Computing Maximum Task Execution Times — A Graph-Based Approach [*]

Peter Puschner and Anton Schedl

Institut für Technische Informatik
Technische Universität Wien
{peter,toni}@vmars.tuwien.ac.at

September 1995

Abstract

The knowledge of program execution times is crucial for the development and the verification of real-time software. Therefore, there is a need for methods and tools to predict the timing behavior of pieces of program code and entire programs.

This paper presents a novel method for the analysis of program execution times. The computation of MAximum eXecution Times (MAXTs) is mapped onto a graph-theoretical problem that is a generalization of the computation of a maximum cost circulation in a directed graph. Programs are represented by *T-graphs*, timing graphs, which are similar to flow graphs. These graphs reflect the structure and the timing behavior of the code. *Relative capacity constraints*, a generalization of capacity constraints that bound the flow in the edges, express user-supplied information about infeasible paths. To compute MAXTs, T-graphs are searched for those execution paths which correspond to a maximum cost circulation. The search problem is transformed into an integer linear programming problem. The solution of the linear programming problem yields the MAXT.

The special merits of the presented method are threefold: First, it uses a concise notation to characterize a program's static structure and possible execution paths. Second, the notation allows for a description of the feasible paths through program code that characterizes the code's behavior sufficiently to compute the exact maximum execution time of the program – not just a bound thereof. Third, linear program solving does not only yield maximum execution times, but also produces detailed information about the execution time and the number of executions of every single program construct in the worst case. This knowledge is valuable for more comprehensive analysis of a program's timing.

[*]This work has been supported by Digital Equipment Corporation under contract EERP/AU-038.

1 Introduction

Computer systems for hard real-time process control must fulfill stringent requirements of reliability, availability, and safety. In these real-time applications the cost of a catastrophic system failure can exceed the initial investment in the computer and the controlled object by far. To prevent such catastrophies, the system design and implementation must guarantee the specified behavior in the value *and* time domain in *all* anticipated operational situations.

One of the central parameters in the design and implementation of predictable real-time applications is the MAximum eXecution Time (MAXT) of the tasks involved. In the design phase estimates for the MAXTs of tasks are used to determine the needed hardware resources, to plan the timing of interactions between tasks, and to allocate tasks to processing units. In the implementation phase, these estimates become time budgets that the tasks must be guaranteed to meet – a violation of a single task's timing constraint would invalidate the results of the design phase. To give such guarantees, the timing behavior of all tasks must be analyzed.

MAXT computation is complex: It has to take into account *a)* the feasible and infeasible paths through the program code and *b)* the hardware characteristics of the target system. In the following we concentrate on aspect *a)* assuming that the execution time of every sequential piece of code is invariable over time.

The paper presents a method that describes feasible program execution paths through a program/code fragment. From that description the maximum execution time of the program – not just an execution time bound – is computed: The MAXT computation is transferred to a graph-theoretical problem resembling the problem of the calculation of a maximum cost circulation in a directed graph. Programs are represented by *T-graphs*, timing graphs, which are similar to flow graphs. A T-graph's structure reflects the structure of a program, the weights of the graph's edges stand for the execution times of sequential code fragments. *Relative capacity constraints*, a generalization of the capacity constraints in flow graphs, express user-supplied information about infeasible paths. To compute MAXTs, T-graphs are searched for a maximum cost circulation which obeys the constraints. The search problem is transformed into an integer linear programming problem. The solution of the programming problem yields the MAXT and detailed information about execution paths of maximum cost (see also [Pus93a, Sch93]).

Several works are related to our research. Our approach has been mainly influenced by Kligerman and Stoyenko. In [Kli86, Sto87] they discuss restrictions for programming languages that are necessary to allow for a computation of execution time bounds for real-time tasks. Their programming language, called Real-Time Euclid, disallows the use of recursions or gotos. Loops are restricted to *for*-loops, for which the maximum number of iterations, and thus the time maximally spent in these loops, can be easily derived, and *time bounded loops*. An algorithm that calculates execution time bounds of RT-Euclid programs by computing the maximum time consumption of all possible communication and execution sequences of the programs is described in detail in [Sto91].

A more formal approach for the computation of worst case execution times is described by Park and Shaw [Sha89, Par91]. The timing schemas described allow Park and Shaw to compute the minimum and maximum execution times of constructs found in most programming languages. Based on this theory a timing tool was implemented. This tool computes execution time bounds by a source code analysis of C programs. It interacts with the user to obtain loop bounds required for the calculations. When the timing tool was tested the computed execution time bounds were very close to the real bounds [Par91].

In [Par93] Park and Shaw extend their previous work. They use regular expressions to characterize feasible paths, thus improving the quality of the computed execution time bounds. When the analyzer tool computes execution time bounds it takes the maximum of the execution time bounds computed for all possible path groups.

Mok and his group [Mok89, Ame85, Che87] produced a timing tool for assembly language programs, which are annotated by TAL scripts. These scripts contain descriptions of the timing properties of code pieces in the form of loop bounds and additional control flow information, which make the calculations less pessimistic. The use of that tool is restricted to the analysis of assembly language programs.

Our present method has its roots in the work of Koza and Puschner [Pus89]. They undertake an analysis of high level language code to compute bounds for the execution times of tasks by extending the bounded loop concept found in [Kli86]. New language constructs describe additional knowledge about the control flow. This allows them to derive significantly tighter execution time bounds for more complex programs. The described tool combines the source program plus the compiled code to compute the MAXT of a program. When the authors experimented with the tool they observed that the tool did not find the correct mapping between the two representations for all input programs.

To avoid this problem, a new task timing analysis tool was developed [Pus93b]. It is based on a single data structure, called *timing tree* [Pos92], which contains all information needed to calculate MAXT bounds. This tool not only allows its users to compute worst case execution time bounds of high quality; it also produces detailed information about the contribution of every statement to this bound and allows programmers to experiment with hypothetical times, i.e., one can predict how local changes in the execution time affect the global timing behavior of a procedure/program.

Li and Malik [Li95a] present a method that is similar to our work. Worst case program execution times are computed with an integer linear programming solver. They identify disjoint sets of program paths and construct a linear programming problem for each path set. All solutions are computed. The maximum is the worst case execution time. Li, Malik, and Wolfe extend this work to incorporate the timing characteristics of modern processors with instruction caches and pipelines, [Li95b]. We do not discuss this work in detail, since the modelling of hardware features is beyond the scope of this paper.

Several new aspects can be found in our work:

- We treat the computation of worst case execution times as a graph-theoretical problem, a generalization of a circulation problem found in graph theory [Jun90].
- T-graphs and relative capacity constraints are a concise notation to characterize a program's static structure and feasible/infeasible execution paths.
- The notation with T-graphs and relative capacity constraints characterizes feasible and infeasible program paths sufficiently to derive exact maximum program execution times (not just execution time bounds) for computer systems in which the execution time of every sequential piece of code is invariable. We prove that property in the paper.
- Our method imposes only a minimum of structural restrictions on programs. This is in constrast to most previous approaches [Wei81, Reh87, Kli86, Pus89, Par91] that were very restrictive, e.g., they disallowed break, return, and other jump statements.
- The graph theoretical problem is mapped onto an integer linear programming problem. A solution of a linear programming problem yields both the maximum execution time and valuable information about the execution times and execution frequencies of every single program construct. Besides, we can rely on existing tools to compute solutions and need not build our own.

