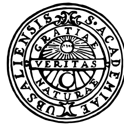
ANDREAS ERMEDAHL

A Modular Tool Architecture for Worst-Case Execution Time Analysis



UPPSALA
UNIVERSITET

Dissertation for the Degree of Doctor of Philosophy in Computer Systems presented at Uppsala University, June 3, 2003.

ABSTRACT

Ermedahl, A. 2003: A Modular Tool Architecture for Worst-Case Execution Time Analysis. Acta Universitatis Upsaliensis. *Uppsala dissertations from the Faculty of Science and Technology 45*. 200 pp. Uppsala. ISBN 91-554-5671-5.

Estimations of the Worst-Case Execution Time (WCET) are required in providing guarantees for timing of programs used in computer controlled products and other real-time computer systems. To derive program WCET estimates, both the properties of the software and the hardware must be considered. The traditional method to obtain WCET estimates is to test the system and measure the execution time. This is labour-intensive and error-prone work, which unfortunately cannot guarantee that the worst case is actually found. Static WCET analyses, on the other hand, are capable of generating safe WCET estimates without actually running the program. Such analyses use models of program flow and hardware timing to generate WCET estimates.

This thesis includes several contributions to the state-of-the-art in static WCET analysis:

(1) A tool architecture for static WCET analysis, which divides the WCET analysis into several steps, each with well-defined interfaces. This allows independent replacement of the modules implementing the different steps, which makes it easy to customize a WCET tool for particular target hardware and analysis needs.

(2) A representation for the possible executions of a program. Compared to previous approaches, our representation extends the type of program flow information possible to express and handle in WCET analysis.

(3) A calculation method which explicitly extracts a longest program execution path. The method is more efficient than previously presented path-based methods, with a computational complexity close to linear in the size of the program.

(4) A calculation method using integer linear programming or constraint programming techniques for calculating the WCET estimate. The method extends the power of such calculation methods to handle new types of flow and timing information.

(5) A calculation method that first uses flow information to divide the program into smaller parts, then calculates individual WCET estimates for these parts, and finally combines these into an overall program WCET. This novel approach avoids potential complexity problems, while still providing high precision WCET estimates.

We have additionally implemented a prototype WCET analysis tool based on the proposed architecture. This tool is used for extensive evaluation of the precision and performance of our proposed methods. The results indicate that it is possible to perform WCET analysis in a modular fashion, and that this analysis produces high quality WCET estimates.

Andreas Ermedahl, Department of Information Technology, Uppsala University, Box 325, SE-75105 Uppsala, Sweden. Email: `andreas.ermedahl@it.uu.se`

ISSN 1104-2516

ISBN 91-554-5671-5

Printed in Sweden by Elanders Gotab, Stockholm 2003.

Distributor: Uppsala University Library, Box 510, SE-75120 Uppsala, Sweden. `acta@ub.se`

Acknowledgements

First of all I would like to thank my supervisor Hans Hansson. During my years as a graduate student Hans has guided me with great enthusiasm and technical knowledge, and he has supported me to grow as a researcher. Also, during the writing of this thesis his thorough reviewing was really invaluable.

The research project I have been working within is a cooperation between researchers located in Uppsala University, C-Lab in Paderborn and Mälardalen University in Västerås. This has convinced me that research is a group activity and this teamwork has allowed me to achieve much more than I possibly could have done on my own.

I would especially like to thank Jakob Engblom who has been my research team-mate in Uppsala during most of my years as a PhD student. Together we planned and started the work that now has resulted in this thesis. I would like to thank Jakob for years of intense and inspiring cooperation and discussions, as well as for his very constructive comments on drafts of this thesis.

Friedhelm Stappert has been involved in the WCET project during the last couple of years, adding fresh perspectives and implementation manpower. Despite the fact that Friedhelm is located at C-Lab in Paderborn in Germany, he, Jakob and I have together managed to produce a WCET tool prototype and write a number of joint research papers. I thank Friedhelm for a very fruitful collaboration.

I thank Jan Gustafsson for introducing me to the area of WCET analysis research. Together we wrote my first conference publication on the subject, and during the last months Jan has given me a lot of valuable and constructive feedback.

Other people involved in the WCET project which I would like to thank for detailed discussions on thesis subjects are Björn Lisper and Christer Sandberg.

Many thanks goes to all my friends and colleagues at the IT-department at Uppsala University for providing me with an excellent working and research environment. This also includes all the people that were part of the department when I started but has graduated or moved on for other reasons.

Mikael Sjödin helped me to a good start in my PhD studies by including me in his research work when I joined the department back in 1996. Mikael also provided constructive discussions on the subjects in this thesis.

I thank Bengt Jonsson, the director of ASTEC, which provided the major part of my project funding.

My years as a PhD student have also provided me with the opportunity to travel and to meet other researchers around the world. I cannot list them all, but would like to mention a few people who have made a special impression on me:

Philippas Tsigas and Marina Papatriantafilou, who encouraged me to go to Hiroshima and make my first conference presentation on my own. Peter Altenbernd, who have taught me that German beer-loving punk-rockers can be both excellent friends and real-time researchers. Chris and Geraldine Exton, who showed me that combining Australians and the Irish can make truly wonderful people. Sang Lyul Min and his PhD students, including Sung-Soo Lim, Kanghee Kim, Woonseok Kim, Sheayun Lee and Hoyoung Hwang, who together gave me a great six month stay at Seoul National University. Lucia LoBello and Giancarlo Iannizzotto, temperamental but wonderful Italian researchers who I got to know during my stay in Korea.

My friends and the players in the HK71 handball team all deserve special thanks for reminding me that there exists a life outside academia.

My deepest gratitude goes to my father Göran and my mother Gunilla, my sisters, my brother, and the rest of my family, for always supporting and believing in me.

Finally, I would like to thank Annelie, the very special person that has been part of my life during the last years. With love, support and a lot of patience she really helped me during the last stressful months of this thesis writing.

This work has been performed within the competence center for Advanced Software TEChnology (ASTEC) at Uppsala University, partially funded by the Swedish Agency for Innovation Systems (Vinnova). The ARTES network provided me with funding for some travels and summer schools. FFDF provided travel funding for my Korean research trip.

Contents

1 Introduction **1**
1.1 Embedded systems . . . 2
1.2 Real-time systems . . . 6
1.3 Execution time estimates . . . 8
1.4 Uses of WCET analysis . . . 11
1.5 The need for WCET analysis tools . . . 12
1.6 Contributions of this thesis . . . 13
1.7 Thesis outline . . . 14

2 WCET Analysis Overview and Previous Work **17**
2.1 Components of static WCET analysis . . . 17
2.2 Flow analysis . . . 18
2.3 Low-level analysis . . . 22
2.4 Calculation . . . 32
2.5 WCET tools . . . 36

3 A Modular WCET Tool Architecture **39**
3.1 Analysis modules and data structures . . . 39
3.2 The basic block graph . . . 40
3.3 The scope graph . . . 41
3.4 The timing model . . . 43
3.5 Separation vs. integration . . . 44

4 Representing Program Flow **47**
4.1 Introduction . . . 47
4.2 Including all possible executions . . . 48
4.3 Flows information characteristics . . . 49
4.4 Expressing flow analysis results . . . 51
4.5 Managing real-world code . . . 51
4.6 Context-sensitive flow information . . . 53
4.7 Flow information locality . . . 54
4.8 Dynamic vs. static flow information . . . 55
4.9 Flow information conversion . . . 57

4.10 Conclusions 57

5 The Scope Graph and Flow Fact Language **59**
5.1 Introduction 59
5.2 The scope graph 60
5.3 Loop bounds 65
5.4 Flow facts 66
5.5 Loop-bound and flow fact semantics 70
5.6 More on complex flows 77

6 Low-level Analysis **85**
6.1 Global low-level analysis 85
6.2 Execution scenarios 86
6.3 Expressing global low-level analysis results 87
6.4 Safe removal of scenarios 89
6.5 Local low-level analysis 90
6.6 The problem of pipeline analysis 91
6.7 Pipeline timing analysis 94
6.8 Timing model 95
6.9 Alternative timing analyses 98

7 Efficient Path-based Calculation **101**
7.1 Introduction 101
7.2 Method overview 102
7.3 Basic path search algorithm 105
7.4 Path search with flow facts 107
7.5 Handling long pipeline effects 112
7.6 Complete example 117
7.7 Possible method extensions 117

8 Extended IPET Calculation **121**
8.1 IPET calculation basics 122
8.2 Expanding the scope graph 124
8.3 Constraint generation 130
8.4 Converting the timing model 136
8.5 Main algorithm and complete example 141

9 Clustered Calculation **145**
9.1 Introduction 145
9.2 Method overview 146
9.3 Clustering of flow facts 148
9.4 WCET calculation using fact clusters 152
9.5 Hardware timing and local calculations 158
9.6 Complete example 163

10 Prototype Tool and Experiments 167
10.1 Prototype implementation 167
10.2 User interaction and feedback 169
10.3 Benchmark programs 171
10.4 WCET estimate precision 173
10.5 Flow facts and WCET precision 175
10.6 Long timing effects and WCET precision 176
10.7 Computation time 177
10.8 Path-based calculation evaluation 178
10.9 Scalability of calculation methods 179
10.10Clustered calculation evaluation 181

11 Conclusions and Future Work 185
11.1 Summary of contributions 185
11.2 Evaluation 186
11.3 Future work in WCET analysis 187

Publications by the Author

During my years as a Ph.D. student I have been involved in a number of different research projects, not all related to WCET analysis, and I have therefore published articles on several topics with a number of different people. The following is a list sorted in chronological order of my publications which have been subject to peer review:

A. Andreas Ermedahl and Jan Gustafsson: Deriving Annotations for Tight Calculation of Execution Time. In *Proceedings of the 3rd International Euro-Par Conference, (Euro-Par'97)*, LNCS 1300, Passau, Germany, August 1997.

B. Jan Gustafsson and Andreas Ermedahl: Automatic derivation of path and loop annotations in object-oriented real-time programs. In *Proceedings of the Joint Workshop on Parallel and Distributed Real-Time Systems at the 11th IEEE International Parallel Processing Symposium (IPPS'97)*, Geneva, Switzerland, April 1997.

C. Andreas Ermedahl, Hans Hansson and Mikael Sjödin: Response-Time Guarantees in ATM Networks. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, San Francisco, California, December 1997.

D. Hans Hansson, Mikael Sjödin and Andreas Ermedahl: Response-Time Guarantees for Networked Control Systems. In *Proceedings of the 9th IFAC Symposium on Information Control in Manufacturing (INCOM'98)*, Nancy - Metz, France, June 1998.

E. Jakob Engblom, Andreas Ermedahl and Peter Altenbernd: Facilitating Worst-Case Execution Times Analysis for Optimized Code. In *Proceedings of the 10th Euromicro Real-Time Systems Workshop (ERTS'98)*, Berlin, Germany, June 1998.

F. Andreas Ermedahl, Hans Hansson, Marina Papatriantafilou and Philippas Tsigas: Wait-Free Snapshots in Real-Time Systems: Algorithms and Performance. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98)*, Hiroshima, Japan, October 1998.

G. Jakob Engblom and Andreas Ermedahl: Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, December 1999.

H. Jakob Engblom and Andreas Ermedahl: Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS'2000)*, Orlando, Florida, USA, December 2000.

I. Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson and Hans Hansson: Execution-Time Analysis for Embedded Real-Time Systems. Accepted for publication in *Journal of Software Tools for Technology Transfer, STTT)*, special issue on ASTEC (forthcoming).

J. Sheayun Lee, Andreas Ermedahl, Sang Lyul Min and Naehyuck Chang: An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors. In *Proceedings of the ACM SIGPLAN 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'2001)*, Snowbird, Utah, USA, June 2001.

K. Jakob Engblom, Andreas Ermedahl and Friedhelm Stappert: A Worst-Case Execution-Time Analysis Tool Prototype for Embedded Real-Time Systems. In *Proceedings of the 1st Workshop on Real-Time Tools (RT-TOOLS'2001)*, Aalborg, Denmark, August 2001.

L. Friedhelm Stappert, Andreas Ermedahl and Jakob Engblom: Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proceedings of the 4th International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES'2001)*, Atlanta, Georgia, USA, November 2001.

M. Andreas Ermedahl: A Unified Flow Information Language for WCET analysis. In *Proceedings of the 2nd Workshop on Worst-Case Execution Time analysis (WCET'2002)*, Vienna, Austria, June 2002.

N. Martin Carlsson, Jakob Engblom, Andreas Ermedahl, Jan Lindblad and Björn Lisper: Worst-Case Execution Time Analysis of Disable Interrupt Regions in a Commercial Real-Time Operating System. In *Proceedings of the 2nd Workshop on Real-Time Tools (RT-TOOLS'2002)*, Copenhagen, Denmark, August 2002.

In addition to the above papers I have been co-authoring a number of technical reports [EES+99, SEE01, EES01, LEMC02] and work-in-progress articles [EES00, ESE00].

Some of these publications form the basis of this thesis. Compared to the original publications, there is a lot of new material in this thesis: each work is extended and the algorithms and methods used are described in more detail. The publications forming the basis of this thesis are:

- Papers I and K which contain the first ideas for the modular WCET tool architecture outlined in Chapter 3. I co-authored the papers and have together with Jakob Engblom and Friedhelm Stappert been the main developers of the WCET tool architecture.
- Paper H and M which deal with the problem of how to represent program flow for WCET analysis. These papers are the basis for Chapter 4 and Chapter 5 respectively. I co-authored the papers and have been the main developer of the flow representation.
- Paper G which contains an early version of the pipeline analysis and the resulting timing model outlined in Section 6.5. Jakob Engblom and I participated equally in the method development and in the paper writing. The paper forms the basis for low-level analysis outlined in Chapter 6. The Ph.D. thesis by Jakob Engblom [Eng02] extends the original work and contains a deeper investigation of processor pipelines than the material presented in this thesis.
- Paper H also forms the basis for the IPET-based calculation method outlined in Chapter 8. Jakob Engblom and I participated equally in the method development and in the paper writing. I am the main developer and responsible for the implementation of the calculation method.
- Paper L which forms the basis for the path-based calculation method outlined in Chapter 7. I co-authored the paper together with Friedhelm Stappert and Jakob Engblom and we all equally participated in the method development.

The clustered calculation method outlined in Chapter 9 has not been previously published and is to our knowledge a completely novel approach for WCET calculation. I am the main developer and responsible for the implementation of the method.

There are also some other publications related to WCET analysis which I co-authored, but which will not be described in more detail in this thesis:

- Paper A and B, which present early work on deriving flow information suitable for WCET analysis. The Ph.D. thesis by Jan Gustafsson [Gus00] and later work of his [GLSB03] contains extensions of these initial ideas.
- Paper E, which deals with the problem of mapping source code WCET flow information to the (optimized) object code (see Section 2.2.3 on page 21 for more information).
- Paper N, which presents a case study of the problems that needs to be addressed when using WCET analysis in an industrial setting.
- Paper [EES00], which deals with how to compare different WCET calculation methods.
- Paper [ESE00], which deals with the problem of validating WCET analysis tools and methods.

To summarize: the publications forming the basis for this Ph.D. thesis are [EES+99], K (WCET tool architecture) H, M (flow representation) G (pipeline timing analysis) H and L (calculation methods). Compared to these publi-

cations, there is a lot of new material in this thesis and each work has been extended and described in more detail.

Almost all of my research has been carried out within the framework of the ASTEC WCET project in close cooperation with several colleagues. The prototype implementation and experiments have been carried out in cooperation with Jakob Engblom (also at Uppsala University) and Friedhelm Stappert (at C-Lab in Paderborn, Germany).

Chapter 1

Introduction

Over the last few decades, our society has become increasingly dependent on computers. Not only the gray PC boxes at our desks, but also the myriad of computer systems *embedded* in everyday things around us. In fact, over 98 percent of all computers sold are used to control vehicles, appliances, power plants, telecommunication equipment, toys, and other products that are intrinsic parts of modern society. Many of these systems are required to react within precise *real-time* constraints to events in the environment.

Take a look around in a modern car. There is an embedded computer controlling the engine, keeping performance up and fuel consumption down by very precise control of the ignition and fuel pump. For your safety, the anti-lock brakes (ABS) are controlled by embedded computers that continuously monitor the behavior of the car to prevent brake locking. In the unlikely event of a collision, yet other embedded computers will detect the crash within milliseconds and deploy the airbags.

Such embedded real-time systems are based on one or more computers. One or more computer programs are running on each compter. Any failure of these embedded computer systems could endanger human life and cause substantial economic losses, and thus, there is a need for software development methods and tools to minimize the risk of failures.

The purpose of *worst-case execution time* (WCET) analysis is to provide information about the worst possible execution time of a computer program before using the program in the final product. WCET estimates are a key component in providing guarantees of satisfactory system behavior, and are especially important when it must be proven that the system will always behave correctly, even in the most stressful situations.

Static WCET analyses are a means of determining the worst-case execution time of a program without actually running it. Such analyses rely on models of program behavior and timing to generate safe WCET estimates which are guaranteed not to underestimate the actual WCET. The alternative analysis method is to test the systems and measure the execution times. This will,

Figure 1.1: Example of products using embedded computers

however, not guarantee that the true worst case will be found, since in general, it is practically impossible to test all possible program behaviors. Static WCET analysis is a concept similar to inspecting the blueprints of a bridge to determine whether it will collapse, instead of building the bridge and driving heavy trucks across it in order to test its strength.

This thesis is about static WCET analysis, in particular about a WCET tool-architecture applicable to a wide spectrum of different embedded computers and programs. The remaining chapters of this thesis will present different parts of the tool in more detail, including methods and algorithms suitable for the particular problems encountered.

The rest of this introduction will give a more detailed background of embedded systems, real-time systems and program execution time. A reader familiar with this background can proceed directly to Section 1.6, where the concrete contributions of this thesis are presented.

1.1 Embedded systems

An embedded system can be said to be "a computer that does not look like a computer", i.e., it is a part of, incorporated within a product. It is a computer used as a mean to achieve some specific purpose, the computer is not the end product in itself.

Contrary to popular opinion, the majority of computers sold are not Intel and AMD systems or servers. The great majority of computers are embedded, used in consumer electronics, vehicles, airplanes, game systems, hand-held devices, networking and communications systems, and many other applications. Figure 1.1 shows products that depend on embedded computers to function properly.

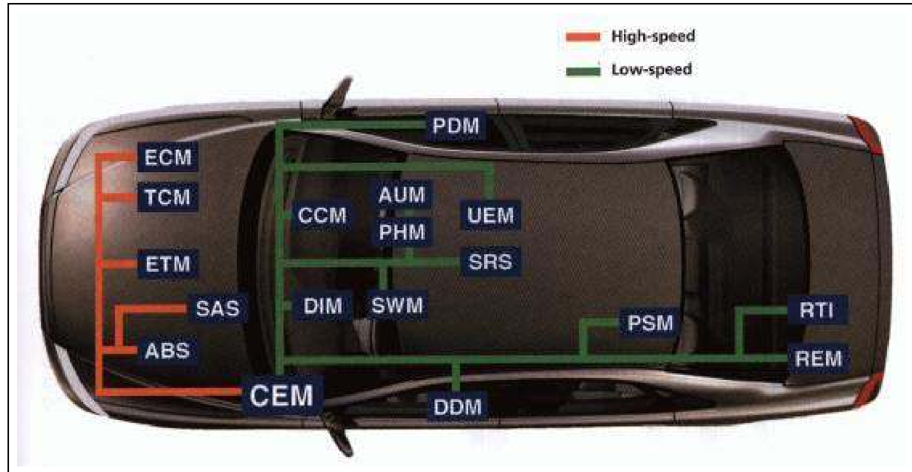


Figure 1.2: Schematic of on-board electronic modules in Volvo S80

In fact, over 98 percent of the total, more than 8 billion processors produced annually, are used in embedded systems [Hal00, Tur02]. The dominating use of computers today is embedded systems, and this will increase even further as we enter an era of pervasive computing with enourmous amount of cooperating computers controlling virtually all the devices in our environment.

In many embedded systems several different embedded computers are included and may need to communicate with each other to fulfill the system objective. For example, a GSM mobile telephone contains at least two processors: a digital signal processor (DSP) specialized for handling encoding and decoding of radio and data signals, and a main processor to run the menu systems, games and other user-interface functions.

As processors become more powerful, more reliable, and less expensive, they also become attractive for use in new areas. In many cases, computers replace sub-systems that were previously controlled entirely by mechanical systems or fixed-function logic implemented as electro-magnetic relays or electronic circuits. But not only do computers replace existing systems or system components, they also have the potential to provide more functionality with higher reliability at lower cost.

For example, it is now common practice to use embedded computers to control many parts of automotive systems. Modern cars have an embedded processor to control the engine. The processor calculates time-angle ratios, which are vital for valve and ignition timing. Outside the engine, automatic transmissions are microprocessor controlled as well. Cars currently available even have adaptive shifting algorithms, modifying shift points based on road conditions, weather, and the driver's individual habits. Anti-lock brakes are generally computer controlled, replacing the hydraulic-only systems of earlier

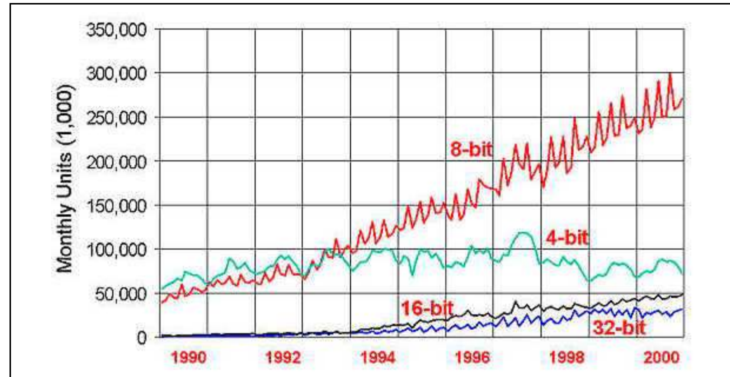


Figure 1.3: Microprocessor unit sales. All types, all markets worldwide [Tur02]

years.

A car such as the Volvo S80 contains more than 30 embedded processors, communicating across several networks. Figure 1.2 illustrates the arrangement of on-board electronic modules in the Volvo S80 [Mel98]. Similarly, BMW 7-series and the Mercedes S-class both contain over 60 processors [Tur02].

Another example of a system containing several embedded processors is a normal PC. Apart from the main processor from Intel or AMD driving the PC there is one processor in the keyboard, another processor in the mouse, a processor in each hard drive and floppy drive, one in the CD-ROM, one in the graphics accelerator, etc., all cooperating to enable the computer to behave in the intended manner.

1.1.1 Properties of embedded hardware

Comparing the embedded processor market with the desktop market, we first note that there is a much larger variety of processors on the embedded market. Contrary to the desktop market situation, there is no specific architecture or manufacturer clearly dominant. There are instead hundreds of processors types to choose from, many very simple, low in cost and specialized for a certain type of application.

As illustrated in Figure 1.3, simpler microprocessors (4-8-16 bit) completely dominate the market in terms of units sold[1]. The list of embedded microprocessors architectures (and manufacturers) available on the very fragmented chip market is very long, including ARM, AMD, Intel, MIPS, SuperH, PowerPC and NEC.

Embedded CPUs are usually much simpler in their design and therefore in most cases much cheaper than desktop processors. The latter incorporate

[1]Desktop processors, however, represent a much large share of the manufacturers' earnings, since the profit per sold unit is magnitudes higher.

many hardware features, including techniques such as caches, branch predictors and speculative execution to boost their performance. Embedded processors do not usually include such features which are generally too expensive, space-demanding and power-consuming. Also, for embedded systems designed for predictability, most of these features are considered to introduce too much time variance into the system. For example, memory in embedded systems is often based on static RAMs, since caches are considered too unpredictable. Caches are also quite demanding in terms of chip area and power consumption, making them less suitable for embedded systems.

Comparing desktop and embedded processors further, we note that embedded processors are often more specialized, intended to perform a specific task. An example of such a specialized embedded CPU is a digital signal processor (DSP). A DSP is targeted to perform intense mathematical calculations, over and over again, and is normally used for processing streams of digital media or signals. Consequently, a DSP is designed to work very differently from normal processors which are more focussed on control-flow decisions and logical operations.

Examples of factors that influence the choise of microprocessor for a particular embedded applications include cost, (i.e., sufficient performance for smallest amount of money), size, peripheral integration, energy consumption, heat emission and the type of task to be performed.

1.1.2 Properties of embedded software

One of the main reasons for the success of computers are that they are *programmable*, allowing one type of computer to be used in a large variety of different applications. Software is the key component in embedded systems, providing added value and required behaviour. The hardware related costs are typically only a small fraction of the total system cost [ART00]. In most embedded systems, the hardware consists of standard electronic components available in large volumes at low cost, whereas the software is to a large extent designed specifically for the application concerned.

