).

1 The idea

Hybrid analysis makes some inferences to determine if there are some def-use associations that will be no more valid after executing a certain basic block. It starts from the information computed by static analysis and then, evermore statically, computes the consequences, if any, of passing through a node.
The analysis considers all the nodes that define the same variable *var* and that are mutually exclusive executed. For convenience we will call this set of nodes B_m. Later, it checks if there are common descendant[1] nodes between the nodes in B_m which use the same variable *var*. If this situation occurs, we already know that static analysis information will contain, for these nodes, multiple def-use associations for variable *var*. The information that hybrid analysis adds, is that if a node $b_i \in B_m$ is executed, we can delete from the common descendant nodes which use variable *var* all the def-use associations related to nodes $b_k \in B_m$ where $k \neq i$.

Below we define a data structure suitable for storing that kind of information.

Definition 5.1 (Suppressible def-use associations)**.** A *suppressible def-use association* for variable n is an ordered tuple $sdu = (b, n, d, u)$ where b is the index of a node, n is a variable name, d is the point where variable n is defined and u is the point where variable n is used.

[1]The notion of descendant and a practical way to compute the set is presented in section 2.

The set containing all the suppressible def-uses associations of a basic block b_i is indicated by $SuppDefUse\,(b_i)$. The intuitive meaning of the data structure is that if block b is executed, we can delete the def-use association $du = (n, d, u)$ from block b_i.

When executing a task in a preemptive real-time system, we can trace which nodes the program executes and dynamically delete the def-use associations which will be no more valid, helping to decrease the amount of memory data for the task, and therefore reduce the context switch time.

In the example below will be presented a possible use case and will be clarified the concept behind hybrid analysis.

Example 5.2 (Possible application of hybrid analysis)**.** The control flow graph in figure 6 is not a complete one, but represents the interesting part of a control flow graph. We can notice that b_5 and b_6 are mutually exclusive executed, which means that if b_5 will be executed, b_6 will not, and vice versa.

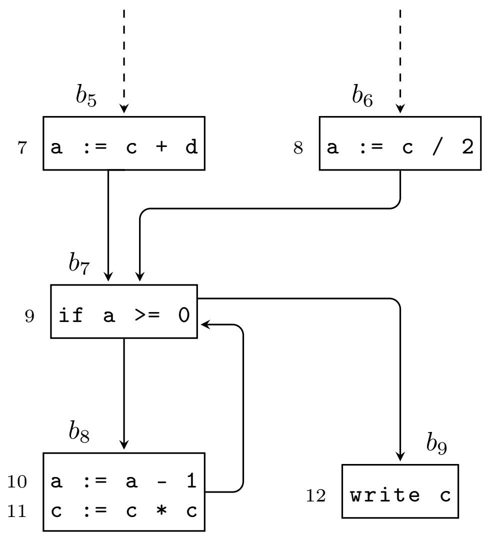

Figure 6: Part of a control flow graph

Both b_5 and b_6 contain a definition for variable a, and both of the nodes have common descendant nodes using the variable a. These nodes are b_7 and b_8.

From static analysis we have this kind of def-use information for b_7 and b_8 related to variable a:

$$DefUse\,(b_7) = \{(a, 7, 9)\,, (a, 8, 9)\,, (a, 10, 9)\,, \dots\} \tag{5.1}$$

$$DefUse\,(b_8) = \{(a, 7, 10)\,, (a, 8, 10)\,, (a, 10, 10)\,, \dots\} \tag{5.2}$$

Hybrid analysis will add the suppressible data structure:

$$SuppDefUse\,(b_7) = \{(6, a, 7, 9)\,, (5, a, 8, 9)\,, \dots\} \tag{5.3}$$

$$SuppDefUse\,(b_8) = \{(6, a, 7, 10)\,, (5, a, 8, 10)\,, \dots\} \tag{5.4}$$

For instance, the meaning of the first element of set 5.3 is that if node b_6 is executed, it is possible to delete the def-use association related to variable a defined in statement 7 and used in statement 9.

