Bishoksan Kafle

Modeling Assembly Program with Constraints

A Contribution to WCET Problem

Dissertação para obtenção do Grau de Mestre em
Lógica Computacional

Orientador: Pedro Barahona, Professor Catedrático,
Faculdade de Ciências e
Tecnologia,
Universidade Nova de Lisboa

Co-orientador: Franck Cassez, Principal Researcher,
NICTA

Júri:

Presidente: Prof. Doutor José Júlio Alferes
Arguente: Prof. Doutor Luis Gomes
Vogal: Prof. Doutor Pedro Barahona

September 2012

Bishoksan Kafle

Modeling Assembly Program with Constraints

A Contribution to WCET Problem

Dissertação para obtenção do Grau de Mestre em
Lógica Computacional

Orientador: Pedro Barahona, Professor Catedrático,
Faculdade de Ciências e
Tecnologia,
Universidade Nova de Lisboa

Co-orientador: Franck Cassez, Principal Researcher,
NICTA

Júri:

Presidente: Prof. Doutor José Júlio Alferes
Arguente: Prof. Doutor Luis Gomes
Vogal: Prof. Doutor Pedro Barahona

September 2012

Dedicated to my Parents

Acknowledgments

First, I would like to thank my supervisor Prof. Pedro Barahona for his kind supervision of this thesis. Without his scientific virtue, wide-knowledge, support and advices this thesis would have never been possible. I am also thankful for his great patience and understanding. My sincere gratitude to my co-supervisor Dr. Franck Cassez, NICTA, Australia for kind supervision, guidance, valuable advices and for the original proposal of the thesis.

I would like to thank the EMCL consortium for the two year scholarship and for giving me a chance to learn from the best. I would also like to thank all the staff members who guided me during my studies here in Europe.

I want to thank all of my friends in Dresden, Lisbon, Bolzano and Vienna who have made my these two years as one of the most beautiful periods in my life from all perspectives. I like to thank Sudeep Ghimire for proof-reading and for being there whenever I was in need.

Finally, my deepest gratitude goes to my beloved family and relatives for always encouraging, supporting and understanding me. In God I remain.

Abstract

Model checking with program slicing has been successfully applied to compute Worst Case Execution Time (WCET) of a program running in a given hardware. This method lacks path feasibility analysis and suffers from the following problems: The model checker (MC) explores exponential number of program paths irrespective of their feasibility. This limits the scalability of this method to multiple path programs. And the witness trace returned by the MC corresponding to WCET may not be feasible (executable). This may result in a solution which is not tight i.e., it overestimates the actual WCET.

This thesis complements the above method with path feasibility analysis and addresses these problems. To achieve this: we first validate the witness trace returned by the MC and generate test data if it is executable. For this we generate constraints over a trace and solve a constraint satisfaction problem. Experiment shows that 33% of these traces (obtained while computing WCET on standard WCET benchmark programs) are infeasible. Second, we use constraint solving technique to compute approximate WCET solely based on the program (without taking into account the hardware characteristics), and suggest some feasible and probable worst case paths which can produce WCET. Each of these paths forms an input to the MC. The more precise WCET then can be computed on these paths using the above method. The maximum of all these is the WCET. In addition this, we provide a mechanism to compute an upper bound of over approximation for WCET computed using model checking method. This effort of combining constraint solving technique with model checking takes advantages of their strengths and makes WCET computation scalable and amenable to hardware changes. We use our technique to compute WCET on standard benchmark programs from Mälardalen University and compare our results with results from model checking method.

Keywords: Worst Case Execution Time (WCET), constraint solving, model checking, static analysis.

Contents

List of Figures

List of Tables

Listings

Chapter 1

INTRODUCTION

"The best way to predict the future is to invent it."
Alan Kay

Hard real-time systems are those that have crucial deadlines. Typical examples of real-time systems include defense and space systems, embedded automotive electronics, air traffic control systems, command control systems etc. They are composed of a set of tasks and are characterized by the presence of a processor running application specific dedicated software. Here, a task may be a unit of scheduling by an operating system, a subroutine, or some other software unit. In these systems, the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced. So each real-time task has to be completed within a specified time frame.

In order to schedule these systems, we need to know some bounds about execution times of each task i.e. the worst-case execution-time (WCET). These bounds are needed for allocating the correct CPU time to the tasks of an application. They form the inputs for schedulability tools, which test whether a given task set is schedulable (and will thus meet the timing requirements of the application) on a given target system. Together with schedulability analysis, WCET analysis forms the basis for establishing confidence into the timely operation of a real-time system [1]. WCET analysis does so by computing (upper) bounds for the execution times of the tasks in the system.

This chapter presents the motivation behind this work, lists the contributions we made and presents the overall structure of the thesis.

1.1 Motivation

There are two main classes of methods for computing WCET [2]: Testing/Measurement and Verification based methods. But only the verification based methods (also known as static methods) are guaranteed to produce safe WCET [3]. Two different kinds of techniques are predominant in static methods, namely, Integer Linear Programming(ILP) based and Model Checking based.

ILP based techniques [2, 3] rely on the construction of a control flow graph (CFG) and the determination of the loop bounds. This can be achieved using user annotations or sometimes

inferred automatically. Hereby, the WCET is computed by solving a maximum cost circulation problem in this CFG. Each edge is associated with a certain cost for executing it. The algorithm implemented in these tools use both the program and the hardware specification to compute the CFG fed to the ILP solver. The architecture of the tool is thus monolithic i.e., it is not easy to add support for a new hardware. But these techniques are fast and can handle large programs [4]. They implement implicit path enumeration technique (IPET) and this considerably simplifies path analysis [5].

On the other hand, the model checking based techniques presented in [3] rely on the fully automatic method to compute a CFG(without user annotations). It describes the model of the hardware as a product of timed automata (independently of the program). The model of the program running on a hardware is obtained by synchronizing the program with the model of the hardware. Computing WCET is then reduced to a reachability problem on the synchronized model and solved using the real time model checker UPPAAL[1]. These techniques [3, 4] are slower and perform better for simplified programs. But allow easy integration of complex hardware models such as cache and pipelines.

Considerable amount of works have been done on both tasks [2, 3, 4, 5, 6], but each of them usually misses the aspect of the other. In this thesis, we purpose a technique which is a combination of these two to take advantage of their strengths and make WCET computation techniques scalable and amenable to changes. The idea behind this is to use ILP based techniques for path analysis and model checking based techniques for integration of complex hardware models.

To achieve this, we compute approximate WCET solely based on the program (only considering clock cycles for each instruction from its manual) without taking into account the hardware characteristics (cache and pipelines) using Constraint Programming (CP) instead of ILP. Based on this approximate value, we propose some feasible and most probable worst case paths to model checking based techniques. Model checking technique then combines the hardware model to this program path suggested by CP and computes the precise WCET. In doing so, we get rid of monolithicity problem of ILP based techniques and the scalability issues of the model checking techniques and yet keeping intact their strengths.

1.2 Structure of this work

The rest of this work is organized as follows. The next chapter provides an overview of timing analysis techniques and WCET. Chapter 3 provides some background knowledge about constraints and constraint solvers. Chapter 4 deals with the path analysis, mainly, the algorithms and results of path validation. Chapter 5 discusses briefly about high-level language generation from assembly language. Similarly, chapter 6 presents some rewriting rules to obtain Constraint Programming Model (CPM) of a high level language. Chapter 7 explains about our technique of computing approximate WCET. Next, chapter 8 concludes this thesis, by providing a summary of this work, and highlighting some possible research directions for future work. Finally, Appendix A is a supplementary chapter which presents CPM of some benchmark programs used to compute WCET.

[1]Real time model checker UPPAAL -`http://www.uppaal.org/`

1.3 Contributions

The major contribution of this research work is to complement model checking technique proposed in [3] with path analysis. To this purpose, we validate the witness trace returned by the model checker while computing WCET and generate test data if it is feasible. We explore further research directions to deal with infeasible traces/paths. We propose a technique based on constraint programming to compute an approximate WCET solely based on the program without taking into account the hardware characteristics (caches and pipelines). Based on this approximate WCET, we suggest some feasible and worst case paths to the model checking tool to integrate complex hardware characteristics in order to compute precise WCET. During this process, we deal with the decompilation of ARM assembly program to C like syntax and rewriting rules from C to constraints model. Further, we compare the results of our technique of approximate WCET computation with the results from Cassez et al. ([3]). Finally, we provide a way of computing an upper bound for over-approximation of WCET computed using model checking technique.

Chapter 2

TIMING ANALYSIS TECHNIQUES AND WCET

"Joy in looking and comprehending is nature's most beautiful gift."
Albert Einstein

This chapter presents some background knowledge about WCET, its challenges and some methods and tools for computing WCET. The knowledge of the maximum time consumption of each program or task or piece of code is a prerequisite for analyzing the worst-case timing behavior of a real-time system and for verifying its temporal correctness. This maximum time needed by each program is assessed by means of WCET analysis. Figure 2.1 taken from [2] depicts several relevant properties of a real-time task. The lower curve represents a subset of measured executions. Its minimum and maximum are the minimal observed execution times and maximal observed execution times, resp. The darker curve, an envelope of the former, represents the times of all executions. Its minimum and maximum are the best-case and worst-case execution times, resp., abbreviated BCET and WCET.

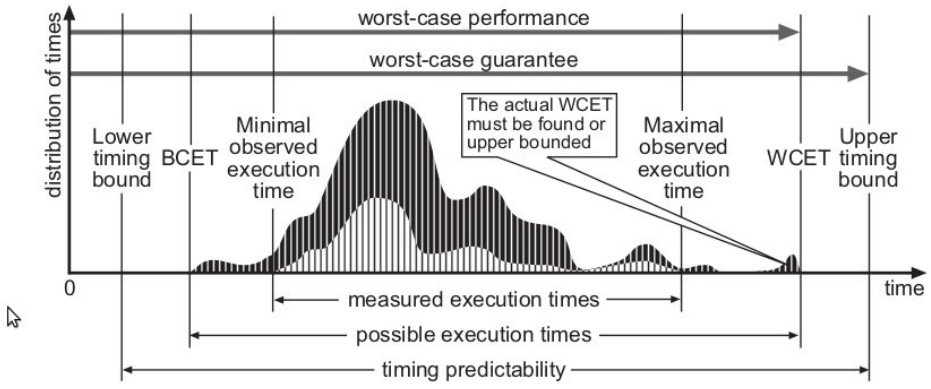

Figure 2.1: *Basic notions concerning timing analysis of systems.*

A task typically shows a certain variation of execution times depending on the input data or

different behavior of the environment. The set of all execution times is shown as the upper curve. The shortest execution time is called the BCET, the longest time is called the WCET. In most cases the state space is too large to exhaustively explore all possible executions and thereby determine the exact worst-case and best-case execution times. Timing analysis is a process of deriving execution-time bounds or estimates. A tool that derives bounds or estimates for the execution times of application tasks is called a timing-analysis tool.

2.1 Overview of Timing Analysis Techniques

Timing analysis attempts to determine bounds on the execution times of a task when executed on a particular hardware. The time for a particular execution depends on the path through the task taken by control (referred to as the program path analysis problem) and the time spent in the statements or instructions on this path on this hardware (referred to as micro-architectural modeling). Both these aspects need to be studied well in order to provide a solution to this problem. The focus of this thesis is on the program path analysis problem. The structure and the functionality (i.e., what the program is computing) of a program determines the actual paths taken during its execution. Any information regarding these helps in deciding which program paths are feasible and which are not. While some of this information can be automatically inferred from the program, this is a difficult task in general (e.g., regarding the information about functionality of a program).

Accordingly, the determination of execution-time bounds has to consider the potential control-flow paths and the execution times for this set of paths. A modular approach to the timing-analysis problem splits the overall task into a sequence of subtasks. Some of them deal with properties of the control flow, others with the execution time of instructions or sequences of instructions on the given hardware. Many of today's approaches to WCET analysis demand that the programmer provide information about (in)feasible execution paths of the code to be analyzed. This path information is described at the high-level language interface. On the other hand, the actual computation of WCET, that uses this path information, takes place at the machine-language level, where the execution times of basic actions can be accurately modeled. Today's practical approaches to WCET analysis therefore have to bridge the gap between these two different representation levels. This makes timing analysis difficult and interesting as a research topic. The progress in this field has led to a number of techniques and tools which will be discussed in the next sections.

2.2 WCET

WCET analysis computes upper bounds for the execution times of programs for a given application, where the *execution time* of a *program* is defined as the time it takes the processor to execute that *program*. Formally, it can be defined as [3]:

Definition 2.1 *The WCET of a program P on a hardware H, represented as $WCET(P, H)$, is the maximum execution time($time(H, P, d)$) of P on H for all input data d, where $time(H, P, d)$ is the execution time of P for input data d on H.*

Mathematically,

$$WCET(P, H) = \max_{d \in D} \; time(H, P, d) \; , \forall d$$

In general, the WCET problem is undecidable as it is equivalent to solving the halting problem. However, real-time systems only use a restricted form of programming, which guarantees that

programs always terminate; recursion is not allowed or explicitly bounded as are the iteration counts of loops. A reliable guarantee based on the worst-case execution time of a task could easily be given if the worst-case inputs for the task were known. Unfortunately, in general the worst-case inputs are not known and are hard to derive. We should bear in mind the following points regarding the WCET [1]:

1. WCET analysis computes upper bounds for the WCET, i.e. it does not guarantee to return the WCET exactly.
2. The WCET bound computed for a piece of code is application- dependent , i.e., a single piece of code may have different WCETs, and thus WCET bounds in different application contexts.
3. WCET analysis assesses the duration that the processor is actually executing the analyzed piece of code, i.e. assuming non-preemptive execution. It is important to note that the results of WCET analysis do not include waiting times due to preemption, blocking, or other interference.
4. WCET analysis is hardware-dependent. WCET analysis therefore has to model the features of the target hardware on which the code is supposed to execute.

In order to make real-time systems temporally predictable and to keep the price of such systems reasonable, the computed WCET has to be [1, 3] :

1. safe i.e., it must not under-estimate the worst case, and
2. tight otherwise either the set of tasks are wrongly declared non schedulable or the cost has to be paid in order to compensate with the pessimism.

Let C be the computed WCET and M be the measured WCET on the real platform for some program P, then the over approximation is given by the formula: $(C - M)/M * 100$. The computed WCET is considered tight when the difference between C and M is lesser or equal to some epsilon(ϵ).

2.2.1 WCET Challenges

Computing the WCET of a program is a challenging and difficult task. It is usually a very hard problem and there are several factors which are responsible for this [1, 3] :

1. WCET analysis has to consider all possible inputs of the program to ensure a real safe upper bound;
2. The hardware on which a program runs usually features a multi-stage pipelined processor and some fast memory components called caches; executing a sequential program is then a concurrent process where the different stages of the pipeline and the caches and the main memory run in parallel;
3. The WCET must be computed on the binary code (or an assembly language equivalent version) where the execution-time of basic actions can be accurately modeled;
4. The characterization of execution path takes place on the source-code level, but the WCET must be computed on the binary code, so the WCET analysis has to bridge this gap between these two different representations. Modern compilers produce smart or optimized compiled programs making it difficult to identify in the machine code the execution paths that have been characterized at the source level.

2.2.2 WCET Methods and Tools

There are two main classes of methods for computing WCET [2, 7] :

1. Testing-based methods or Measurement based methods: These methods attack some parts of the timing-analysis problem by executing the given task on the given hardware or a simulator, for some set of inputs. They then take the measured times and derive the maximal and minimal observed execution times, or their distribution or combine the measured times of code snippets to results for the whole task. The measurements of a subset of all possible executions produce estimates, not bounds for the execution times, if the subset is not guaranteed to contain the worst case. Even one execution would be enough if the worst-case input were known but in general, this is as difficult as computing WCET. These methods might not be suitable for safety critical embedded systems but they are versatile and rather easy to implement. There are some tools which implement these techniques, to mention a few, RapiTime[1] and Mtime [8] etc.

2. Verification-based methods or static methods: This class of methods does not rely on executing code on real hardware or on a simulator, but rather considers the code, combines it with some (abstract) model of the system i.e. hardware, and obtains upper bounds from this combination. Static methods compute bounds on the execution time. The common things among the tools which implement these methods are computation of an abstract graph, the CFG, and an abstract model of the hardware. Then with static analysis tool they can be combined to get WCET. The CFG should produce a super-set of the set of all feasible paths. Thus the largest execution time on the abstract program is an upper bound of the WCET. Such methods produce safe WCET, but are difficult to implement. The price they pay for this safety is the necessity for processor-specific models of processor behavior, and possibly imprecise results such as overestimated WCET bounds. In favor of static methods is the fact that the analysis can be done without running the program to be analyzed which often needs complex equipment to simulate the hardware and peripherals of the target system. In spite of these difficulties of implementation, there are some tools which implement these techniques, to mention a few, Bound-T [2], Chronos [9], SWEET [10] and aiT [3] [11] etc.

Though it is widely discussed by Wilhelm ([12]) that MC is not good for WCET computation, Cassez et al. ([3]) and Dalsgaard et al. ([6]) justify its use to compute WCET and have tools based on model checking techniques. The above mentioned verification tools have following limitations:

1. these methods for computing WCET rely on annotations on the binary program (equivalently assembly program) to analyze. These annotations are often manually asserted which are error-prone.

2. the algorithms and tools that implement these methods are rather monolithic and difficult to adjust to a new hardware.

Cassez et al. ([3]) solve these limitations using model checking technique. However, this technique suffers from the following problems:

[1] Rapita Systems Ltd. Rapita Systems for timing analysis of real-time embedded systems. `http://www.rapitasystems.com/`

[2] Tidorum Ltd. Bound-T time and stack analyser. `http://www.bound-t.com/`.

[3] AbsInt Angewandte Informatik. aiT Worst-Case Execution Time Analyzers. `http://www.absint.com/ait/`.

1. The MC explores exponential number of program paths irrespective of their feasibility. These limit MC's scalability to multiple path programs.

2. The witness trace returned by the MC corresponding to the WCET may not be feasible (executable). This may result in solution which is not tight.

Thus the goal of this thesis is to complement Cassez et al.'s technique [3] with static path analysis. This technique can be summarized in the following three steps [3]:

1. Construction of CFG: The CFG of a program P is built using the technique of program slicing [13], in an iterative manner. The starting point is a partial CFG P_0 built as follows: P is unfolded from the initial instruction and the unfolding process stops when (1) it reaches final instruction or (2) it reaches an instruction for which the next value of register pc is unknown (e.g., a branch instruction with a computed target). The value of target is obtained by further slicing P_0 with an ad hoc slice criterion: the slice enables to compute the possible values of the target. Then P_0 is extended using this new information (performing the unfolding as before) and P_1 is obtained . Repeating this operation will build the full CFG P_n of P. As it assumes that P always terminates this iterative computation is guaranteed to terminate as well.

2. Modeling Hardware: This method is centered around a number of models as it needs to model main memory, caches and pipelines. These are modeled using timed automata (TA). As the focus of this thesis is not in the micro-architectural modeling, we refer the readers to [3] for detailed description. It should be noted that this model is independent from the program description.