Considering the type of programming language used, most embedded systems are programmed in C, C++, and/or assembly language. More sophisticated languages, such as Ada or Java, have found some use, but the need for speed, portability, small code size, and efficient access to the hardware is likely to keep C the dominant language in the foreseeable future [SKO+96]. In embedded system development, several different code sources are often combined, including library code, hand-written assembler, and machine generated C code.

Program constructs used in desktop code differ quite significantly from those used in embedded code. For example, desktop software focusses on arithmetic operations, while embedded software contains more logical and bitwise operations [Eng99b]. The type of algorithms used in embedded systems includes complex decision structures, requiring many mathematical operations. Unstructured code, deeply nested loops, recursion and function pointers is also used in

embedded real-time systems. Much of the complexity comes from automatically generated code, and since the amount of generated code is expected to increase, the problems posed by generated code must be handled.

The most common focus for WCET analysis is user code, but in any system in which an operating system (OS) is used, the timing of operating system services must also be taken into account. Many smaller embedded systems contain no OS, mainly because its demand for system resources are excessive in relation to its function in the particular application. For larger applications responsible for managing several concurrent tasks, it is more common to use an OS. However, compared with those in desktops, the OS's used in such embedded systems are much smaller and include only the functionality needed for handling the particular application. For systems with high demands on predictability and hard timing constraints, it is common to use a real-time OS, such as Enea OSE [Ene03] or SSX5 [Rea03].

1.2 Real-time systems

Real-time systems are computer systems that must react within precise time constraints to events in their environment. The correct behaviour of a real-time system depends not only on the *result* of the computation but also on the *time* at which the result is produced. Most real-time systems are found embedded in products used by people on an everyday basis, as well in more specialized settings such as industrial plants, space shuttles, etc.

As an example of a real-time system, consider a computer-controlled machine on the production line at a bottling plant. The machine's function is simply to cap each bottle as it passes within the machine's field of motion on a continuously moving conveyor belt. If the machine operates too quickly, the bottle will not have arrived. If the machine operates too slowly, the bottle will be too far away for the machine to reach it. Stopping the conveyor belt is a costly operation, because the entire production line must then be stopped. Consequently, the key to correct performance is to have the system running at a steady and predictable pace, i.e., neither too slow, nor too fast.

1.2.1 Hard real-time systems

Real-time systems can be classified roughly as being either *hard* or *soft*. In a hard real-time system, there are one or more activities which must never miss its *deadline*, i.e., the time limit allocated to complete a computation. Failure to meet a deadline could cause catastrophic consequences, including damage to the equipment, major loss in revenues, or even injury or death to users of the system. One example of a hard real-time system is the flight-control system of an aircraft. If action in response to new events is not taken within prescribed deadlines, the aircraft could become unstable, which could potentially lead to a crash.

Another example of a system with hard-real time requirements is the anti-lock braking (ABS) system in a car. When the driver presses the brake pedal the system must actuate the brakes within specified time limits. The computer controlled system must modulate the brake pressure at all four wheels, adjusting the pressure to each wheel independently to prevent wheel locking. If the response time of the system is too high, or if the brake pressures on the different wheels are not correctly correlated, an accident may occur.

1.2.2 Soft real-time systems

In soft real-time systems the meeting of deadlines is desirable, but occasionally missing a deadline has no permanent negative effects.

Consider a cruise-control application in a car, the basic operation of which is to keep a constant speed of the vehicle. If the vehicle is travelling slower than the speed selected by the driver, an embedded computer detects this and sends a signal to the engine controller to accelerate. Similarly, if the vehicle is travelling too fast, the computer detects this and sends a signal to decelerate. The embedded computer needs to sample the speed and send signals sufficiently frequently to meet performance specifications, but not so frequently that it adds unnecessary cost to the system.

If the software occasionally fails to measure the speed in time to be used for the control algorithm, the control algorithm can still use the latest measured value. This is because the amount by which the speed would have changed between the previous sample and the next is so small that the control algorithm can still operate correctly. Missing several consecutive samples, on the other hand, could be a problem, as the cruise control would probably stop meeting application requirements, being unable to maintain the desired speed within a proper error tolerance.

Other examples of soft real-time systems include multimedia, voice over IP and video. For example, in a video playback system it is not fatal to miss an occasional frame, and this is often not even detectable by the user. However, if several subsequent frames are missed, the result would be an annoying blurry picture, but (typically) no one is killed or injured as a consequence of the disturbance. In general, for soft real-time systems, the failure to meet deadlines means that the quality of the service provided is reduced, but the system will still provide useful service.

Considering real-world applications, the distinction of soft and hard real-time systems becomes somewhat fuzzy. For example, an embedded system can have both hard and soft real-time requirements. Actually, the definition of real-time system can be widened to span the spectrum of all computer-based systems [Ste01]. Figure 1.4 illustrates this using some example applications. At one end of the spectrum is non-real-time, where there are no important deadlines (meaning that essentially all deadlines can be missed). These are computer-based systems where the correctness of the result is not really dependent on the point in time when it is produced, such as large computer-based system

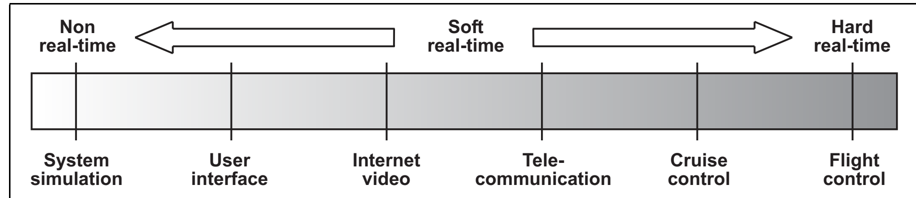


Figure 1.4: The real-time system spectrum

simulations or weather forecast calculations. At the other end is hard real-time, where no deadline is allowed to be missed.

1.2.3 The need for timing analysis

In hard real-time applications, the system must be able to handle all possible scenarios, including peak load situations. The *worst-case* system behaviour must therefore be analyzed and accounted for. If the system is responsible for performing several different concurrent real-time tasks it must be shown that all these tasks can meet their respective deadlines even in the worst-case scenario. For many systems it is important to derive these guarantees *before* the system is put into production. For example, a modern combat aircraft, such as JAS 39 Gripen, contains a number of computers, all which may need to communicate to provide the system functionality [Fre00]. Such aircraft go through very detailed testing and analysis before being used. It is not sufficient to test-fly the aircraft in a certain system configuration to determine if it will be unstable or not.

To derive such overall system timing guarantees, it is necessary to know the execution time demands of the different software tasks in the system. Basically, only if each hard real-time component of the system fulfills its timing requirements can we be sure that the complete system meets its requirements. Thus, WCET analysis provides a solid foundation for constructing safer and better real-time products.

1.3 Execution time estimates

The *worst-case execution time* (WCET) is defined as the longest execution time of a program that could ever be observed when the program is run on its target hardware. There are also other execution time measures that can be used to describe the timing behaviour of a program. The *best-case execution time* (BCET) is defined as the shortest execution time of a program that could ever be observed when the program is run on its target hardware. The BCET can for example, be of interest in control-applications where the output must be sent to the controlled object neither too soon, nor too late. The *average-case execution time* (ACET) lies somewhere in-between the WCET and the BCET, and depends on the execution time distribution of the program.

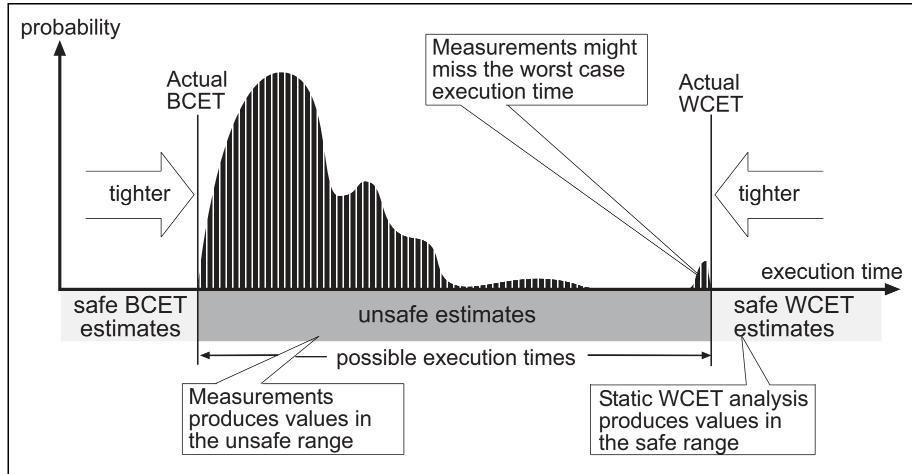


Figure 1.5: Execution time estimates

The goal of execution time analysis is to produce *estimates* of the WCET and BCET. To be valid for use in hard real-time systems, WCET estimates must be *safe*, i.e., guaranteed not to underestimate the real WCET. To be useful, they must also be *tight*, i.e., provide acceptable overestimations of the WCET. Similarly, a BCET estimation should not overestimate the BCET and provide acceptable underestimations.

Figure 1.5 shows how estimates of WCET and BCET relate to the actual WCET and BCET of a program. The example program has a variable execution time, and the curve shows the probability distribution of its execution time. The figure also shows the way measurements and static analysis relate to time estimates (more on this in Section 1.3.3 below).

1.3.1 Problem definition

It should be noted that the definition of WCET is valid only for *one* program in isolation. WCET analysis is therefore performed under the assumption that the analyzed program will be running in isolation and execute undisturbed on the target hardware. This means that interference from background activities, such as direct memory access (DMA) or refresh of DRAM memory are not considered. Similarly, direct interference from the operating system and concurrently running tasks, such as preemptions or interrupts are also ignored in the analysis.

We claim that the assumptions above are reasonable and timing interference caused by such interfering activities should instead be considered in some subsequent analysis, e.g., schedulability analysis [BMSO+96, LHS+96, Sch00]. The problem is thus to derive a safe and sufficiently tight WCET estimate for a single program (task) which executes on a particular hardware platform in a

specific environment.

1.3.2 Sources of execution time variation

The problem that needs to be addressed by WCET analysis is that a computer program typically has no fixed execution time. *Variations* in the execution time occur due to the characteristics of the work the program has to perform and the hardware on which it runs.

Useful computer programs are typically sensitive to their inputs. Consider the Patriot system used to protect military facilities and cities against incoming missiles. The computer system is responsible for detecting an incoming missile, classifying it as a non-friendly object, calculating its trajectory and launching a defensive Patriot missile to intercept the incoming missile. Most of the time, no missile is incoming, and a rather limited amount of computations are needed. However, when a incoming missile is detected a large amount of computation power is needed. Thus, the same software (computer program) can take different amount of execution time depending on the situation.

The hardware on which the program runs is just as important. Obviously, a program runs much faster on a brand new PC than on an old computer. A WCET analysis must consider the timing properties of the particular hardware on which the target program runs. Modern processors are designed to optimize throughput by performance-enhancing features such as caches, pipelines, speculative execution etc. [HP96]. Such features are designed to enhance the average performance, but introduce execution time variability and make it much harder to derive a safe WCET estimate.

In conclusion; both the properties of the software and the hardware must be considered in order to understand and predict the WCET of a program.

1.3.3 Obtaining execution time estimates

The traditional way to determine the timing of a program is by measurements, also known as *dynamic timing analysis*. A wide variety of measurement tools are employed in industry, including emulators, logic analyzers, oscilloscopes, and software profiling tools [Ive98, Ste02]. The methodology is basically the same for all approaches: run the program many times and try different potentially "really bad" input values to provoke the WCET. This is time-consuming and difficult work, which does not always give results which can be guaranteed.

As illustrated in Figure 1.5 measurements are inherently unsafe, guaranteed to produce timing results which are equal to or less than the actual WCET. When using measurements, a safety margin must be added to the result obtained, in the hope that the real worst case lies below the resulting WCET estimate. However, if too much margin is added, resources will be wasted, and if the added margin is too small, the resulting system will be potentially unsafe.

Static WCET analysis avoids the need to run the program by simultaneously considering the effects of all possible inputs, possible program flows, and how

the program interacts with the hardware. This is done by using mathematical models of the software and hardware involved. The result is a worst-case execution time estimate that is greater than or equal to the actual worst-case, and thus safe in all circumstances. The analysis must be repeated after a change in the hardware or software, but the amount of work involved is usually much smaller than for measurements. Also, when using static WCET analysis, there is no need to set up the actual target system.

1.4 Uses of WCET analysis

The main use of WCET analysis is in the development and analysis of real-time systems. In such systems WCET estimates are used to perform scheduling and schedulability analysis, thereby providing timing guarantees for the overall system behaviour, as well as to determine whether timing constraints can be met for certain tasks, and to check that interrupts have sufficiently short reaction times [ABD+95, CRTM98, Gan01]. However, WCET analysis has a much broader application domain; in any product development where timeliness is important, WCET analysis is a natural tool to apply.

Tools for modeling, validation and verification of real-time systems, like UppAal [LPY97], Times [AFM+02], HyTech [HHWT97], Kronos [BDM+98] and SPIN [Hol97] can use WCET estimates to provide guarantees of the overall system behaviour. Typical application areas in which such tools are used include real-time controllers and communication protocols, in particular those in which timing factors are critical.

When developing reactive systems using programming tools such as IAR VisualSTATE, [IAR03], Telelogic Tau, [Tel03], and I-Logix StateMate, [I-L03], feedback relating to the timing of model actions and the worst-case time from input event to output event is very helpful, as demonstrated by Erpenbach et al. [ESS99]. The use of system modelling tools for UML and Statechart [Rat03] could also benefit from accurate timing estimates.

For most embedded system developers, getting some form of timing estimates would be of great value in its own. For time-critical code parts WCET estimates can be used to verify that the execution time is short enough, that interrupt handlers finish fast enough, or that the sample rate of a control loop can be maintained. WCET analysis can also be used to find and target optimizations of the part of the programs where most time is spent. Timing analysis should be able to guide compilers in code optimizations targeting (worst-case) timing of programs.

Another important aspect of embedded software is that only small parts of the applications are usually really time-critical. For example, in a GSM mobile phone, the time-critical protocol code is very small compared to the code for the user interface. Using this fact, ambitious WCET analysis can be performed on the timing-critical parts, provided that they can be identified.

WCET analysis can also be used in embedded system development to select

appropriate hardware. System designers can take the application code they will use and perform WCET analyses for a range of target systems, selecting the cheapest (slowest) chip that meets the performance requirements.

Practical experience of WCET analysis in industry has so far been limited to the space industry [HLS00b, HLS00a] and aerospace industry [FHL+01, TSH+03]. It seems likely that aerospace and automotive industries will be the leading industries in accepting static WCET analysis estimates, since many of their products include resource-constrained embedded safety-critical real-time systems [FHL+01].

1.5 The need for WCET analysis tools

Static WCET analysis is a promising technology that can be used to determine the timing behaviour of programs, especially for programs used in embedded real-time systems. For very simple architectures and programs it is probably possible to derive WCET estimates by hand using code inspection, hardware manual readings and clock-cycle counting. However, due to the complexity of embedded systems hardware and software, *automated tools* are essentail to make it practical to apply static WCET analysis. This thesis will present some steps towards such a tool architecture, including data structures, different analyses, and calculation methods suitable for static WCET analysis.

We believe that a WCET tool should ideally be a component in an integrated development environment, making it a natural part of the embedded real-time programmers' tool chest, the same way as profilers, hardware emulators, compilers, and source-code debuggers. In this way, WCET analysis will be introduced into the natural work-flow of the real-time software engineer. Widespread use of static WCET analysis tools would offer improvements in product quality and safety for embedded and real-time systems, and reduce development time since the verification of timing behaviour is facilitated.

Due to the diversity on the embedded processor market, it is not possible to reach widespread use by only supporting a single target architecture. Instead, there is a need for a WCET tool architecture which is easily *retargetable*, supporting many types of embedded processors and programming environments with minimal retargeting effort. The tool architecture should also be *flexible*, since different target systems require the performance of different types of analyses. The underlying technology needs to be reasonably *efficient*, providing timing estimates fast enough not to stall other development work. Finally, to guarantee the degree of safety of the WCET estimates it must be possible to verify the *correctness* of the analysis methods used.

The WCET tool architecture outlined in this thesis aims at retargetability and flexibility by dividing the WCET analysis task into modules, each with well-defined interfaces, and allowing these modules to be independently replaced. A modular structure also allows the correctness of the tool to be assessed since it is easier to validate the individual modules in isolation. The analysis algorithms

presented have been created with efficiency in mind, limiting the overall tool complexity.

Also, even though static WCET analysis has been known to the research community for some time, it is still difficult to compare the performance and results of the analyses presented by different WCET research groups. A modular WCET tool architecture provides a possibility for researchers to exchange results and compare methods. For example, by having well-defined interfaces between modules, analysis results from one type of tool can be given as input to another tool, allowing each tool to specialize in its particular application domain.

1.6 Contributions of this thesis

The specific contributions of this thesis are:

- A *tool architecture* for the modularization of WCET analysis. The architecture divides the WCET analysis task into modules, each with well-defined interfaces, and allows these modules to be independently replaced. This is an important contribution, since previous work in the WCET analysis area have been more focussed on individual analyses, than on the desired properties of an overall WCET tool architecture. The types of modules in our tool architecture are: *flow analysis;* to determine the possible program flows, *global low-level analysis;* to determine the effects of caches, branch predictors, etc., *local low-level analysis;* to determine the effects of pipelining and to generate execution time for program parts and *calculation;* to combine flow and timing information for calculation of a program WCET estimate.
- A *program flow representation* suitable for WCET analysis. The representation consists of the *scope graph*, a graph representation capturing the dynamic execution behavior of the program, and the *flow fact language*, which is an annotation language used for providing constraints on the program flow. The representation extends the type of flow information previously possible to express and handle in WCET analysis, thereby allowing for calculation of tighter WCET estimates.
- Three different *calculation methods*, each able to use program flow and timing information for deriving a WCET estimate:
  - A *path-based* calculation method which explicitly extracts the longest execution path in the program. Our method is more efficient than previously presented path-based methods and has a computational complexity close to linear in the size of the program.
  - An *implicit path enumeration technique (IPET)-based* calculation method, using integer linear programming (ILP) or constraint programming (CP) techniques for calculating a WCET estimate. The method is able to handle more complex flow and timing information than previously presented IPET methods, thereby allowing for tighter WCET estimates to be derived.

  - A *cluster-based* calculation method using flow information to divide a program into parts in which local WCET calculations can be made. Compared with previously presented calculation methods, we avoid potential complexity problems while keeping the precision of derived WCET estimates.

  The possibility of having three different calculation methods within the same framework pinpoints the benefit of our modular tool architecture.

- A *prototype tool* implementation. The tool is based on the WCET tool architecture outlined and includes machine models for two embedded microprocessors, the NEC V850E and the ARM9. We have performed extensive *experimental runs* to evaluate the correctness, precision and efficiency of our prototype, as well as the individual analyses and calculation modules.

The main focus of this Ph.D. thesis is the overall tool architecture, the program flow representation and the calculation. However, the thesis also contains material on low-level analysis including:

- A *pipeline timing analysis* allowing use of existing trace driven simulators to obtain program timing. Previous research has required the construction of special purpose hardware models to capture timing safely for WCET analysis. The use of simulators reduces the effort required to adapt WCET tools to new hardware architectures and allows for easier verification of the correctness of the hardware model in relation to the real hardware.
- A *timing model* safely capturing the effects of target hardware timing. The timing model allows calculation methods to handle timing effects of different performance enhancing features, such as caches and pipelines, without reverting to detailed hardware modelling. Compared with previous research, the timing model permits calculation methods to safely capture timing effects between instructions in non-adjacent basic blocks, something that has not previously been possible without introducing additional pessimism.

For a more detailed presentation of the timing model and the pipeline analysis we refer to the Ph.D. thesis of Jakob Engblom [Eng02].

1.7 Thesis outline

The remaining chapters of this thesis are organized as follows:

- Chapter 2 gives an overview of static WCET analysis and previous work in the field.
- Chapter 3 presents the modular architecture for WCET analysis tools and gives a short overview of the interface data structures.
- Chapter 4 discusses the issues involved in representing program flow for WCET analysis.
- Chapter 5 presents our flow representation and annotation language.
- Chapter 6 presents our low-level analysis, including the pipeline timing anal-

ysis and the resulting timing model.

- Chapter 7 presents the path-based calculation method.
- Chapter 8 presents the IPET-based calculation method.
- Chapter 9 presents the cluster-based calculation method.
- Chapter 10 presents the prototype implementation and evaluations based on different experimental runs.
- Chapter 11 draws conclusions from the work presented and outlines ideas for future work.

Chapter 2

WCET Analysis Overview and Previous Work

This chapter presents previous work in the area of static WCET analysis, together with a conceptual classification of the phases performed in static WCET analysis.

2.1 Components of static WCET analysis

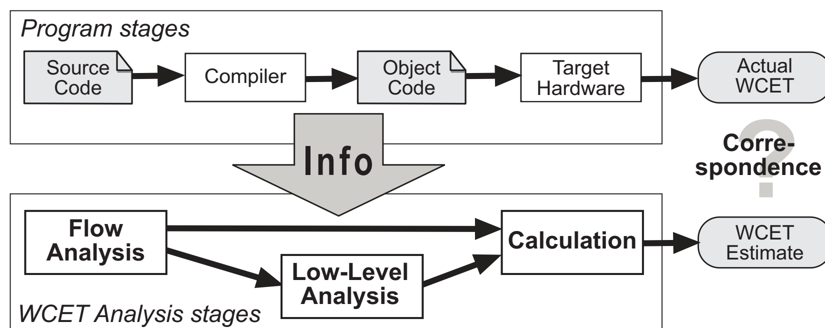


Figure 2.1: Components of WCET analysis

The execution time of a program depends on a number of factors, as illustrated in Figure 2.1. The *program code* defines the possible instructions and execution paths to be executed and the *compiler* transforms the high-level *program source code* to a semantically equivalent *object code*. The object code is executed on the *target hardware* and the *actual WCET* is the largest execution time that could ever be observed when the program is executed.

We divide WCET analysis into the following three distinct phases, closely connected to the different factors that influence the program execution time, and illustrated in Figure 2.1:

- The *flow analysis* analyses the source- intermediate- and/or object code of the program, and determines the possible flows through the program, i.e., the possible sequences of instructions that may be executed.
- The *low-level analysis* analyses the object code and target hardware to determine the timing behaviour for instructions running on the target hardware. For modern processors it is especially important to study the effects of various performance enhancing features, like caches and pipelines.
- The *calculation* combines the results of the flow and low-level analyses to obtain a *WCET estimate* for the program.

The phases serve as a conceptual classification of static WCET analysis and most WCET research groups make a similar division. Some researchers integrate several analysis phases into a single algorithm. Some of the phases can be further divided into different sub-stages, e.g., to analyse different hardware features in isolation. The phase classification is also the base of our modular tool architecture introduced in Chapter 3. The WCET analysis needs input from all the program stages involved in producing the executable program, as illustrated in Figure 2.1.

2.2 Flow analysis

The purpose of the flow analysis phase is to determine possible program flows, i.e., the dynamic behaviour of the program. The result of the flow analysis is information about which functions get called, how many times loops iterate, if there are dependencies between `if`-statements, etc. Since the problem is computationally intractable in the general case[1], a simpler, approximate analysis is normally performed. The analysis should yield *safe* execution information, i.e., all feasible executions must always be covered by the approximation. To be useful, the execution information extracted must also be *tight*, i.e., including as few infeasible executions as possible.

The flow information can be extracted on the source- or object code level and might benefit from information collected during the program compilation. We further divide the flow analysis phase into three sub-phases:

1. *Flow extraction*: Obtaining flow information, either by manual annotations or automatic flow analysis methods.
2. *Flow representation*: Representing the results of the flow extraction, potentially integrating results from several different flow extraction methods.
3. *Calculation conversion*: Converting the represented flow information for the final WCET calculation phase.

[1]The general problem is equivalent to the well-known Halting problem, i.e., that it is impossible to construct a program able to determine if any given program will halt or not.

Not all flow information representations can represent all type of possible program flows and not all calculation methods can take advantage of all type of flow information.

The work presented in this thesis will focus on the last two sub-phases, presenting a general representation for program flow (Section 5) and giving algorithms to convert the flow information to a format suitable for several different calculation methods (sections 7, 8 and 9). No particular flow extraction algorithms will be presented.

2.2.1 Flow extraction

Automatic flow analysis are methods to obtain flow information from the program code with little or no manual intervention. Different approaches have different complexity, generate different amounts of information, and can handle different levels of program complexity. For complex programs it is sometimes very hard (or even impossible) to derive needed flow information, and most automatic flow analysis are complemented with the possibility to provide *manual annotations*. Manual annotations allow the programmer to by hand annotate the program with additional flow information.

Researchers have developed automatic flow analysis methods for detecting infeasible paths[2] and upper bounds for loops.