Section 2 describes the assumptions about programs and system properties made in this work. In Section 3 we show how the computation of the MAXT of a program can be mapped onto the problem of finding a maximum cost circulation in a graph, called T-graph. Since the first solution, that bounds the flow for each edge individually, proves to be unsatisfactory we introduce relative capacity constraints in Section 4. We show that this problem description suits our needs: Feasible execution paths can be exactly characterized and the MAXT can be computed. In Section 5 we illustrate how an ILP problem is constructed to compute the MAXT for a T-graph with relative capacity constraints. The paper is concluded in Section 6.

2 Assumptions

The execution time of a program is determined by two factors, the behavior of the program which depends both on the program structure and the application context as well as the characteristics of the underlying hardware. The method for MAXT analysis presented here concentrates on how to cover the first point. As for the hardware characteristics, we assume that the execution time of every sequential piece of code is invariable. The following list further summarizes the assumptions made in this paper.

- A program/piece of code subject to MAXT-computation has one starting point and one end point. The end point is different from the starting point. Every execution begins or enters the code at the starting point and terminates or leaves the code at the end point.

Note that this assumption does not limit generality. A piece of code with one starting point and one end point can be built from every piece of code with an arbitrary number of starting points and end points. The easiest way to do so is to insert a conditional branch to all starting points at the beginning and to add branches to a new end point to all end points of the original piece of code.

- Every execution can be described as a sequence of executions of the program's or code's constituents (e.g., instructions, statements, blocks, ...) for which the execution time is known. Our method also works if the constituents' execution times are not exactly known but can be bounded. In that case it can, however, only produce solutions which are pessimistic bounds for the MAXT.

- The static structure of the code's constituents, i.e., which constituents may follow each other in execution, is known.

- For each part of code the maximum number of repetitions is known.

- The dynamic characteristics of a program/program part, i.e., the paths that can actually be executed in the given application context, depend on the input data and the program state at the invocation time. We assume that information about this dynamic behavior is available and that the provided information is complete, i.e., all information about feasible/infeasible paths is available to the MAXT analysis.

  It is obvious that the latter assumption will not always hold in practice. The discussion of how difficult it is to provide such execution information is, however, beyond the scope of this paper. In this paper we want to concentrate on the methodological limits of our technique: Assuming that full information is available we investigate to which extent this knowledge can be utilized for the MAXT-computation. In fact, we will show, that our approach allows a complete incorporation of that knowledge into the analysis.

3 The T-Graph Representation of Programs

In this section we discuss the representation we use for the description of the problems to be solved. This representation must describe both static and dynamic properties of the application code under investigation. The static structure of programs/code pieces is represented by directed graphs. The graphs are annotated with restrictions which characterize the feasible/infeasible execution paths.

3.1 Representation of the Static Program Structure

We represent a program or a piece of code by a so-called *T-graph*, timing graph. The edges of the T-graph stand for constituents of the code and are weighted by the execution times of these constituents. The nodes in the graph represent points in the code where the control flow of the program may fork and/or join.

Depending on the used code description language or programming language a T-graph's edges may represent machine instructions, statement sequences of high level language code, pseudo code statements, etc. T-graphs can be generated from any of these codings provided the execution times for the constituents (or at least bounds thereof) can be derived. Of course, it must be guaranteed that every part of the program source is mapped onto a corresponding part of the T-graph. Figure 1 shows, how some typical high level language constructs might be translated into T-graph notation.

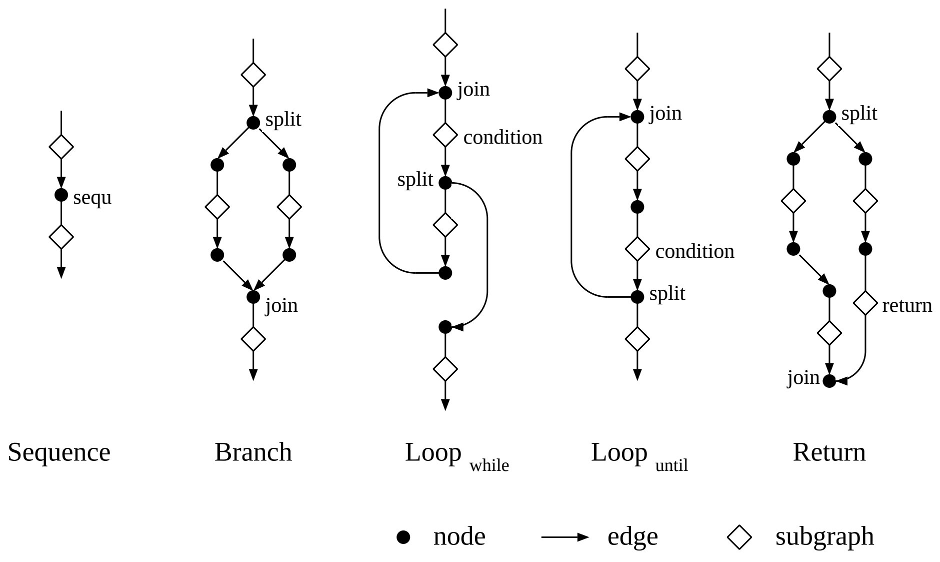


Figure 1: T-Graph Representation of Typical Programming Language Constructs

Formally, a T-graph is a connected directed graph $G = (V, E)$ with vertices (nodes) $V = \{v_i \mid 1 \leq i \leq |V|\}$ and edges (arcs) $E = \{e_i \mid 1 \leq i \leq |E|\}$, where each edge e_i can also be written as an ordered pair (v_j, v_k). A T-graph has the following properties:

1. G has exactly one vertex s with no incoming edges (*starting vertex*).
2. G has exactly one vertex t with no outgoing edges (*terminating vertex*).
3. For every edge e_i there exists at least one directed walk with starting point s and end point t that contains e_i (*no unreachable code*).
4. Every edge has a non-negative integer weight $t_i = \tau(e_i)$ (*execution time*).

Items 1 to 4 define the structure of a T-graph. The semantic interpretation is as follows. Properties 1 and 2 help us to characterize the execution paths whose execution times we want to compute. In the T-Graph all paths start at s and end at t (see also Section 2).

Point 3 guarantees that each program part is part of at least one program path leading from s to t, point 4 assigns an execution time to each edge.

The definition of the T-graph allows that one piece of code is multiply represented by several edges as long as the T-graph correctly represents the control flow through the program code. This is useful when conditional jump instructions whose execution times depend on whether the jump is taken or not have to be represented. These jumps can be mapped onto different edges with different execution times whose successors are the respective alternatives following the branch.

Note that T-graphs can represent arbitrary valid flow graphs, even irreducible ones, with one starting point and one end point. In particular their use is not restricted to the description of 'well-structured' programs. For programs to be analyzed that implies that they may contain arbitrary jump instructions, including goto[1] statements.

3.2 Execution Paths and Execution Times

The previous section introduced how pieces of code can be represented by T-graphs. In this section we will define the terms *execution path* and *execution time*. An execution of a piece of code is characterized by the execution of a sequence of actions of that code. It starts with the first statement (instruction, etc.), follows the control flow as defined by the constructs of the language used, and leaves the code at one of the end statements. For every such execution there exists a corresponding walk from s to t in the T-graph.

Def.: A walk $P_i = (e_{i_1}, e_{i_2}, \ldots, e_{i_m})$ in a T-graph G with $e_{i_1} = (s, v_j)$ and $e_{i_m} = (v_k, t)$ is called *execution path* or for short *path*[2].

Every execution path consists of a finite number of edges. Every edge is weighted by its execution time. We can, therefore, attribute every path an execution time — the sum of the execution times of its edges.