We can notice that when a task is interrupted, in order to delete all the def-use definitions which are no more valid, we have to iteratively check the $SuppDefUse$ set of each node at run-time, adding an overhead to the context switch.
Is it possible to overcome this problem by computing this information in a static way, before the execution of the program and saving it in a new data structure, the $ForwardSuppDefUse$, which is conceptually similar to the $ForwardDefUse$ data structure seen in section 4 of chapter 3.

2 Computing of the $Succ^*$ set

Given a node b_i, the set $Succ^*(b_i)$ is the transitive closure of $Succ(b_i)$ and represents the set containing the descendant nodes of b_i. The set $Succ^*(b_i)$ will be used in section 3 to perform hybrid analysis. The algorithm to compute this set is the following:

```
Add the elements of $Succ(b_i)$ to $Succ^*(b_i)$
repeat
  $NothingAdded =$ true
  for all nodes $b_j \in Succ^*(b_i)$ do
    Add the nodes of $Succ(b_j)$ to $Succ^*(b_i)$
    if at least one new element was added then
      $NothingAdded =$ false
    end if
  end for
until $NothingAdded =$ true
```

The algorithm might be optimized by adding a working list as done for the round-robin algorithm in section 2.2 of chapter 3.

3 Algorithm

Given a control flow graph, the algorithm that performs hybrid analysis is the following:

```
Compute static analysis
for all nodes $b_i \in B$ do
  Compute $Succ^*(b_i)$ set
  for all nodes $b_j \notin Succ^*(b_i)$ and $i \neq j$ do
    for all $d_k = (n_k, a_k)$ such that $d_k \in Gen(b_j)$ and $Gen(b_i)$ contains a definition
    for var $n_k$ do
      for all nodes $b_z \in Succ^*(b_i)$ do
        for all def-use associations $du_y = (n_y, a_y, g_y) \in DefUse(b_z)$ for $d_k$ do
          Add $sdu_z = (i, n_k, a_k, g_y)$ to $SuppDefUse(b_z)$
        end for
      end for
    end for
  end for
end for
```

Chapter 6

Conclusions

The aim of this paper was to contribute to the reduction of the context switch time in preemptive real-time systems. The objective was reached by studying and developing algorithms for data analysis that will be easily integrated in Cheddar.
First of all, an overview about the Cheddar project and about real-time systems was given. We explained the fundamentals of graph theory and program representation as well, including the definitions of basic block and control flow graph.
Afterwards three kind of data flow analysis, and their related algorithms, were presented. We started with static analysis, which statically computes information about the definitions and uses of variables. Static analysis gives safe results that are always valid regardless the path that a program can follow at runtime. Because of that, a basic block must contain information about all the possible executions of the program passing through that node. Sometimes static analysis gives the most precise information about a node: in that case the basic block has minimal information.
Then, we focused on dynamic analysis, which overcomes some problems of static analysis. Dynamic analysis provides precise information for any node in the control flow graph: each node has minimal information, however dynamic analysis needs the path that the program did along the control flow graph and, because of that, this type of data analysis cannot provide general information about a program.
Finally, we introduced a novel technique of data analysis: the hybrid analysis. This kind of analysis computes some inferences statically, and once the program is executed, provides valuable information about the validity of definitions and uses of variables which can be used to decrease the context switch time in real-time systems.
An algorithm was developed for all three types of data analysis. Additionally, all the algorithms were written in Ada and tested with several examples.

The dissertation opens some possibilities for future work. First of all, hybrid analysis information could be used to calculate some other related details about the behaviour of a program. It could be interesting also to extend the idea of hybrid analysis for other variety of analysis, for instance constant propagation analysis or available expression analysis.
What is more, the Ada implementation of the algorithms can be improved in terms of memory efficiency and speed execution, for example by caching some intermediate results and replacing some data structure assignments.

On the personal side, this work allowed me to gain new skills and knowledge about real-time systems, data flow analysis and Ada programming language.

Appendix

1 Instances of printouts

In this section are provided some printouts of the program. For each used example, the control flow graph structure is printed besides.