In the beginning of my doctoral studies I developed a flow analysis method together with Jan Gustafsson [EG97, Gus00]. This analysis is based on abstract interpretation [Cou96, Cou81], works on the program source code level and calculates safe values for variables with respect to loop iterations and function calls. The values are used to derive safe information on loop bounds and infeasible execution paths.

Chapman et al. [CBW94] use symbolic execution, i.e., an execution of a program using symbolic expressions in addition to concrete values, over SPARK Ada to extract program flow information. The method calculates some infeasible paths but manual annotations for loops must be provided.

Altenbernd and Stappert [Alt96, SA00] use symbolic execution on the source code level to derive flow information. The source code is a subset of C. The approach is able to identify some infeasible paths in the program.

Lundqvist and Stenström [LS00] find execution information using symbolic instruction-level simulation of the object code. Their flow analysis is an integrated part of the calculation phase, simultaneously taking pipelining and caching into account.

Colin et al. [CP00] use symbolic evaluation to calculate the number of iterations in inner loops where the iteration count depends on the loop variables of outer loops. However, the initial symbolic formulas must be added manually. Liu and Gomez [LG98] perform symbolic evaluation on a functional language to find executable paths.

[2]An infeasible path is an execution path allowed by the static structure of the program, but not possible when the semantics of the code is taken into account

Healy et al. [HSRW98] use data flow analysis and special algorithms to automatically calculate upper and lower loop bounds for several type of loops. By user-provided loop-invariants the bounds can be further tightened. In [HW99] they present a method using value constraints on variables to find iteration dependent path information inside loops.

Holsti et al. [HLS00b] use Presburger arithmetic to calculate loop bounds for counted loops, analysing programs on the object code level. The approach allows for several types of information (loop bounds, variable value bounds) to be added as annotations to help the automatic flow analysis.

Gerlek et al. [GSW95] present a method for syntactically identifying certain classes of loop induction variables. Such classification is useful for deriving lower and upper bounds of loops.

Ziegenbein et al. [ZWR+01] identify segments of a program that only have a single feasible path by following input-data dependences. Ferdinand et al. [FHL+01] are able to detect some infeasible program paths by analysing the object code using abstract interpretation over processor register values.

2.2.2 Flow representation

The extracted flow information will have to be represented in relation to a *program representation*. The program representation comes in the forms of graphs, syntax trees or program code and can be given in relation to source-, intermediate- or object-code.

Some researchers gives flow information directly or indirectly in relation to the program source code. Kirner et al. [KP01, Kir02] enter manually calculated flow information into the program source code by extending the C language with additional syntax to define scopes, loop limits and path information. Börjesson [Bör95] allows similar flow information to be provided but takes a different approach by `#pragmas` directives instead of altering the language syntax. In [RK02] Kirner et al. includes WCET analysis in the MATLAB/Simulink developing environment by generating their annotated C code from high-level Matlab/Simulink models.

In [CBW94] Chapman et al. extend SPARK Ada, a subset of the programming language Ada83, with additional annotations to facilitate partial proofs of program correctness and WCET calculations. They introduce the concept of *modes*, allowing a program to generate several WCET estimates to reflect a particular system state.

Park [Par93] defines IDL (Information Description Language), to describe the possible paths through a program. IDL uses certain keywords, like `samepath(A,B)` and `nopath(A,B)`, to denote constraints and relate executions of different program entities. The flow information can be given in relation to certain scopes in the graph, for example `always(A) inside L1` means that statement `A` must be executed within `L1`.

Puschner and Koza [PK89] present a program representation in the form of a *syntax tree* (see Section 2.4.1). Flow information is given in respect to

this format, including keywords to express the number of times that several loops are iterated together and the number of times the control-flow can pass through a particular statement. In [Vrc94] Vrchoticky introduce the *timing-tree*; an extension of the syntax-tree to include timing information. The timing-tree works as a format for providing object code timing properties to the source code program level.

Colin and Bernat [CB02] present a program representation, called *scope-tree*, which extends the syntax-tree with a possibility of giving symbolic information on the relative execution frequency of sub-branches in the syntax-tree.

Gustafsson [Gus94] describes WCET analysis for RealTimeTalk, a real-time version of the object-oriented programming language Smalltalk. The program representation is tree-based with additional constructs to support object-oriented features such as message-passing, polymorphism and inheritance.

Recently, many researchers has discovered that the use of linear constraints offers a powerful representation of program flow. The approaches by Li et al. [LM95], Puschner et al. [PS95] and Theiling et al. [TF98] all use the basic block graph and linear constraints to express program flow. The constraints are formulated as constraints over *count variables* holding the number of times program entities can be executed.

Ottosson and Sjödin [OS97] employ Sicstus Prolog constraints over count variables to model possible program flows, thereby allowing more complicated non-linear flows to be expressed, compared to plain linear constraints.

We have developed the *scope graph* program representation, a graph derived from the program basic-block graph, but extended to express the dynamic execution behaviour of the program, and the *flow fact language*, an annotation language used for providing constraints on program flow [EE00]. The scope graph and flow fact language will be presented in Chapter 5.

2.2.3 The mapping problem

One intricate question is to which program code level the flow representation should be related. Flow information can be provided in relation to the source code, the intermediate code in a compiler, or to the object code. Automatic flow analysis is probably easier to perform at the source code or intermediate code, since variables and other entities of interest are harder to identify in (optimised) object code. Also, for the potential WCET-tool user, manual annotations are easier to provide at the source-code level.

However, if the flow information is provided at the source code level, the information must somehow be *mapped* down to the object code to be used in the WCET calculation. In the presence of optimising compilers, this problem is non-trivial since transformations like unrolling loops, inlining functions, and duplicating code can be performed by the compiler [Muc97]. For example, Lundqvist and Stenström [LS98] report a case where entire conditional statements were removed from the program during the compilation process. The mapping problem is very similar to the problem encountered when trying to de-

bug optimised code [Zel84, AT96, Wis94, BW95], something indicating that a WCET tool should preferably be constructed in close cooperation to a compiler framework.

Engblom et al. [Eng97, EAE98] present an approach of an external system that transforms the flow information to reflect the code transformations performed by the compiler. Kirner et al. [KP01] uses the internal debug information propagation facilities in the `gcc` compiler to achieve the same task. In [KP03] the safeness of the flow information transformations is proven using an abstract interpretation framework.

Lim et al. [LKM98] propose an approach where the compiler is assumed to maintain labels identifying relevant locations in the code, allowing loops in the object code to be identified with their source code equivalents. Holsti et al. [HLS00b] rely on compiler generated symbol tables and debug information to map information about loop bounds and variable values from high-level code constructs to the object-code.

One way to avoid the mapping problem is to perform (automatic) flow analysis on the object code of a program [LS00, HLS00a, FHL+01]. However, working on the object code is difficult, since variables migrate between memory and registers, making it difficult to identify and track relevant data objects.

Assuming control over a compiler, it is possible to perform the analysis *inside* the compiler on the intermediate code, which mostly avoids the mapping problem since the analysis can then be performed on the optimised program with full information about variables etc. [HW99, HSR+00]. This also avoids discarding the large amount of program information collected during the compilation process. Following this reasoning, Gustafsson et al. [GLSB03] perform flow-analysis on a compiler-generated intermediate code, thereby avoiding some of the mapping problems.

Another option is to keep the flow information on some source code level and try to assign execution time to the various high-level program constructs [BBMP00, PH98]. For very simple processors and non-optimised code this might be feasible, but due to modern hardware features and compiler optimisations it is extremely difficult to derive such timing which is both safe and tight. The uses of virtual machines in programming languages, like Java and Erlang [AWVW96] complicates the matter even more.

2.3 Low-level analysis

The purpose of low-level analysis is to determine the execution time for each atomic unit of flow (e.g., an instruction, a basic block[3] or a longer execution path) given the architectural features of the target hardware system. To obtain actual timing behaviour of a program the low-level analysis must be performed on the program object code.

[3] A basic block is a sequence of instructions that can be entered only at the first instruction in the sequence and exited only at the last instruction in the sequence [Muc97]

We can further divide the low-level analysis into two sub-phases:

- *Global low-level analysis* determines the timing effect of the machine-dependent factors that needs to be modelled over the entire global program. Examples of such features are *instruction caches*, *data caches*, *branch predictors*, and *translation lookaside buffers (TLBs)*.
- *Local low-level analysis* determines the timing effects of machine-dependent factors that can be handled locally for a few neighbouring instructions. Examples of such effects are *pipeline overlap* and *memory access speed*.

In our tool architecture (see Chapter 3) the global and local low-level analyses are kept as separate steps, and the result from the global low-level analysis is given as input to the low-level analysis. Some researchers integrate the global and local low-level analyses into a single phase. Also, some researchers integrate the low-level analysis with the calculation phase, thereby deriving hardware timing effects simultaneously with the WCET estimate.

2.3.1 Global low-level analysis

For some timing behaviours of a microprocessor, analysis over the whole program (or at least large parts of it) is required in order to obtain a safe and tight result. For example, to determine the cache behaviour of an instruction, the analysis must consider many instructions, arbitrarily remote from the instruction considered. Since exact analysis is normally impossible for global features, an approximate but safe analysis is necessary. For example, when an attempt is made to determine whether a certain instruction is in the cache, a cache miss is assumed unless we can be absolutely sure of a cache hit and that it can be guaranteed that a cache miss is always worse than a cache hit.

Researchers have investigated the behaviour of *instruction caches*, *data caches*, *unified caches* and *branch predictors*.

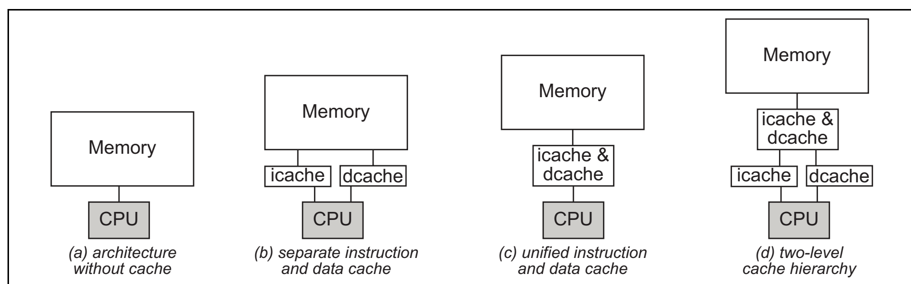


Figure 2.2: Possible memory organisations

Cache analysis

Caches are used as the main solution for bridging the ever-increasing gap between the bandwidth demand of the modern microprocessor and the perfor-

mance of the memory system. For embedded systems designed for predictability caches are not that common. The primary reason is that they are costly both in terms of processor area and power consumption and are considered to introduce too much variability in the system. However, for more high-speed CPUs caches are almost mandatory.

The idea behind a cache is to use a fast intermediate memory between the relatively slow main memory and fast processor to store the most recent blocks fetched from memory. When the processor wants to access a memory block it first checks if the cache contains the block. If so, the access is a *cache hit* and results in a fast access. If not, the access is a *cache miss* and the block is copied from main memory into the cache, where it is stored for this and future uses. Consequently, a cache miss takes much longer time to process since it requires an access to the main memory.

We can distinguish between *instruction caches*, used for providing faster access to the executed instructions, and *data caches*, used for providing faster access to the data manipulated by the instructions. In *unified caches* both the instructions and data are stored in the same cache. To enhance performance further, several levels of caches can be kept between the processor and the main memory forming a *cache hierarchy.* Figure 2.2 illustrates the different concepts.

A cache consists of several locations where blocks from memory can reside. If a block from memory can reside only in one particular location in a cache, the cache is called *direct-mapped.* If a memory block can be placed in several cache locations the cache is called *set-associative.* For set-associative caches a *replacement strategy* is needed to determine which memory block to evict from the cache when adding a new block from memory. Examples of replacement strategies used are least-recently used (LRU), first in first out (FIFO), and random replacement [HP96].

Instruction cache behaviour is rather easy to analyse, since the instruction fetch behaviour can be determined from the program flow (at least when no speculative execution is performed and all instructions addresses are known). Data cache behaviour is harder to determine, since the data access pattern is not fixed but depends on the run-time behaviour of the program, e.g., a single instruction can generate many different data address references.

The first type of cache to be analysed in WCET analysis was direct-mapped caches [LBJ+95, AMWH94], since their behaviour is the easiest to model and predict. Today, most type of analysis are able to handle set-associative caches. Most analyses for instruction caches in the literature [LBJ+95, LMW96, OS97, Sta97, FMW97, HAM+99] assume that the replacement condition is perfect LRU. The uses of less predictable replacement strategies, like pseudo-LRU, may introduce more pessimism in the cache analysis result [Eng02, HLTW03].

Some researchers perform a separate cache analysis phase to determine the instruction cache behaviour. Healy et al. [HAM+99] perform a static cache simulation which generates a categorisation of each instruction cache access as one of *always miss, always hit, first miss, first hit.* The approach has been extended to instruction cache hierarchies with several levels of caches [Mül97]. White

et al. [WMH+97] extends the approach to data caches, classifying instructions in a similar manner, exploiting the data locality inherent in loops over arrays to decrease the number of pessimistic miss categorisations.

Ferdinand et al. [FMW97] use abstract interpretation techniques to conservatively estimate instruction cache hits and misses. The analysis works over a partially unrolled graph allowing the analysis to differ between first and remaining iterations of a loop. Heckmann et al. [HLTW03] presents an extension of the approach working to unified caches and caches with pseudo-LRU replacement strategies.

Some approaches integrate the cache analysis into the calculation phase. Lim et al. [LBJ+95] analyse the instruction cache behaviour by traversing the syntax tree of a program, determining a cache behaviour abstraction for each node. Kim et al. [KMH96] add data cache analysis to the approach.

Li et al. [LMW96] build a cache conflict graph to model conflicting instruction cache accesses. The graph is converted to a set of linear constraints, and solved as part of the calculation phase. Ottosson and Sjödin [OS97] employ constraint techniques to model the execution time gains from using instruction caches and data caches.

Lundqvist and Stenström [LS00] perform instruction and data cache analysis together with flow analysis and calculation in an extended processor simulator. In [Lun02] the data cache analysis is extended to handle more types of complicated data, including data structures with unknown placement in memory.

The cache analysis used by Stappert and Altenbernd [SA00] uses data flow analysis methods to determine instruction and data cache behaviour for programs without loops. The pipeline analysis is performed simultaneously.

Little work has addressed the behaviour of unified caches, mainly due to the complexity introduced by having data and instructions blocks competing for the same locations in the cache. However, Ferdinand et al. [FHL+01] have devised a low-level analysis for a processor employing a unified cache. Value analysis using abstract interpretation is performed on the object code of the program in order to determine the potential target addresses of data access instructions.

An alternative route to limit the unpredictability caused by caches is by locking the contents of the cache. Puaut and Decotigny has performed some work in algorithm for selecting the contents of statically locked caches [PD02]. Vera et al. performs data cache analysis and cache locking for achieving predictable data cache behaviour.

Branch prediction analysis

Branch prediction is another global effect that can have a large effect on the execution time of a program. The outcome of a conditional branch can only be determined quite late in the pipeline and unless some guess is made as to where the program will continue, the pipeline will be stalled until the branch is decided.

Branch predictions are used to guide an out-of-order processor in *speculative execution*. In speculative execution the processor carries on executing instructions in spite of unresolved conditions such as branches or memory accesses. Later, when the condition is resolved, it becomes clear whether the speculation executed can be affirmed. If not, they have to be cancelled and the correct instructions executed.

The simplest form of branch prediction techniques are *static* in that the guesses are statically defined and not dependent on the actual program execution. For example, a simple rule is to assume that all backward branches are always taken, using the fact that programs spend most of their time executing in loops.

To reach higher predictability *dynamic* techniques have been developed. In *one-level* branch prediction each branch is predicted based on its recent execution history of outcomes. Some recent processor designs for desktop and server uses *two-level* dynamic branch prediction techniques [YP93, Gwe95]. In two-level predictors, the recent branch history of several branches is used in order to derive patterns in how a single branch behaves or in how several branch outcomes are correlated.

Colin and Puaut [CP00] model the effect of the local branch predictor scheme for the Intel Pentium processor. The result is a classification of branch predictions similar to the instruction cache reference classification performed by Healy et al. [HAM+99], e.g., always-miss-predicted or first-time-miss-predicted.

Mitra and Roychoudhury [MR01, MR02] model the effect of global branch predictors using linear constraints. The analysis is integrated with the calculation where the effects of branch predictions and the program timing is calculated.

Engblom [Eng03] investigate the execution time variations caused by branch prediction for a number of advanced processors. Using measurements he observes hard-to-predict execution time variation for dynamic branch predictor schemes, and notes that they sometimes cause *inversions*, where executing more iterations of a loop takes less time than executing fewer. The conclusion is that dynamic predictors schemes should be avoided in real-time systems where high execution time predictability is desired.

MMU analysis

For a system using a *memory-management unit* (MMU), there is a need to analyse the worst-case timing of memory accesses in more detail. The MMU is used to translate virtual addresses to physical addresses. A *Translation Lookaside Buffer* (TLB) is a cache holding the most recent address translations, thereby providing a faster mapping. Using a MMU the time required to load a value from memory can be quite high (even for fast memory and no cache), since a TLB miss may require a complex table walk in main memory, involving several memory accesses.

Bernat and Audsley [BA01] present some techniques to reduce the penalty introduced by TLB misses.

2.3.2 Local low-level analysis

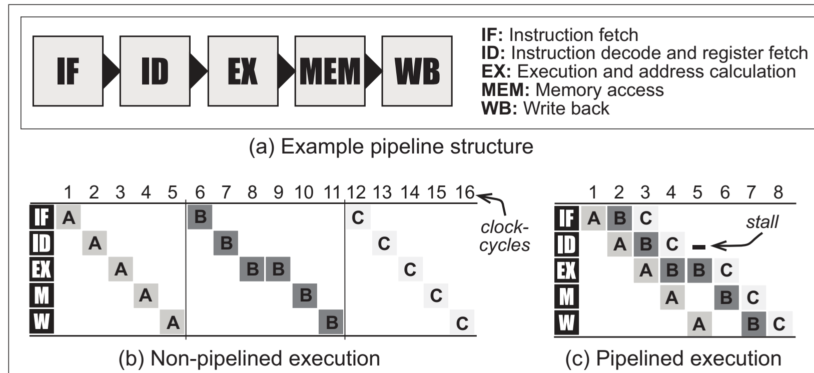


Figure 2.3: Example of pipelined execution

The local low-level analysis handles machine timing effects that depend on a single instruction and its immediate neighbours. Examples of such features are *memory access speed* and *pipeline effects*.

On embedded systems, there are usually several different memory areas, each with different timing. On-chip ROM and RAM are fast, while off-chip memory typically takes several extra cycles to access. A suitable analysis must be able to safely determine which memory accesses that goes to a certain memory area.

Pipeline analysis

The majority of research in local low-level analysis has been directed at pipeline analysis. Just like in global analysis, pessimistic but safe approaches are sometimes necessary, but thanks to the typically simpler behaviour of pipelines, the precision is usually higher for local low-level analysis.

The idea behind pipelining is to increase performance by overlapping the execution of successive instructions. An instruction proceed through a number of stages in order to execute, and while one instruction is in one stage another instruction can be in another stage. Not all instructions need to use all stages and a pipeline can have several parallel paths that instructions can take.

Figure 2.3(a) shows an example pipeline consisting of five stages. In the (`IF`) stage an instruction is fetched from memory, in the (`ID`) stage the instruction is decoded and operands read from registers. In the (`EX`) stage arithmetic operations are performed and the effective target address calculated. In the (`MEM`) stage memory is accessed for data and in the (`WB`) stage the values computed in `EX` or read from memory in (`MEM`) are written back to registers.

Figure 2.3(b) shows an example of non-pipelined execution of instructions `A`, `B` and `C`, where one instruction must complete before the next one can start. The vertical dimension illustrate the stages in the pipeline and the horizontal

dimension represent time. Figure 2.3(c) shows an execution where the instructions overlap in the pipeline. Note that pipelining does not reduce the number of stages needed to complete an instruction, instead it increases the *rate* in which instructions are started and completed, (i.e., the throughput).

Under ideal conditions, the speedup from pipelining equals the number of pipeline stages, but in most cases the speedup is lower due to instruction and resource dependencies. For example, in Figure 2.3(c) the `B` instruction uses the (`EX`) for two clock-cycles causing a stall in the `C` instruction where it has to wait in the (`ID`) stage without doing any work. Examples of dependencies causing stalls are *structural hazards* which occur when two instruction need the same pipeline stage at the same cycle, *data hazards* which occur when one instruction has to wait for some data generated by a previous instruction and *control hazards* which occur when a branch instruction might change the next instruction to execute.

Pipelines come in a wide range of architectural complexity. In general, to reach higher performance, more complex pipelines with higher level of parallelism are required. In simpler pipelines all instructions are executed *in-order*, i.e., in the order specified by the program, and at most one new instruction can be issued per clock-cycle. In *scalar* pipelines the execution can split into multiple branches, e.g., by allowing long running floating point instructions to execute in parallel with the other instructions. A *VLIW* (Very Long Instruction Word) processor allows *several* instructions to start at the same clock cycle. The instructions are processed in-order and are *statically* grouped at compile time. A *superscalar* CPU also allows several instructions to start at the same clock cycle, but the instructions to be issued are *dynamically* grouped during run time. A *superscalar out-of-order* processor allows several instructions to start at the same time, the instructions are issued dynamically and does not need to be issued in the order specified by the program.

The results of global low-level analysis can be incorporated into the final WCET estimate in several different ways. The simplest approach is to assign a certain execution time penalty to bad cases like cache misses, and then add this penalty to the execution time estimate to account for the global effect [LBJ+95, KMH96, TF98, CP00]. A more precise approach is to use the global results as input to the local low-level analysis, and simulate the result of the cache miss on the actual execution of instructions in the processor pipeline [HAM+99, EE99, SA00, FHL+01].

Pipeline reservation tables are commonly used by WCET researchers to provide a detailed model of the behaviour of pipelined processors. Each instruction uses one or several of the pipeline resources in a predetermined manner and the table is used to model the resource usage and interaction between instructions. The concept can be illustrated in a fashion similar to the pipeline diagram shown in Figure 2.3(c).

Atanassov et al. [AKP01] have built a model for the pipelined Infineon C167 processor using constant execution times for each instruction plus formulas that account for the interference between neighbouring instructions, the effect of

memory access times, etc.

Lim et al. [LBJ+95] analyse the pipeline overlap between program basic blocks in conjunction with instruction cache analysis. The target processor is a MIPS R3000/3010, with parallel integer and floating-point pipelines and in-order issue, modelled using reservation tables similar to our pipeline diagrams. Lim et al. [LHKM98] extends the approach to handle in-order superscalar processors. This analysis works over instruction dependence- and latency graphs instead of reservation tables [LHKM98].

Healy et al. [HAM+99] analyse the concatenation of reservation tables for a MicroSPARC 1 processor (similar in complexity to the MIPS R3000/3010), over paths inside loops and functions).

Colin and Puaut [CP01a, CP01a] perform WCET analysis for one of the integer pipelines of a Pentium processor, using reservation tables for pair of instructions.

Altenbernd and Stappert [SA00] performs pipeline analysis for a somewhat simplified model of the PowerPC 604 processor. The pipeline analysis is integrated with the instruction cache analysis and computes a pipeline reservation table for each basic block. Pairs of tables are safely concatenated in the calculation phase to find the longest path.

Wolf and Ernst [WE01] analyse the pipeline behaviour of the scalar StrongARM and SPARClite processors using processor simulators, or special development hardware for the processors, avoiding the need for a special pipeline model. They concatenate pair of paths conservatively.

Schneider and Ferdinand [SF99] analyse the in-order superscalar SuperSPARC I using an abstract pipeline state to model the timing effect of the superscalar pipeline. Ferdinand et al. [FHL+01] analyse the pipeline and cache for the scalar ColdFire 5307 processor, also using an abstract pipeline model.

Lundqvist and Stenström [LS00] perform pipeline analysis for a simplified PowerPC processor with in-order dual instruction issue. The analysis is performed as part of their instruction-level simulation and is integrated with the cache and flow analysis. They use an extended reservation table including information on both pipeline states and possible register and memory contents.

Petters and Färber [PF99, Pet00] perform the pipeline analysis for the superscalar Pentium III and Athlon processors by running code on the real hardware. The analysis is thus intrinsically integrated with the global low-level analysis they use.

Ottosson and Sjödin [OS97] extract pipeline overlap between basic blocks using pipeline reservation tables. The resulting timing yodel assign times to basic blocks and negative timing effects to edges to capture pipeline overlaps both within and between basic blocks. The timing model presented in Chapter 6 is a generalisation of their timing model, allowing for pipeline overlaps between blocks which are not neighbours [EE99].