Def.: The *execution time* τ of a path $P_i = (e_{i_1}, e_{i_2}, \ldots, e_{i_m})$ is the sum of the execution times of its edges,

$$\tau(P_i) = \sum_{j=1}^{m} \tau(e_{i_j}). \qquad (1)$$

[1]We do not want to advertise the use of gotos in a high level programming language. Nevertheless there are other instructions with similar semantics, e.g. exit of loops. Apart from that, assembly language programs with arbitrary jumps are analyzable with our method as well.

[2]Note that this (program) path definition has nothing to do with the classical path definition in graph theory.

The execution times of the edges of a path P_i can be summed up in any order. One can, therefore, count the number of occurrences of every edge $e_j \in E$ on P_i, yielding $f_i(e_j)$, or for short $f_{i,j}$. The execution time of P_i can then be written as follows:

$$\tau(P_i) = \sum_{j=1}^{|E|} f_i(e_j)\tau(e_j) = \sum_{j=1}^{|E|} f_{i,j}t_j. \tag{2}$$

For every application, the number of different execution paths is finite. Since it is possible to compute the execution time for each of these paths, it is also possible to compute the maximum of these times – the maximum execution time.

Def.: The *maximum execution time (*MAXT*)* of a set of paths π in a T-graph G is

$$maxt(\pi) = \max_{P_i \in \pi}(\tau(P_i)) = \max_{P_i \in \pi} \sum_{j=1}^{|E|} f_{i,j}t_j. \tag{3}$$

For real problems a full enumeration of all possible execution paths through every piece of code under scrutiny is infeasible. For that reason, a representation that describes, rather than enumerates execution paths is needed.

3.3 Characterization of Execution Paths – A First Step

An enumeration of all possible execution paths through a piece of code does not make use of the information that is represented by a T-graph. The T-graph has already a lot of information about the possible order of edges in legal execution paths. In the following this information is used as a basis for the description of possible paths for the MAXT computation. We extend the notion of T-graphs, introduce *capacity constraints* that restrict the flow in their edges, and compute the MAXT of the respective code pieces as maximum cost integer circulation in the given *extended T-graphs*.

3.3.1 Circulations

We first introduce circulations as they are defined in graph theory [Jun90]. Later we will modify this common definition to suit our needs.

Given a connected directed graph G, a function $f : E \to \mathbf{R}$ is called a *circulation* if it conserves the flow in every node:

$$\forall v \in V : \sum_{e=(v_j,v)} f(e) = \sum_{e=(v,v_k)} f(e). \tag{4}$$

The capacity constraints $b : E \to \mathbf{R}$ and $c : E \to \mathbf{R}$ restrict the values of f for all edges. A circulation f is called *legal* if

$$\forall e \in E : b(e) \leq f(e) \leq c(e). \tag{5}$$

Finally, let $\gamma : E \to \mathbf{R}$ be a cost function on G. The cost $\gamma(f)$ of a circulation f is then defined as follows:

$$\gamma(f) := \sum_{e \in E} \gamma(e) f(e). \tag{6}$$

3.3.2 T-Graphs and Circulations

The aim is to compute the MAXT of a piece of code as an integer circulation of maximum cost in the T-graph G. For that purpose the T-graph is mapped onto the description of a circulation. The following steps are performed:

1. A backward edge $e_{|E|+1} = (t, s)$ is added to the T-graph G, yielding an *extended T-graph* G' with $V' = V$ and $E' = E \cup \{e_{|E|+1}\}$. The extended T-graph allows for circulations which have a flow of 1 both out of s and into t. We will see, that this modification is necessary to represent execution paths by circulations.

2. For each edge e_i the capacity constraints $b(e_i)$ and $c(e_i)$ are defined as follows:

$$b(e_i) := \begin{cases} 1 & \text{if } e_i = (t, s) \text{ (backward edge)} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

$$c(e_i) := \begin{cases} 1 & \text{if } e_i = (t, s) \\ f_{max}(e_i) & \text{otherwise} \end{cases} \tag{8}$$

   The term $f_{max}(e_i)$ stands for the maximum number of executions of the edge e_i in the path set π, i.e., $f_{max}(e_i) = \max\limits_{P_j \in \pi} f_j(e_i)$.

3. The cost of the edges in the extended T-graph are:

$$\gamma(e_i) := \begin{cases} 0 & \text{if } e_i = (t, s) \\ \tau(e_i) & \text{otherwise} \end{cases} \tag{9}$$

These transformation rules give a formal description how a T-graph and a set of paths that describe the behavior of a piece of code can be characterized by a circulation and capacity constraints. The following comments about the transformation are worth noting:

- For every execution path $P_i = ((s, v_{i_1}), (v_{i_1}, v_{i_2}), \ldots (v_{i_m}, t))$ in G the closed walk $P' = ((s, v_{i_1}), (v_{i_1}, v_{i_2}), \ldots (v_{i_m}, t), (t, s))$ in G' induces an integer circulation in G': we define that every occurrence of an edge e in the closed walk adds 1 to the flow in this edge. Since every closed walk has the property that the number of outgoing edges equals the number of incoming edges for every node, this construction yields indeed a legal circulation.

- Except for the backward edge, the minimum flow for each edge is 0. For the backward edge both the minimum flow and the maximum flow are set to 1. This is to make sure that every valid circulation has indeed a flow of 1 both out of s and into t and thus represents a valid execution path[3].

- For every edge e_i that is member of an execution path $c(e_i) \geq 1$ holds. Only for edges that are not part of any path (dead code) $c(e_i) = 0$.

- In Point 3 of the transformation the cost of every edge was assigned its execution time. Thus, for every path P in G with a corresponding circulation f in G' the execution time of P can be calculated as the cost of the circulation f, i.e., $\gamma(f)$. In particular, this implies that the maximum execution time of the valid execution paths in G can be calculated as the cost of a maximum cost circulation in G'.

- The construction of legal circulations from execution paths induces an equivalence relation on execution paths. Two paths P and P' belong to the same equivalence class if for every edge the number of occurrences in P equals the number of its occurrences in P'. Since the execution time of every program path is only dependent on how often every edge occurs, but independent from the order of the edges, the execution times of all paths of the same equivalence class are equal.

  For the computation of the MAXT as a circulation of maximum cost this implies that it is sufficient to work with equivalence classes, which reduces the complexity of the problem. On the other hand the solution of a MAXT problem with this technique may not only yield a single execution path but the set of paths belonging to one equivalence class.

3.3.3 Shortcomings of the Transformation

Although the previous list of observations might suggest that the construction of a circulation problem as described allows to map the MAXT calculation onto the computation of a maximum cost circulation in the extended T-graph, this is not true in general: Only an upper bound for the MAXT, not necessarily the exact value, can be computed that way. The reason is that essential information about paths get lost in the transformation process. The transformation does not guarantee that maximum cost circulations do indeed represent execution paths. Before we discuss this in more detail, we define the term *circulation subgraph.*

Def.: Let $G = (V, E)$ be an extended T-graph and f a circulation in G. We call the graph $\overline{G} = (V, \overline{E})$ with $\overline{E} = \{e | e \in E,\ f(e) > 0\}$ the *circulation subgraph* of G induced by f.

[3]We assume that every piece of code to be analyzed has at least one valid execution path.

There are two different situations in which maximum cost circulations do not represent valid program paths:

Case 1: The *circulation subgraph* induced by the maximum cost circulation, $\overline{G} = (V, \overline{E})$ with $\overline{E} = \{e | e \in E,\ f(e) > 0$ in the maximum cost circulation$\}$, is not strongly connected, i.e., there is no closed walk $((s, v_{i_1}), (v_{i_1}, v_{i_2}), \ldots, (t, s))$ through the extended T-graph that contains all edges e with $f(e) \geq 1$ in the maximum cost circulation.