3. WCET Computation as a Reachability Problem: The model of a program P running on a hardware is obtained by synchronizing (the automaton of) the program with the (TA) model of the hardware. Computing the WCET is reduced to a reachability problem on the synchronized model and solved using the model-checker UPPAAL. It is assumed that P has a set of initial states I (pc gives the initial instruction of P). P has also a set of final states F (e.g., pc with a particular value). The language $\mathcal{L}(P)$ of P is the set of traces generated by runs of P that starts in I and ends in F. A trace here is a sequence of assembly instructions. As we assume that P always terminates, this language is finite. Then it can be generated by a finite automaton $Aut(P)$. The hardware H (including pipeline, caches and main memory) can be specified by a network of timed automata $Aut(H)$. Feeding H with $\mathcal{L}(P)$ amounts to building the synchronized product $Aut(H) \times Aut(P)$. On this product final states are defined when the last instruction of P flows out of the last stage of pipeline. A fresh clock x is reset in the initial state of $Aut(H) \times Aut(P)$. The WCET of P on H is then the largest value, $max(x)$, that x can take in a final state of $Aut(H) \times Aut(P)$ (we assume that time does not progress from a final state). We can compute $max(x)$ using model-checking techniques with the tool UPPAAL. To do this, a reachability property "(R): Is it possible to reach a final state with $x \geq K$?" is checked on $Aut(H) \times Aut(P)$. If the property is true for K and false for $K + 1$, then K is the WCET of P.

2.3 Previous Work

WCET analysis caught attention about two decades ago ([14], [15], [16], [17]). Substantial progress has been made in this area since then. Several methods/tools have been made available for computing WCET. WCET analysis is usually divided into three parts: a fairly machine-independent flow analysis (or "high-level analysis") of the code, where information about the

possible program flows is derived, a low-level analysis where the execution time for atomic parts of the code is decided from a performance model for the target architecture, and a final calculation where the information from these analyses is put together in order to derive the actual WCET bounds. There are three main categories of calculation methods proposed in the WCET literature: *structure-based* [18], *path-based* [19] and *Implicit Path Enumeration Technique* (IPET) [2, 5, 20, 21].

Path-based methods suffer from exponential complexity and tree-based methods cannot model all types of program-flow, leaving IPET as the preferred choice for calculation because of the ease of expressing flow dependencies and the availability of efficient ILP solvers. IPET constraint systems can be solved using either constraint-programming [22, 23] or ILP [5, 24] with ILP being the most popular. A large number of tools use these techniques, to mention a few: *aiT* [11], *Bound-T*, *Chronos* [9] etc.

Fig. 2.2(a) taken from [2] shows an example control-flow graph with timing on the nodes and a loop-bound flow fact. Fig. 2.2(d) illustrates how a structure-based method would proceed according to the task syntax tree and given combination rules. Fig. 2.2(b) illustrates how a path-based calculation method would proceed over the graph in Fig. 5(a). The reader is refered to [2, 20] for an exhaustive presentation of the first two methods.

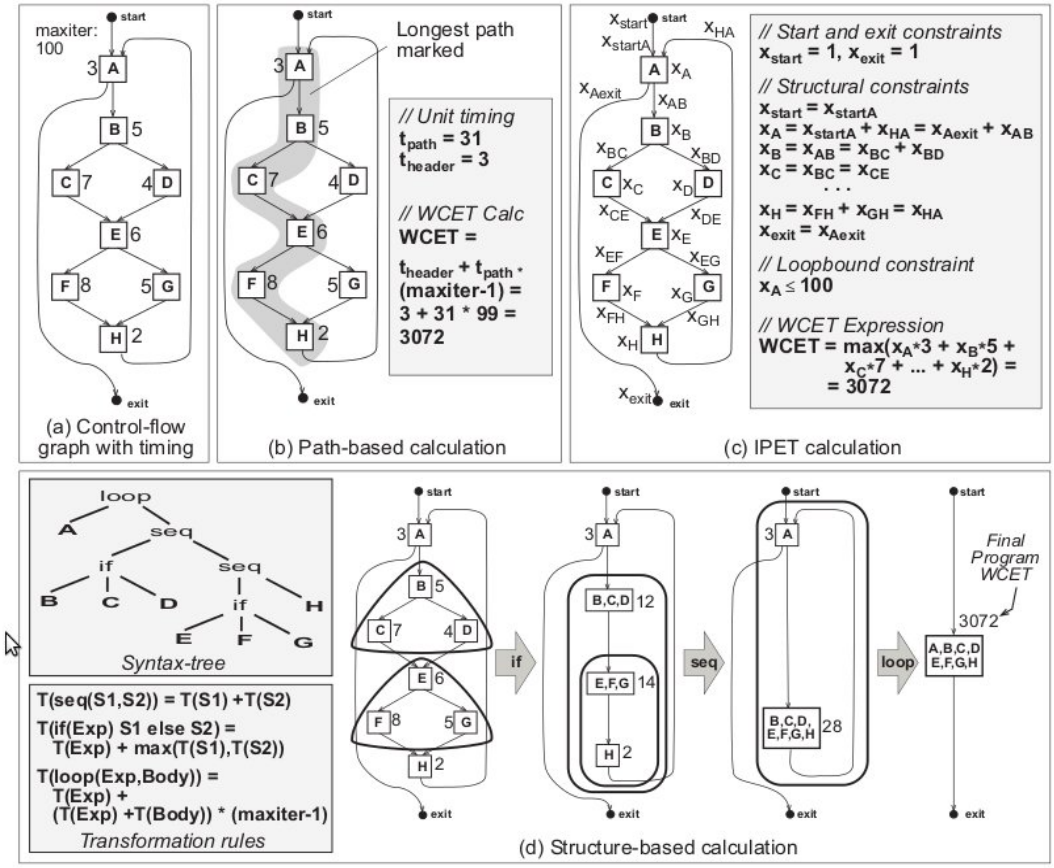

Figure 2.2: *WCET calculation methods.*

2.3.1 IPET

IPET calculation is based on a representation of program flow and execution times using algebraic and/or logical constraints. Each basic block and/or edge in the basic block graph is given a time (t_{entity}) and a count variable (x_{entity}), denoting the number of times that block or edge is executed. The WCET is found by maximizing the $\sum_{i \in entities} x_i * t_i$, subject to constraints reflecting the structure of the program and possible flows.

Figure 2.2(c) shows the constraints and WCET formula generated by a IPET-based calculation method for the program illustrated in Figure 2.2(a). The start and exit constraints states that the program must be started and exited once. The structural constraints reflects the possible program flow, meaning that for a basic block to be executed it must be entered the same number of times as it is exited. The loop bound is specified as a constraint on the number of times node A can be executed.

This thesis aims to model structural as well as functionality constraint of a program automatically to filter out infeasible paths using verification techniques as in [25]. We use constraint programming which allows for more complex constraints to be expressed and provides great facility for automatic loop bounds inference, with a potential risk of larger solution times. The use of constraint logic programming is reported in [26, 27, 28] in the WCET community to handle complex flow analysis and timing variability.

2.4 General Consideration

After having looked at the state of the art for computing WCET, we found that only verification based techniques can produce safe WCET bounds [3]. The mostly used tools use static analysis and ILP and few others use Model Checking techniques to compute WCET. The tools which implement the first technique are scalable but need some manual intervention to provide program annotations (e.g., loop bounds) and are monolithic in nature while those which implement the second technique solve these problems and suffers from scalability issues as they do not filter out infeasible paths while computing WCET. Thus this thesis aims at bridging the gap between these techniques to take advantage of their strengths to compute WCET.

Chapter 3

CONSTRAINTS AND CONSTRAINT SOLVERS

Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.
E. C. Freuder, Constraints, 1997.

A constraint [29] is a restriction on the space of possibilities for some choice; it can be considered as a piece of knowledge that filters out the options that are not legitimate to be chosen, and hence narrowing down the size of the space. Formulating problems in terms of constraints have proven useful for modeling fundamental cognitive activities such as vision, language comprehension, default reasoning, diagnosis, scheduling, and temporal and spatial reasoning, as well as having applications for engineering tasks, biological modeling, and electronic commerce. In this chapter, we provide some background knowledge about constraints and constraint solvers.

3.1 Constraint Satisfaction Problem (CSP)

Basically, a CSP is a problem composed of a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restricts the values the variables can simultaneously take. The task is to assign a value to each variable satisfying all the constraints [30].

Definition 3.1 (CSP) *A CSP can be defined as the triple $\langle X, D, C\rangle$, where $X = \{x_1, ..., x_n\}$ is a finite set of variables, with respective domains $D = \{D_1, ..., D_n\}$ which list the possible values for each variable $D_i = \{v_1, ..., v_k\}$, and a set of constraints $C = \{C_1, ..., C_t\}$. A constraint C_i can be viewed as a relation R_i defined on the set of variables $S_i \sqsubseteq X$ such that R_i denotes the simultaneous legal value assignments of all variables in S_i . Thus, the constraint C_i can be formally defined as the pair $\langle S_i, R_i\rangle$; S_i is called the scope of the constraint.*

Definition 3.2 (Solution CSP) *A solution of the CSP is an n-tuple $\langle v_1, ..., v_n\rangle$ where each $v_i \in D_i$ corresponds to the value assigned to each variable $x_i \in X$, and the assignment satisfies all constraints in C simultaneously.*

Consider the famous n Queens Problem as an example of CSP. One is asked to place n queens on the $n \times n$ chess board, where $n \geq 3$, so that they do not attack each other. One possible

representation of this problem as a CSP uses n variables, $x_1, ..., x_n$, each with the domain $[1..n]$. The idea is that x_i denotes the position of the queen placed in the i^{th} column of the chess board. The appropriate constraints can be formulated as the following dis-equalities for $i \in [1..n-1]$ and $j \in [i+1..n]$:

- $x_i = x_j$ (no two queens in the same row),
- $x_i - x_j = i - j$ (no two queens in each South-West – North-East diagonal),
- $x_i - x_j = j - i$ (no two queens in each North-West – South-East diagonal).

The sequence of values (6,4,7,1,8,2,5,3) corresponds to a solution for $n = 8$, since the first queen from the left is placed in the 6^{th} row counting from the bottom, and similarly with the other queens.

3.2 Constraint Satisfaction Optimization Problem (CSOP)

All solutions are equally good for solving CSPs. In applications such as industrial scheduling, some solutions are better than others. In other cases, the assignment of different values to the same variable gives different costs. The task in such problems is to find optimal solutions, where optimality is defined in terms of some application-specific functions. We call these problems CSOP [30].

Definition 3.3 (CSOP) *A CSOP $\langle X, D, C, f\rangle$ is defined as a CSP together with an optimization function f which maps every solution tuple to a numerical value, where $\langle X, D, C\rangle$ is a CSP, and if S is the set of solution tuples of $\langle X, D, C\rangle$, then $f : S \rightarrow$ numerical value. Given a solution tuple t, we call $f(t)$ the f-value of t. The task in a CSOP is to find the solution tuple with the optimal (minimal or maximal) f-value with regard to the application-dependent optimization function f.*

As an example consider the following problem:

$$min.\ f(x, y) = x^2 + 2y^2 + 2xy - 18$$

subject to the constraint

$$x - y = 1$$

We are looking for the minimum value for $f(x; y)$ over the domain of $x; y$ that satisfy $x-y = 1$.

3.3 Constraint Solvers

There are two broad classes of constraint solvers: *complete solvers* and *incomplete solvers*. We will briefly discuss about them in the the following subsections.

3.3.1 Complete solvers

Complete solvers implement decision procedures that take a given *CSP* and produces a solved form of the problem [31]. Examples of complete solvers include *CLP(R)* and *CLP(B)* that are used for solving real constraints and boolean constraints respectively. Since *CLP(R)* implements simplex algorithms, solving linear constraints is quite efficient. However, for *CLP(B)*, the fact that the underlying representation of boolean functions is based on Boolean Decision Diagrams results in exponential time being required for solving constraints.

3.3.2 Incomplete solvers

The most interesting and fundamental concept in constraint solving that drives incomplete solvers is called constraint propagation. Solvers that implement propagation techniques are based on the observation that if the domain of any variable in some *CSP* is empty, the *CSP* is unsatisfiable. These solvers try to transform a given CSP into an equivalent CSP whose variables have a reduced domain. If any of the domains in the reduced CSP becomes empty, the reduced *CSP*, and hence the original *CSP* are said to be unsatisfiable since both *CSPs* are equivalent. The solvers work by considering each constraint of the *CSP* one by one, and they use the information about the domain of each variable in the constraint to eliminate values from domains of the other variables. These procedures alone may not succeed in getting a solution as the case often, and hence enumeration of variables can be also needed. Therefore, incomplete solvers interleave propagation and enumeration to obtain a solution or to infer the absence of any solution. An incomplete solver can find a solution to a problem, but it can't distinguish between there being no solution and the solver's inability to find it [31]. This means, reaching the fix point would not guarantee the feasibility. In order to guarantee this, we need to label the input variables.

Example of such solver includes *Choco*, *Comet [32]* and *CLP(FD)*. *Choco* is a java library for CSP, CP and explanation-based constraint solving (e-CP). It is built on a event-based propagation mechanism with backtrackable structures. Choco is an open-source software, distributed under a BSD license. The details can be obtained from *Choco's* homepage [1]. *Comet* is a hybrid optimization system, combining CP, local search, and linear and integer programming. It is also a full object-oriented, garbage collected programming language, featuring some advanced control structures for search and parallel programming, supplemented with rich visualization capabilities. The details can be obtained from *Comet's* homepage [2]. *CLP (FD)* is used to solve constraints over finite domains. Boolean constraints can also be modeled here as a special case of finite domain constraints with each variable having domain $D = \{0, 1\}$. Since, propagation may be of no use in the worst case scenario, the *CLP(FD)* solver has an exponential time complexity on the size of the domains. There are different levels of consistency criteria that can be achieved by the constraint propagation algorithm. The most important ones include *node consistency, arc consistency*, and *bound consistency*.

A *CSP* is *node-consistent* if there does not exist a value in the domain of any one of its variables that violates a unary constraint in the *CSP*. In particular, a CSP with no unary constraints is vacuously node consistent. This criterion is of course very trivial but it is very important when it is considered in the context of an execution model that incrementally computes solution from partial solutions. Consider now a CSP of the form:
$< \{x_1, ...x_n\}, \{x_1...x_{n-1} \in \mathbb{N}, x_n \in \mathbb{Z}\}, \{x_1 \geq 0, ..., x_n \geq 0\} >$ where $\mathbb{N}$ denotes set of natural numbers and $\mathbb{Z}$ denotes the set of all integers. Then this CSP is not node consistent, since for the variable x_n the constraint $x_n \geq 0$ is not satisfied by the negative integers from its domain. But when we change the domain of the variable x_n to be $\mathbb{N}$ then this CSP becomes node consistent since for every variable x_i every unary constraint on x_i coincides with the domain of x_i, where $i = 1..n$.

A more demanding consistency criterion is *arc-consistency*. To be considered for *arc-consistency*, a *CSP* must first be node-consistent. In addition, for every pair of variables $\langle x, y \rangle$, for every

[1] `http://choco.emn.fr/`
[2] `http://dynadec.com/`

constraint C_{xy} defined over variables x and y , and for each value v_x in the domain of x, there must exist some value v_y in the domain of y that supports v_x . For example, the *CSP* $\langle\{x, y\}, \{1..5, 1..5\}, \{x + y > 7\}\rangle$ is not arc-consistent because there is no support in the domain of y when x takes 1 or 2 that satisfies the constraint $x + y > 7$. The same holds for y also. An arc-consistent *CSP* which is equivalent to the original *CSP* is obtained by reducing domains of x and y from $\{1, 2, 3, 4, 5\}$ to $\{3, 4, 5\}$.

Another type of consistency criteria is called *bounds consistency* defined on numeric constraints which are arithmetic constraints of equalities or inequalities. For example, the *CSP* $\langle\{x, y\}, \{1..10, 1..10\}, \{x > y\}\rangle$ in not bound-consistent because there are some bound values in the domain of both variables that can never be part of any solution as they do not have any matching value in the other variable to satisfy the given constraint. If x takes the value 1, then there is no any matching value in y that can satisfy the constraint $x > y$. Therefore 1 should not be in the domain of x. Similarly, there is no matching value for x when y takes the value 10. Likewise, 10 should not be in the domain of y. A bound-consistent equivalent *CSP* will be $\langle\{x, y\}, \{2..10, 1..9\}, \{x > y\}\rangle$. Another example can be the *CSP* $\langle\{x, y\}, \{3..10, 1..8\}, \{x = y\}\rangle$. There is no any matching values for y when x takes either 9 or 10 because we have an equality constraint $x = y$. Similarly should y take either 1 or 2, there is no matching value in x that satisfies the given constraint. A bound-consistent equivalent CSP in this case will be $\langle\{x, y\}, \{3..8, 3..8\}, \{x = y\}\rangle$.

Algorithms that impose arc-consistency are polynomial on the number of variables, where as algorithms that impose bounds-consistency are linear on the size of domains of variables. Global constraints have specialized propagation algorithms that exploit the semantics of the constraints to obtain a much faster propagation, and hence a much faster solving of the constraints.

To this end, we have chosen *Comet* as the constraint solver to use in this thesis because of the following reasons:
1. It can handle all the constraints in our case (linear and non-linear),
2. It provides multiple facilities like CP and linear programming (LP),
3. It is free for educational purpose,
4. We have good knowledge of it.

3.4 Constraint programming

CP is an emergent software technology for declarative description and effective solving of large, particularly combinatorial, problems especially in areas of planning and scheduling. It has its roots in computer science, logic programming, graph theory, and the artificial intelligence efforts of the 1980s. CP consists of optimizing a function subject to logical, arithmetic, or functional constraints over discrete or interval variables, or finding a feasible solution to a problem defined by logical, arithmetic, or functional constraints over discrete or interval variables. It is also an efficient approach to solving and optimizing problems that are too irregular for mathematical optimization. This includes time tabling problems, sequencing problems, and allocation or rostering problems.

CP can be characterized pretty well by the equation:

$$CP = Model + Search$$

A CP model looking for a feasible solution has the structure (in *Comet* [32]) as shown in the figure 3.1:

```
import cotfd;
Solver<CP> cp();
//declare the variables
solve<cp> {
    //post the constraints
}
using {
     //non deterministic search
}
```

Figure 3.1: *An example of CP model for a feasible solution in Comet*

First it imports the library that is needed, in this case the finite domain one ($cotfd$). Then specifies the solver to use, which is CP. It can be seen that there is a clear separation between the modeling part, that declares the variables and posts the constraints, and the search part. We now briefly explain about the ingredients of CP.

3.4.1 Variable

The first step in modeling a problem is to declare variables. *Comet* has three primitive types: *int, float* and *bool*. These primitive types are given by value in function or method parameters. There are four types of incremental variables: *integer, floating point, boolean*, and *set over integers*. Incremental variables can be seen as a generalized version of typed variables with extra functionality. Each incremental variable is assigned a domain of values, either automatically or explicitly by the user. They are declared as below.

```
var<CP>{int} x(cp,1..10);
var<CP>{bool} b(cp);
var<CP>{float} f(cp,1,5);
var<CP>{set{int}} s(cp);
```

Discrete integer variables, also called finite domain integer variables (f.d. variables), are the most commonly used. The first line in the above example declares a variable x with the integer interval domain $[1..10]$ in *Comet*. The second line declares a boolean variable b and the third line a float variable f in the range of 1 and 5 etc.