Lindgren et al. [LHT00, Lin02] uses measurements to derive timing for WCET analysis. They derive a system of linear equations from a number of program measurements. The solutions to these equations gives timing for program

fragments, from which, together with flow information, execution time WCET estimates for the entire program can be derived.

Bate et al. [BBMP00] form a WCET model for a Java Virtual Machine (JVM) by calculating an execution time for each JVM instruction in isolation, and a speedup for each pair of JVM instructions when executed in sequence. The assumption is that a very simple ahead-of-time compiler is used to translate Java byte codes to native code for the target platform.

Lundqvist and Stenström [LS99] note that out-of-order processors with dynamic instruction scheduling are subject to *timing anomalies*. These are cases when a shorter execution time of one or several instructions, e.g., a cache hit instead of a cache miss, might lead to a greater program execution time for the whole program.

Bernat et al. [BCP02] takes another approach to handle interaction of complex hardware features by providing *probabilistic* WCET estimates. Each basic block is given a probabilistic execution time distribution, reflecting that it will not always be executed in the worst case manner. By combining several basic block distributions an execution time distribution for the whole program are constructed.

Heckmann et al. [HLTW03] models the pipeline behaviour of the Coldfire MCF 5307 and the PowerPC processors. The analysis is based on abstract interpretation and works over a model integrating several performance enhancing features like speculative execution, branch prediction, out-of-order execution, caching and pipelining. This approach unifies the global and local low-level analysis into one analysis.

Our pipeline analysis method, outlined in Chapter 6, does not need to explicitly model the internal pipeline behaviour of the processor [EE99, Eng02]. Instead we treat the hardware model as a block box, allowing us to use any trace-driven cycle-accurate processor model (or even the hardware itself) to perform WCET analysis.

2.3.3 Other hardware aspects

There are some other common hardware features that might affect the execution time of a program, including *DRAM refresh*, *task switching*, *interrupts* and *DMA controllers*. As mentioned in Section 1.3.1 these factors are not considered when performing WCET analysis, and should instead be accounted for in some subsequent analysis, e.g., schedulability analysis. However, we need to safely derive the effect these features might have on the execution time.

Dynamic random access memory (DRAM), unlike static random access memory (SRAM), has to be periodically refreshed to retain its data. This refresh will occasionally make the memory unavailable, leading to unpredictable delays to a program. The effect can be measured to be a few percent of the total execution time. Atanassov and Puschner [AP01] outlines an approach to determine the impact of the DRAM refresh on the execution time.

Direct memory access (DMA) controllers transfer data between main memory and I/O devices, often in parallel to instruction execution of the CPU. The DMA controller can slow down the execution by "stealing" bus cycles from the executing program.

In multi-tasking systems the execution can switch between different processes. Using preemptive task scheduling a task can be preempted by higher priority processes. Similarly, interrupts invoke executions of interrupt-handler routines, thereby stopping a currently executing task. The cache and pipeline contents of the preempted task can be radically altered when it continues executing, something which might invalidate previously derived low-level analysis results. There are several proposed techniques to account for cache and pipeline related preemption delays in multi-tasking systems [BMSO+96, LHS+96, Sch00].

2.3.4 Hardware modelling

Almost all WCET researchers relies on *models* of the processor on which a program is executed. For pipelined processors, a variety of modelling methods have been used: cycle-accurate simulators [EE99, ZWR+01, Eng02], special-purpose models using reservation tables or similar techniques to model the behaviour of the processor pipeline [LBJ+95, Sta97, HAM+99, CP01a], dependence graphs [LHKM98], pipeline abstractions [SF99, FHL+01], and tables of instruction execution times and inter-instruction effects [BBMP00, AKP01]. All approaches share the common problem of not using the actual hardware but rather a model of it, and thus bringing the quality of the model into question.

Engblom [Eng02] identifies several error sources in constructing a hardware model for WCET analysis, including hardware bugs, manual writing errors and simulator implementation errors. Another complicating factor is that processor manufactures are not generally providing information on the behaviour of their processors on the detailed level needed to construct a safe WCET analysis model. Engblom concludes that a good hardware model must be validated against hardware and to achieve high agreement between the model and the real target hardware, simple processors should preferably be used.

Montán [Mon00] validated and debugged the simulator for a V850E processor. The validation was made by executing small instruction sequences both on the simulator and on an emulator, comparing the results and identifying discrepancies. The resulting simulator is reported to perform close to the real hardware but still containing some differences in execution time compared to the emulator. The simulator is used in our WCET tool prototype (see Section 10).

Atanassov et al. [AKP01] use measurements on hardware to validate a worst-case execution time model for an Infineon C167. They do not attempt to build an exact simulator for the processor, but rather aim for a safe timing model of the target processor suitable for WCET analysis.

Black and Shen [BS98] validates and fine-tunes a model of a PowerPC 604 processor by comparing executions of small programs both on their simulator

and on real hardware. They report that they never manage to get a perfect correspondence between the simulator and the hardware. Fixing an error in the simulator sometimes revealed other errors that had been masked by the first, and the total error could thus increase for some fixes.

Desikan et al. [DBK01] validated a simulator for the superscalar Compaq 21264 processor. They report on a resulting difference between the target hardware and simulator timing ranging from 2% of small microbenchmarks up to 40% for larger programs in the Spec suite.

The question of how to validate the hardware model definitely becomes more important when the complexity of the hardware increases. Especially, for processors using a lot of performance enhancing features, like caches, pipelines, branch predictors etc. the validation process becomes more complicated, since all modelled interactions between these features have to be validated. The quality of a derived WCET estimate is definitely questionable when the timing of the used model differs significantly from the target hardware. Another complicating factor is that many WCET researchers do not work on a concrete system model, but use abstractions, e.g., merging of cache and pipeline states, something which can not be found in a real processor.

We have been working with hardware model validation in mind, by allowing us to use existing trace driven simulators to obtain program timing (see Section 6.5). We believe that the more concrete and executable the hardware model is, the easier the validation process becomes.

2.4 Calculation

The purpose of the calculation is to calculate the WCET estimate for the program, given the program flow and global and local low-level analysis results. There are three main categories of calculation methods proposed in the WCET literature: *tree-based*, *path-based* or *IPET (Implicit Path Enumeration Technique)*.

2.4.1 Tree-based calculation

In a tree-based calculation, the WCET estimate is generated by a bottom-up traversal of a tree corresponding to a syntactical parse tree of the program. The syntax-tree is a representation of the program whose nodes describe the structure of the program (e.g., sequences, loops or conditionals) and whose leaves represent basic blocks. Rules are given for traversing the tree, translating each node in the tree into an equation that expresses its timing based on the timing of its child nodes. The method is conceptually simple and computationally quite cheap, but encounters problems when handling advanced flow information, since the computations are local within a single program statement and thus cannot consider dependences between statements. Also, the method has problem handling unstructured and optimised code, since it makes the syntax-tree

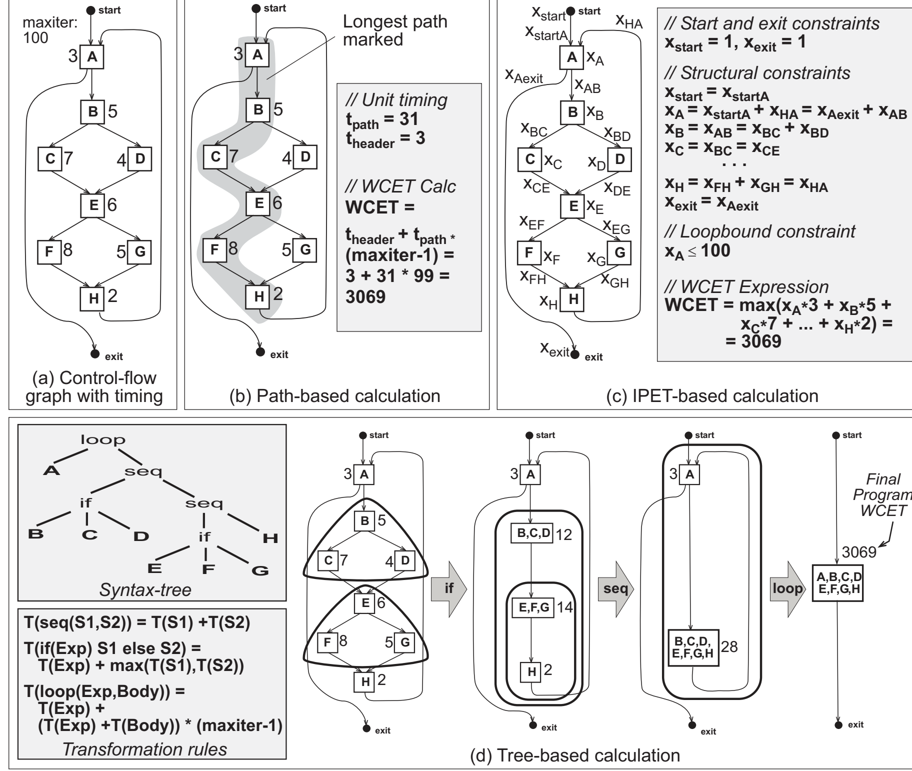


Figure 2.4: Different calculation methods

and transformation rules hard to construct.

Figure 2.4(a) shows an example control-flow graph with timing on the nodes and loop-bound flow information. Figure 2.4(d) illustrates how a tree-based calculation method would proceed over the graph according to the program syntax-tree and given transformation rules. Collection of nodes are collapsed into single nodes, simultaneously deriving a timing for the new node. Since the processing order is pre-defined flow information between non-related program parts, e.g., between `C` and `F`, are hard to handle. Similarly, hardware dependencies between non-local parts of the code are difficult to handle, and must be treated in a pessimistic fashion to guarantee the safeness of the analysis.

The first approach to tree-based calculation was presented by Park and Shaw working on a source-code program level. They dubbed the approach "timing schema" [PS90], using time intervals for the transformation rules, and extracted both BCET and WCET estimates for their example programs. The approach was extended to include pipeline and cache states by Lim et al. [LBJ+95, LHKM98, KMH96]. The method presented by Chapman [Cha95] also builds on a tree-based calculation. Puschner et al. [PPVZ92] use a tree

representation which contains some path and timing information. Bate et al. [BBMP00] perform a tree-based calculation where the number of executions of each type of JVM instruction is propagated, and not actual execution times. Colin and Bernat [CB02] extend the syntax-tree with a possibility of giving symbolic information on the relative execution frequency of sub-branches. Bernat et al. [BCP02] extends the tree-based calculation to include probabilistic timing for the nodes in the tree.

2.4.2 Path-based calculation

In a path-based calculation, the WCET estimate is generated by calculating times for different paths in a program, searching for the overall path with the longest execution time. The defining feature is that possible execution paths are *explicitly* represented. The path-based approach is natural within a single loop iteration, but has problems with flow information stretching across loop-nesting levels.

Figure 2.4(b) illustrates how a path-based calculation method would proceed over the graph in Figure 2.4(a). The loop in the graph is first identified and the longest path within the loop is found. The time for the longest path is combined with the loop bound flow information to extract a WCET estimate for the whole program. The path-based approach is able to capture hardware dependencies between instructions within the same path. For hardware effects going over the analysis boundaries, some type of pessimistic and safe assumption must be made. Some path-based methods have problem handling code containing large amount of branches, since it makes the number of possible paths to consider very large. Another problem for the method is unstructured code, since it makes it difficult to define what constitutes a path or a single iteration.

Healy et al. [HAM+99] look for longest paths inside loops and functions, one loop nesting level at a time, with some special-case handling for cache and pipeline effects between loop levels. Flow information can be used to prune the set of allowable paths, by limiting the number of times a particular path can be executed [HW99].

Stappert and Altenbernd [SA00] investigate the longest paths in a non-looping program, and select the feasible path.

Renaux et al. explicitly extract all the possible paths through a program segment [RGL02]. They use a hardware model consisting of pipeline reservation tables and an abstract cache state.

The method presented by Petters and Färber [PF99] extract a number of paths of interest. The information is used to instrument object code and the timing of the extracted paths are derived using measurements on real hardware.

In Chapter 7 we will present a efficient path-based calculation method. The method is able to handle complex flow information and timing effects between non-local program parts.

2.4.3 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 maximising the sum $\sum_{i \in entities} x_i * t_i$, subject to constraints reflecting the structure of the program and possible flows. The result is a worst-case count for each node and edge, and not an explicit path like in path-based calculation.

Figure 2.4(b) shows the constraints and WCET formula generated by a IPET-based calculation method for the program illustrated in Figure 2.4(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.

IPET constraint systems can be solved using either constraint-solving [OS97, EE99] or integer linear programming techniques (ILP) [PS95, FMW97, HLS00b, LM95], with ILP being the most popular thanks to the availability of efficient solvers. Constraint solving allows for more complex constraints to be expressed, with a potential risk of larger solution times.

The term “implicit path enumeration technique” (IPET) was coined by Malik et al. [LM95], reflecting that the longest path no longer is explicit but implicitly defined. In [LMW96] the approach was extended to simultaneously derive the effects of instruction caches. Puschner et al. [PS95, Pus94] presents a similar approach, with execution time on edges, also showing how some path information can be expressed by numerical constraints. Ferdinand et al. [FMW97, HLTW03] also use IPET, including the timing effects of both pipelining and caches (but not the analysis) in the final calculation. IPET is usually applied across a whole program, but Holsti et al. [HLS00b] apply it on a per-function basis.

Ottosson and Sjödin [OS97] present an IPET-based approach using constraint programming techniques to model possible program flow, pipeline and cache effects in a common framework. Wolf and Ernst [WE01] use IPET over program segments potentially containing several basic blocks. This makes their approach a hybrid between IPET and path-based calculation.

In Chapter 8 we will present an IPET-based calculation method able to handle local flow information and timing effects between non-local parts of the program. Compared to our path-based calculation, our IPET-based approach is able to handle more types of complex flow information.

2.4.4 Parametrized WCET calculation

Sometimes, the input data to a program cannot be bounded in a reasonable fashion, but the data will become known only when the program is executed.

For example, a sorting function in a library will have the data size fixed when called. In such cases, it is possible that instead of generating a single hard WCET estimate, the result of the WCET calculation is a *parametric formula* containing unknowns that are fixed when the WCET estimate is needed in a running system. A simple parametric formula might provide valuable insight into how parameters will affect the program execution time.

Possible parameters are input variables to the program, or entities that stand for unknown program flow information, like an upper bounds on loop iterations in a particular situation. Another example is a real-time operating system, where the WCET of certain interrupt handling routines might depend on the number of processes in the system [CEE+02, CP99]. Another possibility of parametric information is in the hardware timing, e.g., a cache miss might take different time depending on the particular memory configuration used.

There have been a few attempts to construct parametric WCET analysis. Chapman [Cha95] generates a symbolic time expression for a program using a tree-based calculation. Colin and Bernat [CB02] presents the *scope-tree structure*[4] as a way of representing the worst-case execution time of a program symbolically. The approach is tree-based and extends the timing schema calculation model with a possibility of giving symbolic information on the relative execution frequency of subbranches in the syntax-tree.

Vivancos et al. [VHMW01] propose a method for calculating WCET for loops with a parameterised number of loop iterations. A simple symbolic formula for the WCET of a loop is assumed and the loop timing behaviour is iterated until possible convergence. This method takes low-level features like caches and pipelines into account, but is restricted to local analysis of loops and can not take global flow constraints into account. Parametric analysis of nested loops also seems to pose a problem.

Lisper [Lis03] proposes a technique to derive parametric constraints on the program flow. The method is based on abstract interpretation over polyhedra representing possible program variable values. The method outlined propose the use of a parametric integer programming solver to perform a symbolical IPET calculation.

2.5 WCET tools

There have been a number of more or less complete "WCET tools" presented in the literature. Not all research groups have produced a complete and usable tool, and most tools currently available are basically niche products or research prototypes. To call a product a usable WCET tool we believe that there are a number of properties that it should fulfil:

- Basically, it should produce safe and tight estimates of the WCET and provide deeper insight in the timing behaviour of the analysed program and

[4] not to be confused with our scopegraph representation, presented in Chapter 5.

target hardware.

- The tool should be reasonably retargetable, supporting several type of processors with different hardware configurations. It is valuable to provide insight in how different hardware features will affect the execution time.
- The tool should be able to handle optimised and unstructured code. Also, code for which the source-code is not available, e.g., library functions and hand-written assembler, should be possible to handle.
- The tool and analysis should be reasonably automatic, easy to use and should not require any complex user interaction.
- The user should be able to interact with the tool and provide additional information for tightening the WCET estimate, e.g., constraints on variable values and information on infeasible paths.
- The user should be able to specify which part of the code to measure, ranging from individual statements, loops and functions to the whole program.
- The user should be able to view extracted results on both a source code and object code level. The information should provide insight in code parts which are executed, and how often.

Today, there is no tool fully supporting all points above. The main complicating factor is the complexity involved in understanding and safely determining the timing behaviour of modern hardware, but also the limited market for WCET analysis tools. Embedded system developers are in general unaware of the potential benefits of using static WCET analysis.

We believe that a WCET tool should be closely integrated with a compiler, mainly because compilers generate a large amount of information that can be very useful for WCET analysis, but which is not part of standard object-code formats and thus cannot easily be communicated to stand-alone tools. Also, a compiler is the natural place to handle the mapping problem, thereby supporting flow information on both a source and object code level. For time-critical systems a WCET tool might provide feedback helping the compiler to generate code for better timing predictability.

The research group at Florida State University, USA, has attacked a lot of WCET analysis problems, producing a tool which supports many of the above points [HAM+99]. The tool is closely connected to their research *vpo* compiler and can therefore handle optimised code and provide timing information both on a source and object code level [KHR+96, KWH95]. The tool is able to automatically derive some flow information on loop-bounds and infeasible paths [HSRW98, HW99]. Their target processor is MicroSPARC 1. They have developed analysis algorithms supporting several types of caches, including associative instruction caches [HAM+99], multi-level caches [Mül97] and data caches [WMH+97]. The tool is a research prototype and is not commercially available.

The Bound-T tool from Space Systems Finland (SSF) was developed with support of the European Space Agency (ESA) for the use in space applications. They support several target processors, including Intel 8051, Analog Devices

ADSP-21020 and the ERC-32 space SPARC processors. They work on the object code level, having modules for reading and decoding binary files and extracting control-flow graphs. Some flow information is found by automatic flow analysis using Presburger arithmetic. The tool is commercially available, has been used in real systems, and is described in more detail in [HLS00b, HLS00a, HS02].

The AbsInt company [Abs03] is a spin-off from the research at the Universität des Saarlandes, Germany. They provide a tool suite specialised in compiler technology for microcontrollers and DSPs, program optimisation and WCET analysis. A basis for their work is PAG (Program Analysis Generator) [AM95], a tool which supports the generation of static program analysers. Target processors supported for WCET analysis are the Coldfire MCF 5307 and PowerPC 750/755. The analysis works on the machine-code level, reconstructing control-flow graphs from the object code [The01b] and supporting the analysis of several complex hardware features, including speculative execution, out-of-order execution and unified caches [HLTW03]. The tool has several levels of graphical interfaces for detailed hardware analysis information and feedback of WCET execution information [LFW02, TSH$^+$03]. Some flow analysis is performed on the object code level, including dead code detection using range analysis over register values [FHL$^+$01], but loop bounds must be provided manually. The tool is commercially available and has been used at Airbus within EU projects [TSH$^+$03].

The Cinderella tool [Cin97] developed at Princeton University, USA is a research prototype freely available for download. The tool works on the object-code level, and has an interface for providing flow information by linear constraints. Their target processors are the Intel i960KB and Motorola MC68000. Both cache and pipeline behaviour is formulated using linear constraints and solved in the IPET-based calculation phase [LM95, LMW96]. No development has been made in recent years and no attempt to provide commercial support has been made.

The Heptane (Hades Embedded Processor Timing ANalyzEr) tool [Hep03] developed at IRISA ACES team (Ambient Computing and Embedded Systems), Rennes, France is available for download under GPL license. Target architectures supported are Pentium and Mips including analysis of instruction caches, branch prediction and pipeline behaviour. The calculation method used is tree-based. Loops can be annotated using symbolic annotations and evaluated using Maxima, a symbolic computation tool. The tool is is a research prototype and currently under development.

In the following chapters we will present our WCET tool architecture and prototype implementation in more detail.

Chapter 3

A Modular WCET Tool Architecture

To facilitate the design of powerful and portable WCET tools, we have defined a modular WCET tool architecture. The architecture is based on well-defined data structures providing an interface between replaceable modules. The data structures are used to hold the output from different analysis and provide input to others.

3.1 Analysis modules and data structures

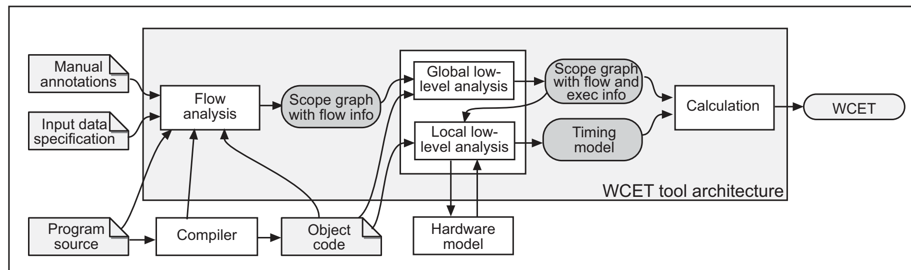


Figure 3.1: WCET tool architecture

Figure 3.1 illustrates our modular WCET tool architecture. Compared to the conceptual view in Figure 2.1, interface data structures have been added and the low-level analysis module has been split into the modules for *global low-level analysis* and *local low-level analysis*. The resulting modules in our WCET tool architecture are:

- *flow analysis:* determines the possible program flow, i.e., the possible sequences of instructions that may be executed.

- *global low-level analysis:* determines the effects of caches, branch predictors, etc. i.e., machine-dependent factors that need to be modelled over the entire global program to be correctly determined.
- *local low-level analysis:* determines the effect of pipelining, i.e., the machine-dependent factors that can be handled locally for a few neighbouring instructions. Generates execution time for program parts.
- *calculation:* combines flow and timing information to calculate a program WCET estimate.

It is possible to combine results of different type of analyses, each adding information to the interface data structures. This makes it easy to extend the tool with new analyses, without disturbing the existing modules.

We use three interface data-structures in our architecture: the *basic block graph*, the *scope graph* and the *timing model*. The basic block graph is a graph representation of the executable program code, the *scope graph* is a graph format for representing program flow and global low-level analysis results, and the *timing model* is a representation for program timing. The following subsections will give a brief introduction to the different data-structures.

3.2 The basic block graph

The object code contains the executable code of the program being analyzed and is target-dependent by nature. The object code can be partitioned into one or more *instructions* and different processors support different instruction sets. Figure 3.2(a) shows an example of a small C function and Figure 3.2(b) shows the corresponding assembler code.

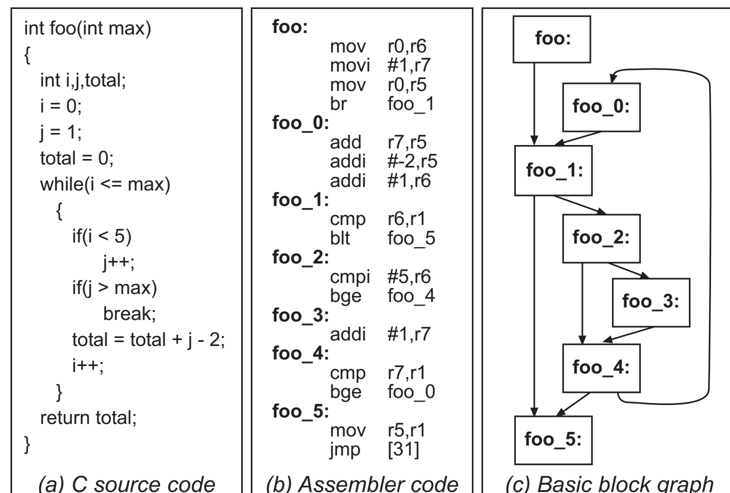


Figure 3.2: Example of basic block graph

All data structures in our tool architecture are based on the possibility of partitioning the instructions in the object code into *basic blocks*. A basic block is a maximal sequence of instructions that can be entered only at the first instruction in the sequence and exited only at the last instruction in the sequence [Muc97]. The generated basic blocks are used to construct a *basic block graph*. Figure 3.2(c) shows the basic blocks graph generated from the assembler code in Figure 3.2(b). The edges between basic blocks represent the potential program flows between the basic blocks. For example, in Figure 3.2(c), there is an arrow from basic block `foo` to `foo_1` since there is an unconditional jump instruction at the end of block `foo` with the first instruction in `foo_1` as target.

The basic block model normally used is the one used in compilers where each machine instruction in the program is only present in one basic block. However, there are some special cases when the basic block model used for WCET analysis needs to be different from the one used by compilers [HLS00b, Eng02].