Figure 2 illustrates Case 1. The left side shows the structure of an extended T-graph. The table to the right lists the maximum capacity $c(e)$ and the cost $\gamma(e)$, i.e., the execution time, for each edge. The minimum capacities are 1 for the backward edge and 0 for all other edges. The last column of the table shows $f(e)$ for each edge in the maximum cost circulation, the thick lines in the graph the respective circulation subgraph.

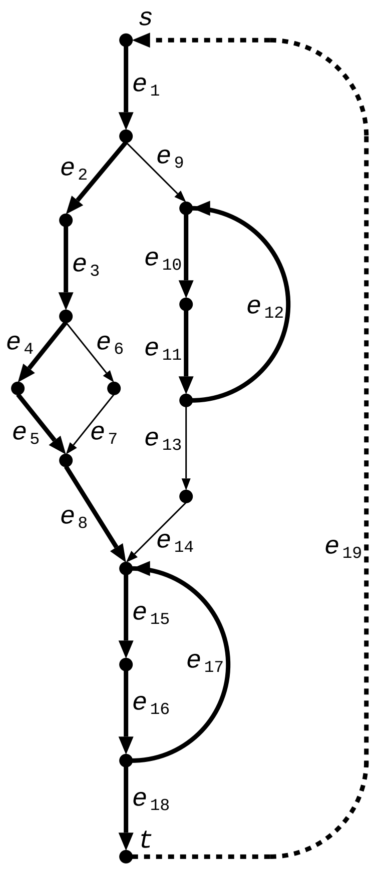
One can see that the circulation subgraph contains at least one walk from s to t. On the other hand, none of these walks contains the edges e_{10}, e_{11}, or e_{12} – there is no flow greater than zero into this subgraph. Observe that it is not even true that the edges of the correct worst case execution path form a subset of the edges of the marked circulation subgraph (see Figure 4, which shows the desired solution). The worst case execution path contains e_9, e_{13}, and e_{14}. It does, however, not contain any edge of the marked walk $(e_2, \ldots, e_8)$.

The reason for the mentioned problem is that a T-graph G does not need to be acyclic. Obviously, any positive flow in a cyclic subgraph of G by itself obeys the flow conservation principle. A non-zero flow can exist in such a subgraph, even if all the edges leading into this subgraph have a flow of zero, i.e., the edges of the cyclic subgraph are not part of a valid execution path.

In order to compute ciruclations that do represent worst case execution paths, we have to make sure that the circulation subgraphs of these circulations are connected, i.e., we map the MAXT calculation to the computation of a maximum cost circulation which is *strongly connected.* In Section 4 we will generalize the capacity constraints by introducing *relative capacity constraints* to cover this connectivity requirement.

Case 2: Although the maximum cost circulation corresponds to a walk through the extended T-graph, there exists at least one edge e for every valid execution path P_i with $f_i(e) < f_{mc}(e)$, where $f_{mc}(e)$ represents the flow through e in the maximum cost circulation, i.e., the maximum cost circulation corresponds to an *infeasible path* [Sha89].

The characterization of execution paths by capacity constraints describes the maximum number of executions of every single edge of a graph separately. It does not allow to describe more complex characteristics of a program's behavior, i.e., that there exists a certain correlation between the number of executions of different edges. Therefore,



e	$c(e)$	$\gamma(e)$	$f(e)$
e_1	1	36	1
e_2	1	8	1
e_3	1	86	1
e_4	1	8	1
e_5	1	112	1
e_6	1	10	0
e_7	1	30	0
e_8	1	46	1
e_9	1	10	0
e_{10}	8	12	7
e_{11}	8	26	7
e_{12}	7	10	7
e_{13}	1	32	0
e_{14}	1	24	0
e_{15}	10	44	10
e_{16}	10	20	10
e_{17}	9	10	9
e_{18}	1	56	1
e_{19}	1	0	1

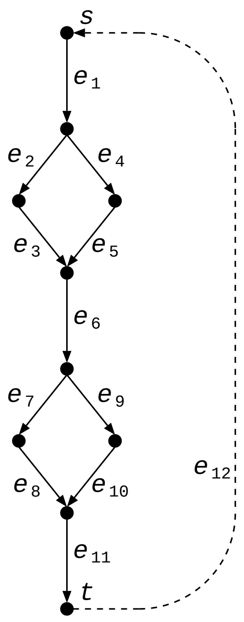
Figure 2: A Maximum Cost Circulation that does not Represent a Valid Path. Thick Lines in the Graph Represent the Circulation Subgraph of the Circulation.

only local, per-edge capacity constraints can be taken into account when computing a maximum cost circulation. The following example illustrates that case.

Consider the T-graph of Figure 3 which consists of two branches in sequence. We assume that the following execution paths are possible:

$$(e_1, e_2, e_3, e_6, e_7, e_8, e_{11}),$$
$$(e_1, e_2, e_3, e_6, e_9, e_{10}, e_{11}),$$
$$(e_1, e_4, e_5, e_6, e_9, e_{10}, e_{11}).$$

The path $(e_1, e_4, e_5, e_6, e_7, e_8, e_{11})$ is assumed infeasible. We observe that the capacity constraints $c(e)$ do not allow to characterize this set of paths sufficiently: The computation of the maximum cost circulation yields a circulation that exactly corresponds to the infeasible path. The cost of this circulation are 378 time units, as opposed to the desired result of 324, the maximum cost of the three feasible paths.



e	$c(e)$	$\gamma(e)$	$f(e)$
e_1	1	44	1
e_2	1	8	0
e_3	1	80	0
e_4	1	10	1
e_5	1	132	1
e_6	1	56	1
e_7	1	8	1
e_8	1	82	1
e_9	1	10	0
e_{10}	1	12	0
e_{11}	1	46	1
e_{12}	1	0	1

Figure 3: Example for an Extended T-Graph.

4 A Characterization of Possible Execution Paths Suited for the MAXT-Computation

In the previous section we tried to map the problem of the computation of the MAXT of programs onto a problem from graph theory – the computation of a maximum cost circulation in a flow network. It turned out that the description of circulations in its existing form does only allow to compute MAXT bounds for pieces of code. It does, however, not allow to characterize program behavior in sufficient detail to be able to derive the worst case execution path(s) in general. In this section we extend the description of circulations such that they can characterize arbitrary sets of execution paths that are both legal and possible. This way, the computation of the exact MAXT of every piece of code – not just a bound thereof – becomes possible.

4.1 Relative Capacity Constraints and Connected Circulations

One problem with the traditional representation of circulations is that the circulation subgraphs for maximum cost circulations might be split into several components (see Case 1 in Section 3.3.3): In every cyclic subgraph a flow greater than zero can exist regardless whether there is any positive flow into that subgraph or not. In the following we will use *relative capcity constraints* to avoid that the flow in a cyclic subgraph which is not fed by a positive flow (except a cycle that contains both s and t) can become greater than zero.

Def.: A *relative capacity constraint* is an inequality or equation of the form

$$\sum_{e_i \in E'} a_i f(e_i) \circ \sum_{e_i \in E'} a'_i f(e_i) + k,$$

where $a_i, a'_i \in \mathbf{Z_0}$, $k \in \mathbf{N_0}$ and $\circ \in \{<, \leq, =\}$, that describes the relation of the flow f in the edges e_i of an extended T-graph G'.