1.1 Static analysis printouts

This instance of static analysis printout is based on the following control flow graph:

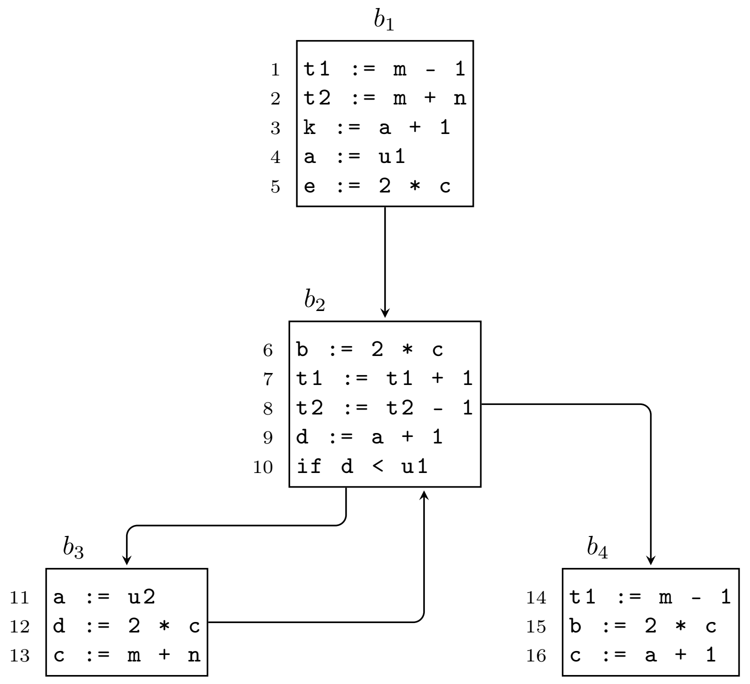

Figure 7: Control flow graph

The printout representing the control flow graph is the following:

```
CFG with 4 blocks:
* Block #1 of type start_node
      with statements:
          Statement #1
              with def_var: t1
              with used_vars: m
          Statement #2
              with def_var: t2
              with used_vars: m, n
          Statement #3
              with def_var: k
              with used_vars: a
          Statement #4
              with def_var: a
              with used_vars: u1
          Statement #5
              with def_var: e
              with used_vars: c
     with previous nodes:
     with next nodes: 2

* Block #2 of type middle_node
     with statements:
          Statement #6
              with def_var: b
              with used_vars: c
          Statement #7
              with def_var: t1
              with used_vars: t1
          Statement #8
              with def_var: t2
              with used_vars: t2
          Statement #9
              with def_var: d
              with used_vars: a
          Statement #10
              with def_var:
              with used_vars: d, u1
     with previous nodes: 1, 3
     with next nodes: 3, 4

* Block #3 of type middle_node
     with statements:
          Statement #11
              with def_var: a
              with used_vars: u2
          Statement #12
              with def_var: d
              with used_vars: c
          Statement #13
              with def_var: c
              with used_vars: m, n
     with previous nodes: 2
     with next nodes: 2

* Block #4 of type terminate_node
     with statements:
            Statement #14
                with def_var: t1
                with used_vars: m
            Statement #15
                with def_var: b
                with used_vars: c
            Statement #16
                with def_var: c
                with used_vars: a
       with previous nodes: 2
       with next nodes:
```