It is possible to use compiled code or handwritten assembler with our technique, since it is neutral regarding how the basic blocks are generated. For such code, control-flow analysis algorithms [Muc97, ASU86] can be used to construct the basic-block graph. If only executable code is available, like for pre-compiled library code, disassembling and more advanced control-flow graph restructuring techniques, such as the one presented by Theiling [The01a], are needed.

The generated basic block graph could include code from several different files and functions. For programs making use of function pointers or dynamic dispatching it can sometimes be hard at compile time to safely derive the program flow, since we can not know the function(s) that will be called at a particular function call site. If so, analyses for determining which variables and functions that potentially might be referenced by different pointers can be used to construct the basic block graph [EH94, Ste96, Das00, HP00].

3.3 The scope graph

The scope graph is a format used to represent the dynamic execution behaviour of a program. The graph consists of nodes and edges where each node is referring to a basic block in the basic block graph and a basic block might be referenced by several different scope nodes. By separating the flow representation format from the basic block format we are able to represent more dynamic execution behaviour and are not restricted by a particular compiler, instruction set or basic block model.

The nodes in the scope graph are partitioned into *scopes* reflecting the dynamic structure of the program in terms of function calls, loops, recursive calls and unstructured code parts. Scopes are necessary in order to carry program flow execution, in particular bounds for all loops and context-sensitive flow information for function calls. Additional flow information can be added to the scope graph using our *flow fact language*.

Figure 3.3 shows an example scope graph generated from the basic block

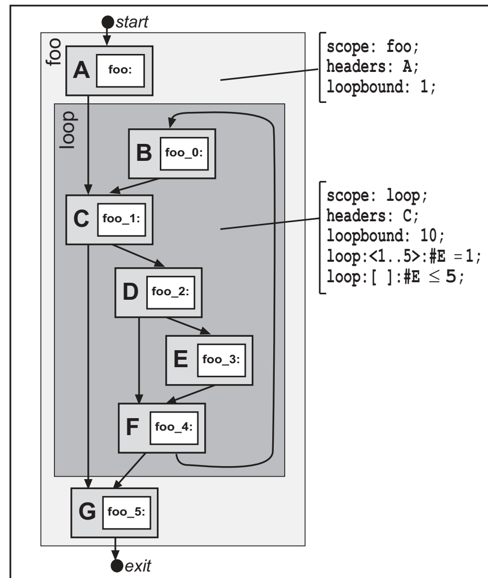


Figure 3.3: Scope graph with flow information

graph given in Figure 3.2(c). For this particular example, the generated scope graph has the same underlying structure as the basic block graph and each basic block is referenced by just one scope node. The scopes in the scope graph correspond to the loop and function found in the basic block graph.

To each scope some flow information has been added. Each scope has a loop bound attached to it, providing an upper bound on the number of times the header node in the scope can be executed for each entry of the scope. The `loop` scope has two additional flow facts. The first flow fact specifies that for each time `loop` is entered, node `E` must be taken during each of the first five loop iterations (but not that the loop needs to iterate 5 times). The second fact specifies that for each time `loop` is entered node `E` can be taken at most five times. Chapter 5 will present the scope graph and flow fact language in more detail.

The scope graph is also used to hold results from global low-level analyses. After applying such analyses the nodes in the scope graph get annotated with information on factors affecting their execution, which cannot be determined from the instructions themselves. An example of such *execution information* is if instructions will hit or miss the cache. Figure 3.4 shows how the nodes in the scope graph given in Figure 3.3 has been extended with such execution information. In Section 6.1 we will outline how results from different global low-level analyses can be added to the scope graph.

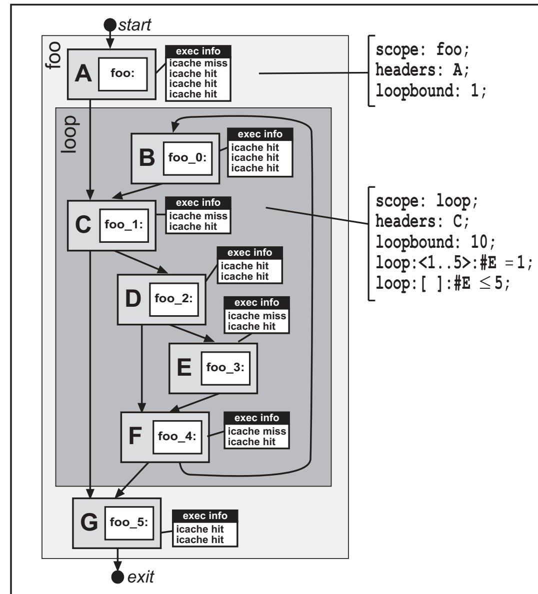


Figure 3.4: Scope graph with flow and execution information

3.4 The timing model

The scope graph is used as an input format to our local low-level pipeline analysis. The analysis derives execution times for a given sequences of instructions annotated with execution facts. We have constructed an analysis which do not require any special-purpose hardware model, but can in principle make use of any trace-driven cycle-accurate *simulator*. Since such processor simulators are standard equipment in embedded system development, the usage of simulators enhance retargetability.

The data structure resulting from the local low-level pipeline analysis is the *timing model*. It is a representation for program timing consisting of times for individual nodes (annotated with execution information), (denoted t_{node}), and timing effects for sequences of nodes, (denoted δ_{seq}). The timing model allows the execution time of a program to be composed from smaller parts, thereby avoiding the need to model complete paths to capture interference between instructions in different basic blocks. The timing model is separated from the hardware model used to derive the timing, i.e., the simulator, thereby avoiding the need for detailed hardware modelling in the calculation. The timing model can alternatively be represented as a *timing graph*: a graph with timing on nodes and timing effects on edges. Figure 3.5 shows the timing model and timing graph generated from the scope graph given in Figure 3.4.

The calculation module takes the input of a scope graph annotated with flow facts, representing possible program flow, and a timing model, representing pro-

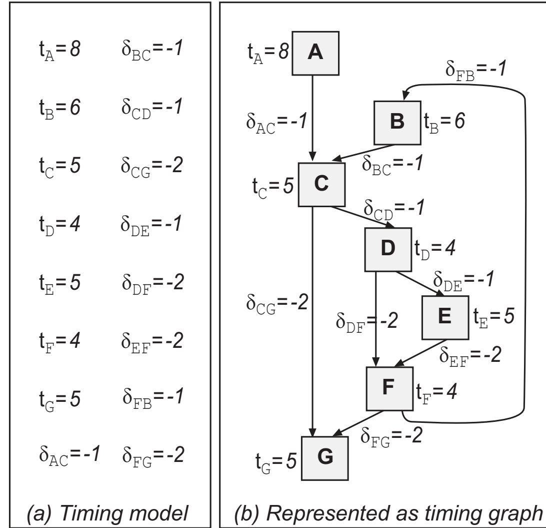


Figure 3.5: Timing model and timing graph

gram timing. Different calculation methods can make different use of flow and timing information, basically trading speed for precision. Chapter 7, Chapter 8 and Chapter 9 will present three different calculation methods. Each calculation method can be replaced independently from the other analysis modules, pinpointing the benefit of our modular tool design.

3.5 Separation vs. integration

We have based our WCET tool architecture on a clear *separation* of modules with distinguished interface data structures. By a modular design the architecture targets the requirements of retargetability, flexibility, efficiency and correctness, as discussed in Section 1.5.

Retargetability is supported since only minor parts of the tool have to be replaced or rewritten to port the tool to new target platforms. Also, the usage of cycle-accurate simulators in the pipeline analysis instead of constructing special purpose hardware models, further supports retargetability.

Flexibility is supported by well-defined and expressive interfaces, making it easy to vary and replace the different modules in our WCET architecture. Also, by having well-defined interfaces data structures we can string together results from various analyses with different complexity and resulting information.

Showing the correctness of a tool is supported by the modular structure since it is easier to validate the individual modules in isolation. To guarantee that a WCET estimate produced by a WCET analysis tool is safe and tight, each analysis phase must be safe and tight in its own right. Otherwise, errors

in one module could mask errors in other modules [EES01]. Also, the usage of cycle-accurate simulators allows for easier validation against the real target hardware, see Section 2.3.4.

Efficiency is supported by separating each type of analysis into a distinguished module, making each analysis simpler and implementation easier. The integration of some analyses may cause excessive solution times due to the complexity of the integrated problem, even if each separate problem is quite easy to solve. For example, some researchers report on analysis times of minutes or even hours even for quite small programs [LMW96, OS97].

So, considering all the benefits of separation why do not all WCET researchers produce modular WCET analysis tools? The main reason is the complexity involved in constructing models for and analyses of modern hardware. For simple processors such as our target architectures, i.e., the NEC V850E and the ARM9, with relative few performance enhancing features, the behaviour of each feature is relatively independent, and can be separately analysed from each other. For example, a cache analysis can provide input on possible cache hits and misses of instructions to the pipeline analysis.

However, for more advanced processors, techniques like instruction prefetching, out-of-order execution and control-speculation introduce interdependencies between different processor components, such as caches, pipelines, prefetch queues, and branch prediction units. For example, in a processor making use of speculative execution, instructions are speculatively fetched and executed. Such instructions might cause cache misses, or even evict "useful" instructions from the cache, even though the speculation turned out to be wrong and the speculatively executed instructions should be discarded.

Performing separate analysis of such processor components might potentially lead to pessimistic WCET estimates. This is because conservative assumptions needs to be made in each individual analysis on the potential state of other interfering components. For modern processors using a lot of such performance enhancing features, the overall pessimism introduced by separate low-level analysis might become very high. The work by Heckmann et al. [HLTW03] contains a good discussion on the need for integrated low-level analysis for advanced hardware architectures. The authors show that for their target processors an integrated global and local low-level analysis obtain much better WCET estimates than the one from separate analysis. However, the integration complicates the design of the analysis and increases the analysis running time and memory demands.

Our modular tool architecture make no direct demands on what type of analysis to perform in the different modules. For complex hardware architectures it is definitely possible to use a more integrated type of hardware analysis, but keep the flow analysis and the calculation modules intact.

Chapter 4

Representing Program Flow

In this chapter we will look more deeply into how program flow should be represented for WCET analysis. The question we will answer are: What characteristics must a suitable flow representation fulfill and what design options are there to choose from?

4.1 Introduction

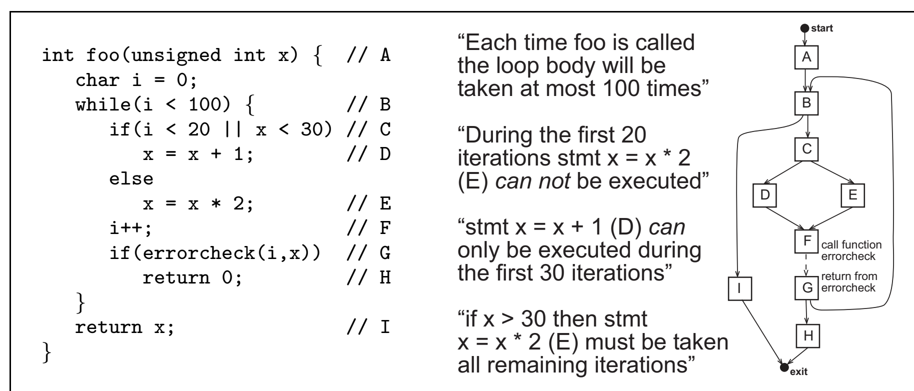


Figure 4.1: Example program, flow information and control-flow graph

Figure 4.1 shows an example function `foo`, its corresponding control-flow graph and some flow information. The capital letters `A`, `B`, etc. indicate the basic blocks found in the control-flow graph to the right.

The flow information provided are examples of properties which we would like to express for the given code fragment. For example, that for each time `foo` is called the loop body can be executed at most 100 times. The bound is just an upper bound, since the execution can leave the function before all iterations

have been executed due to the `errorcheck` outcome. The `i < 20` condition in node `C` gives that statement `E` can not be executed during the first 20 iterations of the loop, and depending on the input value of `x` statement `D` or `E` might be executed during the following 10 iterations.

The given function can be executed in several different ways depending on the possible input values of `x`. The flow information needs to be valid for all these possible executions. Comparing the provided flow information we note the last flow fact is more dynamic in its nature since it depends on the current run-time value of the `x` variable. More detailed knowledge of possible values of input variable `x` might allow us to tighten the flow information.

For this particular example the flow information can be given either in terms of the program code, e.g., statement `x = x + 1`, or the corresponding executable object-code, here illustrated as a control-flow graph, e.g., node `D`. In the case of optimizing compilers the correspondence between the code levels is not always that straightforward.

The given example indicates some things to take into consideration and choices to make when constructing a representation for program flow suitable for WCET analysis. The following sections will discuss and describe these choices in more detail.

4.2 Including all possible executions

One noticeable consequence of performing the flow analysis as an analysis step separate from the low-level analysis and calculation is that a safe flow representation suitable for WCET analysis must include *all* possible executions of the program. This is because without any timing and hardware information we do not know which execution path that gives the worst case timing. Only during the calculation phase, when information on both possible flows and hardware timing are available, certain execution paths can be safely excluded.

```
char bar(char x) {
   if( x < 20 )        // A
      x = x + 1;       // B
   else
      x = x + 2;       // C
   return x;           // D
}
```

Figure 4.2: Simple example program

The most complete form of program flow representation suitable for WCET analysis would therefore be a simple explicit enumeration of all possible execution paths through a program. Every possible execution would be represented using an ordered list of statements or instructions. For example, see the small example program given in Figure 4.2. The function can be executed in two

different ways depending on the input value of x. The two possible executions of the example program are therefore: ⟨A,B,D⟩ and ⟨A,C,D⟩.

One of the strong features of computer programs is that the same code fragment can be repeatedly executed, e.g., using loops or functions. Unfortunately, this makes the set of possible executions paths to grow rapidly in size. The explicit list representation of the dynamic behaviour of a program might be suitable for small programs with one or a few possible execution paths, but is too expensive for most real-world programs. Instead, we need a compact approximate way of representing the dynamic behavior of the program. The approximation must be:

- *Safe* – no feasible executions of the program should be excluded.
- *Tight* – as few infeasible executions as possible should be included.

4.3 Flows information characteristics

Investigating previous presented flow representations for WCET analysis we note that they roughly can be partitioned into two separate parts: a *program representation* and an additional *flow information language*.

The program representation is more or less closely connected to the program code. This can range from including flow information annotation as a part of the programming language or as additional annotations outside the programming language (close connection) to having a special program format separate from the real program (loose connection). The program representation comes in the forms of graphs, syntax trees or program code and can be given in relation to the source-, intermediate- or object- program code. See Section 2.2.2 for a more detailed discussion of program representations.

The flow information language allows for constraining and relating executions of different programs entities found in the program representation, e.g., functions, loops, statements, nodes or edges. The expressiveness of the language gives the type of flows that we are able to express. To be useful the language must have a clear and precise semantics.

Investigating the connection between the program representation and the flow information in more detail we note that some flow information is mandatory and some flow information is not.

For a given program representation the amount of possible executions are given by the *structure* of the program: all execution paths that can be traced through the statements in the program code or the nodes and edges in a program graph are considered possible, regardless of the semantics of the code. This set is usually infinite, since all looping constructs can be taken an arbitrary number of times. For example, for the program given in Figure 4.2 the structure gives that statement B and statement C can not be executed during the same call of function bar.

The set of possible executions are made finite by introducing *basic finiteness information*, where all loops in the code are bound with some upper limit on the

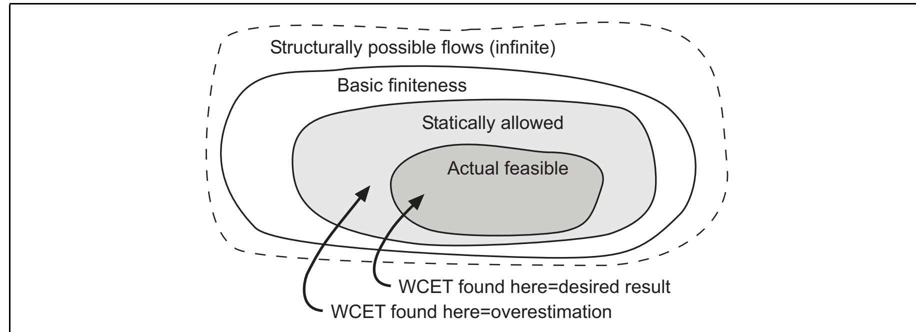


Figure 4.3: Relation between possible executions and flow information

number of executions. Such flow information is required to allow the extraction of a WCET bound for a program. For example, in Figure 4.1 the loop body execution bound provides such basic finiteness information.

Adding more flow information allows the set of executions to be narrowed down further to a set of *statically allowed paths*. From this set the calculation will extract the execution path with the worst case timing. A good flow representation should allow us to make the set of statically allowed paths both safe and tight in comparison to the set of *actual execution paths*. This additional flow information is not mandatory but will allow us to produce tighter WCET estimates. For example, in Figure 4.1 the flow information provided for nodes D and E is not mandatory.

```
int baz(int x) {
   if( x < 5 )         // A
      x = x + 1;       // B
   else
      x = x * 2;       // C
   if( x > 10 )        // D
      x = sqrt(x);     // E
   return x;           // F
}
```

Figure 4.4: Program with an infeasible path

The feasible paths through a program are not always easy to determine. For example, a detailed investigation of the program in Figure 4.4 will conclude that the path $\langle$A,B,D,E,F$\rangle$ is not possible to take due to the limitations on the possible values of the x variable, i.e., if x is less than 5 in A then it can't be larger than 10 after executing B. Such execution paths are called *infeasible* or *false* paths, i.e., an execution path allowed by the static structure of the program, but not possible when the semantics of the code is taken into account.

Knowledge of possible variable values might sometimes allow us to reduce the set of possible executions even further. For example, if the property $x \geq 10$

always hold when calling `baz`, only ⟨`A,C,D,E,F`⟩ is a feasible execution path.

The extreme case of an infeasible path is *dead* code, i.e., a piece of code which is never possible to execute during any executions. Dead code can be safely removed from the program without altering the program behaviour. Flow information on infeasible paths and dead code are not mandatory but will allow us to tighten the WCET estimate.

A infeasible path does not always have to be infeasible. For example, the second flow information given in Figure 4.1 is specifying that `E` cannot be executed during the first 20 iterations of the loop. For the remaining iterations `E` is no longer infeasible.

We conclude that a flow information language should allow us to express both basic finiteness flow information as well as additional tightening flow information, like information on infeasible paths and dead code.

4.4 Expressing flow analysis results

As presented in Section 2.2.1 researchers have developed analyses for automatically deriving program flow information suitable for WCET analysis. A suitable flow representation must be expressive enough to include results from such automatic flow analyses as well as user-provided manual annotations.

An intuitive way of providing flow information is by limiting the number of times different program entities, e.g., loops, statement, nodes or edges, can be taken. This can either be done using precise bounds, e.g., that a loop is iterated exactly ten times, or upper or lower bounds, e.g., that a certain statement cannot be taken more than five times. It is also beneficial if the executions of different program entities can be related, e.g., that node `A` and node `B` will be executed the same number of times.

A language for representing such flow information can consist of named special relations between entities, e.g., using constructs like Park's `samepath(A,B)` [Par93]. An alternative is to use a more generic style based on mathematics, e.g., by relations between variables which represent the number of times certain entities can be taken, e.g., $x_{\mathtt{A}} = x_{\mathtt{B}}$ [LM95, PS95]). The benefit of a generic math-based language is that it can express flows that are hard to put in words and that there is no immediate limit to the types of flows that can be expressed. On the other hand, a special purpose constructs may be easier to understand and express complex relations in a compact way, but requires that new language constructs are invented in order to express new flows.

For a potential WCET-tool user the use of graphical interfaces might offer a third and more suitable paradigm for expressing flow information [KHR+96].

4.5 Managing real-world code

The flow representation must allow us to express the flows found in real-world programs. Researchers have investigated embedded software [Eng99a], the Enea

OSE operating system [CEE+02], the RTEMS operating system [CP01b] and common signal-processing algorithms [EY97], trying to determine the characteristics of the code and program flow for embedded software. The results are not in complete agreement, indicating the diversity of algorithms and coding style employed in the embedded and real-time systems field.

Engblom compared properties of embedded programs and desktop programs and its implications for WCET analysis [Eng99b, Eng99a]. He concludes that embedded code has characteristics different from desktop code. For example, desktop software focusses on arithmetic operations, while embedded software contains more logical and bitwise operations. Most of the investigated embedded code was quite simple, containing single nested loops, simple decision structures, etc. However, he reports on some program instances with highly complex control-flow. For instance, deeply-nested loops and highly complex control structures occur in some programs, and more problematically, recursion and unstructured code [Eng99a].

Colin and Puaut investigated the code properties of the RTEMS operating system kernel [CP01b]. The code contained a small number of loops and no nested loops, unstructured code or recursion. They found some cases of function pointers usage and loops with iteration bounds depending on the current system state, e.g., on the number of tasks allocated in the system. Complicated features found were algorithmic timing behaviour of memory allocation routines, which were dependent on the stack and heap layout.

Carlsson et al. investigated the timing behaviour of disable-interrupt (DI) regions found in the Enea OSE operating system kernel [CEE+02]. Within the DI regions no usage of function pointers and only a few cases of nested loops where found. The authors found program parts whose execution is dependent on the system state and configuration used, e.g., number of tasks and particular flag values.

Ernst and Ye investigated the control flow properties of some signal-processing algorithms [EY97]. They note that even though code contained a lot of decisions and loops, the decisions where written is such a way that there was only a single path through the data, regardless of the input data. Similarly, Patterson [Pat95] notes that most branches in the Spec benchmarks are independent of input data to the program.

For a general purpose WCET analysis tool, the flow representation must be able to correctly express flows found in recursive or unstructured code. Recursion is used and allowed in many programming languages, and even though not recommended, embedded programmers might write unstructured code, e.g., due to uses of `goto`:s and jumps into loops. Also, unstructured (object) code might be produced by program generators, optimizing compilers or programs written as state machines [Eng99a].

We conclude that a suitable flow representation should not be too tightly coupled to a particular programming language or instruction set. This will also make the WCET tool easier to retarget to new target platforms. The representation should be able to handle both unstructured code, recursion and

pointer dependent flows, since such phenomena occur in real-world embedded code.

4.6 Context-sensitive flow information

Many types of program analyses, including WCET flow analysis, can be formulated as *data-flow analyses*. This is the collective name for any analysis, based on the program control flow, which tries to derive information on program behaviour and how it manipulates its data. The analysis is often formulated as a fixed-point iteration over a set of equations defined from the program structure according to the analysis being performed.

Analyses which consider data going in-between functions are called *inter-procedural data-flow analyses*. Investigating such analyses, like pointer analysis [EH94, Ste96, HP00, Das00], we note that the quality of the result might benefit from considering that the data flowing into a function from one call-site is different from the data flowing into the same function at another call-site.

The same reasoning holds for WCET flow information. Assume that the function given in Figure 4.1 is called from several different call-sites in a program. For some of these call sites we might be able to derive more detailed bounds on the input variable `x`, allowing for more tight flow information to be given for these call instances.

```
int main() {
   ...
   foo();
   ...
   bar(32);
   ...
}
```

(a) function `main`

```
void foo() {
   j=0;
   while(j<5) {
      bar(j);
      j++;
   }
   bar(100)
}
```

(b) function `foo`

```
void bar(int b) {
   ...
   for(i=0;i<b;i++) {
      ...
   }
   ...
}
```

(c) function `bar`

Figure 4.5: Code for illustrating different levels of context sensitivity

An analysis which allows us to derive different data for different calling contexts is called a *context-sensitive* analysis. A context-sensitive analysis might produce tighter results, but might be more costly, since a function needs to be analysed multiple times in different contexts.

Look at Figure 4.5 for an illustration of various levels of context-sensitivity. Function `bar` gets called at three different call-sites and contains a `for` loop whose loop bound depends on the input parameter `b`.

Figure 4.6 shows three different call-graphs for three different levels of context-sensitivity, corresponding to the code in Figure 4.5. In Figure 4.6(a) each function only exists in one version in the graph. A flow-analysis working

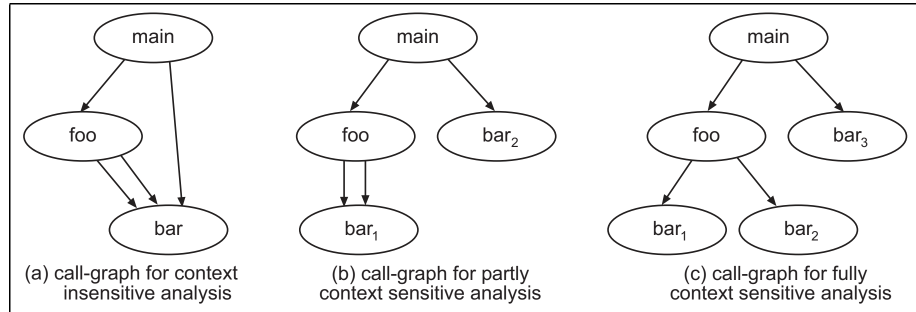


Figure 4.6: Call-graphs with different context sensitivity

over such a graph would derive safe flow information valid for all different invocations of `bar` i.e., an upper loop bound for `bar` of 100. This bound is safe but not tight since the loop would iterate much less times for some invocations.