3.4.2 Constraints

Constraints act on the *domain store* (the current domain of all variables) to remove inconsistent values. Behind every constraint, there is a sophisticated filtering algorithm, that prunes the search space by removing values that don't participate in any solution satisfying the constraint. The domain store is the only possible way of communication between constraints: whenever a constraint C_1 removes a value from the domain store, this triggers the detection of a possible inconsistent value for another constraint C_2 . This inconsistent value is in turn removed, and this propagates, until reaching a $fixedpoint$, which means that no constraint can remove a value. This is the basic idea of the $fixpoint$ algorithm. As soon as a variable's domain becomes empty, the domain store fails, meaning that there is no possible solution, and the search has to

backtrack to a previous state and try another decision. To summarize, a constraint system must implement two main functionalities:

1. *Consistency Checking*: verify that there is a solution to the constraints, otherwise tell the solver to backtrack

2. *Domain Filtering*: remove inconsistent values, i.e., values not participating in any solution

A constraint can be posted to the CP solver with the post method. The constraint must always be posted inside a $solve\,\{\}$, $solveall\,\{\}$ or $suchthat\,\{\}$ block, otherwise $Comet$ does not guarantee the results. The following example posts the constraint that variables x and y must take two different values.

```
cp.post(x != y);
```

3.4.3 Search

Search in CP consists in a non-deterministic exploration of a tree with a backtracking search algorithm. The default exploration algorithm is Depth-first Search. Other search strategies available are: Best-First Search, Bounded Discrepancy Search, Breadth-First Search, etc. The following example in *Comet [32]* explores all combinations of three 0/1 variables using a depth-first strategy:

```
import cotfd;
Solver<CP> cp();
var<CP>{int} x[1..3](cp,0..1);
solveall <cp> {
}
using {
label(x); //search part
cout << x << endl;
}
```

Figure 3.2: *An example of CSP in Comet*

This produces the following output.

```
x[0,0,0]
x[0,0,1]
x[0,1,0]
x[0,1,1]
x[1,0,0]
x[1,0,1]
x[1,1,0]
x[1,1,1]
```

Constraint programming has been extended to constraints over other domains among the most important ones being *boolean constraints*, *real linear constraints* and *finite domain constraints*.

Boolean Constraints: A constraint is called boolean if each variable in the constraint has a

domain $D = \{0, 1\}$. Such constraints are particularly useful for modeling digital circuits, and boolean constraint solvers can be used for verification, design, optimization etc. of such circuits. An example of boolean constraint is shown below where the domain of the variables x, y, z is D:

$(x \vee y) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg z)$

Real Linear Constraints: Such constraints have variables that can take any real value. Unlike finite domains, real domains are continuous and infinite. The solver called $clp(R)$ is bundled into many Prolog implementations as a library package which is used to solve real constraints. In addition to all the common arithmetic constraints, $clp(R)$ solves a number of linear equations over real-valued variables, covers the lazy treatment of nonlinear equations, features a decision algorithm for linear inequalities that detects implied equations, removes redundancies, performs projections (quantifier elimination), allows for linear in-equations, and provides for linear optimization. They are not present in *Comet*. An example of such constraint is shown below where the domain of the variables x, y is real number:
$2.4 * x^5 \leq 10.1 + 4.2 * y^6$

Finite Domain Constraints All variables in such constraints get associated with some finite domain, either explicitly declared by the program, or implicitly imposed by the finite-domain constraint solver. By finite domain, we mean some subset of integers. Therefore, only integers and unbound variables are allowed in finite domain constraints.

Finite-domain constraint solvers mainly deal with two classes of constraints called *primitive constraints* and *global constraints*. All other types of constraints are automatically translated to conjunctions of primitive and global constraints, and then solved. Classes of *primitive constraints* include (among others):

- **Arithmetic constraints**: Examples include $x + y = 5$ which constraints x and y to take values that can be summed up only to 5, and $x > y$ which constraints the value taken by x to be always greater than the value taken by y, and
- **Propositional constraints**: are complex constraints formed by combining individual constraints using propositional combinators. The main propositional combinators include $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ which play roles similar to logical conjunction, disjunction, implication and bi-implication respectively. For example, given constraints C_1 and C_2 , the propositional constraint $C_1 \wedge C_2 (C_1 \vee C_2)$ is satisfied if and only if both(either one of C_1 or C_2) is satisfied. Another propositional constraint $C_1 \Rightarrow C_2$ evaluates to true if C_1 is false or C_2 is true. An important property of this constraint is that if C_1 evaluates to true, then C_2 should necessarily evaluate to true. This can be used to specify constraints that are needed only under some condition. For this reason, such constraints are also called conditional constraints.

Some of the most important global constraints defined in *Comet* include *alldifferent, atleast, atmost* and *cardinality* etc. The *alldifferent* function allows to state that each variable in an array of CP variables takes a different value. For example:

```
1 import cotfd;
2 Solver<CP> cp();
3 var<CP>{int} x[1..5](cp,1..6);
4 solve<cp>
5 cp.post(alldifferent(x));
```

A solution is $x = [4, 2, 5, 1, 6]$ since all the values are different. Details about other global constraints defined in *Comet* can be obtained from its manual [32].

An important concept in search is labeling of variables. The labeling functions are used on variables over finite domains to check satisfiability or partial satisfiability of the *constraint store* (the constraint store contains the constraints that are currently assumed satisfiable.) and to find a satisfying assignment [31]. A labeling function is of the form *label* ($\langle variable \rangle$), where the argument is a variable over finite domain. Whenever the interpreter evaluates such a function, it performs a search over the domains of the variable to find an assignment that satisfies all relevant constraints. Typically, this is done by a form of *backtracking*: variables are evaluated in order, trying all possible values for each of them, and backtracking when inconsistency is detected.

The first use of the labeling function is to actually check satisfiability or partial satisfiability of the constraint store. When the solver adds a constraint to the constraint store, it only enforces a form of local consistency on it. This operation may not detect inconsistency even if the constraint store is unsatisfiable. A labeling function over a set of variables enforces a satisfiability check of the constraints over these variables. As a result, using all variables mentioned in the constraint store results in checking satisfiability of the store.

The second use of the labeling function is to actually determine an evaluation of the variables that satisfies the constraint store. Without the labeling functions, variables are assigned values only when the constraint store contains a constraint of the form $x = value$ and when local consistency reduces the domain of a variable to a single value. A labeling function over some variables forces these variables to be evaluated. In other words, after the labeling functions have been considered, all variables are assigned a value. Typically, constraint solvers are written in such a way that labeling functions are evaluated only after as many constraints as possible have been accumulated in the constraint store. This is because labeling functions enforce search, and search is more efficient if there are more constraints to be satisfied. Consider the example presented in the figure 3.2. When the solver solves this *CSP*, The function label(x) is evaluated as all constraints are satisfied in the constraint store (in fact there are no constraints), forcing a search for a solution of the constraint store. Since the constraint store contains exactly the constraints of the original *CSP*, this operation searches for a solution of the original problem. Please refer to chapter 6.1.2 for sophisticated labeling policies.

3.5 Partial Conclusion

In order to understand better the matter of the discourse, in this chapter, we introduced some concepts related to constraint programming and constraint solvers. Next, we chose *Comet* as the constraint solver to be used in this thesis.

Chapter 4

PATH FEASIBILITY ANALYSIS

Representation is the essence of programming.
Fred Brooks, 1995.

This chapter deals with path analysis, mainly, the algorithms and results of path validation. The execution time of a given program depends on the actual program trace (or program path) that is executed. Determining a set of program paths to be considered which can give WCET is a core component of any analysis technique for WCET. In the context of our application, this path is returned by the MC as a witness path for the computed WCET. This means that the WCET is the execution time of this path in the given hardware model. So if the path is infeasible the corresponding WCET is subjected to change. In practice, we may get tighter WCET value, which is of importance in the context of real time embedded system. We can determine the feasibility of a path through static analysis. Static analysis is a technique that is performed without actually executing the path taking into consideration all the inputs of the program from where the path is extracted. In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. In our case we perform analysis on the assembly code.

The input path to our system is a witness trace returned by the MC. We symbolically execute this path and create a set of constraints on the program's variables. Then a constraint solver (in our case *Comet*) is used to solve these constraints. A solution to the set of constraints is test data that will drive execution down the given path. If it can be determined that the set of constraints is inconsistent, then the given path is shown to be non executable. The technique that will be described here also aids in path validation. We have also developed a path analysis tool which produces the result of path feasibility (whether feasible or not), given a path. Here we use the terms feasible, valid, consistent and executable indistinctly. Finally, we applied our technique to validate some paths derived while computing WCET using method proposed in [3] on WCET benchmark programs [33] and present some results.

This tool has the following capabilities:

1. Classifies the given path as feasible or infeasible. Not all program paths are feasible and, therefore, classification of feasible and infeasible paths is of value in analyzing programs.This helps us refine the computed WCET in order to get the tighter result which is of prime importance.

2. Generates test data to drive execution down a program path.

3. As the tool operates directly on binaries (Assembly Code), it is able to analyze even proprietary software.

4.1 Some assumption about the program (path)

It is assumed that the program from which a path is extracted is completely stored in the memory. So all the instructions in a given path have some memory address (hexadecimal or decimal number) associated with them. We assume that the references to *stack* is via specialized register *sp* only and references to memory cells do not depend on the input variable. If some register or memory position is never assigned any value in the program then it is considered as an input value. Moreover we assume that the path does not contain any loop i.e. the path is finite. An example of a path taken from famous *binary search (bs-O2)* program is shown below. The first column represents the memory address and the second column after ':' is the operator of assembly language also known as instruction (e.g., *add, mov* etc.) and the columns that follow are the operands to the corresponding operator. Please refer to 5.2 for the formal grammar of the subset of assembly language we handled. It should be noted that a path does not have any procedure declaration but only the sequence of assembly instructions which may come from multiple procedures/functions.

```
13648 : add r3,r3,#3
13652 : mov r2,#0
13656 : add r1,r3,r2
13660 : asr r1,r1,#1
13664 : ldr ip,[r0,r1 lsl #3]
13668 : cmps ip,#0
13672 : subeq r3,r2,#1
13676 : beq 0003578 13680
13680 : subgt r3,r1,#1
13684 : addle r2,r1,#1
13688 : cmps r2,r3
13692 : ble 0003558 13656
13656 : add r1,r3,r2
13660 : asr r1,r1,#1
13664 : ldr ip,[r0,r1 lsl #3]
13668 : cmps ip,#0
13672 : subeq r3,r2,#1
13676 : beq 0003578 13680
13680 : subgt r3,r1,#1
13684 : addle r2,r1,#1
13688 : cmps r2,r3
```

4.2 Path Based Analysis

Path based analysis consists of analyzing just one path at a time which is conceptually simple and often simpler to implement. Now, it is important to know how this path is retrieved from MC in our case. WCETs are reported in [34] along with .xml file and .q file for some of the WCET benchmark programs from Mälardalen University [33]. This .xml file is the uppaal model i.e., program plus hardware model for model checker UPPAAL, the query .q contains the reachability property formulated in timed computation tree logic (TCTL), that is, "Is it possible to reach a final state with $GBL\_CLK \geq K + 1$?" where GBL_CLK is the global

clock which is reset at the beginning of each verification process and K is some integer value. When this property is false and was true for K, in this case K is the computed WCET. We can simply load this .xml file in MC UPPAAL and run this query over it. The MC returns a witness trace corresponding to the computed WCET K and we can save the trace in a file. This is a huge file with sequence of program instructions along with cache and pipeline modeling. The file can be parsed to obtain solely the trace corresponding to the program. Then this trace can be validated with our technique.

Our main contribution to the state of the art is to deal with path feasibility for low level language like ARM assembly which is of great importance for embedded systems as they execute low level code and to generate test data if the path is feasible.

4.3 Extracting Path Constraints

There is no algorithm that can tell us whether an arbitrary statement is reachable. Nor is there an algorithm for deciding the feasibility of program paths [35, 36]. However, if some restrictions are put on the programs, the problem becomes decidable. In the present work, we assume that numeric expressions involves finite domain integers. Similar to some existing works [36, 37, 38], we generate test data in two steps: firstly extract a set of constraints from a given path, and then solve the constraints. Given a path, we can obtain a set of constraints called path predicates or path constraints. The path is executable if and only if these constraints are satisfiable. Basically, there are two different ways of extracting path constraints [36] :

1. Forward expansion: It starts from the first instruction, and builds symbolic expressions as each statement in the path is interpreted.

2. Backward substitution: It starts from the final instruction and proceeds to the first, while keeping a set of constraints on the input variables.

In this thesis, we focus on the *forward expansion* method. This method is chosen because of the implementation points of views, equally *backward substitution* method can be chosen. It is not obvious to derive path constraints from assembly code, as a constraint has to be derived from one or more instructions. The path constraints are derived taking into account the semantics of each assembly instruction. Further difficulties arise from the fact that assembly code involves registers stack, memory etc. and we need to model them properly in order to reflect their characteristics.

4.3.1 Modeling Register, Stack and Memory

The ARM microprocessor has 16 general-purpose registers: $r0 - r15$. Some aliases are used for certain registers e.g., $r13$ is also referred to as SP, the stack pointer. $r14$ is also referred to as LR, the link register. $r15$ is also referred to as PC, the program counter. The details about the registers and their special purpose can be found in ARM Architecture Reference Manual [39]. We consider that the variables corresponding to registers, stack and memory assume values from a finite domain (FD).

In sequel, we model each register as an array of integer variable which represents its evolution along the path. In *Comet*, we declare this in the following way:

```
1 int maxValue =2^31-1;
2 int minValue =-2^31;
3 range Values = minValue..maxValue;
4 int max_assignment= 10;
5 Solver<CP> cp();
6 var<CP>{int} r0[0..max_assignment](cp, Values);
```

The 3^{rd} line in the above code defines finite domain for variables in which we solve our path constraints. The 4^{th} line declares the number of assignments that a certain register can take i.e. evolution history of this register, which is 10 in this case. Then the last line declares a vector $r0$ which take values from integer interval domain *Values* and its index ranges over $0..max\_assignment$. *CP* is the solver we use to solve the constraints.

Similarly, we model stack and memory as matrices of integer and in sequel, they are represented by S and D respectively. The first index represents stack or memory position and the second represents the evolution of certain position (stack or memory) along the path. In *Comet*, we declare this in the following way:

```
1 range fIndexS = 1024..2068; //declares the range of first index of S
2 range fIndexD = 6024..8068; //declares the range of first index of D
3 var<CP>{int} S[fIndexS, 0..max_assignment](cp, Values);
4 var<CP>{int} D[fIndexD, 0..max_assignment](cp, Values);
```

The 3^{rd} line declares the matrix S which assume values from finite domain *Values* and its first index ranges from $1024..2068$ and the second index from $0..max\_assignment$. Similarly, The 4^{th} line declares the matrix D which assume values from finite domain *Values* and its first index ranges from $6024..8068$ and the second index from $0..max\_assignment$. We distinguish between the memory and the stack while modeling, as this is the case in the assembly language also.

4.3.2 Maintaining Version for the Variables

A challenge of translating programs from procedural languages like assembly language to a constraint system in a declarative language like *Comet* is how to represent procedural language's variables, which have state, with declarative language's variables that are stateless. For example, the statement $add\, r1, r1, \#1$ is a valid assignment statement in Assembly that changes the state of the variable $r1$. The semantics of this statement is that it takes the current value of $r1$, adds 1 to it, and assigns the sum back to the same variable $r1$. In procedural languages like Assembly, no matter what a variable contains, it is possible to assign a new value to it . But the same statement $add\, r1, r1, \#1$ will never hold in *Comet*. This is because *Comet* considers both occurrences of $r1$ to have the same value so it will never succeed in finding any such value that can be added to one and still remains the same! One way to solve this problem in declarative language is to replace the statement $add\, r1, r1, \#1$ with another statement $add\, r2, r1, \#1$, and using the variable $r2$ in the place of other subsequent occurrences of $r1$.

While implementing a parser, we have taken into account the concept of introducing versions for variables. As the parser reads the instructions along the path, it generates constraints in static single assignment (SSA) form. SSA is a property of an intermediate representation (IR), which says that each variable is assigned exactly once. Existing variables in the original IR are split into versions, new variables typically indicated by the original name with a subscript,

so that every definition gets its own version. The concept of maintaining versions for variables works like this: every time a parser reads a new variable, it instantiates the version of the variable to its initial value 0. The version of a variable will be updated every time the parser comes across an assignment statement where some expression is assigned to this variable. After an assignment the current version of the variable will be the new version. The name of the variable in the constraint system the parser generates will be the name that the parser reads from the program qualified by the current value of its version at the end. Assume the parser has just read variables $r0, r1$ and $r2$. At this moment, the current version of these three variables is 0 by definition. The assignment statement $add\, r2, r1, r0$ will be represented as $r2[1] = r1[0] + r0[0]$ or the assignment statement $add\, r1, r1, \#1$ that we considered above will be represented as $r1[1] = r1[0] + 1$ in the resulting constraint system which clearly solves the problem that has been discussed above. Since the assignment statements have updated the current versions of $r1$ and $r2$ to 1, another assignment statement $add\, r2, r1, r0$ will be represented as $r2[2] = r1[1] + r0[0]$ in the constraint system. Here $r2$ has been assigned twice which causes its current version to be 2, $r1$ has been assigned only once which causes its current version to be 1, and $r0$ was not assigned at all which causes its current version to remain 0. In a clear picture, this can be seen as following:

```
add r2, r1, r0
add r1, r1, 1
add r2, r1, r0
```

In the constraints system, this looks like:

```
r2[1] = r1[0]+ r0[0]
r1[1] = r1[0]+1
r2[2] = r1[1]+r0[0]
```

However, introducing versions for variables may cause some confusion due to inconsistent update of variable versions which is caused by the presence of conditional statements along the path. For example ARM assembly language consists of instructions like $moveq, addeq$ whose semantics is that if the result of last comparison is equal then perform mov or add operation otherwise these operations will not be executed. Let us consider the following sequence of instructions:

```
cmp r0, r1
addgt r0, r0, #1
addle r1, r1, #1 // if(r0>r1){r0=r0+1} else {r1=r1+1}
add r2, r0, r1 // r2 = r0+r1
```

Lets try to give some idea of transformation without any particular syntax of *Comet*. The condition $r0 > r1$ will be represented as $r0[0] > r1[0]$, the assignments $r0 = r0 + 1$ as $r0[1] = r0[0] + 1$ and $r1 = r1 + 1$ as $r1[1] = r1[0] + 1$. But how to represent $r2 = r0 + r1$? If we simply put $r2[1] = r0[1] + r1[1]$ after *if..else* statement, this is wrong since either $r0[1]$ or $r1[1]$ will be invalid depending on the evaluation of the condition. In order to match changes of versions in different branches of a conditional statement, the easiest way is to add a matching assignment statement in the other branch. In our example, adding $r1[1] = r1[0]$ in the *if* block and $r0[1] = r0[0]$ in the *else* block guarantees that no matter what the condition is, it is safe to represent the assignment $r2 = r0 + r1$ as $r2[1] = r0[1] + r1[1]$ in the constraints system being generated. This is true because if the condition is true variable $r0$ will be assigned a new value and its version will be updated like before but what is new here is the added assignment $r1[1] = r1[0]$ which will cause the update in the version of the variable $r1$ parallel to that

of $r1[1] = r1[0] + 1$ in the *else* block. Similarly, if the condition is false, the assignment $r0[1] = r0[0]$ in the *else* block will do the matching of the version of variable $r0$ with that of the assignment $r0[1] = r0[0] + 1$ in the *if* block. It may be the case that the *else* block may not be present. The following example will illustrate this case:

```
cmp r0, r1
addgt r0, r0, #1 // if(r0>r1){r0=r0+1}
add r2, r0, r1 // r2 = r0+r1
```

The safe way to transform this is as shown in the following figure.

```
if(r0[0]>r1[0]){
  r0[1] = r0[0] + 1;
}else{
   r0[1] = r0[0] ;
}
r2[1]=r0[1] + r1[0];
```

In doing so, the version of $r0$ is correctly updated irrespective of whether the *if* condition holds or not, so at the end we always get the right assignment for $r2$.