The results of static analysis are the following:

```

Static info:
Info def_uses block # 1:
      Def_in:
      Def_out:
            (t1,1)
            (t2,2)
            (k,3)
            (a,4)
            (e,5)
      Use_out:
            (m,1)
            (m,2)
            (n,2)
            (a,3)
            (u1,4)
            (c,5)
      Def_use_assos:
      Forward_def_use_assos:
         in block # 2:
            (t1,1,7)
            (t2,2,8)
            (a,4,9)
         in block # 4:
            (a,4,16)
      This block contains minimal information

Info def_uses block # 2:
      Def_in:
            (t1,1)
            (t2,2)
            (k,3)
            (a,4)
            (e,5)
            (a,11)
            (d,12)
            (c,13)
            (b,6)
            (t1,7)
            (t2,8)
      Def_out:
```

```
                (b,6)
                (t1,7)
                (t2,8)
                (d,9)
                (k,3)
                (a,4)
                (e,5)
                (a,11)
                (c,13)
        Use_out:
                (c,6)
                (t1,7)
                (t2,8)
                (a,9)
                (d,10)
                (u1,10)
        Def_use_assos:
                (c,13,6)
                (t1,1,7)
                (t1,7,7)
                (t2,2,8)
                (t2,8,8)
                (a,4,9)
                (a,11,9)
                (d,9,10)
        Forward_def_use_assos:
            in block # 2:
                (t1,7,7)
                (t2,8,8)
                (d,9,10)

Info def_uses block # 3:
        Def_in:
                (b,6)
                (t1,7)
                (t2,8)
                (d,9)
                (k,3)
                (a,4)
                (e,5)
                (a,11)
                (c,13)
        Def_out:
                (a,11)
                (d,12)
                (c,13)
                (b,6)
                (t1,7)
                (t2,8)
                (k,3)
                (e,5)
        Use_out:
                (u2,11)
                (c,12)
            (m,13)
            (n,13)
      Def_use_assos:
            (c,13,12)
      Forward_def_use_assos:
         in block # 2:
            (c,13,6)
            (a,11,9)
         in block # 3:
            (c,13,12)
         in block # 4:
            (c,13,15)
            (a,11,16)
      This block contains minimal information

Info def_uses block # 4:
      Def_in:
            (b,6)
            (t1,7)
            (t2,8)
            (d,9)
            (k,3)
            (a,4)
            (e,5)
            (a,11)
            (c,13)
      Def_out:
            (t1,14)
            (b,15)
            (c,16)
            (t2,8)
            (d,9)
            (k,3)
            (a,4)
            (e,5)
            (a,11)
      Use_out:
            (m,14)
            (c,15)
            (a,16)
      Def_use_assos:
            (c,13,15)
            (a,4,16)
            (a,11,16)
      Forward_def_use_assos:
```

1.2 Dynamic analysis printouts

In order to compute dynamic analysis the execution path is provided. It must be written in a text file named *dynamic_path.txt*, located in the main folder of the program, and must contain the sequence of number, representing the basic blocks, separated by new lines.
Here the dynamic path provided for this example:

```
1
2
3
4
5
7
4
5
7
4
8
```

The considered control flow graph is the one in example 3.4 of chapter 3. The printout representing the control flow graph is the following:

```

CFG with 8 blocks:
* Block #1 of type start_node
      with statements:
           Statement #1
               with def_var: a
               with used_vars:
           Statement #2
               with def_var: b
               with used_vars:
           Statement #3
               with def_var: c
               with used_vars: a
           Statement #4
               with def_var: d
               with used_vars: b
      with previous nodes:
      with next nodes: 2

* Block #2 of type middle_node
      with statements:
           Statement #5
               with def_var:
               with used_vars: c
      with previous nodes: 1
      with next nodes: 3, 4

* Block #3 of type middle_node
      with statements:
           Statement #6
               with def_var:
               with used_vars: d
      with previous nodes: 2
      with next nodes: 4

* Block #4 of type middle_node
      with statements:
           Statement #7
               with def_var:
                with used_vars: d
      with previous nodes: 2, 3, 6, 7
      with next nodes: 5, 8

* Block #5 of type middle_node
      with statements:
           Statement #8
               with def_var:
               with used_vars: c, d
      with previous nodes: 4
      with next nodes: 6, 7

* Block #6 of type middle_node
      with statements:
           Statement #9
               with def_var: c
               with used_vars: c, d
      with previous nodes: 5
      with next nodes: 4

* Block #7 of type middle_node
      with statements:
           Statement #10
               with def_var: d
               with used_vars: d, c
      with previous nodes: 5
      with next nodes: 4

* Block #8 of type terminate_node
      with statements:
           Statement #11
               with def_var:
               with used_vars: c
      with previous nodes: 4
      with next nodes:
```