Figure 4.6(b) shows a context-expanded version of the call-graph in Figure 4.6(a). Two separate copies of `bar` have been created and the context is the function from which `bar` is called. A flow analysis working over this graph would derive a safe but not tight upper loop bound for the loop in $\texttt{bar}_1$ of 100 and a tight upper bound for the loop in $\texttt{bar}_2$ of 32 (see below).

Figure 4.6(c) shows a further context-expanded version of the call-graph. Here function `bar` gets copied for each separate call-site. A potential flow analysis would now derive upper loop-bounds of 5, 100 and 32 for $\texttt{bar}_1$, $\texttt{bar}_2$ and $\texttt{bar}_3$ respectively. Note further that the upper loop-bound given for $\texttt{bar}_1$ is safe but not tight since the loop in $\texttt{bar}_1$ will not always iterate all 5 iterations. A more advanced flow analysis would derive a total upper bound of 15 (i.e., 1+2+3+4+5=15) on the `for` loop, as a total for *all* invocations of `bar` called from the `while` loop.

Such context-sensitive graph expansion can be automated. The most common approach is to consider the *call-string*, the sequence of function calls leading to a certain invocation of a function, as the context [Mar99, The02]. A context does not always need to be a function, e.g., in the instruction cache analysis outlined in [The02] the analysis will differ between the first and remaining iterations of a loop or a recursive function, allowing for a derivation of more detailed instruction cache analysis results (see Section 6.1).

We conclude that WCET flow analyses might benefit from allowing different levels of context-sensitivity and that a suitable flow representation should allow us to represent results from such analyses.

4.7 Flow information locality

Closely related to the concept of context-sensitivity is the ability to provide flow information in a *local scope*. Here one key observation is that flow information

is usually *local* in its nature, specifying something valid for a small part of a program, maybe for a few iterations of a loop or a particular invocation of a function.

For example, the flow information derived for function `foo` in Figure 4.1 is actually valid *each time* the execution enters `foo`. If `foo` is called from several different places in the program the flow information will be valid for *each such call*. The local *scope* for this flow information is therefore function `foo`. If we want to express the same information in a global program scope we first have to derive the number of times `foo` is called and constrain the total number of executions of `D` and `E` in relation to this value.

Other type of contexts suitable for providing local flow information are loops, recursive call nests, and pieces of non-reducible code. Also, the ability to express context-local flow information is suitable when using flow analysis algorithms not working over the complete program at once, e.g., when using a flow analysis algorithm which derives flow information for each function separately.

In loops or recursive code, each statement can be executed a repeated number of times. For such constructs flow information can be valid for a local context consisting of one or a few iterations. For example, flow analysis methods like [Gus00, HW99] derive information on feasible executions valid for some iterations of a loop or surrounding loops. For example, the second and third flow facts given in Figure 4.1 are only valid for some specific iterations.

We conclude that a suitable representation should allow us to provide flow information at varying levels of locality, from the complete program down to a single loop or iteration.

4.8 Dynamic vs. static flow information

As mentioned above, flow information for WCET analysis should be valid for all possible program executions. Flow information can be further divided into two types: *dynamic* and *static* flow information.

Dynamic flow information is information depending on the current run-time state of the program, e.g., depending on variable values or register contents. In Figure 4.1 the last flow fact contains dynamic flow information, since its effect depends on the run-time value of the `x` variable.

Static flow information is information independent on the current run-time program state. The first three flow facts given in Figure 4.1 are examples of such static flow information, since they does not depend on any run-time state values.

Not all calculation methods can make use of dynamic flow information, and such information must therefore be safely converted to a static equivalent. Such conversion might sometimes introduce some pessimism and the resulting static flow information might not include the same amount of information as the original dynamic one. For example, the methods outlined in [HSRW98] and [CP00] allow users or automatic flow analysis methods to provide symbolic formulas

on the execution bounds of loops and other program entities. Using a symbolic computation tool the provided dynamic flow information is converted to static flow information.

Another option is to use a calculation method which can make use of dynamic flow information. During the calculation phase potential unbound variable values will be safely bounded and the provided flow information will be fixed. A parametric WCET calculation method (see Section 2.4.4) goes one step further by allowing flow information to include unbounded flow parameters, e.g., unbounded loop bounds or variable values. When generating a final WCET estimate the unbound flow parameters need to be bounded. For example, the method outlined in [CB02] allows symbolic flow information to be given as input to the calculation phase, potentially generating a simplified parameterised WCET formula. It should be noted that allowing dynamic and parameterised flow information might introduce more complexity into the calculation phase.

```
for(i=0;i<MAX;i++)    // L1
  for(j=0;j<i;j++)    // L2
    for(k=0;k<j;k++) // L3
       A;
```

(a) example program

"For each entry of L1 the loop body will iterate exactly **MAX** times"

"For each entry of L2 its body will iterate as many times as current value of i"

"For each entry of L3 its body will iterate as many times as current value of j"

"For each entry of L3 stmt A will be taken as many times as the loop-body"

(b) dynamic flow info

$$exec(\texttt{A}) \leq \sum_{i=0}^{\texttt{MAX}} \sum_{j=0}^{j<i} \sum_{k=0}^{k<j} 1$$

(c) flow info conversion

"**MAX** is upper bounded by 10"

"For each entry of L1 stmt A will be executed at most 120 times"

(d) static flow info

Figure 4.7: Example of dynamic and static flow information

Figure 4.7 gives an illustration of the difference between dynamic and static flow information. Figure 4.7(a) shows an example program consisting of three nested loops, `L1`, `L2` and `L3`. The number of executions made within the inner loops are dependent on the iteration count value of the surrounding loop. Figure 4.7(b) shows examples of dynamic flow information valid for the given program fragment. `MAX` is a constant not known at compile time. Assuming that the calculation method used can not handle this type of parametric dynamic flow information the information must be converted. Figure 4.7(c) gives an example of a symbolic formula extracted from the dynamic flow information which derives an upper bound on the number of times statement `A` can be executed. Figure 4.7(d) illustrates the resulting static flow information. The value of the `MAX` variable has been upper bounded and the resulting information does not contain the same information as the original one, i.e., the order in which the different executions are made has been lost in the conversion.

4.9 Flow information conversion

Regardless of the flow representation used, the extracted flow information must be converted to a format suitable for the calculation method. As will be discussed in later chapters, not all calculation methods can take advantage of all types of flow information. For example, if the calculation method can handle dynamic flow information such information can be provided as is, otherwise it has to be converted to static flow information suitable for the particular calculation method used.

Similar to the safety and tightness demands on the flow representation the conversion must also be safe, i.e., never exclude execution paths which are considered possible by the flow information, and tight, i.e., including as few extra execution paths as possible compared to the provided flow information. A flow representation close to a calculation format will simplify the conversion but might limit the expressiveness of flow information.

4.10 Conclusions

In conclusion we can distinguish the following desired properties for a flow representation suitable for WCET analysis:

- The representation should allow us to give both basic finiteness and additional tightening flow information.
- The representation should allow us to express flows found in normal code including selections, functions and loops, but also more complicated code including recursion, pointers and unstructured code.
- The representation should not be closely connected to a particular target architecture or programming language.
- The representation should allow us to express flow information from flow analysis methods and manual annotations with varying level of context sensitivity and locality.
- The representation should allow us to express static flow information. Depending on the particular calculation method used dynamic flow information might be converted or provided as is.
- The representation should be possible to convert to a format suitable for the used calculation method. The conversion must be safe and as tight as possible.

Chapter 5

The Scope Graph and Flow Fact Language

In this chapter we present our program flow representation. It consists of a *scope graph*, which is a graph representation capturing the dynamic execution behaviour of the program, and a *flow fact language*, which is an annotation language for providing additional constraints on the program flow.

5.1 Introduction

The scope graph is a directed graph consisting of nodes and edges. Each node in the scope graph refers to a basic block, and one basic block might be referenced by several different nodes. By separating the flow representation format from the basic block format we are able to represent more dynamic execution behaviours and are not restricted by a particular compiler, instruction set or basic block model.

The nodes and edges in the scope graph are partitioned into *scopes*, corresponding to the differentiating and repeating environments found in the basic block graph, including loops, functions, recursive call nests and non-reducible code parts. Scopes carry program flow information, in particular bounds for all loops and flow information with varying level of context sensitivity and locality.

The scopes are organised in a *scope-hierarchy*. We say that a scope c is a *child* of another scope p if the execution of p encapsulates the execution of c or if the execution must pass through p to reach c. The exact scope layout depends on the flow analysis algorithms employed.

To express more precise program flow information, each scope in the scope graph might carry a set of *flow facts*. The flow fact language combines the expressive power of IPET, using constraints to limit possible executions of entities in the scope graph, with the ability to give the flow information in a scope-local context. The given flow information can be further restricted to be valid for

only a few specific iterations of a particular scope.

5.2 The scope graph

A scope graph $sg = \langle N_{sg}, E_{sg}, S_{sg}, start, exit \rangle$ is a directed graph consisting of a *set of nodes* N_{sg}, a *set of edges* $E_{sg} \subseteq N_{sg} \times N_{sg}$, a *set of scopes* S_{sg} and specially identified start and exit nodes. An edge $e \in E_{sg}$ goes from a *source* node, $\texttt{source}(e) \in N_{sg}$, to a *target* node, $\texttt{target}(e) \in N_{sg}$. When referring to a particular edge we use arrow-concatenation of the source and target node names, e.g., edge $u \rightarrow v$ goes from node u to node v. The term *entities* will in the following be used to refer to both nodes and edges.

In the scope graph there exists a special start node, $start \in N_{sg}$, with no predecessors, and a special exit node, $exit \in N_{sg}$, with no successors. Every node in N_{sg} is reachable from $start$ and from every node in N_{sg} it is possible to reach $exit$.

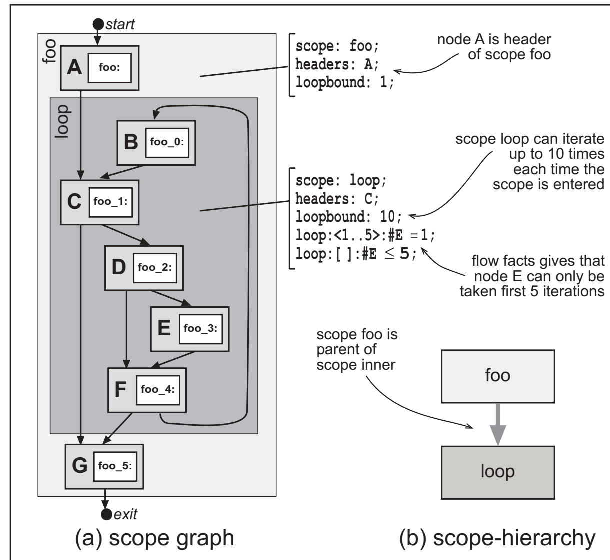


Figure 5.1: Example scope graph and scope-hierarchy

Figure 5.1 shows a scope graph generated from the basic-block graph in Figure 3.2 on page 40. The scopes have been decorated with loop bounds and some flow facts. In this particular example, each node except *start* and *exit* corresponds to exactly one basic-block. For the given scope graph we have that $N_{sg} = \{start, \texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}, \texttt{G}, exit\}$, $E_{sg} = \{start \rightarrow \texttt{A}, \texttt{A} \rightarrow \texttt{C}, \texttt{B} \rightarrow \texttt{C}, \texttt{C} \rightarrow \texttt{D}, \texttt{C} \rightarrow \texttt{G}, \texttt{D} \rightarrow \texttt{E}, \texttt{D} \rightarrow \texttt{F}, \texttt{E} \rightarrow \texttt{F}, \texttt{F} \rightarrow \texttt{G}, \texttt{F} \rightarrow \texttt{B}, \texttt{G} \rightarrow exit\}$ and $S_{sg} = \{\texttt{foo}, \texttt{loop}\}$.

We define the following useful notations for extracting attributes of a scope graph sg. The $\langle type \rangle$ notation is used to clarify the presentation by making the

type of the attribute explicit.

$$\langle \textit{set of nodes} \rangle \ \texttt{nodes(}sg\texttt{)} \equiv N_{sg}$$
$$\langle \textit{set of edges} \rangle \ \texttt{edges(}sg\texttt{)} \equiv E_{sg}$$
$$\langle \textit{set of scopes} \rangle \ \texttt{scopes(}sg\texttt{)} \equiv S_{sg}$$
$$\langle \textit{node} \rangle \ \texttt{start_node(}sg\texttt{)} \equiv start$$
$$\langle \textit{node} \rangle \ \texttt{exit_node(}sg\texttt{)} \equiv exit$$

5.2.1 Scopes

The scopes in a scope graph define a partitioning of the nodes and edges in the scope graph. Intuitively, each scope corresponds to a certain repeating or differentiating execution environment in the program, such as a loop, a function, a recursive call-nest or a piece of non-reducible code, and encapsulates the execution of the program code within this context. For example, the scope graph illustrated in Figure 5.1(a) contains the two scopes `foo` and `loop`.

More formally, a scope $s = \langle N_s, E_s, H_s, C_s, b_s, F_s \rangle$ in a scope graph sg consists of a set of *nodes* $N_s \subseteq N_{sg}$, a set of *edges* $E_s \subseteq E_{sg}$, a set of *header nodes* $H_s \subseteq N_s$, a set of *child scopes* $C_s \subseteq S_{sg}$, an *upper loop-bound* $b_s \in \mathcal{Z}^+$, and a set of *flow facts* F_s.

The nodes N_s of a scope s form a subset of the nodes in the scope graph, i.e., $N_s \subseteq N_{sg}$. Each node in a scope graph belongs to exactly one scope. For example, scope `foo` in Figure 5.1 contains the nodes `A` and `G`. The edges E_s of a scope s are a subset of the edges in the scope graph, i.e., $E_s \subseteq E_{sg}$. Each edge in a scope graph belongs to exactly one scope. An edge is defined to belong to the same scope as its source node, i.e., $e \in E_s \equiv \texttt{source(}e\texttt{)} \in N_s$. For example, the edges `C`→`G` and `F`→`G` in Figure 5.1 both belong to scope `loop`. We will use `scope(`*ent*`)` to denote the scope that the entity (node or edge) *ent* belongs to.

Each scope s has a non-empty set of *header nodes* $H_s \subseteq N_s$ (usually just one node per scope) with the property that no other node in the scope can be executed more than once without passing any of the header nodes. Also, it should not be possible for the execution to enter and later exit a scope without passing one of the header nodes in between. In Figure 5.1 node `A` is header of scope `foo` and node `C` is header of scope `loop`.

Each scope s also contains an upper loop-bound b_s and a set of flow facts F_s, both used for restricting the possible executions over the scope and its descendant scopes. The syntax and the semantics of these different constructs will be described in more detail in the following sections.

We define the following useful notations related to a scope s:

$$\langle \textit{set of nodes} \rangle\ \texttt{nodes}(s) \equiv N_s$$
$$\langle \textit{set of edges} \rangle\ \texttt{edges}(s) \equiv E_s$$
$$\langle \textit{set of nodes} \rangle\ \texttt{headers}(s) \equiv H_s$$
$$\langle \textit{set of scopes} \rangle\ \texttt{children}(s) \equiv C_s$$
$$\langle \textit{integer} \rangle\ \texttt{loopbound}(s) \equiv b_s$$
$$\langle \textit{set of flow facts} \rangle\ \texttt{flow_facts}(s) \equiv F_s$$

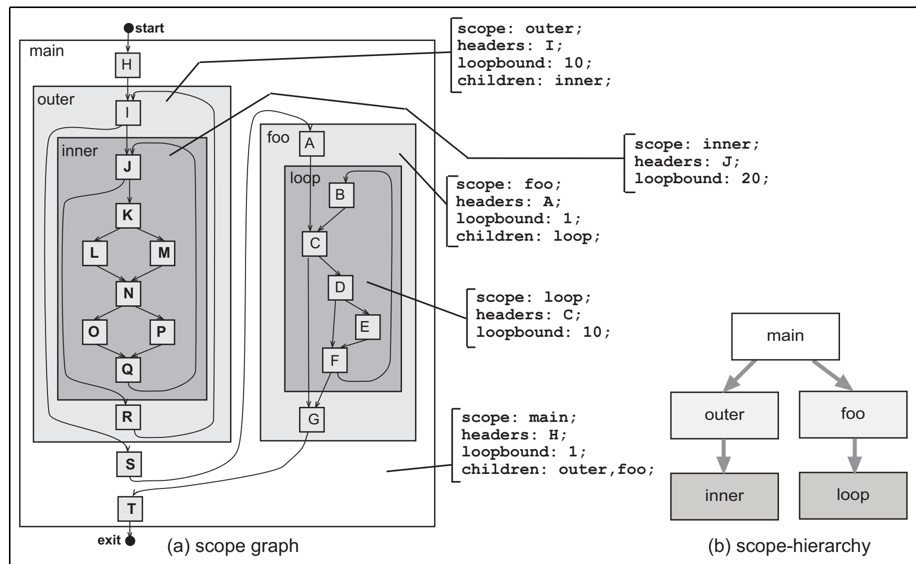


Figure 5.2: Scope graph and scope-hierarchy for two functions

A scope graph could be created for the whole program or some smaller fragment of the basic block graph. This allows us to produce worst case timing estimates for e.g., the complete program, individual functions or smaller parts of the program. Figure 5.2 shows an extension of the scope graph given in Figure 5.1 where function `foo` is called by function `main`. Function `main` includes two nested loops, resulting in scopes `outer` and `inner`. No flow information except loop bounds is included.

5.2.2 The scope-hierarchy

The scopes in the scope graph are partially ordered by the *child* relation. Roughly, a scope c is a child of a parent scope p if the execution within p encapsulates the execution of c. For example, c might correspond to a loop whose body is completely nested within a surrounding loop p. In Figure 5.2

the code in scope `inner` is nested within the code of scope `outer` and therefore `inner` is a child to scope `outer`. Another possibility is that c corresponds to a function called from within another function p. In Figure 5.2 `foo` is called within scope `main`, and therefore scope `foo` becomes a child to scope `main`.

Using the child relation the scopes in the scope graph are organized in a *scope-hierarchy*, a directed tree with scopes as vertices and edges from a scope to all its children. There exists a special *root scope* from which all scopes in the scope-hierarchy are reachable. The only nodes in the root scope are *start* and *exit*. The root scope will not be explicitly shown in our figures. Figure 5.2(b) shows the scope-hierarchy for the scope graph in Figure 5.2(a).

We use the child relation to define some other relations between the scopes in the scope-hierarchy. The *parent* of a scope s is the scope in the scope-hierarchy which has s as a child. All scopes except the root-scope has a parent. The scope being parent to a scope s is defined as:

$$\langle \textit{scope} \rangle \ \texttt{parent(}s\texttt{)} \equiv p : s \in \texttt{children(}p\texttt{)}$$

Each scope has zero or more *ancestor scopes*, i.e., scopes above it in the scope-hierarchy. The immediate ancestor is the parent scope. For example, in Figure 5.2 on page 62 scopes `main`, and `foo` are ancestors to scope `loop`. Scope `foo` is a parent of scope `loop`. The set of ancestor scopes of a scope s is recursively defined as:

$$\langle \textit{set of scopes} \rangle \ \texttt{ancestors(}s\texttt{)} \equiv \\ \{\texttt{parent(}s\texttt{)}\} \cup \texttt{ancestors(parent(}s\texttt{))}$$

Each scope has zero or more *descendant scopes*, i.e., scopes below it in the scope-hierarchy. The immediate descendants of a scope s are its child scopes, C_s. For example, in Figure 5.2 on page 62 scopes `outer`, `inner`, `foo` and `loop` are descendants of `main`. The set of descendants of a scope s is defined as the set of scopes having s as ancestor:

$$\langle \textit{set of scopes} \rangle \ \texttt{descendants(}s\texttt{)} \equiv \{d \mid s \in \texttt{ancestors(}d\texttt{)}\}$$

A scope without any descendants is called a *leaf scope*. For example, in Figure 5.2 scopes `inner` and `loop` are leaf scopes. The leaf scopes in a scope graph sg are defined as follows:

$$\langle \textit{set of scopes} \rangle \ \texttt{leaves(}sg\texttt{)} \equiv \{s \mid s \in \texttt{scopes(}sg\texttt{)} \wedge \texttt{descendants(}s\texttt{)} = \emptyset\}$$

The *complete subtree* for a scope s is the set of scopes consisting of s and all scopes in the scope graph having s as ancestor. The complete subtree is a subset of the scopes in the scope-hierarchy in the form of a tree with s as root. For example, in Figure 5.2 on page 62 scopes `main`, `outer`, `inner`, `foo` and `loop` are all included in the complete subtree of scope `main`. The set of scopes included in the complete subtree of a scope s is defined as follows:

$$\langle \textit{set of scopes} \rangle \ \texttt{complete_subtree(}s\texttt{)} \equiv \{s\} \cup \texttt{descendants(}s\texttt{)}$$

A set of scopes *sub* is a (complete or non-complete) *subtree* of a scope s if *sub* forms a connected tree with s as root. For example, {`main`}, {`main`,`outer`}, {`main`,`outer`,`foo`} and {`main`,`outer`,`inner`} are examples of sets forming subtrees with `main` as root in the scope graph given in Figure 5.2. A set of scopes *sub* is a subtree of a scope s if the following predicate is true.

$$\begin{aligned}&\langle \textit{boolean} \rangle\ \texttt{is_subtree(}s, sub\texttt{)} \equiv\\ &\quad s \in sub \wedge\\ &\quad sub \subseteq \texttt{complete_subtree(}s\texttt{)} \wedge\\ &\quad \forall c \in (sub - \{s\}) : \texttt{parent(}c\texttt{)} \in sub\end{aligned}$$

5.2.3 Special scope entities

For each scope in the scope graph we define some special nodes or edges useful for providing flow information. One example is the header-nodes which allow us to provide information on iterations within the scope.

An *in-edge* $e = u \rightarrow v$ of a scope s is an edge having a source node u in a scope not included in the complete subtree of s, and having target node v in a scope included in the complete subtree of s. Note that an in-edge of a scope s does not need to have its source node in a parent scope of s, and does not need to have its target node within s. For example, in Figure 5.2, the edge `H`→`I` is an in-edge of scope `outer` and edge `I`→`J` is an in-edge of scope `inner`. The set of in-edges for a scope s is defined as follows:

$$\begin{aligned}&\langle \textit{set of edges} \rangle\ \texttt{in_edges(}s\texttt{)} \equiv\\ &\quad \{e \mid \texttt{scope(source(}e\texttt{))} \notin \texttt{complete_subtree(}s\texttt{)} \wedge\\ &\quad\quad \texttt{scope(target(}e\texttt{))} \in \texttt{complete_subtree(}s\texttt{)}\}\end{aligned}$$

An *in-node* of a scope s is a target node of an in-edge. A scope can be entered at several in-nodes, allowing for arbitrary code structuring, for example unstructured jumps into loops and functions. Note that an in-node of a scope s does not need to be a header-node of s. In Figure 5.2 node `B`, `C` and `K` are in-nodes of `outer`, `inner` and `foo` respectively. The set of in-nodes for a scope s is defined as follows:

$$\langle \textit{set of nodes} \rangle\ \texttt{in_nodes(}s\texttt{)} \equiv \{n \mid \exists e \in \texttt{in_edges(}s\texttt{)} : n = \texttt{target(}e\texttt{)}\}$$

An *out-edge* of a scope s is an edge having its source in a scope within the complete subtree of s and having its target in a scope outside the complete subtree of s. A scope might have several out-edges, allowing for non-local jumps out of loops and multiple function return sites. An edge can be an out-edge of several scopes. In Figure 5.2 on page 62 edges `I`→`S` and `J`→`R` are out-edges of `outer` and `inner` respectively. Edges `C`→`G` and `F`→`G` are both out-edges of scope

loop. The set of out-edges for a scope s is defined as follows:

$$\langle \textit{set of edges} \rangle\ \texttt{out_edges}(s) \equiv \{e \mid \texttt{scope(source}(e)\texttt{)} \in \texttt{complete_subtree}(s) \wedge \texttt{scope(target}(e)\texttt{)} \notin \texttt{complete_subtree}(s)\}$$

An edge going to a header node of a scope s and having a source node in a scope located in the complete subtree of s is a *back-edge* of s. In Figure 5.2 on page 62 edges R→I, Q→J and F→B are back-edges of `outer`, `inner` and `loop` respectively. The set of back-edges for a scope s is defined as follows:

$$\langle \textit{set of edges} \rangle\ \texttt{back_edges}(s) \equiv \{e \mid \texttt{target}(e) \in \texttt{headers}(s) \wedge \texttt{scope(source}(e)\texttt{)} \in \texttt{complete_subtree}(s)\}$$

5.3 Loop bounds

To limit the set of possible executions over a scope graph, every scope s must be given a *loop-bound* b_s. A loop-bound is an upper limit on the number of iterations which can be executed within the scope each time the scope has been entered. It is an upper bound valid for *each possible entry* of the scope. We express a loop-bound by an integer larger than or equal to zero. An *iteration* of a scope s is defined to start when a header of s is executed and end when a back-edge or an out-edge of s is executed.