4.3.3 Updating Arrays

Transforming assembly instruction to constraint system is straight forward if the instruction does not involve array manipulation that is manifested by str instruction. In this case, the data have to be written in the memory or in the stack. In order to model correctly the whole stack or memory has to be copied and only the specific position will receive the new value. The reason behind this is the representational difference between Assembly and Constraint system. Let's clarify this issue with some example. Let's assume that the current versions of all variables are 0 and the *update_history* for the array is also 0, which means that the array has not been updated so far.

```
1. str r0, [sp,#12]
2. str r1, [r0]
```

The semantics of these instruction are: the first one stores the value of $r0$ in the stack S (recall our assumption about the program path that the access to the stack is only through the special register *sp*) at position $sp + 12$ whereas the second one stores the value of $r1$ in the memory D at position $r0$. At this point let's recall the declaration of S and D.

```
var<CP>{int} S[fIndexS, 0..max_assignment](cp, Values);
var<CP>{int} D[fIndexD, 0..max_assignment](cp, Values);
```

It has to be guaranteed that the positions accessed or written for S and D must be within their range. The transformation of above lines of code in the *pseudo language* looks like following:

```
1. S[(sp[0]+12),1] =r0[0];
   forall(i in fIndexS: i != (sp[0]+12)){
       S[i,1]=S[i, 0];
   }
2. D[(r0[0],1] =r1[0];
   forall(i in fIndexD: i != (r0[0])){
       D[i,1]=D[i, 0];
   }
```

The instructions are translated into sequence of constraints. The first line stores the value of $r0$ at $sp + 12$ of S and the following $forall$ loop copies the old array to the new one as it was except for the position which was updated. Same reasoning applies for the update of D. We can see that constraint variables ($sp[0], r0[0]$) appear as the index of the array, in the literature, this is known as element constraint. In the same way, we can index a matrix with a row and column variable. It is worth to point out that writing in array is a costly operation as the whole array has to be copied. We can write a bit smarter code which looks elegant and is shown below.

```
1. forall(i in fIndexS){
       S[i,1]=(i == (sp[0]+12))*r0[0] + (i != (sp[0]+12))*S[i, 0]
   }

2. forall(i in fIndexD){
       D[i,1] = (i == r0[0])*r1[0] + (i != r0[0])*D[i, 0]
   }
```

There are only two possibilities, whether i is equal to $sp + 12$ or not. So we have boolean expression and the flow is controlled by this. If the case is true, the array at this position is updated with the recent value. If not, the positions of the array will receive the old values. This is how the new copy of the array is generated.

4.3.4 Constraints Generation Algorithm

Now we are going to present an algorithm based on the *forward expansion* which generates the set of constraints from a given path. Let PC be the set of path constraints corresponding to the path P_a. As we know that a path is the sequence of assembly instructions, we denote by $P_a[i]$ the i^{th} instruction of the sequence. Similarly, len denotes the length of P_a i.e. the total number of instructions in the sequence. Then PC can be obtained in the following way:

Algorithm 1: *extractPathConstraints(P_a): set of Constraints*

$len \leftarrow Length(P_a)$
$i = 0$;
$PC = \{\}$;
while $i < len$ **do**
  if $P_a[i]$ is assignment instruction **then**
    $PC = PC \cup genAssignConsInSSAForm(P_a[i])$;
  else if $P_a[i]$ is cmp or tst instruction **then**
    $saveOperands(cmpOp1, cmpOp2)$;
  else if $P_a[i]$ is conditional instruction **then**
    $PC = PC \cup genCondConstraintsInSSAForm(P_a[i], cmpOp1, cmpOp2)$;
  else
    DO NOTHING
  end if
  i ++;
end while
return PC

The function *saveOperands(.,.)* saves the operands $cmpOp1$ and $cmpOp2$ of the comparison instructions (*cmp* or *tst*) which can be used later. The function *genAssignConsInSSAForm(.)* and *genCondConstraintsInSSAForm(.,.,.)* generates constraints in SSA form depending on the semantics of the instruction in question. The later one considers the result of the last compar-

ison instructions while generating constraints. It should be noted that the concept of version explained above is the same as SSA form. Let PS be the size of the path and SM be *max(size of stack, size of memory)* . This algorithm runs in polynomial time with respect to PS and the space needed is bounded by $|PS| * |SM|$. This space bound is because of the fact that in the worst case, for an instruction (e.g., str instruction) we need to generate SM number of constraints which occupies $|SM|$ space.

We assume that assignment instructions are all those that do not contain branching instructions like (b, bl, bx and $b\,\langle cond\rangle$) etc. where $\langle cond\rangle$ can be one of *le, gt, eq, cs, ne, cc* etc. Similarly, we classify instructions as conditional which have the form $b\ \langle cond\rangle$. For more details one can consult ARM Manual [39]. Let us consider the following path and run this algorithm to produce the set of constraints.

Example

```
1. 7932: mov r2, #80
2. 7936: add r1, ip, r2
3. 7940: ldr r4, [sp,#12]
4. 7944: cmp r4, r0
5. 7948: bne #7920
6. 7920: mov r3, #1
7. 7924: mov r2, r3
```

Figure 4.1: *An example of a path in assembly language*

Let's assume that the current versions of all the variables are 0. According to our assumption, instructions at 1, 2, 3, 6, 7 correspond to assignment instructions, line 4 to comparison instruction and line to 5 conditional instruction. Let's analyze this example in details. When the parser finds the first instruction (at line 1), it recognizes the type of instruction(assignment instruction) and its semantics(i.e. copy the value of the second operand to the first). Then it updates the version of register $r2$ to 1 and assigns 80 to it. After this the parser advances to the second instruction and discovers another assignment instruction and its semantics (i.e. add second operand with the third and put the result in the first). It updates the version of the first operand $r1$ to its next version which is 1 and assigns the sum of second operand and third operand to it. As the second and third operand's value is not modified according to the semantics of the add instruction, their current version is used, which is 0 for ip and 1 for $r2$. Then, the next instruction that the parser finds is ldr, which may access the stack or the data memory. According to our convention, the access to the stack is only through the register sp. The parser finds this information and accesses the stack with the interpreted index that is $sp[0] + 12$ and loads this value in $r4$, updating the version of $r4$ to 1. Now the parser finds the comparison instruction cmp, in this case the parser saves its operands to some local variables of the program. So far we have seen that the constraints generation is straight forward for assignment instruction however it is not the case for conditional instruction because it has to consider the operands of the previous comparison instruction plus the jumping address and the memory address of the instruction that follows. The jumping address is the one that comes as its first operand. If this is equal to the memory address of the instruction that follows it, then the jumping took place. As in this example, the jumping address 7920 followed by bne for the instruction at line 5 is equal to the memory address of the following instruction at line 6. In this case there is a jump. So the constraint is generated taking into account the semantics of bne, i.e., the last saved operands

are different from each other, $r4[1]! = r0[0]$. Suppose the instruction that follows line 5 has a memory address which is $currentaddress + 4$, then there is no jump but sequential execution of instructions (this is because each instructions are of 4 bytes in ARM Assembly). Then, in this case we need to negate the constraint generated when there was a jump, $!(r4[1]! = r0[0])$ or equivalently $(r4[1] == r0[0])$. The same logic of assignment instruction can be applied to the instruction at line 6 and 7. The set of constraints generated for the path in figure 4.1 is shown below.

```
{
 1. r2[1] = 80 ,
 2. r1[1] = ip[0]+ r2[1] ,
 3. r4[1] = S[sp[0]+12,0],
 4. r4[1] != r0[0],
 5. r3[1] = 1 ,
 6. r2[2] = r3[1]
}
```

Figure 4.2: *The system of constraints for the path in figure 4.1*

4.4 Constraint Solving

After obtaining the path constraints, we have a CSP. Now solving this CSP would tell us whether the path is feasible or not. CSP represents a very general kind of problem. The choice of the constraints solver depends on the type of constraints that we generate. If the constraints are linear we can use Linear solver, if they are boolean we can use boolean solver and non linear solver in case they are non linear. In our case, the system of constraints can be non-linear if the assembly instruction contains MUL (multiplication), or DIV (division) over integers. In order to use the solver, we need to generate constraints in line with its syntax. The above path constraints can be written in $Comet$ in the following way:

```
cp.post(r2[1] = 80);
cp.post(r1[1] = ip[0]+ r2[1]);
cp.post(r4[1] = S[sp[0]+12,0]);
cp.post(r4[1] != r0[0]);
cp.post(r3[1] = 1);
cp.post(r2[2] = r3[1]);
```

It should be noted that the translation is straight forward. These constraints can be posted in the *solve* $\{...\}$ or *subject to* $\{...\}$ with the method $post$ on the solver cp.

4.5 Experiment and Results

We have studied many path examples from [34]. In most cases, the feasibility of paths can be decided very quickly. Some experimental results are summarized in the table 4.1. In this table, $Benchmark$ is the program from where the path is extracted,$Description$ gives information about the program, PL: the length of the path, $Feasibility$: the result of feasibility analysis i.e. $feasible$ if it is feasible, $infeasible$ otherwise, CGT and CST denotes the constraint generation time and constraint solving time respectively. For our experiment, we use a computer which has Intel Core 2 Duo CPU with 2.20GHz speed and 2048 MB memory with Ubuntu 11.10 system, and measure the timings in milliseconds.

Table 4.1: *Path Feasibility Results*

SN	Benchmark	Description	PL	Feasibility	CGT(ms)	CST(ms)
1	**fib-O0**	Simple iterative Fibonacci calculation, used to calculate fib(300)	4591	**feasible**	548	13420
2	**fib-O1**		1853	**feasible**	418	32
3	**fib-O2**		803	**feasible**	201	20
4	**bs-O0**	Binary search for the array of 800 integer elements(array not given)	344	**infeasible**	143	144
5	**bs-O1**		192	**feasible**	138	11304
6	**bs-O2**		145	**feasible**	134	8508
7	**insertsort-O0**	Insertion sort on a reversed array of size 11	1708	**infeasible**	396	164
8	**insertsort-O1**		805	**infeasible**	248	92
9	**insertsort-O2**		729	**feasible**	164	2888

Note: In table 4.1, file-Ox indicates that the file was compiled using gcc -Ox. (optimization option).

Comments on the result

1. Almost 33% of program paths are infeasible while 66% are feasible.
2. While computing WCET, the MC explores the paths in a program irrespective of their feasibility and the longest path is chosen in terms of the time this path takes to be executed in the abstract model of the hardware. When a program has several paths, the MC has to make many choices (explore branches) and there are more changes of making wrong choices before concluding a worst case path i.e. choosing an infeasible path. This is manifested by $bs - Ox$ and $insertsort - Ox$.
3. CGT are proportional to the path length, i.e. if the path is longer then it takes more time to generate constraints. This is because our algorithm for constraint generation is polynomial in the length of the path.
4. CST varies according to the program. The very high value for $fib - O0$ is justified because the path consists of instructions which write multiple times to the stack. In this case, the whole stack has to be updated and the number of constraints generated also increases significantly which take more time to solve.

4.6 Partial Conclusion

In this chapter, we presented an algorithm for constraints generation for assembly code. We applied it to generate constraints for paths returned by MC while computing WCET on some benchmark programs from from Mälardalen University using method described in [3]. We also presented some results of solving these constraints which indicates whether a given path is valid or not. The results showed that approximately 33% of the paths are infeasible.

Chapter 5

DECOMPILATION OF ASSEMBLY PROGRAM

"No computer has ever been designed that is ever aware of what it's doing; but most of the time, we aren't either."
Marvin Minsky

This chapter describes the process of decompilation of assembly program. A decompiler is a program that reads a program written in a machine language (assembly language) – the source language and translates it into an equivalent program in a high-level language – the target language [40]. This is illustrated in figure 5.1.

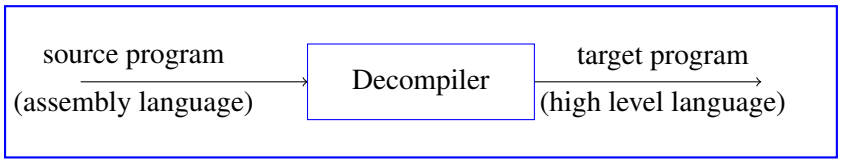

Figure 5.1: *A decompiler*

5.1 Source and Target Program

In our case, the source program is written in subset of ARM assembly language and the target program is the subset of standard C.

5.1.1 Subset of ARM Assembly Language

We build a parser for the source program in an incremental manner i.e., as we verify more benchmark programs more new instructions are added to the parser. Currently the parser can handle the following instructions with their standard syntax: *bx, stmdb, ldmia, mov, str, ldr, tst,bl, b, sub, add, mvn, asr, lsl*. Most of these instructions can be made conditional by attaching some conditional mnemonic at their end. The set of conditional mnemonics we handle are : *eq, le,gt, ne, lt, ge, cs, hi, ls, cc*. The conditional instructions are executed only when the condition is *true*. Please refer to [39] for the syntax and semantics of such instructions. A part of the formal grammar expressed in standard *BNF* notation is shown below. At the moment, we only

handle a single procedure (function) i.e., a program for us is a single procedure which can be recursive but does not call another procedure.

```
<program> ::= <procedureDeclaration> :
              <assemblyInstruction>
              (<EOL> <assemblyInstruction>)* //<EOL> is the end of line

<procedureDeclaration> ::= <INTEGER_NUMBER>(<IDENTIFIER>)
    //<IDENTIFIER> is a valid identifier and <<INTEGER_NUMBER>> is a
    positive integer number

<assemblyInstruction> ::= <INTEGER_NUMBER> :
                        <oneOperandInstruction>
                        |
                        <twoOperandInstruction>
                        |
                        (
                          <addInstruction>
                          |
                          <subInstruction>
                          |
                          <ASRInstruction>
                         )


<oneOperandInstruction> ::= bx lr

<twoOperandInstruction> ::= <lslInstruction> | <movInstruction>
                          | <stmdbInstruction> | <strInstruction>
                          | <ldmiaInstruction> | <ldrInstruction>
                          | <cmpInstruction>
                          | <branchRelatedInstructions>
                           ⋮
⋮
```

Figure 5.2: Formal Grammar of Assembly Language handled

5.1.2 Subset of C Language

The language C is standard. So instead of presenting a formal grammar for it, we would like to mention some restrictions we put in it. The language we generate have the following property.

1. The program is completely structured i.e. there are no conditional jumps, there are no exit from loop bodies (there are no 'break' or 'return' statements in loop bodies)
2. It does not contain 'switch' statements.
3. It has no library calls or external function calls.
4. Only Data Types are integers and single/multi dimensional array of Integers.

These restrictions allows us easy transformation from C to CPM, which we deal in the next chapter.

5.2 Decompilation Phases

A decompiler, or reverse compiler, attempts to reverse the process of a compiler which translates a high-level language program into a binary or executable program. The structure of decompilers is based on the structure of compilers; similar principles and techniques are used to perform the analysis of programs. So the main phases of decompilation among others are:

- Syntax analysis: The syntax analysis, is the process of analyzing a text, made of a sequence of tokens (e.g., words), to determine its grammatical structure with respect to a given (more or less) formal grammar. These words can be represented in a parse tree.
- Control flow analysis: A control flow graph of each subroutine in the source program is also necessary for the decompiler to analyze the program. This representation is suited for determining the high-level control structures used in the program. The control flow analyzer phase attempts to structure the control flow graph of each subroutine of the program into a generic set of high-level language constructs. This generic set must contain control instructions available in most languages; such as looping and conditional transfers of control.
- Code generation: The final phase of the decompiler is the generation of target high-level language code, based on the control flow graph and assembly code. Variable names are selected for all local stack, argument, and register-variable identifiers. Control structures and intermediate instructions are translated into a high-level language statement.

Decompilation is well established technique and the detailed description of it can be found in [40]. Decompilation is not the purpose of this thesis rather it is just an intermediate step to get to the main purpose, so we will only cover few basic things here.
Next, we will present an example program taken from [34], which will be used to clarify different steps we follow in this chapter. This is the $Fibonacci$ program which calculates the *Fibonacci-number* 300. we are not interested in the actual result of the computation, but only in the time it takes to compute it.

```
/*
00003214 <fibO0>:
00:  sub sp, sp, #32
04:  str r0, [sp, #4]
08:  mov r3, #1
0c:  str r3, [sp, #16]
010: mov r3, #0
014: str r3, [sp, #20]
018: mov r3, #2
01c: str r3, [sp, #12]
020: b 050 <fib+0x50>
024: ldr r3, [sp, #16]
028: str r3, [sp, #24]
02c: ldr r2, [sp, #16]
030: ldr r3, [sp, #20]
034: add r3, r2, r3
038: str r3, [sp, #16]
03c: ldr r3, [sp, #24]
040: str r3, [sp, #20]
044: ldr r3, [sp, #12]
048: add r3, r3, #1
04c: str r3, [sp, #12]
050: ldr r2, [sp, #12]
054: ldr r3, [sp, #4]
058: cmp r2, r3
05c: ble 024 <fib+0x24>
060: ldr r3, [sp, #16]
064: str r3, [sp, #28]
068: ldr r3, [sp, #28]
06c: mov r0, r3
070: add sp, sp, #32
074: bx lr
*/
```

Figure 5.3: *An example of source program: fib-O0*

5.2.1 Reconstruct CFG from Assembly Code

CFG construction takes two steps [41]. The first step partitions the code into a set of *basic blocks* (BBs). The second step looks at the branches in the code and fills in the CFG's edges to represent the flow of control. Informally, the code in a BB has: one entry point, meaning no code within it is the destination of a jump instruction anywhere in the program; one exit point, meaning only the last instruction can cause the program to begin executing code in a different BB. Under these circumstances, whenever the first instruction in a BB is executed, the rest of the instructions are necessarily executed exactly once, in order. The code may be source code, assembly code or some other sequence of instructions. More formally:

Definition 5.1 *A sequence of instructions forms a BB if : the instruction in each position dominates, or always executes before, all those in later positions, and no other instruction executes between two instructions in the sequence.*

This definition is more general than the intuitive one in some ways. For example, it allows unconditional jumps to labels not targeted by other jumps. This definition embodies the properties that make BB easy to work with when constructing an algorithm. The blocks to which

control may transfer after reaching the end of a block are called that block's successors, while the blocks from which control may have come when entering a block are called that block's predecessors. The start of a BB may be reached from more than one location. BBs become the nodes in the CFG.

5.2.1.1 Partitioning assembly instructions into basic blocks

The basic outline of the algorithm that we follow to partition assembly instructions into basic blocks is as follows [42] :

The algorithm for generating basic blocks from a sequence of code is simple: the parser scans over the code, marking block boundaries, which are instructions which may either begin or end a block because they either transfer control or accept control from another point. Then, the sequence is simply "cut" at each of these points, and basic blocks remain.

Input : A sequence of assembly instructions.
Output: A list of basic blocks.

- Step 1. Identify the leaders in the code. Leaders are instructions which come under any of the following 3 categories :
  - The first instruction is a leader.
  - The target of a conditional or an unconditional goto/jump instruction is a leader.
  - The instruction that immediately follows a conditional or an unconditional jump instruction is a leader.
- Step 2. Starting from a leader, the set of all following instructions until and not including the next leader is the basic block corresponding to the starting leader. Thus every basic block has a leader. Instructions that end a basic block (also called trailers) include:
  - Unconditional and conditional branches, both direct and indirect
  - Returns to a calling procedure
  - The return instruction itself.