Note: Without restriction any relative capacity constraint of the given form can be written in standard form

$$\sum_{e_i \in E'} b_i f(e_i) \circ k$$

where $b_i = a_i - a'_i$ for all i. In the rest of the paper both notations will be used.

The following theorem describes the minimal requirement that guarantees that a T-graph together with a set of relative capacity constraints describe only circulations that correspond to valid execution paths from s to t. We need the definition of *implicating edges* for this theorem.

Def.: Let e_i be an edge in E and E_{sub} a subset of E with $e_i \notin E_{sub}$. We define that e_i *implies* E_{sub} if every execution path containing e_i contains at least one edge of E_{sub}.

Theorem: Let $G' = (V', E')$ be an extended T-graph, $G_{cyc} = (V_{cyc}, E_{cyc})$ a cyclic subgraph of G' that contains neither s nor t, and E_{imp} the set of edges e_i that imply E_{cyc}.

If the flow of at least one edge e_c of each cyclic subgraph G_{cyc} is bounded by a relative capacity constraint of the form

$$f(e_c) \leq \sum_{e_i \in E_{imp}} a_i f(e_i),$$

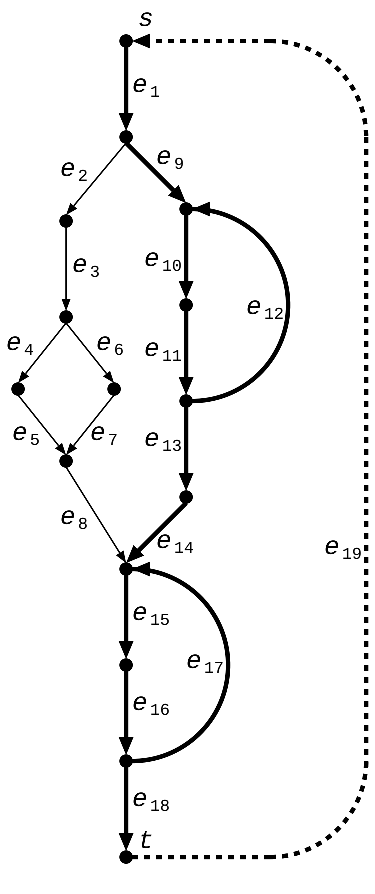
where at least one a_i is greater than zero, then the circulation subgraphs of all legal circulations are strongly connected, i.e., every legal circulation corresponds to at least one closed walk $((s, v_1), \ldots, (v_j, t), (t, s))$.

Proof: The correspondence between walks and connected circulations has been discussed earlier. Remains to be shown that the above conditions imply that the subgraph of the circulation is strongly connected.

Indirect proof: We assume that the circulation subgraph is not strongly connected although there exists a relative capacity constraint for each cycle. It follows that one cycle with edges $E_{cyc'}$ is not reachable from s in the circulation subgraph. This implies that $f(e_c) > 0$ for all edges $e_c \in E_{cyc'}$ and that $f(e_i) = 0$ for the edges $e_i \in E_{imp'}$ that imply $E_{cyc'}$. Since the flow of at least one edge $e_c \in E_{cyc'}$ is constrained by a relative capacity constraint and since $f(e_i) = 0$ for all $e_i \in E_{imp'}$ we get $f(e_c) = 0$. Hence, there is no cyclic positive flow in the cycle $E_{cyc'}$. This contradicts the assumption and proves the theorem. □

According to the theorem one can use relative capacity constraints as the basis for the computation of maximum cost circulations that are strongly connected. All one has to do for that purpose is to specify one relative capacity constraint for each cycle (loop) in the extended T-graph.

The example shown in Figure 4 uses the T-graph that has already been used before. In contrast to the earlier description *relative* capacity constraints describe the flow in the edges. We use one capacity constraint for the backward edge, $f(e_{19}) = 1$, to specify that circulations must have a flow of one out of s and into t. Two further relative constraints limit the flow in the cycles (loops). Together with the structural description of the extended T-graph and the cost of the edges this information is sufficient to compute a maximum cost circulation (last column of the table). Implicitly we assume that $f(e) \geq 0$ for all edges. The given relative capacity constraints replace the traditional, absolute capacity constraints $c(e)$ that had been defined for all edges in Figure 2.



e	$\gamma(e)$	$f(e)$
e_1	36	1
e_2	8	0
e_3	86	0
e_4	8	0
e_5	112	0
e_6	10	0
e_7	30	0
e_8	46	0
e_9	10	1
e_{10}	12	8
e_{11}	26	8
e_{12}	10	7
e_{13}	32	1
e_{14}	24	1
e_{15}	44	10
e_{16}	20	10
e_{17}	10	9
e_{18}	56	1
e_{19}	0	1

$f(e_{19}) = 1$
$f(e_{10}) \leq 8f(e_9)$
$f(e_{15}) \leq 10f(e_8) + 10f(e_{14})$

Figure 4: Maximum Cost Circulation of an Extended T-Graph with Relative Flow Constraints.

4.2 Complete Description of Possible Execution Paths – Exact MAXT Computation

In the previous section we have shown how we can use relative capacity constraints to meet the "connectivity requirement" for circulations that represent execution paths of programs. In this section we show a property of the code representation by extended T-graphs and relative capacity constraints that is much stronger: This program code representation is strong enough to characterize the possible and impossible (infeasible) paths through any piece of code in a way that the computation of the maximum execution time is possible. We formulate this important property in the following theorem.

Theorem: By means of extended T-graphs and relative capacity constraints the behavior of every piece of code can be described such, that its actual worst case execution time – not just a bound thereof – can be computed (*Completeness* of the representation).

Proof: Let G' be an extended T-graph. We have already shown that an appropriate set of restrictions for capacities allows to compute an upper bound for the maximum execution time of a piece of code. It remains to be shown, that the selection of an adequate set of relative capacity constraints makes it possible to compute the exact maximum execution time, not just a bound thereof. We proof that by induction with respect to the number of paths through the extended T-graph[4].

1. A single execution path $P_1 = (e_{i_1}, \dots, e_{i_m})$ can be assigned an n-tuple $F(P_1) = (f_{1,1}, \dots, f_{1,|E|})$ where $f_{1,j}$ is the number of occurrences of e_j on P_1. One can construct a set of restrictions R_1 that characterize the path,

$$R_1 := \{f(e_j) \leq f_{1,j} | 1 \leq j \leq |E|\}.$$

Maximizing the cost of all possible circulations under these constraints yields:

$$maxt(P_1) = \tau(P_1) = \sum_{j=1}^{|E|} f_{1,j} t_j = \max_{R_1} \sum_{j=1}^{|E|} f(e_j) t_j.$$

Hence the theorem holds if exactly one path is feasible.

2. In the second step of the inductive proof we assume that for a set of n paths described by an extended T-graph and restrictions the maximum execution time can be calculated.

Claim: A set of $n + 1$ paths can also be characterized in a way that the maximum execution time can be computed.

[4]Note that if we applied the inductive steps of the proof to really compute the MAXT, we would rather use the equivalence classes of paths introduced earlier than single paths in every inductive step. We feel, however, that the reader can follow the proof more easily if we handle one path in one step.

Let $R_n := \{\sum a_{kl}x_l \leq c_k\}$ be the set of restrictions for n paths after transformation into standard form [5]. $\overline{R}_{n+1}$ be the set of restrictions which characterize the path P_{n+1}. Since $P_{n+1} \notin \{P_1, \dots, P_n\}$, two cases can be distinguished: Either P_{n+1} or a path out of $\{P_1, \dots, P_n\}$ is the path with the maximum execution time. Therefore, either the restrictions R_n or $\overline{R}_{n+1}$ describe the worst case.