The results of dynamic analysis are the following:

```

Dynamic info:
Info def_uses block # 1:
      Def_in:
      Def_out:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Use_out:
            (a,3)
            (b,4)
      Def_use_assos:
            (a,1,3)
            (b,2,4)
      Forward_def_use_assos:
         in block # 1:
            (a,1,3)
            (b,2,4)
         in block # 2:
            (c,3,5)
         in block # 3:
            (d,4,6)
         in block # 4:
            (d,4,7)
         in block # 5:
            (c,3,8)
            (d,4,8)
         in block # 7:
            (d,4,10)
            (c,3,10)
         in block # 8:
            (c,3,11)

Info def_uses block # 2:
      Def_in:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Def_out:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Use_out:
            (c,5)
      Def_use_assos:
            (c,3,5)
      Forward_def_use_assos:

Info def_uses block # 3:
      Def_in:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Def_out:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Use_out:
            (d,6)
      Def_use_assos:
            (d,4,6)
      Forward_def_use_assos:

Info def_uses block # 4:
      Def_in:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Def_out:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Use_out:
            (d,7)
      Def_use_assos:
            (d,4,7)
      Forward_def_use_assos:

Info def_uses block # 5:
      Def_in:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Def_out:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Use_out:
            (c,8)
            (d,8)
      Def_use_assos:
            (c,3,8)
            (d,4,8)
      Forward_def_use_assos:

Info def_uses block # 7:
      Def_in:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Def_out:
            (d,10)
            (a,1)
            (b,2)
            (c,3)
      Use_out:
            (d,10)
            (c,10)
      Def_use_assos:
            (d,4,10)
            (c,3,10)
      Forward_def_use_assos:
         in block # 4:
            (d,10,7)
         in block # 5:
            (d,10,8)
         in block # 7:
            (d,10,10)

Info def_uses block # 4:
      Def_in:
            (d,10)
            (a,1)
            (b,2)
            (c,3)
      Def_out:
            (d,10)
            (a,1)
            (b,2)
            (c,3)
      Use_out:
            (d,7)
      Def_use_assos:
            (d,10,7)
      Forward_def_use_assos:

Info def_uses block # 5:
      Def_in:
            (d,10)
            (a,1)
            (b,2)
            (c,3)
      Def_out:
            (d,10)
            (a,1)
            (b,2)
            (c,3)
      Use_out:
            (c,8)
            (d,8)
      Def_use_assos:
            (c,3,8)
            (d,10,8)
      Forward_def_use_assos:

Info def_uses block # 7:
      Def_in:
            (d,10)
            (a,1)
            (b,2)
            (c,3)
      Def_out:
            (d,10)
            (a,1)
            (b,2)
            (c,3)
      Use_out:
            (d,10)
            (c,10)
      Def_use_assos:
            (d,10,10)
            (c,3,10)
      Forward_def_use_assos:
         in block # 4:
            (d,10,7)
         in block # 7:
            (d,10,10)

Info def_uses block # 8:
      Def_in:
            (d,10)
            (a,1)
            (b,2)
            (c,3)
      Def_out:
            (d,10)
            (a,1)
            (b,2)
            (c,3)
      Use_out:
            (c,11)
      Def_use_assos:
            (c,3,11)
      Forward_def_use_assos:
```