  Instructions which begin a new basic block include
  - Procedure and function entry points
  - Targets of jumps or branches
  - "Fall-through" instructions following some conditional branches

  Note that, because control can never pass through the end of a basic block, some block boundaries may have to be modified after finding the basic blocks. In particular, fall-through conditional branches must be changed to two-way branches, and function calls throwing exceptions must have unconditional jumps added after them. Doing these may require adding labels to the beginning of other blocks. Let's run this algorithm for the example presented in the figure 5.3. According to the step 1, the leaders are instructions at line: 2 (first instruction), 11 and 26 (instruction that immediately follows a branching instruction). According to the step 2, the trailers are instructions at line: 10 and 25 (branching instructions) and 31 (return instruction). The basic blocks supposed to be the sequence of instructions from line 2-10, 11-25 and 26-31. However, instruction at line 22 is a start to a basic block for being a target of a branching instruction. Therefore the

pre-assumed basic block 11-25 is divided into two basic blocks 11-21 and 22-25. So the final basic blocks are the sequence of instructions from line 2-10, 11-21, 22-25 and 26-31. This is shown clearly in the figure 5.4.

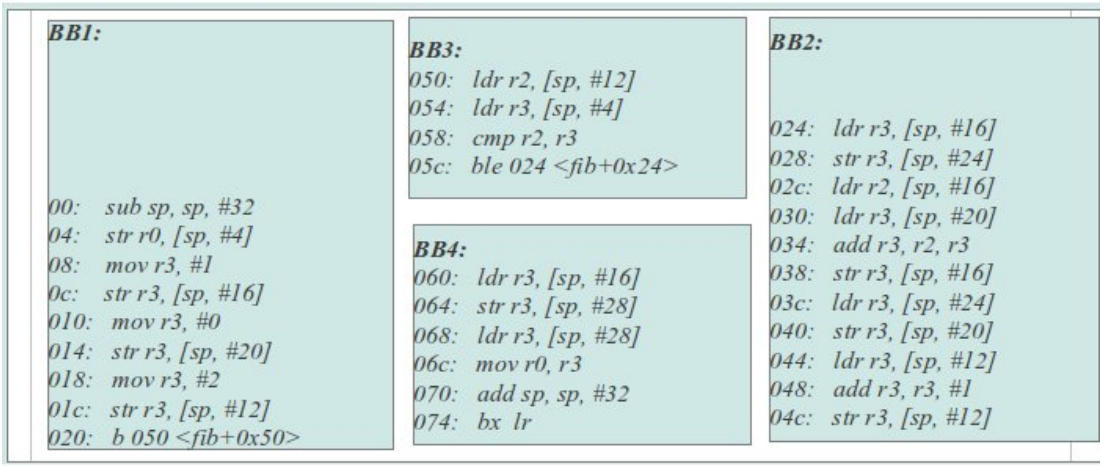

Figure 5.4: *Partition of program into basic blocks*

5.2.1.2 CFG from List of Basic Blocks

Once the assembly program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the flow graph are the basic blocks. There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B. There are two ways that such an edge could be justified:

- There is a conditional or unconditional jump from the end of B to the beginning of C.
- C immediately follows B in the original order of instructions, and B does not end in an unconditional jump.

We say that B is a predecessor of C, and C is a successor of B. Often we add two nodes, called the entry and exit, that do not correspond to executable instructions. There is an edge from the entry to the first executable node of the flow graph, that is, to the basic block that comes from the first instruction of the assembly code. There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program. If the final instruction of the program is not an unconditional jump , then the block containing the final instruction of the program is one predecessor of the exit, but so is any basic block that has a jump to code that is not part of the program.

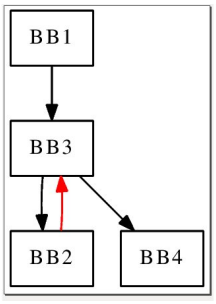

Figure 5.5: *CFG of fib-O0*

From the above BBs, we know that there is a direct jump from the last instruction of BB1 to the first instruction of BB3, so we create the corresponding edge from BB1 to the start of BB3. Similarly, a conditional branching from the last instruction of BB3 to the beginning of BB2 and BB4 would create two edges. And there is a edge going from end of BB2 to the beginning of BB3 as it is sequentially followed. The complete CFG can be seen in the figure 5.5. Additionally, we can add two nodes, called entry and exit nodes. In such case, there are edges from entry to BB1 and BB4 to exit node.

5.2.2 Loops

In oder to generate HLL, *loops* need to be identified in a CFG. We say that a set of nodes L in a flow graph is a loop if [42]

- There is a node in L called the loop entry with the property that no other node in L has a predecessor outside L. That is, every path from the entry of the entire flow graph to any node in L goes through the loop entry.
- Every node in L has a nonempty path, completely within L, to the entry of L.

We implemented the algorithm of Johnson presented in [43] to detect loops. In the figure 5.5, according to the above definition, BB2 and BB3 form a loop with BB2 as the loop entry.

5.2.3 HLL Code Generation

The code generator generates code for a predefined target high-level language. The following examples make use of the C language as target language. In fact, we can call it a pseudo language which has C like syntax. As in our case, we assume that all variables take integer values in a finite domain, we abstract away from the burden of declaration of the types of the variables also. It should be noted that we refer to it as C, subset of C or pseudo language and all of them mean the same. The modeling of registers and memory are not the same as explained in the chapter 4. As we are transforming from procedural ($assembly$) to procedural (C) language, the language itself (C) maintains the states of its variables. So registers are simple int variable and memory is an *array of int* in C. So the transformation is rather simpler than in the previous chapter. The basic idea here is to generate codes in C based on the semantics of the assembly instruction. We do so by generating code for BBs and establishing flow control from CFG as described in [40].

5.2.3.1 Generating Code for a Basic Block

Consider the control flow graph of figure 5.5 after control flow analyses. For each basic block, the instructions in the basic block are mapped to an equivalent instruction of the target language. Transfer of control instructions (i.e., bl, bx, $b\,\langle cond\rangle$ etc.) where $\langle cond\rangle$ is ble,bge, bgt etc. are dependent on the structure of the graph (i.e. they belong to a loop or a conditional jump or be equivalent to a goto), and hence, code is generated for them according to the control flow information, described in the next subsection (subsection 5.2.3.2). This section illustrates with an example on how to generate code for all other instructions of a basic block. It should be noted that the code generation is similar to what has been explained in chapter 4 except for maintaining the versions for the variables. The figure 5.6 shows the code generated for the basic blocks of figure 5.4 except for the transfer of control instructions.

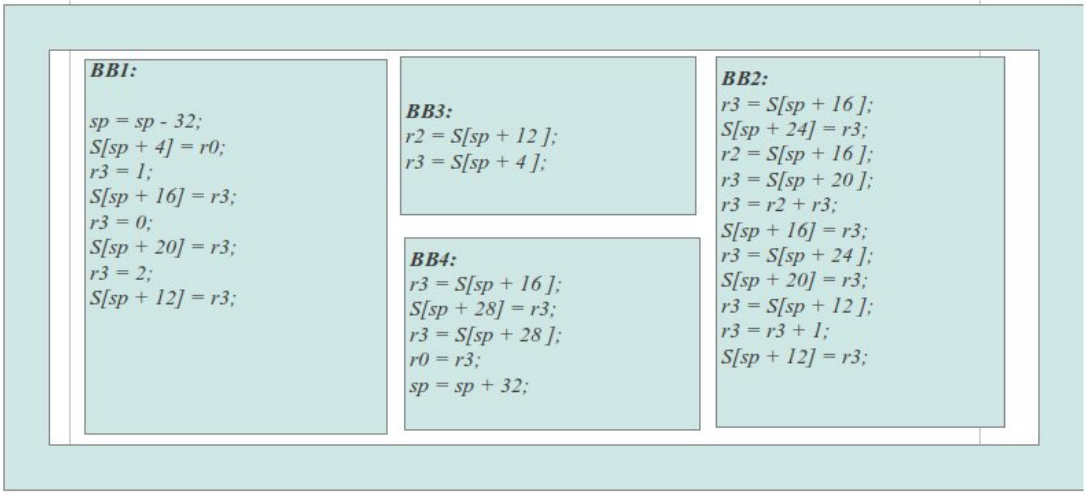

Figure 5.6: *Code generation for BBs except for transfer of control instructions*

5.2.3.2 Generating Code from Control Flow Graphs

The information collected during control flow analysis of the graph is used in code generation to determine the order in which code should be generated for the graph. Consider the graph in figure 5.5 with structuring information where *BB1* is the root node.

The generation of code from a graph can be viewed as the problem of generating code for the root node, recursing on the successor nodes that belong the structure rooted at the root node (if any), and continue code generation with the follow node of the structure. Recall that the follow node is the first node that is reached from a structure (i.e., the first node that is executed once the structure is finished). Follow nodes for loops conditionals are calculated during the control flow analysis phase. Other transfer of control nodes transfer control to the unique successor node; hence the follow is the successor, and termination nodes (i.e., bx) are leaves in the underlying depth-first search tree of the graph, and hence terminate the generation of code along that path. Figure 5.7 shows the complete code generated corresponding to the assembly code shown in the figure 5.3.

```
{
sp = sp - 32;
S[sp + 4] = r0;
r3 = 1;
S[sp + 16] = r3;
r3 = 0;
S[sp + 20] = r3;
r3 = 2;
S[sp + 12] = r3;

Loc0:
r2 = S[sp + 12 ];
r3 = S[sp + 4 ];
while(r2 <= r3)
{
r3 = S[sp + 16 ];
S[sp + 24] = r3;
r2 = S[sp + 16 ];
r3 = S[sp + 20 ];
r3 = r2 + r3;
S[sp + 16] = r3;
r3 = S[sp + 24 ];
S[sp + 20] = r3;
r3 = S[sp + 12 ];
r3 = r3 + 1;
S[sp + 12] = r3;

goto Loc0;
}
r3 = S[sp + 16 ];
S[sp + 28] = r3;
r3 = S[sp + 28 ];
r0 = r3;
sp = sp + 32;

}
```

Figure 5.7: *Complete HLL code for fib-O0*

When generating code for the loop body or the loop follow node, if the target node has already been traversed by the code generator, it means that the node has already been reached along another path, therefore, a goto label needs to be generated to transfer control to the target code. So is the case for *BB3* in the above example. The complete algorithm for code generation can be found in chapter 7 of [40]. It can be seen that this is a naive code generation. The BBs can be optimized before generating the code. Many techniques of BB optimizations are discussed in [42]. But the techniques that we employed for the BB optimization are elimination of extraneous loads and stores instructions and the algorithm for this is presented in [44], register's name elimination (only in some places). As we can see, the HLL code still has statements like *goto* which need to be removed. For our prototype tool, we removed this manually. However techniques for this are explained in [40]. So after possible optimization (automatic or manual) we obtained final code as shown in the figure 5.8. This code will be used in the next phase to be translated to CPM.

```
fib-O0(a)
{
  s04 = a ;
  s16 = 1;
  s20 = 0;
  s12 = 2;
  while(s12 <= s04){
    s24 = s16;
    s16 = s16 + s20;
    s20 = s24;
    s12 = s12 + 1;
  }
  a = s16;
}
```

Figure 5.8: *Complete HLL optimized code for fib-O0*

At the end, we would like to copy this note from Cifuentes ([40]). It is hard to capture the semantics of the program and that decompilation is economically impractical, but it could aid in the transportation process.

This project made use of known technology to develop a decompiler of assembler programs. No new concepts were introduced by this research, but it raised the point that de-compilation is to be used as a tool to aid in the solution of a problem, but not as tool that will give all solutions to the problem, given that a 100% correct decompiler cannot be built.

5.3 Mapping between Assembly Language and HLL

Decompilation is an intermediate step towards computing WCET as shown in the toolchain 7.1. As the time for BB can be determined precisely at assembly level, we take note of it once the basic blocks are formed. Timing for each instruction can be found in ARM reference manual [1]. Regarding the timings, the manual states that *All accesses are from cached regions of memory. If an instruction causes an external access, either when prefetching instructions or when accessing data, the instruction takes more cycles to complete execution*. However in our case, we always assume data cache misses i.e., the address to read data from or write data to are never in cache and the penalty for this is 34 cycles. This number is obtained from some expert in the field. But for instructions, we assume that there is no instruction cache miss. Without cache analysis, the safe thing to assume is that there is always a cache miss while computing WCET. As we do not pretend to compute precise WCET and instruction cache miss applies to all instructions unlike data, we do not take this into consideration. This affects the approximate WCET we compute by $instruction\_cache\_miss\_penalty * number\_of\_instructions$ cycles. According to this, we have computed the following timings for the basic blocks of figure 5.4.

$$time\_BB1 = 169;$$

$$time\_BB2 = 351;$$

$$time\_BB3 = 89;$$

[1] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0091a/BEIEDGJJ.html

$$time\_BB4 = 132;$$

These timings are used in CPM to compute an approximate WCET. We also preserve a mapping between the assembly code and the high level code as we need this while generating worst case paths. This will be discussed further in the next chapters.

5.4 Partial Conclusion

This chapter presented some techniques and examples to generate high level language from assembly code. Even though decompilation is economically impractical, in many cases its use is justified (e.g., in this thesis). This is an intermediate step in computing WCET according to our approach.

Chapter 6

MODELING HLL WITH CONSTRAINTS

"If you optimize everything, you will always be unhappy."
Donald Knuth

In this chapter, we discuss with examples about the rewriting rules to convert a program in C into a constraint model. If we want to use constraints programming as a tool to optimize/measure certain behavior of source code(e.g., runtime) written in C like language(we refer it as C or its subset), there must be a way of transforming the original program into a system of constraints on which the actual optimization process can be done using constraint solvers. The success of the optimization process depends not only on how efficiently the solver can solve/optimize the constrains system but also on how accurately the original program can be transformed into the system of constraints. The accuracy of the transformation is very crucial because a significant difference between the original program and the corresponding system of constraints will make any judgment that has been made about the original program based on the system of constraints unacceptable. Therefore, the transformation should always keep the semantics and logic of the original program for the given behavior, and the only significant difference allowed is the representation used. In this study, the source program is written in C and its corresponding constraint system uses *Comet* syntax. The responsibility of keeping the semantics and logic of the source program in the resulting system of constraints falls on the parser which is the component doing the task of transforming the C program into a constraint programming (*Comet*). The parser reads a program written in C, transforms each of the C structures it gets on the way into *Comet* structures, and generates a system of constraints equivalent to the original program it has read. This needs the parser to be capable of reading, recognizing and parsing all types of C structures. However, since the focus of this work is not to build a complete parser to convert C programs to their equivalent constraint systems, this parser handles programs written is some subset of C with characteristics as mentioned in section 5.1.2. The parser reads structures from the subset of C , and generates the corresponding constraints in *Comet*. The issue of transforming C structures from the source program into these constraints will be discussed in detail in section 6.1 where the equivalent constraints generated by the parser for each and every structure in the C subsets are shown. This type of conversion for program verification was also discussed by Tewodros in his Master Thesis [25].

6.1 Transformation of HLL to CPM

Constraint solvers are used as optimization tools in our work, a parser is required that can generate a semantically equivalent system of constraints for the input program. The system of constraints should be modeled in such a way that the constraint solver can solve it as efficiently as possible. The input is the program obtained in the previous chapter. In this section, we discuss with examples about the rewriting rules to convert a program in C (program in short) into a CPM. The parser generates a semantically equivalent system of constraints for the input program. It works in two steps. In the first step, it parses HLL to obtain information about the variables and their nesting so that we can declare them correctly in the constraint model. It also collects information about the conditionals, loop's condition etc. which leads to new variables. In the second step, it writes to the output file, the declaration of library to import, the specification of the solver we are going to use, constants that will be used in the transformation along with the variables. Then it generates the corresponding constraints and writes to the file. The resulting file is modified manually to supply the optimization function and the execution times for the basic blocks. In some cases, the labeling function is also modified manually to gain performance.

Now we look into the different constructs of the program and their transformation into the constraint model. Basically, we deal with assignments, conditionals and loops. As an example consider the program from figure 5.8. We will use this example as far as it covers different aspects of transformation.

6.1.1 Rewriting Rules for different kinds of instructions

The structural transformation from HLL to CPM is easy as the control structure in both the languages have similar constructs. The problem arises when we need to maintain different states in CPM for a HLL variable and this makes it difficult and tricky and sometimes tedious. The input program is divided into code blocks also known in the literature as basic blocks, each block being a sequence of instructions which are executed once. In our example program, there are four blocks as explained in chapter 5. And each such block has a context corresponding to its nesting in *conditional* (if-then-else) and *loop* (for, while and do) statements.

6.1.1.1 Declaration

We have compiled the following working rules for the declaration of variables in CPM. The study about variables declaration is done in the first phase of parsing.

1. Any variable v in the program should be mapped into an array variable $v[< ctx >]$ where
   - $< ctx >$ is a tuple of indices corresponding to the deepest block where an $assignment$ to that variable (i.e. $v = Expr$) occurs, where $Expr$ is some arithmetic or boolean expression. If a variable is only read in the program we just declare it as a simple constraint programming f.d. variable without states.

2. All $loops$ have ranges $1..n$ for some $n \in \mathbb{N}$ representing the maximum number of iterations of the loops, the variables within loop have corresponding ranges $0..n$, to model the state of the variables before the loop is executed. If a variable is assigned more than once outside of the loop then we can assign different name to this variable and use accordingly in the subsequent reading.

3. In CPM, a *boolean* variable is assigned to each block controlled by *if-else* statement whereas *boolean array* variable is assigned to each block in a *loop*. These *boolean* variables control the execution of these blocks in a given context.

Following the above rules, the declaration section of CPM looks as follows for our example program of figure 5.8. The explanation of each declaration is provided as comment and the compliance rule is mentioned.

```
1  //importing finite domain integer library as we are solving our
       problem in finite domain
2   import cotfd;
3  //defining the domain of vars
4
5  //maxInt is a constant which represent the maximum value of integer
       that we consider
6  int  maxInt = 2^32 -1;
7  //represents our finite domain
8  range NonNeg = 0..maxInt;
9
10 //maximum number of times loop is iterated(n)
11 int  maxIters = 300;
12
13 //counting iterations
14 range Iters = 1..maxIters; //range of loop blocks (rule #2)
15 range Iter0 = 0..maxIters; //range of var. in loop(range of states:
       rule #2 )
16
17 // Solver declaration, in our case is constraint programming solver
18 Solver<CP> cp();
19
20 //The following variable are associated with the solver and their
       value is in the range NonNeg.
21
22 //variables s12, s16, s20 are assigned within the while loop and
       outside of the loop,
23 //so their indices starts from 0 to maximum number of possible
       iterations:maxIters (rule #2 and rule #1)
24   var<CP>{int} s12[Iter0](cp,NonNeg);
25   var<CP>{int} s16[Iter0](cp,NonNeg);
26   var<CP>{int} s20[Iter0](cp,NonNeg);
27
28 //whereas variable s24 is assigned only within the loop, so its
       indices are in the range Iters
29   var<CP>{int} s24[Iters](cp,NonNeg);
30
31 //var a is assigned once and is assigned outside of the loop, but
       it was read before it was assigned(this may mean that it was
       assigned somewhere before)
32 // so we need to maintain two states for this variable. (rule #1)
33   var<CP>{int} a[1..2](cp,NonNeg);
34 //var s04 is assigned only once and it is assigned outside of the
       loop, so we do not need to maintain state for this. (rule #1)
35   var<CP>{int} s04(cp,NonNeg);
36 //The following two variables are related to the loop
37 //nits collects the number of effective iterations(iterations
       actually taken place according to CPM)
```

```
38  var<CP>{int} nits(cp,NonNeg);
39 //wh1 is an array variable which encodes the condition of the while
       statement and controls the block within while (rule #3)
40  var<CP>{bool} wh1[Iter0](cp);
```

6.1.1.2 Assignments

Assignments are the most common type of instructions that appear in programming and assignment changes the state of a variable in question.