A trick that is used in linear programming, the Big M Method [Hil88], is applied to construct a disjunction of R_n and $\overline{R}_{n+1}$: Let M be a new, large constant that is greater than the right hand sides of all inequalities introduced so far. Then either the restrictions

$$\sum a_{kl}x_l \leq c_k \text{ for all } R \in R_n \quad \text{and} \quad f(e_j) \leq f_{n+1,j} + M \text{ for all } R \in \overline{R}_{n+1}$$

or

$$\sum a_{kl}x_l \leq c_k + M \text{ for all } R \in R_n \quad \text{and} \quad f(e_j) \leq f_{n+1,j} \text{ for all } R \in \overline{R}_{n+1}$$

hold. Using an additional, binary variable y, we build a new set of restrictions for the disjunction of these two sets as follows:

$$\begin{array}{rcll} \sum a_{kl}x_l & \leq & c_k + yM & \text{for all } R \in R_n \\ f(e_j) & \leq & f_{n+1,j} + (1-y)M & \text{for all } R \in \overline{R}_{n+1} \\ y & \leq & 1 & \end{array}$$

Rewriting the formulas so that variables only occur on the left side and constants on the right yields the set of restrictions of the standard form for $n+1$ execution paths. Hence we can write:

$$maxt(\{P_1, \dots, P_{n+1}\}) = \max(maxt(\{P_1, \dots, P_n\}, maxt(P_{n+1})) =$$

$$= \max(\max_{R_n} \sum_{j=i}^{|E|} f(e_i)t(e_i), \max_{\overline{R}_{n+1}} \sum_{i=1}^{|E|} f(e_i)t(e_i)) = \max_{R_{n+1}=R_n \oplus \overline{R}_{n+1}} \sum_{i=1}^{|E|} f(e_i)t(e_i),$$

where the $\oplus$ operator stands for the disjunction of restrictions as introduced above.

This result proves the claim. Thus, we have shown that for any set of paths the worst case execution time can be computed. $\square$

4.3 Describing Software Behavior Concisely

Relative capacity constraints are means to describe software behavior. Using them we can avoid that the computation of the maximum execution time for a piece of code yields

[5] Standard form means that the restrictions are rewritten such that variables only occur on the left side and the right side contains only one constant. For each $f(e_j)$ in the original form there is a corresponding x_l in the standard form of the restriction; the exact mapping from $f(e_j)$ to x_l is unimportant for our proof and thus omitted.

the execution time of a path set that consists solely of infeasible paths. Hence, the use of relative capacity constraints provides a solution to the problem Case 2 encountered with the original representation.

The above proof was conducted on a single path basis. A large number of constraints were needed to describe all feasible paths. For practical use, representing code behavior by extended T-graphs and relative capacity constraints – as we did before – is more adequate. In most cases an extended T-graph and a few relative capacity constraints are sufficient to exactly characterize the possible behavior of a piece of code. Note that relative capacity constraints are not restricted to the description of loop bounds but may characterize any relation between the number of executions of different code statements (see Table 1).

We use our second example (Figure 3) to illustrate the use of relative capacity constraints for the exclusion of infeasible paths from the analysis. In this example we have two branching statements with two branches in sequence, potentially allowing four different paths. One execution path, however, is assumed infeasible. In order to characterize the set of possible paths exactly we state which behavior is not allowed, i.e., which edges must never occur on the same path. Here, the edges e_4 and e_5 on the one hand and e_7 and e_8, on the other hand, must never be executed in a single execution. Due to flow conservation it is sufficient to say that e_4 (e_5) and e_7 (e_8) are exclusive. If the flow in e_4 equals 1 the flow in e_7 must be 0 and vice versa. The sum of the flows can maximally assume 1. We express this by the relative constraint $f(e_4) + f(e_7) \leq 1$. Together with the extended T-graph and the minimal constraint set $f(e_{12}) = 1$ and $f(e_i) \geq 0$ for all i this inequality is already sufficient to completely describe the behavior of the code.

Table 1 gives further examples on how some common dependencies can be described by relative capacity constraints. This list is by far not complete – as we showed before, any dependency can be expressed with such constraints. It gives, however, an impression how relative capacity constraints can be used.

While the construction of a set of capacity constraints for a specific application might not be too hard, reading and understanding the meaning of a given set of constraints can be difficult. The use of a user-friendlier language to describe characteristics of execution paths, as, e.g., used in [Par93], is thus certainly advantageous in practice. Note, that in this work we do not discuss description languages in more detail. We rather emphasize that it is possible to describe how often the parts of a program are executed on the feasible execution paths and to compute the maximum execution times from that knowledge as shown above.

5 MAXT Calculation with Linear Programming

The previous section introduced a representation for the analysis of maximum execution times of programs. Programs are described by extended T-graphs and a set of constraints which describe the set of possible paths through the graphs. The problem of calculating

Inequalities	*Semantics*
$f(e_i) \geq 1$	e_i appears at least once on every execution path.
$f(e_i) = f(e_j)$	e_i and e_j are executed the same number of times.
$f(e_i) = 3f(e_j)$	e_i executes three times as often as e_j.
$f(e_i)+f(e_j)+f(e_k) \leq K$	The sum of the number of executions of e_i, e_j, and e_k is bounded by K.
$f(e_i) \leq Mf(e_j)$	1) Every execution of e_j makes M executions of e_i possible. If e_i is part of a loop that is preceeded by e_j the inequality resembles a marker [Pus89]. 2) Assuming that M is a "very large" constant this can be understood as kind of an implication. Every path with at least one occurrence of e_i must also include e_j.
$f(e_i) \leq My$, $f(e_j) \leq M - My$, $y \leq 1$	Occurrences of e_i and e_j exclude each other (Big M Method).
$3f(e_i) + f(e_j) \leq 9$, $f(e_i) \leq 2$	For every execution of e_i (max. 2) the maximum number of executions for e_j decreases by 3. The total number of executions of both parts together is also bounded.

Table 1: Examples of Inequalities and Equations Describing the Dynamic Execution Behavior of Code.

the MAXT now corresponds to the problem of finding the circulation with maximum cost in the graph. To find that circulation, either a new calculation method has to be developed or the problem can be transformed in a way that existing methods can be applied. We use Integer Linear Programming (ILP), a well known method in the field of Operations Research. We noticed that the basic structure of an ILP problem is well suited for our needs. In fact we developed the T-graph with regard to how such a T-graph can be transformed into an ILP problem.

One has to perform the following steps to compute the MAXT of a code fragment:

1. Build an extended T-graph and the corresponding circulation model out of the structure of the program and of the knowledge about feasible and infeasible paths.

2. Transform the T-graph and the circulation model into a Linear Programming prob-

lem.

3. Solve the Linear Programming problem with existing standard tools.

5.1 Setting Up the Linear Programming Problem

The aim of ILP is to find the maximum value of an objective function subject to the assumption that a set of functional constraints hold. The standard ILP problem can be characterized with the following determinants:

- n decision variables $x_1, x_2, \dots, x_n$,
- the objective function $Z = \sum_{i=1}^{n} c_i x_i$, which has to be maximized,
- m functional constraints $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$ for all $i \in [1, m]$, and
- nonnegativity constraints $x_j \geq 0$ for all $j \in [1, n]$.

We now show the construction of the ILP problem from the T-graph and the circulation model.