Figure 5.3 gives the grammar of loop-bounds and the semantics of loop-bounds is given in Section 5.5.4.

LoopBound	→	`loopbound:` $\mathcal{Z}^+$

Figure 5.3: Loop-bound grammar

Note that a loop-bound should be interpreted as a local constraint, i.e., not as a bound on the total number of times a loop is iterated through the whole program, but a bound valid for each entry of the scope, independent of the in-edge or the invocation history executed before entering the scope. If the scope can be entered several times, e.g., being the inner loop in a loop-nest, the loop-bound given must be a safe upper loop bound valid for each such entry. For example, a loop-bound of 20 for scope `inner` in Figure 5.2(b) means that `inner` can iterate at most 20 times each time scope `inner` is entered.

Further restrictions on possible flows, e.g., lower loop-bounds, different loop-bounds for different entries, or bounds valid for the complete program execution should be given using flow facts.

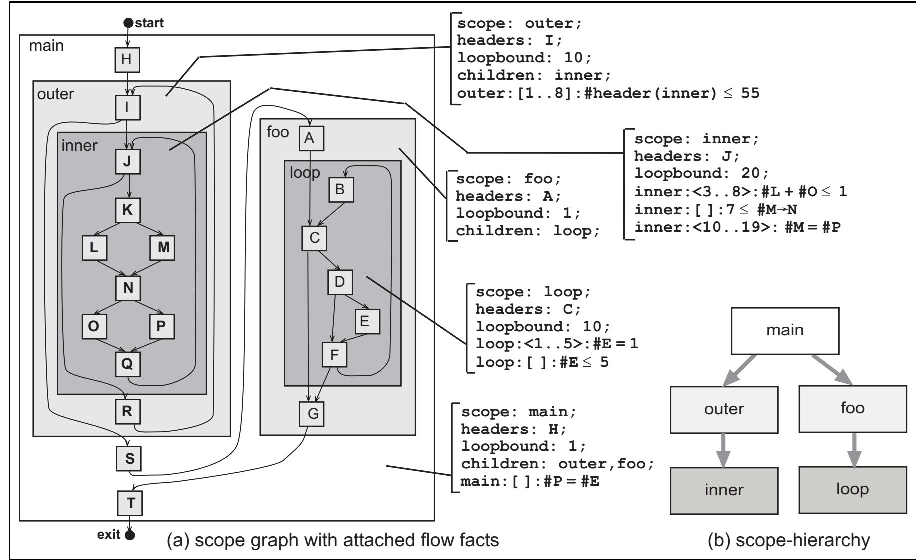


Figure 5.4: Scope graph and flow facts

5.4 Flow facts

To express more complex program flow information than just basic loop-bounds each scope s can carry a set of *flow facts* F_s. The flow facts combine the expressive power of IPET, using constraints to limit possible executions of scope graph entities, with the ability to give the flow information in a scope-local context. The given flow information can be further restricted to be valid for only a few specific iterations of the particular scope. Figure 5.4 shows the scope graph given in Figure 5.2 with some attached flow facts.

Each flow fact $f \in F_s$ consists of three parts, a *defining scope*, a *context specification* and a *constraint expression*. Intuitively, the constraint is only defined for the executions within the complete subtree of the defining scope when the execution is within the iterations given by the context specification.

For example, flow fact `loop : <1..5> : #E` $= 1$ in Figure 5.4 has `loop` as defining scope, `<1..5>` as context specification and `#E` $= 1$ as constraint expression.

Figure 5.5 gives the grammar of the specification language for flow facts. For each non-terminal *NT* in this grammar and flow fact f we will use `NT(`f`)` to denote the corresponding expression, e.g., `constraint(`f`)` gives the constraint expression of flow fact f and `min(range(context(`f`)))` gives the *min* iteration of the range in the context specification of flow fact f.

FlowFact	$\rightarrow$	*Scope* : *Context* : *Constraint*
Scope	$\rightarrow$	*Id*
Context	$\rightarrow$	`< >`
	|	`<` *Range* `>`
	|	`[ ]`
	|	`[` *Range* `]`
Range	$\rightarrow$	*Min* . . *Max*
	|	$\mathcal{Z}^{+}$
Min	$\rightarrow$	$\mathcal{Z}^{+}$
Max	$\rightarrow$	$\mathcal{Z}^{+}$
Constraint	$\rightarrow$	*Expression Relop Expression*
Expression	$\rightarrow$	*CountVariable*
	|	*Integer*
	|	*Expression Op Expression*
Relop	$\rightarrow$	$\leq$ | $=$ | $\geq$
Op	$\rightarrow$	$+$ | $-$ | $*$

Figure 5.5: Flow fact grammar

5.4.1 Defining scope

The *defining scope* is the scope over which the flow fact will constrain program execution, i.e., the constraint should hold for the executions within the complete subtree of the defining scope. Like loop-bounds, each flow fact is considered *local* to its defining scope, meaning that the fact is only allowed to constrain executions of entities located in the complete subtree of the defining scope. We will illustrate the defining scope of a flow fact by graphically attaching the flow fact to this scope.

5.4.2 Context specification

Operator	**Type**	**Iterations**
< >	foreach	all
[ ]	total	all
<*range*>	foreach	range
[*range*]	total	range

Figure 5.6: Context specification operators

The *context specification* gives the iterations for the defining scope over which the executions of entities should fulfil the given constraint. The type of the context is either *total*; written as [ ] or [*range*], or *foreach*; written as < > or <*range*>.

For a total context, ([ ] or [*range*]), the sum of all executions of entities

within the given range should together fulfil the given constraint. For example, flow fact `inner : [3..8] : #F` $\rightarrow$ `G` ≥ 4 specifies that for each entry of `inner`, when the iteration counter of `inner` is between three and eight, the edge `F`$\rightarrow$`G` must be taken at least four times.

For a foreach context, (`< >` or <*range*>), the execution of entities within *each* iteration within the given range should fulfil the given constraint. For example, flow fact `loop : <6..10> : #N` $= 0$ specifies that when the iteration counter of `loop` is between six and ten, for each such iteration, node `N` can't be executed.

A range r in a context specification for a fact with defining scope s specifies that the constraint should be valid when the execution of s (and its descendant scopes) is within the range of iterations given by r. To better understand the ranges imagine that each scope s has a fictitious *iteration counter* holding the current iteration number of s. The counter is incremented with one every time one of the header nodes of s are taken and reset to zero whenever the execution leaves the complete subtree of s. A range is written as $min..max$, where $min \leq max$ and $min, max \in \mathcal{Z}^+$ and a range specification valid over a single iteration can be written as just i, instead of $i..i$.

Note that a ranged fact does not specify that the execution must enter the range if the defining scope is entered, just that *if* the range is entered then the constraint should hold for the executions made within the range. For example, range 3..5 for a scope s tells us that the corresponding fact should be valid when the iteration count of s is between three and five, but it does not specify that we must execute three or more iterations in s when entering s.

`[ ]` and `< >` is a more compact syntax to specify that a fact should be valid for *all* iterations of the defining scope. E.g., flow fact `loop : <> : #N` $+$ `#P` ≤ 1 specifies that for no iteration of `loop` both of the nodes `N` and `P` can be executed. We define predicate `is_total(`f`)` to be true if flow fact f has a total context and predicate `is_foreach(`f`)` to be true if f has a foreach context.

5.4.3 Constraint expression

The constraint in a flow fact is specified as a relation between two arithmetic expressions involving *execution count variables* and *constants*. An execution count variable, `#`*entity*, corresponds to an entity (node or edge) in the scope graph, and represents the number of times the entity is executed in the context given by the context specification.

A fact can only refer to count variables corresponding to entities located in the complete subtree of the defining scope of the fact. For example, flow fact `main : [ ] : #P` $=$ `#E` in Figure 5.4 is defined in scope `main`, but refers to executions of entities located in scope `inner` and scope `loop`. But, a fact in `inner` could not refer to node `H`.

For each scope s we define a special header count variable `#`$header(s)$, referring to the number of times the header nodes of scope s are executed within the context given by the fact that uses the variable. The `#`$header(s)$ count variable can be used to limit the number of iterations of a scope beyond the basic

limitation offered by the loop-bound. If the scope has several header nodes the header count variable will refer to a sum of header node executions within the context given by the fact.

For each scope s we also define an entry count variable, $\#entry(s)$, referring to the number of times s is entered via any of its in-edges. If the scope can be entered at several different in-edges the entry count variable will represent the sum of executions of these in-edges in the context given by the flow fact. Consequently, the entry count variable $\#entry(s)$ can only be referred to by facts with defining scope among the ancestors of s.

We do not make any direct demands on the type of constraint expression allowed. Basically, the type of constraint expressions needed depends on the type of flow information that we should express and the power of the calculation method. More advanced flow information requires more advanced calculation methods.

5.4.4 Flow fact examples

We now explain the meaning of the example flow facts attached to the scope graph in Figure 5.4.

- The flow fact `loop : <1..5> : #E = 1` expresses that for each entry of `loop`, for each iteration 1 to 5 of `inner`, node `E` must be executed if the iteration is executed.
- The flow fact `loop : [] : #E ≤ 5` expresses that for each entry of `loop`, node `E` can not be executed more than 5 times.
- The flow fact `inner : <3..8> : #L + #O ≤ 1` expresses that, for each entry of `inner`, for iteration 3 to 8 of inner it is not possible to take both node `L` and node `O`. This means that during the covered iterations any execution passing both `L` and `O` is infeasible.
- The flow fact `inner : [] : 7 ≤ #M→N` expresses that for each entry of `inner`, edge `M→N` must be taken at least 7 times.
- The flow fact `inner : <10..19> : #M = #P` expresses that for each entry of `inner`, for each iteration between 10 and 19, node `M` must be taken the same number of times as node `P`.
- The flow fact `outer : [1..8] : #header(inner) ≤ 55` is defined in scope `outer` but refers to the header node of scope `inner`. It expresses that for each entry of scope `outer`, for the first eight iterations of `outer`, the header node of `inner` can not be taken more than 55 times. This is useful for expressing a triangular loop behaviour.
- The flow fact `main : [] : #P = #E` is defined in scope `main` but refers to entities located in scopes `inner` and `loop`. It expresses that for each entry of scope `main` node `P` should be taken exactly the same number of times as node `E`.

Observe that all the flow facts in the scope graph should be valid, i.e., there is an implicit conjunction of the flow facts. Also, note that flow facts are allowed to overlap in the ranges of their context specifications and refer to the same execution count variables. This is useful when we want to combine the results from several different flow-analysis methods.

5.5 Loop-bound and flow fact semantics

In this section we define semantics for our flow information language. We will define the semantics of loop-bounds and flow facts as *restrictions* on the program flow. The exact restrictions are defined by reasoning over all possible execution paths in a scope graph.

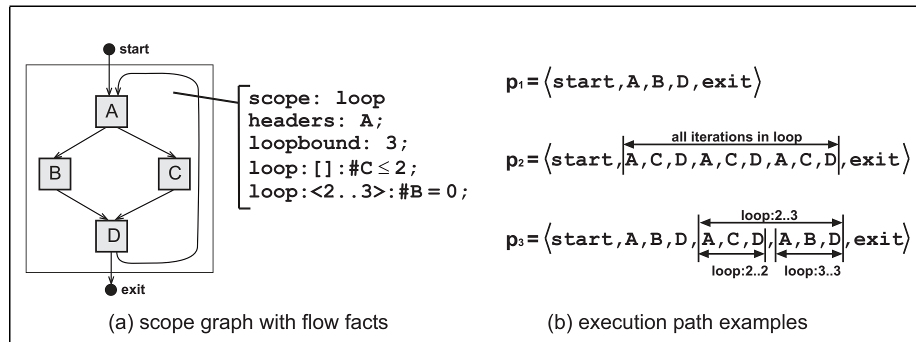


Figure 5.7: Example scope graph and execution paths

5.5.1 Execution paths

An *execution path* over a scope graph is represented as an ordered list of nodes in the scope graph. The edges in a path are implicitly given by their source and target nodes. Note that we do not allow two edges in a scope graph to have the same target and source nodes, i.e., the scope graph is not a multi-graph. In Figure 5.7(b) three example paths p_1, p_2 and p_3 are shown for the scope graph in Figure 5.7(a).

Each node in a path gets an index corresponding to the its position in the list. A node might occur several times in a path, for example path p_3 has multiple occurrences of node `A`, but each occurrence will have a different index. The indices are used to separate between different occurrences of the same node. All indices of nodes in a path p can be represented as a possible infinite set of positive numbers and is defined by $\texttt{indices}(p)$. For example, the set of indices for path p_1 is $\texttt{indices}(p_1) = \{1, 2, 3, 4, 5\}$.

The i:th node in a path p is denoted $\texttt{node}(p, i)$ where $i \in \texttt{indices}(p)$. For example, for path p_1 in Figure 5.7(b) the second node taken, $\texttt{node}(p_1, 2)$, is `A`.

An execution path must be continuous in the scope graph, meaning that it can only go between nodes which have edges connecting them. More formally, a path p over a scope graph sg is a finite or infinite sequence $p = \langle n_1, n_2, n_3, \ldots \rangle$ of nodes in sg such that $\forall i, i+1 \in \texttt{indices}(p) : \exists e \in \texttt{edges}(sg) : \texttt{source}(e) = \texttt{node}(p,i) \wedge \texttt{target}(e) = \texttt{node}(p, i+1)$.

The indices are also used to identify edges taken within a path. The i:th edge executed in a path p is defined to be $\texttt{edge}(p,i)$ and is the edge going between $\texttt{node}(p,i)$ and $\texttt{node}(p,i+1)$. For example, for path p_1 in Figure 5.7(b) the third executed edge, i.e., $\texttt{edge}(p_1, 3)$, is B→D.

5.5.2 Subpaths

As mentioned in Section 5.4, both flow facts and loop-bounds are locally defined, meaning that they have a scope-local semantics. An execution path might include several entries to a scope and for a path to fulfil a given loop-bound it must be satisfied for each such path passage.

Checking that a path fulfils a flow fact is more complicated since a flow fact should be valid for each path passage when the execution has entered the defining scope and the iteration counters are within the ranges given by the context specification.

We introduce the concept of *subpaths* to define semantics for loop-bounds and flow facts. A subpath is a continuous sequence of nodes extracted from an execution path. More formally, a path *sp* is a subpath of an execution path p if there exists a continuous sequence of nodes in p with exactly the same nodes in the same order as in *sp*. The subpath *sp* of path p starting at a node with index b and ending at a node with index e (when $e \geq b$) is defined as:

$$
\begin{aligned}
&\langle \mathit{path} \rangle\ \texttt{subpath}(p, b, e) \equiv \\
&\quad sp : (\,|\,\texttt{indices}(sp)\,| = (e - b + 1)) \wedge \\
&\qquad (\forall 1 \leq i \leq b + e - 1 : (b + i - 1) \in \texttt{indices}(p) \wedge \\
&\qquad\quad \texttt{node}(sp, i) = \texttt{node}(p, b + i - 1))
\end{aligned}
$$

For example, the path $\langle \texttt{A}, \texttt{B} \rangle$ is a subpath of p_1 given in Figure 5.7(b). Observe that the same subpath might occur several times in a path. In path p_2 in Figure 5.7(b) the subpath $\langle \texttt{A}, \texttt{C}, \texttt{D} \rangle$ occurs three times. We will later use the fact that a subpath can be recursively divided into smaller subpaths.

The set of all subpaths of a path p are defined as:

$$
\begin{aligned}
&\langle \mathit{set\ of\ paths} \rangle\ \texttt{subpaths}(p) \equiv \\
&\quad \{ sp \mid \exists b, e \in \texttt{indices}(p) : b \leq e \wedge sp = \texttt{subpath}(p, b, e) \}
\end{aligned}
$$

Whether an execution path p satisfies a certain flow fact $f = \mathit{scope} : \mathit{context} : \mathit{constraint}$ is determined by considering all subpaths of p that are within the *scope* and *context* specification and checking that *constraint* is satisfied within each such subpath.

To extract the subpaths of a path p that are within a given context we first extract all subpaths of p that start with an entry and ends with an exit of the defining scope. Within each resulting subpath, the subpaths corresponding to the range in the context specification are extracted. If the fact is a foreach fact each such subpath is divided into a number of smaller subpaths corresponding to the iterations of the defining scope. Finally, for each extracted subpath the number of different entity executions are collected and checked against the given constraint. If all counted executions in all extracted subpaths fulfil the constraint, then the path fulfils the flow fact.

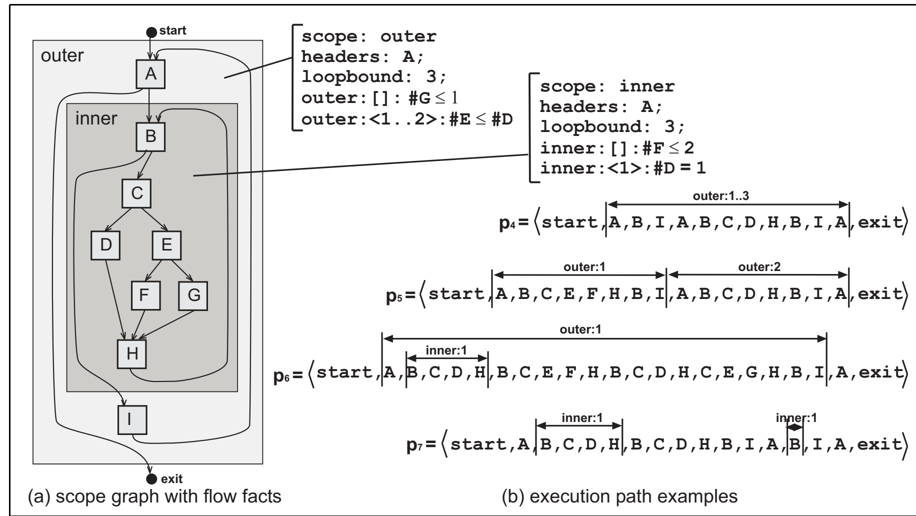


Figure 5.8: Example scope graph and execution paths

Figure 5.8 shows a scope graph with two scopes and some flow facts, together with some execution paths. Within path p_5 we extract two subpaths corresponding to the first and second iteration of outer, as illustrated by outer:1 and outer:2, respectively. The executions of entities within each subpath are collected and validated against the #E = 1 constraint. Since node E but not node D is executed within outer:1 we conclude that the subpath, and consequently path p_5, does not fulfil the given fact outer : <1..2> : #E ≤ #D.

5.5.3 Extraction of path entities

To extract subpaths corresponding to ranges we first need functionality to extract and order executions of nodes and edges that fulfil specific properties.

We use the inclusion $ent \in V$ to define that an entity *ent* (node or edge) in a scope graph fulfils a certain property V. For example, node B in Figure 5.8 is a header node of inner since B ∈ headers(inner). We can now extract the nodes or edges in a path that fulfils a certain property. For example, an

execution node with index i within a path p is an execution of a header node of scope s if $\texttt{node}(p, i) \in \texttt{headers}(s)$.

Given a certain index i of an entity in an execution path p we define the index of the *next* entity j in p for which a certain property V holds. This is entity index j such that $i < j$ and there exists no other entity with index k such that and $i < k < j$ and fulfilling V:

$$\begin{aligned}
&\langle index \rangle\ \texttt{next}(p, i, V) \equiv \\
&\quad j : (j \in \texttt{indices}(p) \wedge i < j \wedge (\texttt{node}(p, j) \in V \vee \texttt{edge}(p, j) \in V) \wedge \\
&\qquad (\neg \exists k \in \texttt{indices}(p) : i < k < j \wedge (\texttt{node}(p, k) \in V \vee \texttt{edge}(p, k) \in V)))
\end{aligned}$$

For example, $\texttt{next}(p_2, 1, \texttt{headers}(\text{loop}))$ for path p_2 in Figure 5.7(b) on page 70 denotes the index 2. The index corresponds to the first execution of `A` in p_2 after the execution of the *start* node. Similarly, $\texttt{next}(p_2, 2, \texttt{headers}(\text{loop}))$ denotes index 5, and corresponds to the second execution of the header node.

The index of the t:th succeeding entity fulfilling a property V given a current entity with index i are recursively defined by applying `next(p, i, V)` t times. If there does not exists such an entity the result is undefined.

$$\begin{aligned}
&\langle index \rangle\ \texttt{succ}(p, i, V, t) \equiv \\
&\quad j : \texttt{if}\ \ t = 1\ \ \texttt{then}\ \ j = \texttt{next}(p, i, V)\ \ \texttt{else} \\
&\qquad j = \texttt{succ}(p, \texttt{next}(p, i, V), V, t-1)
\end{aligned}$$

For example, $\texttt{succ}(p_2, 1, \texttt{headers}(\text{loop}), 3)$ for path p_2 in Figure 5.7(b) on page 70 denotes the index 8, corresponding to the third execution of `A`.

5.5.4 Loop-bound semantics

We are now ready to give the precise semantics for loop-bounds. Loop-bounds are locally defined, meaning that an execution path should fulfil the given loop-bound each time the execution enters the scope of the loop-bound, i.e., for each entry of the scope the execution should not iterate more than loop-bound times before leaving the scope.

More formally, for an execution path p to fulfil a loop-bound b of a scope s the following property should hold: If k is an $b+1$:th succeeding execution in p of an header node of s after an execution i of an in-edge of s in p then there must exists an execution j of an out-edge of s in p such that $i < j < k$.

$$\begin{aligned}
&\langle boolean \rangle\ \texttt{satisfy_loopbound}(p, s) \equiv \\
&\quad (\forall k \in \texttt{indices}(p) \wedge \texttt{node}(p, k) \in \texttt{headers}(s) \wedge \\
&\qquad (\exists i \in \texttt{indices}(p) \wedge \texttt{edge}(p, i) \in \texttt{in_edges}(s)\ : \\
&\qquad\quad k = \texttt{succ}(p, i, \texttt{headers}(s), \texttt{loopbound}(s) + 1))\ : \\
&\qquad (\exists j \in \texttt{indices}(p) \wedge \texttt{edge}(p, j) \in \texttt{out_edges}(s) : i < j < k))
\end{aligned}$$

All the execution paths in Figure 5.7(b) fulfil the loop-bound of scope `loop`. This is because none of the paths execute the header node more than three

times before exiting `loop`. Observe that it is mandatory to give loop-bounds for all scopes in the scope graph.

5.5.5 Range subpaths

The next step is to define the subpaths of a path that corresponds to a range specification over a scope.

We define the indices of nodes that start a subpath for a range $r = min..max$ over scope s. These indices are the min:th executions of a header node in s since s was last entered. If the execution left s by an out-edge before the header node was taken for the min:th time the node is not a valid start node for the range.

$$
\begin{aligned}
&\langle \textit{set of indices} \rangle\ \texttt{range_begin}(p, s, r) \equiv \\
&\quad \{ j \mid\ (\exists i \in \texttt{indices}(p) : \texttt{edge}(p, i) \in \texttt{in_edges}(s) \wedge \\
&\qquad j = \texttt{succ}(p, i, \texttt{headers}(s), \texttt{min}(r)) \wedge \\
&\qquad (\neg \exists k \in \texttt{indices}(p) : \texttt{edge}(p, k) \in \texttt{out_edges}(s) \wedge i < k < j)) \}
\end{aligned}
$$

For example, `range_begin(`p_3`,loop,2..4)` for the the scope graph given in Figure 5.7 denotes the index set $\{5\}$. This corresponds to the second occurrence of node `A` in p_3. `range_begin(`p_1`,loop,2..4)` is an empty index set, since the execution left the scope before the second iteration started.

Similarly, we can define all indices of edges in an execution path that ends a subpath for a range $r = min..max$ over a scope s. These are the indices of the edges at which a back-edge for s is taken for the $max - min + 1$:th time since a begin-node for the subpath was taken, plus the indices of the out-edges leaving s before the whole range of iterations was executed (whatever comes first).

$$
\begin{aligned}
&\langle \textit{set of indices} \rangle\ \texttt{range_end}(p, s, r) \equiv \\
&\quad \{ j \mid (\exists i \in \texttt{range_begin}(p, s, r) \wedge \\
&\qquad j = \texttt{next}(p, i, \\
&\qquad\qquad \{ \texttt{succ}(p, i, \texttt{back_edges}(s) \\
&\qquad\qquad\quad \texttt{max}(r) - \texttt{min}(r) + 1) \} \cup \\
&\qquad\qquad \{ \texttt{next}(p, i, \texttt{out_edges}(s)) \})) \}
\end{aligned}
$$

For example, `range_end(`p_1`,loop,2..4)` denotes the index set $\{10\}$, corresponding to the only occurrence of the out-edge `D`$\rightarrow$$exit$ in the path.