1. Every variable *assignment*: $v = Expr$ in a program is rewritten into a conditional constraint in CPM and is posted, that is,
   $cp.post(control \implies v[< ctx(v) >] == contextedExpression)$
   where
   - $control$ is the *boolean(array)* variable controlling the block where this assignment occurs. If the assignment occurs outside of the control structure(loop or conditional), $control$ is replaced by $TRUE$.
   - $< ctx(v) >$ is the tuple of indices corresponding to the nesting of the block and
   - *contextedExpression* is the expression of the right hand side, where the program variables are replaced by the corresponding array variables of CPM.

We consider the following instructions to show the use of above rules. For variable assignments that occur inside the loop, we present examples later.

```
1  s04 = a ;
```

Its representation in CPM:

```
1 //rule #1
2 //here the contol is always true, so we do not need to put it
3 //a[1] is the contexted expression(adequate state of a), as s04 is
      assigned only once, we do not need its context
4  cp.post(s04 == a[1]) ;
```

6.1.1.3 Loop statements

We need to be careful while transforming loop as it involves many details mentioned below.

1. For every $loop$ (be it for, while or do) in the program, a *forall loop* is created in CPM:
   forall(i in 1..n) ctxLoopBlock
   where
   - n is an upper bound on the number of loop iterations
   - $ctxLoopBlock$ is the conversion of loop block according to the conversion rules

2. For every such loop, a *boolean array* variable *loop-name*$[< ctx >, loop - index]$ is created to control the corresponding loop block
   where
   - *loop-name* is a new non-ambiguous name; and
   - $< ctx >$ is the tuple of indices corresponding to the nesting of the block where the loop appears; and
   -$loop - index$ is an index which maps into different loop iterations, with range $0..n$.

3. The 0^{th} instance of the *boolean array* variable should be posted to *True*. Taking into account the context where it occurs as well as the possibility that the block where it occurs is executed, the following two constraints are posted:
$cp.post(control => (loop - name[< ctx >, 0] == True))$
$cp.post(!control => (loop - name[< ctx >, 0] == False))$

4. The other instances of the *boolean array* variable depend on the type of the loop.

   (a) In *while* loop, all iterations are dealt in a same way, since not even the first iteration may execute, hence the following constraints are posted:
   $cp.post(loop - name[< ctx >, i - 1] => (loop - name[< ctx >, i] == iterCondition))$
   $cp.post(!loop-name[< ctx >, i-1] => (loop-name[< ctx >, i] == False))$
   where
   -$iterCondition$ is the loop condition of the program expressed with adequate array variables. These two points are semantically equivalent to
   $cp.post(loop-name[< ctx >, i] == (loop-name[< ctx >, i-1]$ && $iterCondition))$,
   which is done to guarantee that whenever the i^{th} iteration of the loop is false, all the future iterations are false too.

   (b) In *do..while* loop, the first iterations is always executed, and the constraint is modified depending on the iteration count i as shown below
   if $(i == 1)$
   $cp.post(loop-name[< ctx >, i-1] => (loop-name[< ctx >, i] == True)))$
   if $(i > 1)$
   $cp.post(loop - name[< ctx >, i - 1] => (loop - name[< ctx >, i] == iterCondition)))$
   but always
   $cp.post(!loop-name[< ctx >, i-1] => (loop-name[< ctx >, i] == False))$
   to guarantee that whenever the i^{th} iteration of the loop is false, all the future iterations are false too.

   (c) *for* loop can be dealt in many ways. One way is to convert it to *while* or *do..while* loop and deal with it as explained before. The following lines of codes shows how to convert *for* loop to *while* loop.

```
for(BEFORE_STATEMENT; FINISH_STATEMENT; ITERATE_STATEMENT)
{
   LOOP_CODE
}
```

```
BEFORE_STATEMENT
while(FINISH_STATEMENT)
{
   LOOP_CODE
   ITERATE_STATEMENT
}
```

Now, let's generate the constraints corresponding to *while* loop from our example 5.8.

```
1  //As this while block(the whole while structure) is free that is
       not under any control statement,we post the following constraint
2 //to indicate that its 0 instance is always true. rule #3,
3  cp.post((wh1[0] == true) );
4
5  forall(i in Iters) //rule #1
6   {
7    //wh1 coressponds to rule #2, and it controls this while loop
8    //current state of while is previous state of while together in
         conjunction with the condition itself, this is done to make
         sure that
9    //whenever the last iteration is false, all the future iterations
         are false
10    cp.post( wh1[i] == (wh1[i-1] && (s12[i-1] <= s04))); rule #4.a
11
12   //whenever the ith condition is true for the loop, all the
         statements within while are posted for this iteration.
13   //The following code explains the assignment rule #1 as the
         control here is wh1[i].
14    cp.post( wh1[i] =>
15              (
16                (s24[i] == s16[i-1])      &&
17                (s16[i] == s16[i-1] + s20[i-1]) &&
18                (s20[i] == s24[i])        &&
19                (s12[i] == s12[i-1] + 1)
20              )
21           );
22   }
23
24   //These four instructions can be posted seperately also as shown
         below.
25    cp.post( wh1[i] => (s24[i] == s16[i-1]) );
26    cp.post( wh1[i] => (s16[i] == s16[i-1] + s20[i-1]));
27    cp.post( wh1[i] => (s20[i] == s24[i]) );
28    cp.post( wh1[i] => (s12[i] == s12[i-1] + 1) );
```

6.1.1.4 Conditional Statement

In this, we consider $if..then$ and $if..then..else$. They are different in HLL but in CPM both of them take the same form in terms of the structure(not in terms of functionality).

1. For every *if instruction* in the program, a *boolean array* variable $if - name[< ctx >]$ is created to control the corresponding $then$ and $else$ blocks
   where
   - $if - name$ is a new non-ambiguous name, and
   - $< ctx >$ is the tuple of indices corresponding to the nesting of the block where the *if statement* appears.

2. The boolean $if - name$ is subjected to two conditional constraints, namely:
   $cp.post(control \implies (if - name[< ctx >] == ctxCondition))$ and
   $cp.post(!control => (if - name[< ctx >] == False))$;
   where
   - $control$ is the *boolean array* variable of the block where the *if statement* occurs;

- $ctxCondition$ is the *if condition* of the program expressed with the adequate array variables

(a) For every assignment appearing in the *else* block of the program a similar constraint is posted in CPM but conditioned by the negation of the *if-name* array variable $cp.post(!if-name[< ctx >] => ctxAssignement]$ where $ctxAssignement$ is the $assignment$ of the program expressed with the adequate array variables

(b) For every variable assigned either in the $then$ or in the $else$ block, a corresponding assignment must be considered in the other block but the value of variable in the new $context$ will be its value in the last $context$. If $else$ block does not exist, we should consider the $else$ block and a corresponding assignment must be considered as in the previous case.

We consider the following example to explain this situation. Details of conditional statement was explained in chapter 4.

```
{
  s16 = x; //x is the input parameter
  if(s16==11)
   {
     a = s16;
   }else{
     b = s16;
   }
}
```

The transformation of this into CPM looks as follows:

```
 //here ifcond is the boolean variable which controls this
    conditional statement.
 // This has to be declared first as it was explained in declaration
    section. rule #1

//rule #2, as the if condtion is not under control of any external
   condition, we can simply post single constraint below
 cp.post(ifcond == (s16[1] == 11));
 //if part
 //variable a = 16 is executed when the if condition is true, so
    whenever ifcond holds we need to post this constraint. But b
    gets updated when this condition is false.
 //so when ifcond holds we need to explicitely say that b does not
    change that means it takes its previous value. Similar logic
    applies to a in else block.
 cp.post(ifcond => (a[1] == s16[1])); // rule #2.a
 cp.post(ifcond => b[1] == b[0]); // rule #2.b

 //else part
 cp.post(!ifcond => (a[1] == a[0])); // rule #2.a
 cp.post(!ifcond => b[1] == s16[1]); // rule #2.a
```

Suppose now that we are just given the *if* block without *else* block. But in CPM we need to generate the else block also as shown below to make sure that the previous value for a is copied when the condition is false.

```
1
2  cp.post(ifcond == (s16[1] == 11));
3  //if part
4  cp.post(ifcond => (a[1] == s16[1])); // rule #2.a
5
6  //else part
7  cp.post(!ifcond => (a[1] == a[0])); // rule #2.a
```

6.1.1.5 Special case: Array assignment

Each array variable is indexed by a tuple of indices, an index for each loop condition in the deeper nested block in which the assignment occurs. When multiple assignments are made to the same variable within a block, an extra index should be used to distinguish the different states of the variable during the block execution. Lets consider the following hypothetical code in HLL with two nested loops, where D is an array and is assigned twice inside the loop:

```
1  r6 = 2;
2  while(r6 != 11)   //lets represent by outer, i is the iteration
       count
3      {
4       while(r4 != 0) //lets represent by inner, j is the iteration
            count
5          {
6           D[3] = 1;
7           r2 = D[3];
8           D[7] = 3;
9           r4 = (0 == r2);
10         }
11      r6 = r6 + 1;
12     }
```

Let's only consider the instructions where the array *D* is updated in the above example and leave the rest of the instructions for the moment. Remember that updating array is a costly process and was discussed in chapter 4. Let n and m be the maximum iteration count for outer while loop(represented as *outer*) and for inner while loop(represented as *inner*) respectively. Also consider that the indices of D ranges over *IndexRangeD* that is $0..p$ where p is some integer. Assume that the array D is assigned only inside both of the loops. In the resulting constraint system, the array variable D has 4 indices, i.e. one for the outer loop(i), one for inner loop(j), one to account for multiple assignments made within the block(1 for the first assignment and 2 for the second assignment), one for the array index(k which ranges over $IndexRangeD$).

The following list shows the codes corresponding to the update of array D within the block. The update occurs only when both the conditions controlling the block are true. According to the first instruction, the current state of D at index 3 will receive 1 and all other will have the value from the previous iteration. But if this is the first iteration then it takes the value of D before starting the loop. In case of the second instruction, the previous value for D will be from the first update.

```
forall(i in 1..n) //where n is the max number of iteration for the
   outer loop
{
  forall(j in 1..m) //where m is the max number of iteration for the
     inner loop
  {
    forall(k in IndexRangeD){// where IndexRangeD is the range in
       which indices of D lie.

      cp.post( (outer[i] && inner[i,j]) => D[i,j,1,k] == (k==3)*1 +
         (k!=3)*D[i-1,j-1,2,k]);
      cp.post( (outer[i] && inner[i,j]) => D[i,j,2,k] == (k==7)*3 +
         (k!=7)*D[i,j,1,k] );

    }
  }
}
```

6.1.1.6 Code Block

A code block is a sequence of one or more statements where each statement can be a *declaration*, an *assignment*, an *if..then..else* statement or a *while* loop. When the parser reads a generic code block S_1 ; S_2 ; S_3 , it transforms the code block into the sequence of statements C_1 , C_2 , C_3 where C_i refers to the constraint corresponding to the statement S_i of the original program.

6.1.1.7 Basic Block Timing, Optimization function and Search

So far we have discussed about transforming HLL into CPM. This step is also an intermediate step towards the computation of approximate WCET. In order to produce the CPM needed for computation of an approximate WCET, we need to augment this model with timings of the BBs in the program and pose optimization problem in this augmented model. The complete CPM of the example program *fib-O0* presented in the figure 5.8 is shown in appendix A. It should be noted below that maximizing clocks would maximize the number of loop iterations also which is manifested by the presence of $nits$ in the formulation of clock constraints(line 51). The search for loop's truth values(*wh1*) are made in $using$ block. The last block prints the result of computation and thus the approximate WCET.

6.1.2 Labeling

The concept of labeling was previously introduced in 3.4.3. The order of labeling variables makes difference during search in finite domain constraint solvers. The performance of these solvers is subjected to labeling algorithm designed. Search in *Comet* can be deterministic, which means that, at each node of the search tree, you have the possibility to control the subtrees under that node.

Consider the following pseudo code example of famous *binary search* program where *lowerIndex* and *upperIndex* represent the lower and upper indices of array *S* and *Element* is the item we are looking for.

```
while (lowerIndex <= upperIndex){ //while condition: while
          //compute midpoint

      if (S[midPoint]==Element) { //if condition: if1
          //Element found
      }
      else {
         if ( S[midPoint] > Element) { //if condition: if2
            //search in the lower part of the array
         }
        if ( S[midPoint] < Element) { //if condition: if3
            //search in the upper part of the array
         }
     }
}
```

In order to know the the truth values of *while* and *if* conditions, we need to label them so that the solver tries to instantiate their values. There are many different ways of doing this, some of them guide the search process while some others just add extra burden to the solver. One of the first attempt of labeling the above code is the following.

```
//label all iterations of while
label(while);
//label all iterations(iteration--because it is within while) of if1
label(if1);
//label all iterations(iteration--because it is within while) of if2
label(if2);
//label all iterations(iteration--because it is within while) of if3
label(if3);
```

If we label like this, the solver first instantiates *while* condition for each iterations, lets say with 1, upto maximum iterations. Then does the same with the rest of the *if* conditions. Labeling all of them at one time is costly and makes backtracking difficult. This can be seen as horizontal labeling. A second attempt is the following code.

```
forall(i in 0..30){
  //label first iterations of while, if1, if2, if3 and successively
  label(while[i]);
  label(if1[i]);
  label(if2[i]);
  label(if3[i]);
}
```

In the above case, the solver instantiates the first iterations of *while* condition and its nesting. This can be seen as vertical labeling. This solution is better than the first one because labeling is done step by step for each iteration, making search more efficient. However looking at the semantics of the above code fragment, we know that if *while* condition is false, then the rest of the code will not be executed. Similarly, *if2* or *if3* will be executed only if *if1* is false. These two attempts do not take this into consideration. This is an important thing to consider to make search efficient. A third attempt takes this into consideration and produce the following code for labeling. This avoids unnecessary labeling and performance was increased by several times in our experiment. We have opted this method in our tool implementation.

```
1    forall(i in 0..30)
2      //label while condition in the first step
3      try<cp> {
4        cp.post(while[i] == 1);
5        //label if conditions(if1, if2, if3) only if while is true
6        try<cp> {
7          cp.post(if1[i] == 1);
8        }
9        |
10       //label if2 and if3 only if while is true and if1 is false
11        {
12         cp.post(if1[i] == 0);
13         try<cp> {
14           cp.post(if2[i] == 1);
15         }
16         |
17         {
18           cp.post(if2[i] == 0);
19         }
20         try<cp> {
21           cp.post(if3[i] == 1);
22         }
23         |
24         {
25           cp.post(if3[i] == 0);
26         }
27       }
28   }
29   |
30   {
31     cp.post(while[i] == 0);
32   }
```

If the above fragment of code (6.1.2) is used for optimization(maximization or minimization) purpose, we can be bit smarter while choosing the right value from the finite domain that will help the solver to attain its maximum value faster. Let's assume that the maximum value of the objective function is in function of number of iterations of *while* condition i.e., executing *while* more times will increase the value of the objective function. So as to reach the maximum value faster we can try to post *while* with *true* first and then with *false*. If it is a minimization problem then post *while* with *false* first and then with *true*. It is to be noted that in *Comet* the order of labeling is also important. The result is different if *if1* is labeled before *while* and *while* is labeled before *if1*. This shows care should be taken while labeling variables.

Based on the above observation we can define some transformation rules for labeling the control structures(e.g., *if, while*) of a program. Given a program we can obtain all the control structures out of it. We can define a grammar in such a way that a *control* is either *if* or *while* or their sequence or their arbitrary nesting. Let $control < ctx >$ be the context of *control* w.r.t. its nesting in *loop* where $< ctx >$ represents the tuple of indices corresponding to its nesting. In the absence, $[< ctx >]$ can be eliminated.

```
control <-- if
       <-- while
       <-- control; control //a control comes sequentially after
           other
       <-- control1(control2) //control2 is nested inside control1,
           where control1, control2 are controls
       <-- if control1 then control2
```

6.1.3 Rules for Generating Labeling Function

In this subsection, we present labeling rules for different constructs of HLL like *if, while etc.*
case 1: *if*

```
   try<cp> {
             cp.post(if[<ctx>] == 1);
          }
          |
          {
            cp.post(if[<ctx>] == 0);
          }
```

case 2: *while*

```
 forall(i in 0..maxIters) //where maxIters is the maximum number of
    possible iterations for while
  try<cp> {
             cp.post(while[i,<ctx>] == 1);
          }
          |
          {
            cp.post(while[i,<ctx>] == 0);
          }
```

case 3: *control1; control2*

```
//control1
   try<cp> {
             cp.post(control1[<ctx>] == 1);
          }
          |
          {
            cp.post(control1[<ctx>] == 0);
          }
//control2
   try<cp> {
             cp.post(control2[<ctx>] == 1);
          }
          |
          {
            cp.post(control2[<ctx>] == 0);
          }
```

case 4: *control1(control2)*

```
//control1
   try<cp> {
            cp.post(control1[<ctx>] == 1);
           //control2
            try<cp> {
                    cp.post(control2[<ctx>] == 1);
                  }
                  |
                  {
                    cp.post(control2[<ctx>] == 0);
                  }
          }
          |
          {
            cp.post(control2[<ctx>] == 0);
          }
```

case 5: *if control1 then control2*
In order to simplify the reading of the rule, let's define a function called *label(control)* in the following way.

```
label(control[<ctx>]) == try<cp> {
                   cp.post(control[<ctx>] == 1);
                  }
                  |
                  {
                    cp.post(control[<ctx>] == 0);
                  }
```

Now the transformation for *if..then..else..* looks like following:

```
//control1
   try<cp> {
            cp.post(if[<ctx>] == 1);
            //control1
            label(control1[<ctx>]);
           }
          }
          |
          {
            cp.post(if[<ctx>] == 0);
           //control2
            label(control2[<ctx>]);
          }
```

6.2 Partial Conclusion

In this chapter, we presented some rewriting rules to transform HLL(C like) to CPM. In addition to this, we explained with examples the transformation of different constructs(e.g., assignments, conditionals, loops etc.) of HLL to CPM. This transformation is one of the fundamental

step in computing WCET according to our approach. We also discussed about efficient way of labeling CP variables.

Chapter 7

WCET COMPUTATION

"The computing field is always in need of new cliches."
Alan Perlis

In this chapter, we study an efficient way, based on IPET (described in 2.3.1), of determining the approximate worst case running time of a program in ARM architecture. Based on this approximate WCET, we also suggest some feasible paths which can lead to the worst case. The more precise WCET can be calculated using the method proposed by Cassez et al. ([3]) in these paths, taking into account the hardware characteristics(caches and pipelines) which bring tricky dependencies to WCET. Our approach complements the method proposed in [3] in the following ways:

1. Avoids the need for generating substantial exponential number of paths, and makes this method highly scalable.

2. As the paths we propose are feasible, the obtained WCET is tight which is not the case in [3].

3. Provides the over approximation of computed WCET so that the buyers of the tools get confidence in the product they are buying.

In order to achieve this, the road map we follow is to translate the assembly code into HLL and model HLL with CP. Using IPET based approach on CPM, we can compute the approximate WCET. Taking advantage of the CP solver like *Comet*, we can obtain the paths which gives WCET greater or equal to certain threshold value and supply these paths to the model checker. Finally, we apply our technique to WCET benchmark programs from Mälardalen University [1] slightly modified by Cassez et al. and present some results.