1. The execution time of a path through the whole program is the weighted sum of the edges' execution frequencies, where the weights of the edges are their execution times. The objective function of the ILP problem is constructed by using the cost function of the circulation:

$$Z = \sum_{i=1}^{|E'|} f_i t_i .$$

   For each i there is a constant value t_i, which is the execution time of the code represented by edge e_i ($t_i = \tau(e_i)$). The decision variables f_i represent the execution frequencies of the edges e_i. The goal is to find the set of frequencies f_i, where the function Z yields the maximum value.

2. The frequency of each edge e_i underlies a set of restrictions due to the control flow of the program and data dependent implications between program parts. These restrictions are derived from the structure of the T-graph and the relative capacity constraints of the circulation.

- A set of functional constraints describe the structure of the T-graph. For each node v_j an equation of the following form is built:

$$\sum_{e_m=(v_i,v_j)} f_m = \sum_{e_n=(v_j,v_k)} f_n$$

  The flow in the edges going into v_j equals the flow in the edges leaving node v_j (flow conservation).

- The capacity constraints of the circulation model are directly transformed into the functional constraints of the ILP problem:

$$\sum_{e_j \in E'} a_{ij} f_j \circ \sum_{e_j \in E'} a'_{ij} f_j + k,$$

  where a_{ij}, $a'_{ij} \in \mathbf{Z_0}$, $k \in \mathbf{N_0}$ and $\circ \in \{\leq, <, =, >, \geq\}$. These constraints describe restrictions for loops and other dependencies between execution frequencies of program parts. An additional constraint, $f_{|E'|} = 1$, restrics the backward edge.

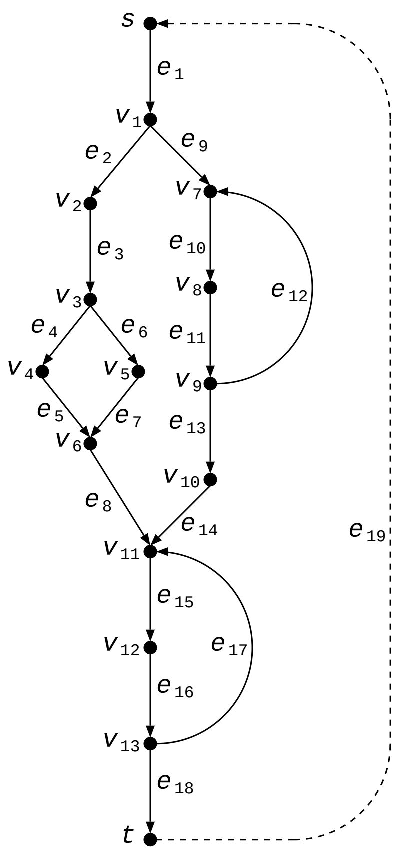
- For each edge the nonnegativity constraint $f_i \geq 0$ is built. This completes our problem. In most software packages the nonnegativity constraints are formed automatically.

Figure 5 and Figure 6 illustrate the transformation of a circulation model into an ILP problem. They show an extended T-graph with constraints for all edges and the corresponding ILP problem. The objective function is the sum of the execution frequencies (f_i) of all edges weighted by their execution times. The ILP problem contains one equation for each node to describe the flow conservation. Additionally, the flow in the two loops is restricted by two inequalities which are easily derived from the restrictions of the T-graph. A further equation, $f_{18} = 1$, restricts the flow in the whole T-graph. The nonnegativity constraints for all edges complete our ILP problem.

The ILP problem can be solved with standard tools availabe on the market. The solution yields the MAXT of a program (solution of the ILP problem) as well as the execution frequencies of its constituents (settings of the decision variables), see Figure 6.

5.2 Benefits of the Approach

The previous section described, how integer linear programming can be used to derive the maximum execution times of programs. The solution of an ILP problem gives us valuable additional and extensive information about the code's timing. This section lists the information we get from a solution of the ILP problem.



e	$t(e)$
e_1	36
e_2	8
e_3	86
e_4	8
e_5	112
e_6	10
e_7	30
e_8	46
e_9	10
e_{10}	12
e_{11}	26
e_{12}	10
e_{13}	32
e_{14}	24
e_{15}	44
e_{16}	20
e_{17}	10
e_{18}	56
e_{19}	0

$f(e_{19}) = 1$
$f(e_{10}) \leq 8f(e_9)$
$f(e_{15}) \leq 10f(e_8)+10f(e_{14})$

Figure 5: Example of an Extended T-Graph with Constraints for Each Edge

1. The maximum execution time of a program, i.e., the maximum time of all paths through the T-graph obeying the restrictions.

2. The set of paths with maximal execution time (worst case paths). The solution shows, which program parts belong to the paths that are most time-consuming. The set of execution paths is characterized by the values of the decision variables greater than zero.

3. The contribution of each program part: The MAXT of any program part can easily be derived using the values of the decision variables weighted by their execution times. So it is possible to calculate the MAXT *of program parts* as well.

4. The number of executions of every single edge in the computed worst case: After having solved the ILP problem, the decision variables contain the execution frequencies for all edges. This information helps to find program parts where improvements

Objective function of the ILP problem:

$$\begin{aligned} Z &= 36f_1 + 8f_2 + 86f_3 + 8f_4 + 112f_5 + 10f_6 + 30f_7 + 46f_8 + 10f_9 + 12f_{10}+ \\ & \quad 26f_{11} + 10f_{12} + 32f_{13} + 24f_{14} + 44f_{15} + 20f_{16} + 10f_{17} + 56f_{18}. \end{aligned}$$

Graph description:

$$\begin{aligned} s:&\ f_{19} &&= f_1 \\ v_1:&\ f_1 &&= f_2 + f_9 \\ v_2:&\ f_2 &&= f_3 \\ v_3:&\ f_3 &&= f_4 + f_6 \\ v_4:&\ f_4 &&= f_5 \\ v_5:&\ f_6 &&= f_7 \\ v_6:&\ f_5 + f_7 &&= f_8 \\ v_7:&\ f_9 + f_{12} &&= f_{10} \\ v_8:&\ f_{10} &&= f_{11} \\ v_9:&\ f_{11} &&= f_{12} + f_{13} \\ v_{10}:&\ f_{13} &&= f_{14} \\ v_{11}:&\ f_8 + f_{14} + f_{17} &&= f_{15} \\ v_{12}:&\ f_{15} &&= f_{16} \\ v_{13}:&\ f_{16} &&= f_{17} + f_{18} \\ t:&\ f_{18} &&= f_{19}. \end{aligned}$$

Backward edge and loops:

$$\begin{aligned} f_{10} &\leq 8f_9 \\ f_{15} &\leq 10f_8 + 10f_{14} \\ f_{19} &= 1. \end{aligned}$$

Nonnegativity constraints:

$$f_i \geq 0 \text{ for } 1 \leq i \leq 19.$$

Solution of the ILP problem:

$$Z = 1262$$

Settings of the decision variables:

$$\begin{array}{lllllll} f_1 = 1, & f_2 = 0, & f_3 = 0, & f_4 = 0, & f_5 = 0, & f_6 = 0, & f_7 = 0, \\ f_8 = 0, & f_9 = 1, & f_{10} = 8, & f_{11} = 8, & f_{12} = 7, & f_{13} = 1, & f_{14} = 1, \\ f_{15} = 10, & f_{16} = 10, & f_{17} = 9, & f_{18} = 1, & f_{19} = 1. \end{array}$$

Figure 6: Description of a MAXT-Problem as a Linear Programming Problem

have substantial influence on the quality of the MAXT. An optimizing compiler can use this information to concentrate on promising program parts.