1.3 Hybrid analysis printouts

The control flow graph of this example is the same of section 1.2. The results of hybrid analysis are the following:

```

Hybrid info:
Info def_uses block # 1:
      Def_in:
      Def_out:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Use_out:
            (a,3)
            (b,4)
      Def_use_assos:
            (a,1,3)
            (b,2,4)
      Forward_def_use_assos:
         in block # 1:
            (a,1,3)
            (b,2,4)
         in block # 2:
            (c,3,5)
         in block # 3:
            (d,4,6)
         in block # 4:
            (d,4,7)
         in block # 5:
            (c,3,8)
            (d,4,8)
         in block # 6:
            (c,3,9)
            (d,4,9)
         in block # 7:
            (d,4,10)
            (c,3,10)
         in block # 8:
            (c,3,11)
      Suppressible_Def_Uses:
      This block contains minimal information

Info def_uses block # 2:
      Def_in:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Def_out:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Use_out:
            (c,5)
      Def_use_assos:
            (c,3,5)
      Forward_def_use_assos:
      Suppressible_Def_Uses:
      This block contains minimal information

Info def_uses block # 3:
      Def_in:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Def_out:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
      Use_out:
            (d,6)
      Def_use_assos:
            (d,4,6)
      Forward_def_use_assos:
      Suppressible_Def_Uses:
      This block contains minimal information
```

```
Info def_uses block # 4:
      Def_in:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
            (c,9)
            (d,10)
      Def_out:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
            (c,9)
            (d,10)
      Use_out:
            (d,7)
      Def_use_assos:
            (d,4,7)
            (d,10,7)
      Forward_def_use_assos:
      Suppressible_Def_Uses:
            (7,d,4,7)

Info def_uses block # 5:
      Def_in:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
            (c,9)
            (d,10)
      Def_out:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
            (c,9)
            (d,10)
      Use_out:
            (c,8)
            (d,8)
      Def_use_assos:
            (c,3,8)
            (c,9,8)
            (d,4,8)
            (d,10,8)
      Forward_def_use_assos:
      Suppressible_Def_Uses:
            (6,c,3,8)
            (7,d,4,8)

Info def_uses block # 6:
      Def_in:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
            (c,9)
            (d,10)
      Def_out:
            (c,9)
            (a,1)
            (b,2)
            (d,4)
            (d,10)
      Use_out:
            (c,9)
            (d,9)
      Def_use_assos:
            (c,3,9)
            (c,9,9)
            (d,4,9)
            (d,10,9)
      Forward_def_use_assos:
         in block # 5:
            (c,9,8)
         in block # 6:
            (c,9,9)
         in block # 7:
            (c,9,10)
         in block # 8:
            (c,9,11)
      Suppressible_Def_Uses:
            (6,c,3,9)
            (7,d,4,9)

Info def_uses block # 7:
      Def_in:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
            (c,9)
            (d,10)
      Def_out:
            (d,10)
            (a,1)
            (b,2)
            (c,3)
            (c,9)
      Use_out:
            (d,10)
            (c,10)
      Def_use_assos:
            (d,4,10)
            (d,10,10)
            (c,3,10)
            (c,9,10)
      Forward_def_use_assos:
         in block # 4:
            (d,10,7)
         in block # 5:
            (d,10,8)
         in block # 6:
            (d,10,9)
         in block # 7:
            (d,10,10)
      Suppressible_Def_Uses:
            (6,c,3,10)
            (7,d,4,10)

Info def_uses block # 8:
      Def_in:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
            (c,9)
            (d,10)
      Def_out:
            (a,1)
            (b,2)
            (c,3)
            (d,4)
            (c,9)
            (d,10)
      Use_out:
            (c,11)
      Def_use_assos:
            (c,3,11)
            (c,9,11)
      Forward_def_use_assos:
      Suppressible_Def_Uses:
            (6,c,3,11)
```

2 Source code

The Ada implementation is divided into three main files:

- *cfg.ads*: contains the definitions of the data structures as well as the definitions of the methods and procedures used in file *cfg.adb*;
- *cfg.adb*: here are defined the procedures for printing the results both on screen and on file;
- *run.adb*: where the algorithms of static, dynamic and hybrid analysis are implemented. There are also auxiliary functions.

The source code is available upon request.

Bibliography

[1] K. D. Cooper, T. J. Harvey, and K. Kennedy. *Iterative data-flow analysis, revisited.* Tech. rep. Rice University, 2004.

[2] A. Facchini. *Algebra e matematica discreta.* Zanichelli, 2000.

[3] U. P. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice.* CRC Press, 2009.

[4] R. Mascetti. *Fregasega di Ada: bitumazioni pratiche antanizzate.* Monicelli, 1975.

[5] J. W. McCormick, F. Singhoff, and J. Hugues. *Building Parallel, Embedded, and Real-Time Applications with Ada.* Cambridge University Press, 2011.

[6] R. Melandri and G. Necchi. *Fregasega di Ada II: sbiriguda a posterdati avanzata.* Monicelli, 1982.

[7] G. Perozzi. *Fregasega di Ada III: teoria del vaffanzum.* Monicelli, 1985.

[8] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. "Cheddar: a Flexible Real Time Scheduling Framework". *International ACM SIGADA conference, Atlanta* (2004).

[9] *The Cheddar project: a free real time scheduling analyzer.* Website. 2011. URL: `http://beru.univ-brest.fr/~singhoff/cheddar/#Ref1`.