7.1 OVERVIEW OF THE METHOD AND TOOL CHAIN

As mentioned in the chapter 2, Computing the WCET of a program is usually a very hard problem. Our approach to deal with this problem consists of three steps:

1. We first convert assembly program to HLL. In doing so, we maintain the mapping between two different levels of code and the time(without considering the effects of the special hardware characteristics) for the basic blocks. This is described in chapter 5.

[1] Mälardalen WCET Research Group. WCET Project – Benchmarks. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

2. We model HLL with constraints to obtain the CPM of a program. This is described in chapter in 6.

3. We compute approximate WCET. For this we Optimize(maximize) a characteristic function(execution time of the program) on CPM and the maximum value corresponds to an approximate WCET.

Next, we discuss about this last step. Then, we provide some probable worst case paths which can lead to WCET. The overview of our tool chain is presented in the figure 7.1.

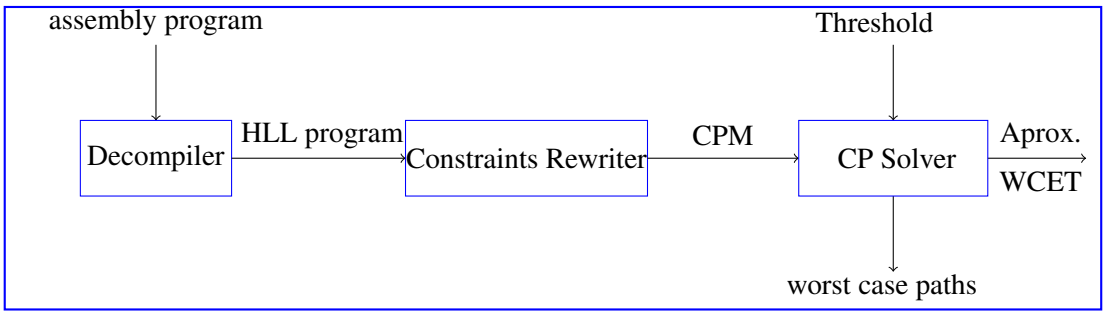

Figure 7.1: *Tool Chain Overview(Aprox. WCET Tool)*

In this, the rectangular box corresponds to a module and the arrow shows the the connections between different modules. The text on top of the arrow shows the input for the next module. The input to our decompiler is the program in assembly language, and it outputs the program in HLL which will be fed into the constraints rewriter module. This module produces CPM of the HLL program which becomes input to the next module called constraints solver. We augment CPM with the following CSOP and supply it to the CP solver(*Comet*).

$$\max \sum_{i=0}^{N} c(i) * t(i) \tag{7.1}$$

where
- N is the number of BBs
- $c(i)$ is the execution count of BB_i
- $t(i)$ is the runtime of BB_i

The solver solves this CPM and maximizes the expression 7.1. The maximum value corresponds to the approximate WCET. This is how the approximate WCET of a given assembly program is computed.

Based on this approximate WCET, we can supply a threshold(lesser or equal to WCET) and taking advantage of the constraints solver we can generate paths which can give WCET(maximum value of the characteristic function) greater or equal to the given threshold. This is done using *solveall* method of *Comet*. The code corresponding to this can be presented as follows(all other code remains same as in A.1 except for printing the paths which is done within the search). For this particular example, knowing the truth value of *while condition* is enough to derive the paths. So we only print this.

```
solveall<cp>{
 //nits is the number of times the loop is iterated effectively,
//as we infer loop bounds automatically, we can calculate this
 cp.post( nits == sum(i in Iters)(wh1[i]));
//timeX is the rough time for the basic block X.
 cp.post(clocks == timeBB1
                //for the first time BB3 is always executing
                + timeBB3
                +timeBB4
                + nits * (timeBB2+timeBB3)
         );
 //get all paths where clocks>=max_clock,
 //where max_clock is the threshold value we supply
 cp.post(clocks>=max_clock);
}

⋮

using {
 label(wh1);
 cout<<wh1<<endl;
}
```

Figure 7.2: *Code to obtain worst case paths*

From these truth values, a complete path can be derived as all other blocks will be executed once except the one within the loop. For example, if a loop is executed 10 times in the worst case, the block within the loop is repeated 10 times in a path in an appropriate place where loop occurs in a sequential program. However, the generated paths are high level paths as they were obtained from high level program. Since the transformation from HLL to CPM maintains all the logical structure of HLL(if, while etc.), and we have preserved a mapping between HLL and assembly program, we can translate these paths into low level paths. In our implementation, we have a module which transforms each high level paths to low level paths. These new set of paths will be inputs to Cassez et al. tool(see figure 7.3 for tool chain integration). The precise WCETs can be calculated using his method on these paths. The maximum of all these values corresponds to the WCET.

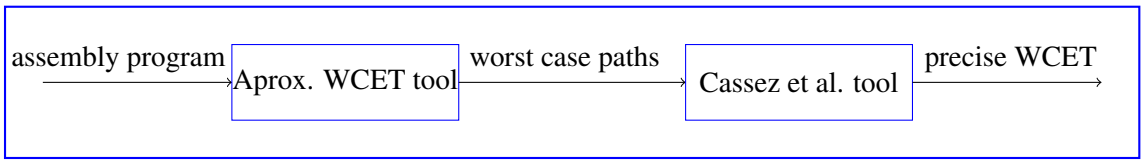

Figure 7.3: *Tool Integration with Cassez et. al WCET Tool*

The advantages of our approach are:

- The over approximation(OA) is bounded from above by the following expression

$$OA \leq (C - P_c)/P_c * 100 \tag{7.2}$$

where
- C: WCET computed by Cassez et al. tool (path may be infeasible)

- P_c: WCET computed by Cassez et al. tool on the worst path according to CPM (feasible path).

- We can produce the values of input data which produce WCET, thanks to the state of the art constraint solver.

- Li and Malik [5] pointed out that functionality constraints(i.e. what program is computing) of a program are hard to automatize but in our case we do so by transforming HLL to CPM and cut off all kinds of infeasible paths including infeasibilities caused by functionality of a program. This is something which has not been done in WCET community.

We applied our technique to the WCET benchmark programs(slightly modified by Cassez et al.) and the results are summarized in section 7.2. These modifications are documented in [3].

7.2 Experimental Results

Table 7.1 shows approximate WCET results obtained using our tool chain. We took three benchmark programs from WCET benchmark programs from Mälardalen University for our experiment. They are:

- fib-O0 = Simple iterative Fibonacci calculation, used to calculate fib(300)

- bs-O2 = Binary search for the array of 800 integer(only Array is given not the element)

- insertsort-O2 = Insertion sort on a reversed array of size 11

We have considered programs with different characteristics for our experiment:

1. programs compiled with different optimizations options(e.g., -O2, -O0) because they stress different parts of the hardware and the WCETs differ.

2. different sets of programs: single path program(fib-O0) and multiple paths program(bs-O2 and insertsort-O2) to see how different programs affect the computation time.

The second column in the table 7.1 shows a benchmark program, the third column gives description of a program, the fourth column represents the computed approximate WCET(measured in number of processor cycles) and the last column represent the time to compute the WCET. We use a computer which has Intel Core 2 Duo CPU with 2.20GHz speed and 2048 MB memory with Ubuntu 11.10 system, and measure the timings in milliseconds. For approximate WCET computation, we assume the following:
- *In-order exec*: The instructions are executed in order.
- *No instruction cache misses*: This is not important to us as we are not computing the real WCET but approximate one.
- *Always data cache misses*: The address to read data from or write data to are never in cache and the penalty for this is 34 cycles. In both of these cases, there will be cache update.

Table 7.1: *Approximate WCET Computation*

SN	Benchmark	Description	A. WCET	A. CT(ms)
1	**fib-O0**	Simple iterative Fibonacci calculation, used to calculate fib(300)	131950	36
2	**bs-O2**	Binary search for the array of 800 integer elements(array given)	842	648
3	**insertsort-O2**	Insertion sort on a reversed array of size 11	9504	208

7.2.1 Comments on the Results

1. The result shows that the WCET computation times are higher for multiple paths program as the solver needs to search and backtrack many times in order to maximize the characteristic function. In case of the single path program, there is no search, so the solver produces the result much faster.

2. Without using heuristic the computation time for *insertsort-O2* was 6534 ms. To obtain better result we modify the labeling function(search function) in such a way that the search for the elements of the array starts from upper bound of their domains towards the lower bound. This was done with the knowledge of the program in consideration. If the elements in the array are in reversed order, sorting them takes longer time. With this heuristic during search we obtain CT as low as 208 ms. Using the labeling strategies presented in 6.1.2, we obtained WCET computation time for *bs-O2* as low as 648 which was 6736 with naive labeling.

3. The highest approximate WCET for *fib-O0* is justified by having instructions in assembly language of *fib-O0* which access and update stack multiple times. As we consider that there are always data cache misses and the penalty for each such miss is 34 cycles, the WCET is higher. If we do not consider data cache miss we obtain results as low as 15348. This shows how important is cache analysis for WCET computation.

4. The lowest approximate WCET for *bs-O2* is because of the presence of few instructions which access the main memory and low penalty for them.

7.3 Comparison between two WCET computation approaches

Table 7.2 compares our result with the result from Cassez et al. The second column represents the benchmark programs, the third column *A. WCET* represent the approximate WCET computed with our method, and A. CT(ms) in the fourth is the time to compute this approximate WCET. Similarly, *P. WCET* is the precise WCET (i.e. WCET computed taking into consideration the hardware characteristics) computed using Cassez et al.'s method and *P. CT* is the time to compute this. The times are measured in milliseconds.

Table 7.2: *Comparison between two approaches of WCET computation*

SN	Benchmark	A. WCET	A. CT(ms)	P. WCET	P. CT(ms)
1	**fib-O0**	131950	36	8098	2320
2	**bs-O2**	842	6736	628	15800
3	**insertsort-O2**	9504	208	1326	11680

7.3.1 Comments on the Comparison

1. The results are obtained much faster in our case than Cassez's case, the highest being 64 times faster. This is manifested by constraint solvers implementing the path pruning techniques unlike model checkers. This shows that constraint solvers are normally better suited for path analysis than model checkers. But exception applies to some programs(*bs-O2*) in which case the time gain is not so significant in comparison to other cases of the experiment because there are many equally weighed(i.e. paths which produce equal approximate WCET) feasible and probable worst case paths(e.g., when the searched element is not present in the array). So pruning paths is difficult.

2. Approximate WCETs in our case are way larger than the precise WCETs. This is because in our model we have not taken into consideration the caches and pipelines which have huge effects on WCETs. In case of *bs-O2* the results are almost the same as this program accesses memory very few times and the effects of data cache misses are not so significant. The small difference is due to the pipeline effects.

3. The most important is that approximate WCETs preserve the ordering with respect to precise WCETs. We obtained approximate WCETs higher than the precise WCETs as expected.

7.4 Paths Study

We discussed briefly in section 7.1 on how we can provide most probable worst case paths for the tool chain integration. In this section, we focus our study around the number of these paths necessary to guarantee that these include the worst case path. In fact, we generate sufficient number of feasible paths if there are. In order to study the bound on the number of these paths to generate, we implemented a method which compares paths(compare the sequence of memory locations of the two paths) generated by constraint solver with the witness path returned by MC. The comparison makes sense only if the path returned by MC is feasible. If the comparison succeeds, then these paths are the same. If not, we count the number of failure until we find the same path. This limiting number will be the number of paths that need to be generated by the constraint solver. In principle, this number could be substantially large in comparison to the total number of feasible paths. If we consider the example of *bs-O2*, there are several paths which correspond to not finding the element in the array and have the same cost according to CPM and we can put them in one class. We may have several such classes depending upon the cost of the paths. So we can supply one representative of each class with high cost to the MC.

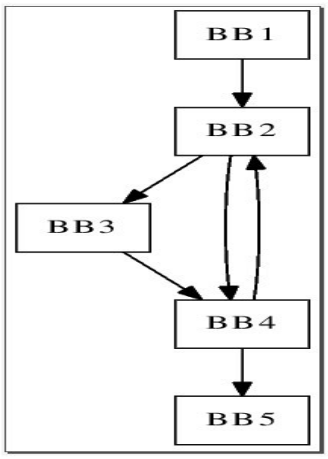

Figure 7.4: *CFG bs-O2*

We compared paths generated by the constraint solver to the path from MC for the above three programs for which we computed approximate WCET. For *fib-O0* and *insertsort-O2* the first path from constraint solver coincided with the path from MC. So in these cases, generating single path is sufficient which is manifested by having a single path with the highest cost and we call this a clear winner. For *fib-O0*, only one path is feasible for a given number so this is the obvious answer. And for *insertsort-O2* the worst case should occur when the array is completely reversed. In this sense of analysis, both solvers(CS and MC) produce the same path. However this is not the case in *bs-O2*. There is no unique situation under which the worst case scenario can occur but multiples(e.g., there are multiple paths which can lead to not finding an element in the array). Figure 7.4 is the CFG of *bs-O2*. The edge *BB2* to *BB4* is taken when the element is found. Simply looking at the graph the worst case should occur when we do not visit *BB4* directly from *BB2* but through *BB3* because in this case there is extra cost of visiting *BB3*. The CPM reports this as the worst case, however the model checker returns as the worst case when the element is found in the last iteration of the search loop(i.e., *BB4* visited directly from *BB2*). We can reproduce the case of MC when we put negative number as the cost of visiting *BB3* in the code example A.3. According to CPM, this path falls in the third class, having the third highest cost. CPM reports second class with paths corresponding to not finding the element in less than $log_2 N$ step where $log_2 N$ is the worst number of steps we have to try during the search. The difference is accounted because of the caches and pipelines considerations in one and not in the other method. This shows that it is not always the case that the worst path according to CPM is the real worst path. Now the case arises when the witness path from MC is infeasible(this can be verified using the methodology described in chapter 4). It does not make sense the comparison as we never generate infeasible paths. So in conclusion, if there is no clear winner according to CPM we supply representatives of few classes(if there are) with high costs as the most promising paths. In case of having a clear winner, we can supply just a single path. This conclusion was drawn from our empirical study in few benchmark programs and more reliable guarantee can be obtained by studying more benchmark programs.

7.5 Partial Conclusion

In this chapter, we described a procedure of computing approximate WCET(without considering the hardware characteristics) of a program. We applied our technique to compute WCET

of some standard benchmark programs and compared our results with the results from Cassez et al. [3]. The results showed that our timings to compute WCET are shorter than his timings. Based on this approximate WCET, we generated some most probable feasible worst case paths which will be used to compute precise WCET using Cassez et al.'s [3] technique. We also provided a way to compute an upper bound for over approximation on computing WCET using his method.

Chapter 8

CONCLUSIONS AND FUTURE WORKS

"One finds limits by pushing them."
Herbert Simon

In this thesis, we developed a technique to validate program paths written in low level language(ARM Assembly) and generate input data if the path is valid. We implemented a path analysis tool and used it to validate paths[1] produced as the result of computation of WCET using model checking technique [3]. The results show that almost 33% of the paths are infeasible. This means that the computed WCET may not be tight, and this leads further to path analysis.

The above mentioned fact gave us motivation to do path analysis. So we provided a comprehensive methodology to program path analysis in the context of WCET, which deals with structural as well as functionality constraints of a program automatically and also deduces loop bounds. We developed a technique to compute approximate WCET(without taking into account the hardware characteristics) using IPET. We also implemented a prototype tool chain and used it to compute approximate WCET on some standard benchmark programs from Mälardalen University. We compared our results with the results from Cassez et al. [3]. The comparison revealed that our method was faster by large magnitude than Cassez et al.'s. Based on this approximate WCET, we propose some most probable worst case paths which can give WCET. We supply these paths to Model checking tool proposed in [3] to compute precise WCET on these paths. Moreover, we provided the theoretical upper bound of over approximation while computing WCET using Cassez et al. method. For the programs we have studied our technique scales well, but overall scalability can not be guaranteed.

For future works, we want to extend the set of supported instructions so that we can study more benchmark programs and any conclusion made based on large set of programs becomes stronger and more reliable. In addition to this, we would like to extend our technique to handle multiple procedures/functions instead of a single one. We would also like to look into the seamless integration of CS and MC so that we can combine the benefits of both to take WCET computation to a new height. Further, we would like to study the global scalability of our technique(WCET computation) by studying more of standard benchmark programs as well as programs from other benchmarks. Taking a comprehensive set of benchmark programs, we can

[1] `http://www.irccyn.fr/franck/wcet/`

guarantee that the paths supplied by CS contains the real worst case path with certain probability. This will be really useful information and we would like to focus in this direction also. We also would like to use linear solver in the future when the constraints are linear because they are known to be faster than CP solver.

During this research, we realized that using CS for path validation is not so beneficial. If a path is infeasible, we have no way to know the cause of in-feasibility. It means that the solver does not provide any hints about inconsistent core(the root cause of in-feasibility). Many state of the art tools like SAT and Satisfiability Modulo Theories(SMT) solvers have this facility. One recommendation for path validation is to use SMT solver. If the inconsistent core could be known, we may possibly refine the constraint model using this information. Some model checkers gives this facility however we are not aware of any constraints solver doing this. To the best of my knowledge, CS for the next generations should fill up these gaps.

Bibliography

[1] Peter P. Puschner and Alan Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2/3):115–128, 2000.

[2] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[3] Jean-Luc Béchennec and Franck Cassez. Computation of wcet using program slicing and real-time model-checking. *CoRR*, abs/1105.1633, 2011.

[4] Benedikt Huber and Martin Schoeberl. Comparison of implicit path enumeration and model checking based wcet analysis. In Niklas Holsti, editor, *WCET*, volume 10 of *OASICS*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.

[5] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 16(12):1477–1487, 1997.

[6] Andreas Engelbredt Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. Metamoc: Modular execution time analysis using model checking. In Björn Lisper, editor, *WCET*, volume 15 of *OASICS*, pages 113–123. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.

[7] Franck Cassez. Timed games for computing wcet for pipelined processors with caches. In *11th Int. Conf. on Application of Concurrency to System Design (ACSD'2011)*, pages 195–204. IEEE Computer Society, June 2011.

[8] Bernhard Rieder, Peter P. Puschner, Ingomar Wenzel, and Ingomar Wenzel. Using model checking to derive loop bounds of general loops within ansi-c applications for measurement based wcet analysis. In *WISES*, pages 1–7, 2008.

[9] Xianfeng Li, Yun Liang, Tulika Mitra, Abhik Roychoudhury, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. 2007.

[10] Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *STTT*, 4(4):437–455, 2003.

[11] Christian Ferdinand, Reinhold Heckmann, and Reinhard Wilhelm. Analyzing the worst-case execution time by abstract interpretation of executable code. In *ASWSD*, pages 1–14, 2004.

[12] Reinhard Wilhelm. Why ai + ilp is good for wcet, but mc is not, nor ilp alone. In *In Verification, Model Checking and Abstract Interpretation (VMCAI), LNCS 2937*, 2004.

[13] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[14] Eugene Kligerman and Alexander D. Stoyenko. Real-time euclid: A language for reliable real-time systems. *IEEE Trans. Software Eng.*, 12(9):941–949, 1986.

[15] A. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating tight execution time bounds of programs by annotations. *IEEE Real-Time Syst. Newsl.*, 5(2-3):81–86, May 1989.

[16] Peter P. Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, 1989.

[17] Alan Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15:875–889, 1989.

[18] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2/3):249–274, 2000.

[19] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. Computers*, 48(1):53–70, 1999.

[20] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In Jan Gustafsson, editor, *WCET*, volume MDH-MRTC-116/2003-1-SE, pages 99–102. Department of Computer Science and Engineering, Mälardalen University, Box 883, 721 23 Västerås, Sweden, 2003.

[21] Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Systems*, 13(1):67–91, 1997.