Apart from the above-mentioned advantages, our approach gives the user the ability to manipulate parameters of the derived ILP problem to get more information about the program's timing behavior:

- Manipulating the coefficients of variables: The coefficients of the variables correspond to the execution times of the respective edges. A variation of these coefficients can help the user to learn about the global effects of local improvements within a piece of code. The effects on the MAXT can be analyzed before the improvements are really implemented. This helps the user to find program parts, where improvements reduce the maximum execution time of the overall program.

- Adding new functional constraints: If the person who analyzes a program finds new dependencies between program parts after having calculated the MAXT, new constraints describing these dependencies can be added to the ILP problem. If the solution does not violate any of the new restrictions, no improvements can be derived. The old solution is also valid for the new problem. If, on the other hand, the solution violates the added constraints, a new solution can be calculated. This new solution is better than or equal to the former MAXT.

6 Conclusion

This paper presented a new method for calculating the maximum execution times of programs. The method is derived from a graph-theoretical problem, the computation of a maximum cost circulation in a graph. In that problem, capacity constraints bound the minimum and maximum flow for every edge. We showed that the characterization of the flow on a per-edge basis is not sufficient to describe all feasible/infeasible paths through a program. Hence, we introduced relative capacity constraints. Relative capacity constraints allow us to relate the flows through edges to each other. We proved that a program description in the form of a T-graph and a set of relative capacity constraints allows us to characterize feasible execution paths sufficiently to compute exact MAXTs of programs, not just execution time bounds.

We have shown that our method is not restricted to programs of a specific structure. In contrast, it can also deal with programs which contain jumps, like break or return statements, and whose flow graph is irreducible. As a minimum requirement, however, the flow in all circles of those T-graphs has to be bounded by relative capacity constraints.

To obtain the MAXT for a program given in our description with T-graphs and relative capacity constraints we translate the description into an ILP problem. This ILP can be solved with standard tools available on the market. The solution of the ILP problem yields the MAXT of a program and produces detailed information about the execution times and execution frequencies of all program constructs.

References

[Ame85] P. Amerasinghe. *A Universal Hardware Simulator.* Undergraduate Honors Thesis, Dept. of Computer Sciences, University of Texas, Austin, TX, USA, Dec. 1985.

[Che87] M. Chen. *A Timing Analysis Language – (TAL).* Programmer's Manual, Dept. of Computer Sciences, University of Texas, Austin, TX, USA, 1987.

[Hil88] F. S. Hillier and G. J. Lieberman. *Operations Research.* Internationale Standardlehrbücher der Wirtschafts- und Sozialwissenschaften. R. Oldenbourg; Wien, München, 1988.

[Jun90] D. Jungnickel. *Graphen, Netzwerke und Algorithmen.* BI-Wissenschaftsverlag, Mannheim, Wien, Zürich, 2^{nd} edition, 1990.

[Kli86] E. Kligerman and A. Stoyenko. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, Sep. 1986.

[Li95a] Y. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 95–105, La Jolla, CA, USA, June 1995.

[Li95b] Y. S. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proc. 16th Real-Time Systems Symposium*, pages 198–307, Pisa, Italy, Dec. 1995.

[Mok89] A. K. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating Tight Execution Time Bounds of Programs by Annotations. In *Proc. 6th IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–80, Pittsburgh, PA, USA, May 1989.

[Par91] C. Y. Park and A. C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, 24(5):48–57, May 1991.

[Par93] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62, 1993.

[Pos92] G. Pospischil, P. Puschner, A. Vrchoticky, and R. Zainlinger. Developing Real-Time Tasks with Predictable Timing. *IEEE Software*, 9(5):35–44, Sep. 1992.

[Pus89] P. Puschner and Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, Sep. 1989.

[Pus93a] P. Puschner. *Zeitanalyse von Echtzeitprogrammen.* PhD Thesis, Technisch-Naturwissenschaftliche Fakultät, Technische Universität Wien, Wien, Österreich, Dez. 1993.

[Pus93b] P. Puschner and A. Schedl. A Tool for the Computation of Worst Case Task Execution Times. In *Proc. Euromicro Workshop on Real-Time Systems*, pages 224–229, Oulu, Finland, June 1993.

[Reh87] P. H. Rehm. A C Language Cross Development Environment for Real-Time Programming. Master's Thesis, University of California at Irvine, Irvine, CA, USA, Jul. 1987.

[Sch93] A. Schedl. Zeitanalyse von Echtzeitprogrammen mittels linearer Optimierung. Master's Thesis, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria, Sept. 1993.

[Sha89] A. C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, SE-15(7):875–889, July 1989.

[Sto87] A. Stoyenko. *A Real-Time Language With A Schedulability Analyzer.* Dissertation, Computer Systems Research Institute, University of Toronto, Canada, Dec. 1987.

[Sto91] A. Stoyenko, V. Hamacher, and R. Holt. Analyzing Hard-Real-Time Programs for Guaranteed Schedulability. *IEEE Transactions on Software Engineering*, SE-17(8):737–750, Aug. 1991.

[Wei81] Y. Wei. *Real-Time Programming with Fault Tolerance.* PhD Thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA, June 1981.

A Terminology in Graph Theory

In the literature relevant to graph theory some terms are used with different meanings. This appendix describes the term semantics used in this paper and mentions variations to other terminologies in literature.

A *(directed) graph* $G = (V, E)$ consists of a finite set $V = \{v_1, v_2, ..., v_n\}$ of elements called *vertices*, and a set $E = \{(v_{i_1}, v_{j_1}), (v_{i_2}, v_{j_2}), ..., (v_{i_m}, v_{j_m})\}$ of *ordered*[6] pairs of members of V called *edges*.

A *subgraph* S of a graph $G = (V, E)$ is a graph $S = (V', E')$ such that V' is contained in V, E' is contained in E, and the endpoints of any edge in E' are also in V'.

A *path* is a finite sequence of edges of the form $P = ((v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}))$, i.e., a finite sequence of edges in which the terminal vertex of each edge coincides with the initial vertex of the following edge.[7]

A graph is said to be *connected* if there is at least one chain joining every pair of distict vertices.

[6] In literature this definition often belongs to the term *directed graph* or *digraph*, while edges of a *graph* consist of *unordered* pairs of members of V.

[7] In literature this definition sometimes corresponds to the term *walk* while *path* is defined as a sequence of edges where all vertices and edges are *distinct*.

A *circulation* in a graph G is a vector $f = [f_1, f_2, ..., f_n]$ of n non-negative integral[8] numbers, where f_i represents the *flow* through edge e_i. This vector must comply with the *flow conservation condition* at each vertex, i.e., for every vertex v_j of G the sum of the flows in edges incident to v_j is equal to the sum of the flows in edges incident from v_j:

$$\sum_{e_m=(v_i,v_j)} f_m = \sum_{e_n=(v_j,v_k)} f_n$$

The *capacity* of an edge e_i is a non-negative integer $c(e_i)$ which denotes the maximum permissible value of the flow in the edge e_i.

A circulation is *feasible* or *legal* if and only if $b(e_i) \leq f_i \leq c(e_i)$ for each edge e_i, where $b(e_i)$ is the lower bound of flow through e_i and $c(e_i)$ is the capacity of e_i.

The *weight* w_i of an edge e_i is a real[9] number assigned to e_i. In our paper this weight belongs to the execution time (t_i) of the corresponding piece of code represented by the edge e_i.

[8]In graph theory the flow through an edge is defined as a vector of *real* numbers instead of integers.

[9]In the examples in this paper all weights are *integral* numbers since we are used to specify the execution times of pieces of code in CPU cycles.