[22] Greger Ottosson and Mikael Sjodin. Worst-case execution time analysis for modern hardware architectures. In *In Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97*, pages 47–55, 1997.

[23] Jakob Engblom and Andreas Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99),*, pages 88–95, December 1999.

[24] ANDREAS ERMEDAHL. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Box 325, Uppsala, Sweden, June 2003.

[25] Tewodros Awgichew Beyene. Constraint based Certification of Imperative Programs. Master's thesis, Universidade Nova de Lisboa, Portugal, October 2011.

[26] Adrian Prantl, Jens Knoop, Markus Schordan, and Markus Triska. Constraint solving for high-level wcet analysis. *CoRR*, abs/0903.2251, 2009.

[27] Adrian Prantl, Markus Schordan, and Jens Knoop. Tubound - a conceptually new tool for worst-case execution time analysis. In Raimund Kirner, editor, *WCET*, volume 8 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

[28] Amine Marref and Guillem Bernat. Predicated worst-case execution-time analysis. In Fabrice Kordon and Yvon Kermarrec, editors, *Ada-Europe*, volume 5570 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 2009.

[29] Rina Dechter. *Constraint processing*. Elsevier Morgan Kaufmann, 2003.

[30] Edward P. K. Tsang. *Foundations of constraint satisfaction*. Computation in cognitive science. Academic Press, 1993.

[31] Krzysztof R. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

[32] Dynamic Decision Technologies Inc. Comet tutorial @ONLINE. `http://dynadec.com/`, 2010.

[33] Wcet benchmark programs from mälardalen wcet research group @ONLINE. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[34] Wcet benchmark programs from franck cassez's homepage @ONLINE. `http://www.irccyn.fr/franck/wcet/`.

[35] Elaine J. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM J. Comput.*, 8(4):587–598, 1979.

[36] Jian Zhang and Xiaoxu Wang. A constraint solver and its application to path feasibility analysis. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):139–156, 2001.

[37] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Software Eng.*, 17(9):900–910, 1991.

[38] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[39] Arm architecture reference manuals @ONLINE. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.architecture/index.html`.

[40] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, QUEENSLAND UNIVERSITY OF TECHNOLOGY, Australia, July 1994.

[41] Keith D. Cooper, Timothy J. Harvey, and Todd Waterman. Building a control-flow graph from scheduled assembly code. Technical report, 2002.

[42] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[43] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975.

[44] B.C. Housel. *A Study of Decompiling Machine Languages into High-Level Machine Independent Languages.* PhD thesis, Purdue University, Computer Science, August 1973.

Appendix A

CPM FOR SOME BENCHMARK PROGRAMS

Listing A.1: *CPM: fib-O0*

```
1
2  //fib-O0 (300) but it does not calculate the fib nr
3  import cotfd;
4  //defining the domain of vars
5  int  maxInt = 2^16;
6  range NonNeg = 0..maxInt;
7  //max nr of iterations
8  int  maxIters = 300;
9
10
11 //time for basic blocks
12 int maxTime=100000;
13 int timeBB1=169;
14 int timeBB2=351;
15 int timeBB3=89;
16 int timeBB4=132;
17 range Time=0..maxTime;
18
19 //counting iterations
20 range Iters = 1..maxIters;
21 range Iter0 = 0..maxIters;
22
23
24 Solver<CP> cp();
25   var<CP>{int} s12[Iter0](cp,NonNeg);
26   var<CP>{int} s16[Iter0](cp,NonNeg);
27   var<CP>{int} s20[Iter0](cp,NonNeg);
28   var<CP>{int} s24[Iters](cp,NonNeg);
29   var<CP>{int} a[ 1..2](cp,NonNeg);
30   var<CP>{int} s04(cp,NonNeg);
31
32   var<CP>{int} clocks(cp,Time);
33
34   var<CP>{int} nits(cp,NonNeg);
35   var<CP>{bool} wh1[Iter0](cp);
```

```
36
37
38  int t1 = System.getCPUTime();
39
40  //optimization function, the clock cycle of the processor
41  maximize<cp>
42    clocks
43
44  subject to {
45
46    cp.post( nits == sum(i in Iters)(wh1[i]));
47    //this can be deduced from the CFG presented in chapter 4
48    cp.post(clocks == timeBB1
49                  + timeBB3 //for the first time BB3 is always
                        executing
50                  +timeBB4
51                 + nits * (timeBB2+timeBB3)
52                );
53    cp.post(
54
55      (s04 == a[1]) &&
56      (s16[0] == 1) &&
57      (s20[0] == 0) &&
58      (s12[0] == 2));
59
60    cp.post((wh1[0] == true) );
61
62    forall(i in Iters){
63      cp.post( wh1[i] == (wh1[i-1] && (s12[i-1] <= s04)));
64
65
66      cp.post( wh1[i] => (
67           (s24[i] == s16[i-1])      &&
68          // (s16[i] == s16[i-1] + s20[i-1]) &&
69           (s16[i] == s16[i-1] ) && //to avoid the calculation of fib nr
70           (s20[i] == s24[i])        &&
71           (s12[i] == s12[i-1] + 1)  ));
72
73  //cout << i << " - " << clocks << endl;
74    }
75    cp.post( (a[2] == s16[nits]) );
76  }
77  using {
78  //search
79
80   //we want to search the truth value of each iteration
81    label(wh1);
82
83  }
84
85  int t2 = System.getCPUTime();
86
87  //printing of results
88  if (cp.getSolution() == null)
89    cout << "no solution obtained in " << t2-t1 << " milisecs" << endl;
90  else {
```

```
  cout << nits << " iterations in " << clocks << " clock cycles" <<
      endl;
  cout << " with result fib(" << a[1] << ") = " << a[2] << endl;
  cout << " obtained in " << t2-t1 << " milisecs" << endl;
}
```

Listing A.2: *CPM: insertsort-O2*

```

/*
HLL for insertsort-O2

D is the input array
 insertsort-O2(D){
  n = 11; // D size
  r6 = 2;
  r0 = 2;
  do{             // 1 to n
    r1 = r0-1
    ip = D[r0];
    r2 = D[r1];
    if (ip<r2){
      r3 = r0;
      do{           // 1 to n
        D[r0] = r2;
        r2 = D[r3-2];
        r4 = (r1>0 && r2>ip);
        r0 = r1;
        D[r3-1]=ip;
        r1 = r1-1;
        r3 = r3-1;
      }while(r4 != 0);
    }
   r6=r6+1;
   r0=r6;
   }while (r6 != 11);
}
*/

import cotfd;

int  T[1..5]=[498,98,147,18,24];

int  d = 10;
int  m = d-1;
int  n = d;
range Outs0 = 0..m;
range Outs1 = 1..m;
range Inns0 = 0..n;
range Inns01= 0..n+1;
range Inns1 = 1..n;
range Inds0 = 0..d;
range Inds01= 0..d+1;
range Inds1 = 1..d;
range Ints = 0..d; // d
```

```
48 range Time = 0..2^30;
49
50 Solver<CP> cp();
51   var<CP>{int} r0[Outs0, Inns01](cp,Inds01);
52   var<CP>{int} r1[Outs0, Inns0](cp,Inds0);
53   var<CP>{int} r2[Outs0, Inns0](cp,Ints);
54   var<CP>{int} r3[Outs0, Inns0](cp,Inds0);
55   var<CP>{int} r6[Outs0](cp,Inds01);
56   var<CP>{int} ip[Outs0](cp,Ints);
57   var<CP>{int} D[Outs0, Inns0,1..2,Inds0](cp,Ints);
58   var<CP>{int} clocks(cp,Time);
59   var<CP>{int} out_c(cp,Time);
60   var<CP>{int} inn_c[Outs0](cp,Time);
61   var<CP>{bool} outer[Outs0](cp);
62   var<CP>{bool} ifc[Outs0](cp);
63   var<CP>{bool} r4[Outs1, Inns1](cp);
64   var<CP>{bool} inner[Outs0,Inns0](cp);
65
66 int t1;
67 int t2;
68 int t3;
69 int t4;
70
71 int t0 = System.getCPUTime();
72
73 maximize<cp> clocks subject to {
74 //solve<cp> {
75
76 // time
77 t1 = System.getCPUTime();
78   cp.post(out_c == sum(i in Outs0) outer[i]);
79   forall (i in Outs0) cp.post(inn_c[i] == sum(j in Inns0)
       inner[i,j]);
80   //cp.post(clocks == T[1]+out_c*(T[2]+T[3]*sum(j in
       Outs1)(inn_c[j])+T[4])+T[5]);
81     cp.post(clocks == T[1]+ T[5]+
82               sum(i in Outs1)
83                 (
84                  (i<out_c)*
85                     (T[2]+
86                       ifc[i]*(T[3]* inn_c[i]+T[4])+
87                       (!ifc[i])*T[4]
88                     )
89                  )
90
91                 )
92                 ;
93
94 // initial vector
95
96 forall(k in Inds0) cp.post(D[0,n,2,k] == (k>0)*(d-k+1));
97 //cp.post(D[0,n,2,0] == 0);
98 //forall(k in 1..d) cp.post(D[0,n,2,k] > 0);
99 // forall(k in 1..d-1) cp.post(D[0,n,2,k] > D[0,n,2,k+1]);
100
101   cp.post(true => (r0[0,n+1] == 2));
```

```
102  cp.post(true => (r6[0] == 2));
103  cp.post(true => (outer[0] == true));
104
105  forall(i in Outs1){
106    if (i == 1) cp.post( outer[i-1] => (outer[i] == true ));
107    if (i > 1) cp.post( outer[i-1] => (outer[i] == (r6[i-1] !=
         d+1)));
108    cp.post(!outer[i-1] => (outer[i] == false));
109
110    cp.post( outer[i] => r1[i,0] == r0[i-1,n+1]-1);
111    cp.post(!outer[i] => r1[i,0] == r1[i-1,n]);
112    cp.post( outer[i] => ip[i] == D[i-1,n,2,r0[i-1,n+1]]);
113    cp.post(!outer[i] => ip[i] == ip[i-1]);
114
115    cp.post( outer[i] => (r2[i,0] == D[i-1,n,2,r1[i,0]]));
116    cp.post(!outer[i] => (r2[i,0] == r2[i-1,n]));
117
118
119    cp.post( outer[i] => (ifc[i] == (ip[i] < r2[i,0])));
120    cp.post(!outer[i] => (ifc[i] == false));
121
122    cp.post( !ifc[i] => (inner[i,0] == false));
123    cp.post( !ifc[i] => (r2[i,n] == r2[i,0]));
124    cp.post( !ifc[i] => (r0[i,n] == r0[i-1,n+1]));
125    cp.post( !ifc[i] => (r1[i,n] == r1[i,0]));
126    forall(k in Inds0) cp.post( !ifc[i] => (D[i,n,2,k] ==
         D[i-1,n,2,k]));
127
128    cp.post(  ifc[i] => r3[i,0] == r0[i-1,n+1]);
129    cp.post(  ifc[i] => r0[i,0] == r0[i-1,n+1]);
130    forall(k in Inds0) cp.post( ifc[i] => (D[i,0,2,k] ==
         D[i-1,n,2,k]));
131    cp.post(  ifc[i] => (inner[i,0] == true));
132
133    forall(j in Inns1){
134      if (j == 1) cp.post( inner[i,j-1] => (inner[i,j] == true));
135      if (j > 1) cp.post( inner[i,j-1] => (inner[i,j] == r4[i,j-1]));
136      cp.post(!inner[i,j-1] => (inner[i,j] == false ));
137      forall(k in Inds0){
138        cp.post( (inner[i,j] && k == r0[i,j-1]) => D[i,j,1,k] ==
             r2[i,j-1] );
139        cp.post( (inner[i,j] && k != r0[i,j-1]) => D[i,j,1,k] ==
             D[i,j-1,2,k]);
140        cp.post(! inner[i,j]          => D[i,j,1,k] == D[i,j-1,2,k]);
141      }
142      cp.post( inner[i,j] => r2[i,j] == D[i,j,1,r3[i,j-1]-2]);
143      cp.post(!inner[i,j] => r2[i,j] == r2[i,j-1]);
144      cp.post( inner[i,j] => (r4[i,j] == ((r1[i,j-1] > 0) && (r2[i,j]
           > ip[i]))) );
145      cp.post(!inner[i,j] => (r4[i,j] == false) );
146      cp.post( inner[i,j] => r0[i,j] == r1[i,j-1]);
147      cp.post(!inner[i,j] => r0[i,j] == r0[i,j-1]);
148      forall(k in Inds0){
149        cp.post( (inner[i,j] && k == r3[i,j-1]-1) => D[i,j,2,k] ==
             ip[i] );
```

```
150         cp.post( (inner[i,j] && k != r3[i,j-1]-1) => D[i,j,2,k] ==
                D[i,j,1,k]);
151         cp.post(! inner[i,j]            => D[i,j,2,k] == D[i,j,1,k]);
152       }
153       cp.post( inner[i,j] => r1[i,j] == r1[i,j-1]-1);
154       cp.post(!inner[i,j] => r1[i,j] == r1[i,j-1]);
155       cp.post( inner[i,j] => r3[i,j] == r3[i,j-1]-1);
156       cp.post(!inner[i,j] => r3[i,j] == r3[i,j-1]);
157     }
158    cp.post( outer[i] => r6[i] == r6[i-1]+1);
159    cp.post(!outer[i] => r6[i] == r6[i-1]);
160    cp.post( outer[i] => r0[i,n+1] == r6[i]);
161    cp.post(!outer[i] => r0[i,n+1] == r0[i,n]);
162   }
163 t2 = System.getCPUTime();
164 cout << endl << " ==== posted all in " << t2-t1 << " ms =======" <<
        endl;
165 }
166
167 using {
168   forall(i in Outs1) {
169     // label(outer[i]);
170     // try<cp> cp.label(outer[i],true); | cp.label(outer[i],false);
171     tryall<cp>(v in 0..1 ) by (-v) cp.post(outer[i] == v);
172     // label(ifc[i]);
173     try<cp> cp.label(ifc[i],true); | cp.label(ifc[i],false);
174     forall(j in Inns0)
175       //label(inner[i,j]);
176       try<cp> cp.label(inner[i,j],true); | cp.label(inner[i,j],false);
177   }
178   forall(k in Inds1)
179     label(D[0,n,2,k]);
180
181 t3 = System.getCPUTime();
182 cout << endl << " ==== labelled all in " << t3-t2 << " ms =======" <<
        endl;
183 }
184
185 t4 = System.getCPUTime();
186
187 if (cp.getSolution() == null)
188   cout << "no solution obtained in " << t4-t0 << " milisecs" << endl;
189 else {
190    cout << " wcet time of " << clocks << " clock cycles " << endl;
191   cout << " obtained in " << t4-t0 << " milisecs" << endl;
192   cout << " with initial vector: " << endl;
193 forall(k in Inds0) cout << D[0,n,2,k] << " "; cout << endl;
194   cout << " and sorted vector: " << endl;
195 forall(k in Inds0) cout << D[m,n,2,k] << " "; cout << endl;
196   cout << endl;
197
198 }
```

Listing A.3: *CPM: bs-O2*

```
 //bs-O2 : This method is extracted from main method of the assembly
    code.
//so the return element is not shown.
/*
 bs-O2(x){
  r3=799;
  ip=0;
  do{
    r1 = (ip + r3)/2;
    r4 = S[8*r1];
    if (r4 == r0)
       r3=ip-1;
    else {
      if (r4 > r0) r3 = r1 - 1;
      if (r4 < r0) ip = r1 + 1;
    }
  }while(ip<=r3);
}

*/

import cotfd;

int maxInt = 2000;
range NonNeg = 0..maxInt;
int maxIters = 12;
int n = 1600; //size of array
range Iters = 1..maxIters;
range Iter0 = 0..maxIters;
range Iter1 = 1..maxIters+1;
//time for basic blocks
int maxTime=10000;
int timeBB1=49;
int timeBB2=61;
int timeBB3=6;
int timeBB4=12;
int timeBB5=3;
range Time=0..maxTime;

int S[0..n-1];

Solver<CP> cp();
var<CP>{bool} wh1[Iter1](cp);
var<CP>{bool} if1[Iter0](cp);
var<CP>{bool} if2[Iter0](cp);
var<CP>{bool} if3[Iter0](cp);
var<CP>{int} nits(cp,NonNeg);
var<CP>{int} r0[1..2](cp,-1..maxInt);
var<CP>{int} r1[Iters](cp,NonNeg);
var<CP>{int} r3[Iter0](cp,-1..maxInt);
var<CP>{int} r4[Iters](cp,NonNeg);
var<CP>{int} ip[Iter0](cp,NonNeg);
var<CP>{int} r5[Iter0](cp,-1..maxInt);
var<CP>{int} clocks(cp,Time);

int t1 = System.getCPUTime();
```

```
//the elements of the array
S[0]=0;    S[1]=807;
S[2]=2;    S[3]=1249;
        ⋮
S[1596]=1596;  S[1597]=1669;
S[1598]=1598;  S[1599]=1096;

maximize<cp>
clocks
subject to
 {

  cp.post(nits == sum(i in Iters) (wh1[i]));
  cp.post(clocks == timeBB1 + timeBB5 +
              sum(i in Iters)(((i<=nits)*(timeBB2 +
                if1[i]*timeBB4 + (!if1[i])*(timeBB3+timeBB4))))) ;

  cp.post(r3[0]== 799);
  cp.post(ip[0]== 0);

//initilization
 cp.post(if1[0]==false);
 cp.post(if2[0]==false);
 cp.post(if3[0]==false);
 cp.post(wh1[1] == true);

  forall(i in Iters){

    cp.post( wh1[i] => (r1[i]== ((ip[i-1]+r3[i-1])/2)));
    cp.post( wh1[i] => (r4[i]== S[2*r1[i]]));
    //value found
    cp.post( wh1[i] => (if1[i] == (r4[i]==r0[1])));
    cp.post( !wh1[i] => (if1[i] == false));

    cp.post((wh1[i] && if1[i]) =>(r3[i]==ip[i-1]-1));
    cp.post((wh1[i] && if1[i]) =>(ip[i]==ip[i-1]));

    cp.post((wh1[i]) => (if2[i] == (r4[i] > r0[1])));
    cp.post((!wh1[i]) => (if2[i] == false));
    cp.post((wh1[i] && !if1[i] && if2[i]) => (r3[i] == r1[i]-1));
    cp.post((wh1[i] && !if1[i] && if2[i]) => (ip[i] == ip[i-1]));

    cp.post(wh1[i]=>if3[i]==(!if1[i] && !if2[i]));
    cp.post(!wh1[i] => (if3[i] == false));

    cp.post((wh1[i] && if3[i]) => (ip[i]==r1[i]+1));
    cp.post((wh1[i] && if3[i]) => (r3[i]==r3[i-1]));

    cp.post( wh1[i+1] == (wh1[i] && (r3[i] >= ip[i])));
  }

}

using {
  //r0[1] is the element we are searching for
```

```
  label(r0[1]);

  forall(i in Iters){


    try<cp>{
      cp.post(wh1[i]==1);
           try<cp>{
             cp.post(if1[i]==1);
            }
            |
            { cp.post(if1[i]==0);
                try<cp> {cp.post(if2[i]==1);}|{cp.post(if2[i]==0);}
            }
     }
    |
    { cp.post(wh1[i]==0);}
 }

}

int t2 = System.getCPUTime();
cout << endl;

if (cp.getSolution() == null)
  cout << "No solution found " << endl;
else {
  cout << " The solution is obtainted in " << t2-t1 << "
      miliseconds" << endl;
  cout << " Computed WCET is " << clocks<< " cycles" << endl;
}
```