A subpath for a range r over scope s in a path p can now be represented as all entities inbetween a range subpath begin-node and the next range subpath end-edge. The set of all subpaths over a path p for a given range r over a scope s is defined as:

$$
\begin{aligned}
&\langle \textit{set of paths} \rangle\ \texttt{range_subpaths}(p, s, r) \equiv \\
&\quad \{ sp \mid \exists b \in \texttt{range_begin}(p, s, r) : \\
&\qquad e = \texttt{next}(p, b, \texttt{range_end}(p, s, r) \wedge sp = \texttt{subpath}(p, b, e) \}
\end{aligned}
$$

For example, $\texttt{range_subpaths(}p_3\texttt{,outer,2..3)}$ denotes set $\{\langle \texttt{A}, \texttt{C}, \texttt{D}, \texttt{A}, \texttt{B}, \texttt{D}\rangle\}$. The subpath corresponds to the second and third iterations of `outer` over p_3 and is illustrated by `outer:2..3` in Figure 5.7. Observe that the extracted subpath indirectly includes all the edges between the nodes as well as the $\texttt{D}{\rightarrow}exit$ edge.

5.5.6 Iteration subpaths

Foreach contexts specifies that the given constraint should be valid for each single iteration of the included range. We therefore need functionality to divide a range subpath into smaller subpaths corresponding to single iterations of the scope. As mentioned in Section 5.3 an iteration for a scope s starts at a header node of s and ends at the next back-edge or out-edge of s in the path.

For a subpath sp we define the indices of all the nodes that start an iteration of a scope s as well as indices of all the edges that end an iteration of the scope s as:

$$\langle \textit{set of indices} \rangle\ \texttt{iter_begin(}sp, s\texttt{)} \equiv \{i \mid \texttt{node(}sp, i\texttt{)} \in \texttt{headers(}s\texttt{)}\}$$

$$\langle \textit{set of indices} \rangle\ \texttt{iter_end(}sp, s\texttt{)} \equiv \\ \{i \mid \texttt{edge(}sp, i\texttt{)} \in (\texttt{back_edges(}s\texttt{)} \cup \texttt{out_edges(}s\texttt{)})\}$$

For example, $\texttt{iter_begin(}sp_3\texttt{,loop)}$ denotes the index set $\{1, 4\}$, i.e., all indices within the subpath corresponding to an execution of the header node `A`. Index set $\{3, 6\}$ are denoted by $\texttt{iter_end(}sp_3\texttt{,loop)}$. The first index corresponds to an execution of the back-edge $\texttt{D}{\rightarrow}\texttt{A}$ while the second index corresponds to an execution of the out-edge $\texttt{D}{\rightarrow}exit$.

An iteration *iter* over a scope s can now be defined as all entities executed inbetween an iteration begin node b and the next iteration end edge e. The iteration subpaths over a path sp for a scope s are defined as (note the similarity with the definition of a range subpath):

$$\langle \textit{set of paths} \rangle\ \texttt{iter_subpaths(}sp, s\texttt{)} \equiv \\ \{iter \mid (\exists b \in \texttt{iter_begin(}sp, s\texttt{)}\ : \\ e = \texttt{next(}sp, b, \texttt{iter_end(}sp, s\texttt{))} \wedge iter = \texttt{subpath(}sp, b, e\texttt{))}\}$$

For example, $\texttt{iter_subpaths(}sp_3\texttt{, loop)}$ denotes the iteration set $\{\langle \texttt{A}, \texttt{B}, \texttt{D}\rangle\}$. This corresponds to the last subpath `loop:3..3` for p_3 as illustrated in Figure 5.7(b) on page 70.

5.5.7 Constraint satisfaction

A flow fact constraint contains count variables corresponding to entities in the scope graph. As previously mentioned, a flow fact is only allowed to refer to entities located in the complete subtree of its defining scope. Let $\texttt{cvars(}c\texttt{)}$ denote the set of all count variables in a constraint c, and let $\texttt{entity(}v\texttt{)}$ denote

the entity that count variable v refers to. All execution indices for an entity corresponding to a count variable v within a path sp are defined as follows:

$$\langle \textit{set of indices} \rangle\ \texttt{collect}(sp, v) \equiv$$
$$\{i \mid i \in \texttt{indices}(p) \wedge (\texttt{node}(sp, i) = \texttt{entity}(v) \vee \texttt{edge}(sp, i) = \texttt{entity}(v))\}$$

For instance, $\texttt{collect}(\langle \textit{start}, \texttt{A}, \texttt{B}, \texttt{D} \rangle, \texttt{\#B})$ denotes the index set $\{2\}$.

We can now check for the validity of a subpath (or iteration) sp by replacing each count variable with the number of times its corresponding entity got executed within the subpath. The notation $c[v/sum]$ means that we replace v with sum in the constraint c. For the subpath to be valid the resulting constraint must be true.

$$\langle \textit{boolean} \rangle\ \texttt{satisfy_constr}(sp, c) \equiv$$
$$c[\forall v \in \texttt{cvars}(c)\ /\ |\texttt{collect}(sp, v)|\ ]$$

For example, $\texttt{satisfy_constr}(\langle \textit{start}, \texttt{A}, \texttt{B}, \texttt{D} \rangle,\ \texttt{\#B} = 0)$ is false. After replacing entity variable `#B` with the the number of occurrences of `B` within the path the given constraint becomes `1 = 0`. This is false and therefore the path $\langle \textit{start}, \texttt{A}, \texttt{B}, \texttt{D} \rangle$ does not fulfil the given $\texttt{\#B} = 0$ constraint.

5.5.8 Flow fact semantics

After defining the subpaths corresponding to ranges and iterations and giving predicates to check if a subpath fulfils a given constraint we can now define the semantics of flow facts, i.e., if an execution path fulfils a given flow fact.

For a path to satisfy a total fact, all the subpaths given by the context specification should fulfil the given constraint. The subpaths of a path p given by the context specification of a flow fact f are given by:

$$\langle \textit{set of paths} \rangle\ \texttt{total_subpaths}(p, f) \equiv$$
$$\{sp \mid sp \in \texttt{range_subpath}(p, \texttt{scope}(f), \texttt{range}(\texttt{context}(f)))\}$$

A path p fulfils a given total fact f if all total subpaths fulfil the given constraint.

$$\langle \textit{boolean} \rangle\ \texttt{satisfy_total}(p, f) \equiv$$
$$\forall sp \in \texttt{total_subpaths}(p, f)\ :\ \texttt{satisfy_constr}(sp, \texttt{constraint}(f)))$$

For example, $\texttt{satisfy_total}(p_2,\ \texttt{loop : [] : \#C} \leq 2)$ for path p_2 in Figure 5.7 is false. The subpath given by the context range is $\langle \texttt{A}, \texttt{C}, \texttt{D}, \texttt{A}, \texttt{C}, \texttt{D}, \texttt{A}, \texttt{C}, \texttt{D} \rangle$. The number of occurrences of `C` are counted, which gives a total of 3. Since this contradicts the given constraint we conclude that path p_2 does not satisfy the flow fact.

For a path to satisfy a foreach fact all the iterations within the subpaths given by the context specification should fulfil the given constraint. The iterations of

a path p within the subpaths given by the context ranges of a flow fact f are:

$$\langle \textit{set of paths} \rangle\ \texttt{foreach_subpaths(}p, f\texttt{)} \equiv$$
$$\{iter \mid \exists sp \in \texttt{total_subpaths(}p, f\texttt{)} : iter \in \texttt{iter_subpaths(}sp, \texttt{scope(}f\texttt{))}\}$$

A path p fulfils a given foreach fact f if all foreach subpaths fulfil the given constraint.

$$\langle \textit{boolean} \rangle\ \texttt{satisfy_foreach(}p, f\texttt{)} \equiv$$
$$\forall iter \in \texttt{foreach_subpaths(}p, f\texttt{)}\ : \texttt{satisfy_constr(}iter, \texttt{constraint(}f\texttt{))}$$

For example, $\texttt{satisfy_foreach(}p_3\texttt{,loop : <2..3> : \#B} = 0\texttt{)}$ for path p_3 in Figure 5.7 is false. The subpath given by the range in the context specification is $\langle \texttt{A}, \texttt{C}, \texttt{D}, \texttt{A}, \texttt{B}, \texttt{D} \rangle$. Within this subpath the iteration subpaths are $\langle \texttt{A}, \texttt{C}, \texttt{D} \rangle$ and $\langle \texttt{A}, \texttt{B}, \texttt{D} \rangle$. Within each iteration the number of occurrences of B are collected and summed up. The first iteration fulfils the given constraint, but the second iteration does not, since B is taken once. Thus, the flow fact is not satisfied.

We use the type of the flow fact (total or foreach) to validate that a path p fulfils a given flow fact f as:

$$\begin{aligned}&\langle \textit{boolean} \rangle\ \texttt{satisfy_flow_fact(}p, f\texttt{)} \equiv\\ &\quad (\texttt{is_total(}f\texttt{)} \wedge \texttt{satisfy_total(}p, f\texttt{)}) \vee\\ &\quad (\texttt{is_foreach(}f\texttt{)} \wedge \texttt{satisfy_foreach(}p, f\texttt{)})\end{aligned}$$

Finally, a path p is a valid execution over an scope graph sg if it is an execution path over sg and satisfy all loop-bounds and flow facts for the scopes in sg:

$$\begin{aligned}&\langle \textit{boolean} \rangle\ \texttt{is_valid_path(}p, sg\texttt{)} \equiv\\ &\quad (\forall s \in \texttt{scopes(}sg\texttt{)} : \texttt{satisfy_loop_bound(}p, s\texttt{)} \wedge\\ &\quad\quad (\forall f \in \texttt{flow_facts(}s\texttt{)} : \texttt{satisfy_flow_fact(}p, f\texttt{)}))\end{aligned}$$

Investigating the paths given in Figure 5.8 we conclude that only path p_4 and p_6 fulfil the given flow facts. Neither p_5 or p_7 fulfil fact $\texttt{inner : <1> : \#D} = 1$ and p_5 does not fulfil fact $\texttt{outer : <1..2> : \#E} \leq \texttt{\#D}$.

5.6 More on complex flows

The scope graph and flow fact language together fulfil some, but not all of the desired flow representation properties outlined in Chapter 4. Loop-bounds allow us to express basic finiteness flow information, and the flow facts allow us to express additional tightening flow information. The scope graph is not closely related to any particular programming language, but since it relates to the object code, it is rather sensitive to program changes and needs to be regenerated every time the program is recompiled.

The flow fact language is rather expressive, using a math-based language to give constraints on possible flows. It combines the expressiveness with the ability to give the flow information in a local context, ranging from a single iteration of a loop to the whole program execution. The flow fact language can only express static flow information, and requires that given dynamic flow information is converted to a static format.

In the following subsections we will see how the scope graph format can be adapted to express flows found in recursive and unstructured code as well as context-sensitive flow information.

5.6.1 Context-sensitive flow information

One of the desired properties of a flow representation is the ability to allow context-sensitive flow information to be expressed, i.e., that the flow information derived for a function (or some other specific code part) is different due to the context in which the function was called. Such context is for example the preceding function invocation history.

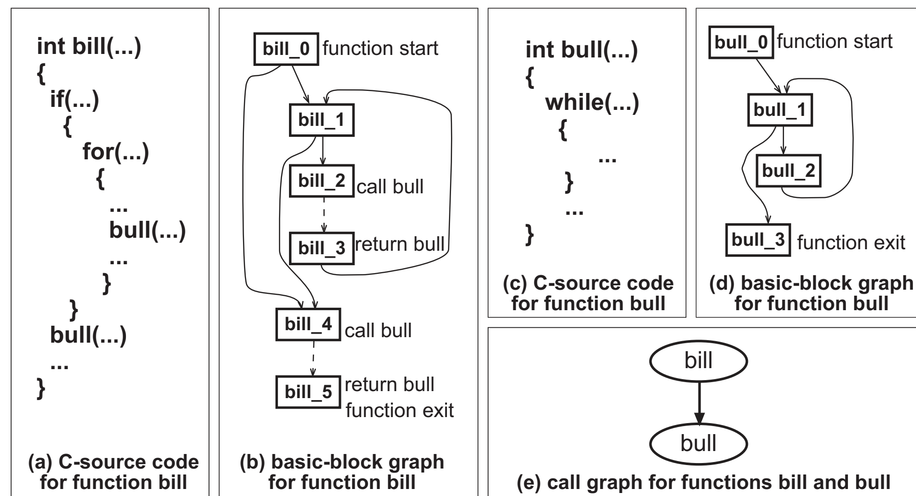


Figure 5.9: Function `bill` calls function `bull` twice

The scope graph format supports context-sensitive flow information by allowing a function (or some other specific code part) to exist in several different copies in the scope graph. For each copy, flow information valid for this particular invocation could be individually provided. Similarly, flow information valid independently from where the function got called, (i.e., context-insensitive flow information), could be multiplied to all the different copies. Since a scope node holds a reference to a basic block, this gives that the same basic block might be referenced by several different scope nodes.

Since the scope-hierarchy must be tree-formed one function can potentially exist in several copies in the scope graph, all with identical flow information. For programs including a lot of code and functions, the resulting scope graph can potentially get rather large due to this scope duplication. However, this is the price we have to pay for allowing flow information reaching down into descendant scopes.

For an illustrative example of the need to create several copies of the same function, consider Figure 5.9 and Figure 5.10. Figure 5.9(a) and Figure 5.9(c) depicts pseudo C-code for two example functions `bill` and `bull`. Function `bill` calls `bull` at two different call-sites, once inside `for` loop and once after the loop. Figure 5.9(b) and Figure 5.9(d) show the corresponding basic-block graphs.

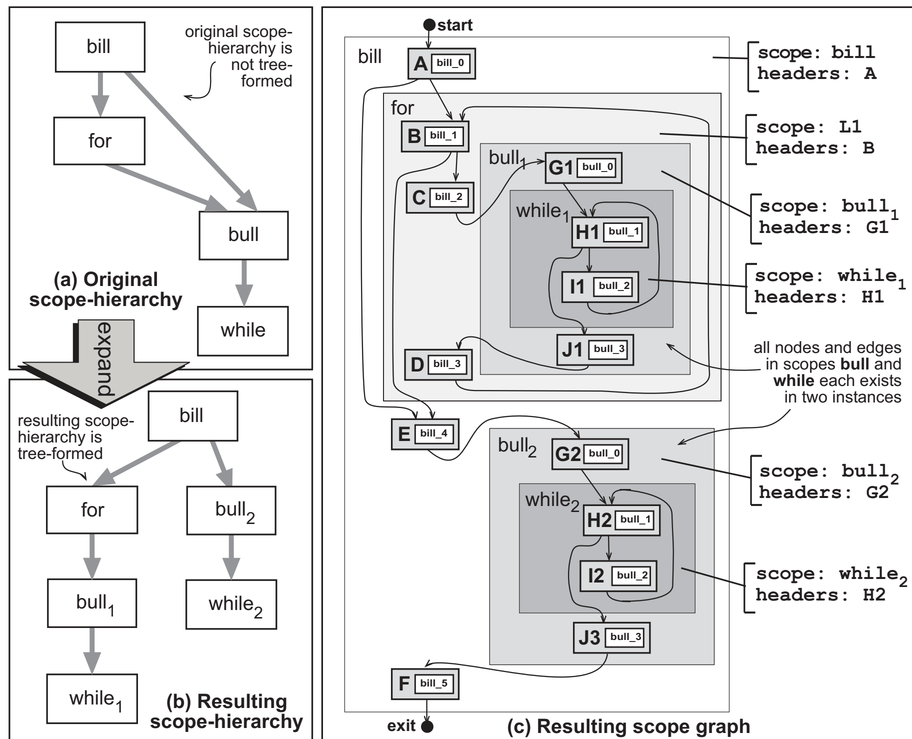


Figure 5.10: Resulting expanded scope graph for `bill` and `bull`

Figure 5.10(a) illustrates how a scope-hierarchy covering both `bill` and `bull` would look like when not differentiating between the contexts of the two calls to `bull`. This scope-hierarchy is a directed acyclic graph (DAG) and not a tree and an extra expansion-pass is therefore needed. Figure 5.10(b) illustrates how the resulting tree-formed scope-hierarchy would look like. Both scopes `bull` and `while` exists in two versions in the resulting scope-hierarchy. The algorithm for converting a DAG-formed scope-hierarchy to a tree can be made as a depth-first

traversal of the DAG.

Figure 5.10(c) illustrates the resulting scope graph. Each node in the original `bull` and `while` scopes exists in two version in the resulting scope graph. Flow information valid for `bull` independent from where `bull` got called should get copied into both versions of `bull`, while context-sensitive flow information should only be given to the copy it is valid for. For example, flow information only valid when `bull` is called within the `for` loop, should be added only to the $\texttt{bull}_1$ scope. Note that each basic block for function `bull` is referenced by two different nodes in the scope graph. For example, basic block `bull_1` is referenced both by node `H1` and node `H2`.

5.6.2 Recursive code

Recursive code is code where a function calls itself directly or indirectly through other functions. Programmers often use recursion as an alternative to writing iterative code, e.g., when traversing complex data-structures. Since recursion, like normal loops, allow repeated execution of a piece of code, the number of times a function can recursively call itself must be upper bounded, since otherwise a WCET estimate for the program can not be produced.

We do not make any direct demands on how to create a scope graph for recursive call nests, but here only give some general remarks. The exact scope graph layout is up to the flow analysis algorithms employed.

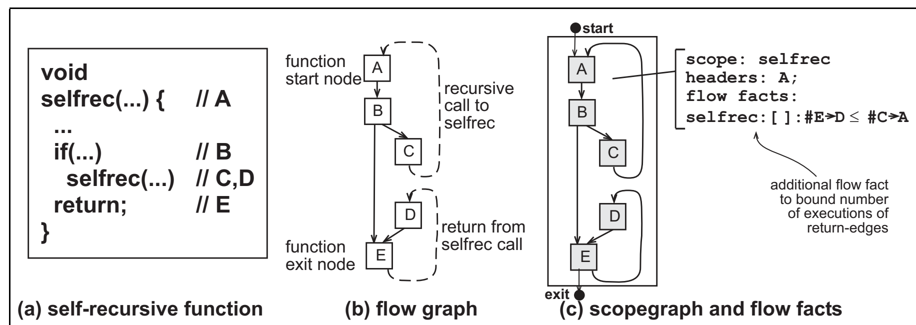


Figure 5.11: Example of self-recursive function

Figure 5.11(a) gives the code of a self-recursive program `selfrec` and Figure 5.11(b) depicts the corresponding basic-block graph. Investigating the basic block graph layout we note that one recursive call actually give raise to *two* loops: one loop due to the recursive call-edge and one loop due to the corresponding return-edge. The concept of ranged flow facts and iterations given earlier in this chapter is not directly applicable to recursive code. This means that it is not possible to specify flow information valid for a particular recursive call, using the the semantics given in Section 5.5.

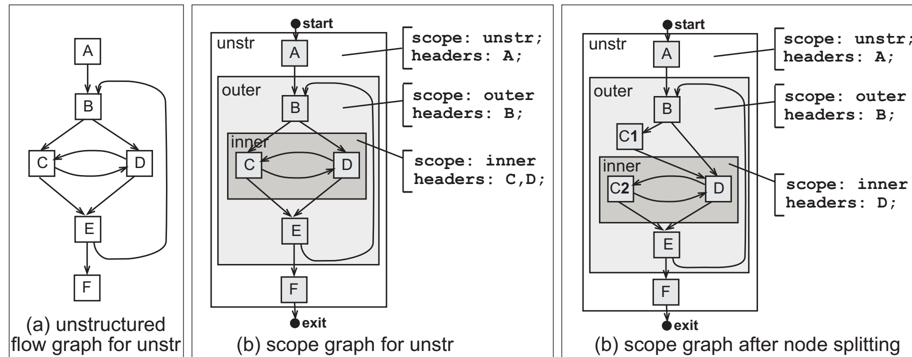


Figure 5.12: Example of unstructured graph

Figure 5.11(c) illustrates a potential extension of the scope concept for handling recursive code. Both loops have been encapsulated within a single scope, thereby allowing flow information to be specified for a certain recursive call. To bound the executions of the second loop we have added a flow fact specifying that the back-edge E→D (corresponding to a recursive function call) should be upper bounded by the C→A back-edge (corresponding to a return from recursive function call). Note that the first iteration within the C→A loop will correspond to the last iteration within the E→D loop, and the second iteration within the C→A loop will correspond to the second last iteration within the E→D, etc.

This way of handling recursive functions allows us to uniquely specify a certain recursive call in a recursive loop. However, the definition of a header node is no longer valid, since it is possible to iterate within the scope without taking the header node. Flow facts with ranged context specifications over such recursive scopes are therefore not supported by the current scope graph implementation. Giving specific flow information within mutually recursive call nests is even more tricky. However, we believe that it should be possible to adapt the iteration concept for such code constructs and plan to do it for future work.

5.6.3 Unstructured code

An unstructured code part is a loop (nest) with not just one but several possible nodes where the execution can enter the loop (nest) [Muc97]. To create scopes from unstructured code some type of loop-identification algorithm is preferably used. There exists a number of suitable algorithms presented in literature, each with different ways to identify and define loops in unstructured code and with different analysis complexity [Ste93, VSL96, Hav97, Ram00].

We do not prescribe any particular loop-identification algorithm to use when constructing a scope graph from unstructured code. The loose definition of a header node, i.e., that its should not be possible to executed any other node

in the scope more than once without passing one of the header nodes, allow us to use most loop-identification algorithms. Instead, we outline some cases that needs to be especially considered when providing flow information for unstructured code.

The first case is when we have a scope with multiple headers. Figure 5.12(a) depicts an example of an unstructured control flow graph for a function `unstr`. Figure 5.12(b) illustrates a possible corresponding scope graph. The loop identification employed has identified two loops `outer` and `inner`, both resulting in a corresponding scope in the scope graph.

As mentioned in Section 5.3 an iteration of a scope is defined to start when a header of the scope is executed and end when a back-edge or an out-edge of the scope is executed. When giving flow information over the scope graph given in Figure 5.12(b) and in particular the `inner` scope we must make sure that the flow information is valid independently of which node `inner` was entered and of which header node an iteration started.

Since many type of flow analysis algorithms and calculation methods for WCET analysis require structured code, an alternative is to employ some kind of graph transformations to convert the unstructured graph into a structured one. One example of such technique is *node-splitting* [Muc97]. Figure 5.12(c) depicts how the original scope graph presented in Figure 5.12(b) could look like after applying node splitting. Node `C` has been split into two distinct copies, with the result that the `inner` scope only can be entered at the `D` node.

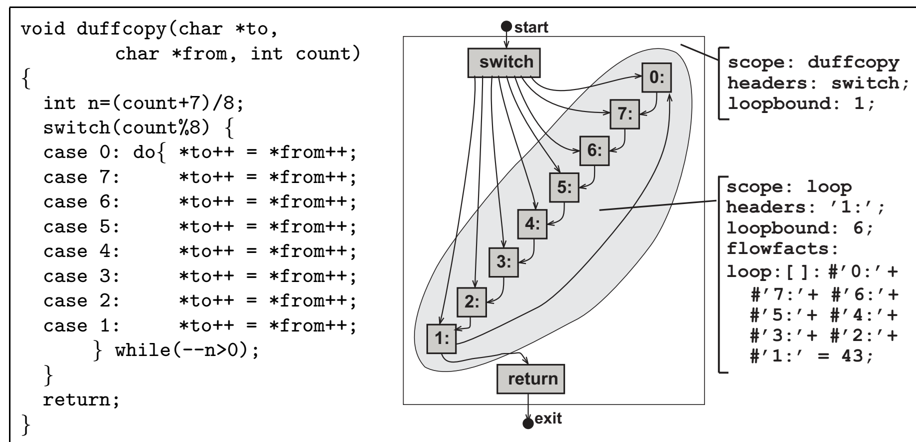


Figure 5.13: Example of multiple-entry loop: `duff`

The other case we need to consider is when unstructured code produces scopes with in-nodes which not are header nodes. This allow us to avoid scopes with multiple headers but complicates the iteration concept. In a scope with in-nodes not being header-nodes, it is possible that one or several nodes in the scope are executed once before the first execution of a header node. Since an

iteration is defined to start when a header node is taken, we can not say that these nodes are executed in the first iteration of the scope. Instead, we introduce the concept of iteration "0", used to hold the execution of entities in the scope taking place before the header node in the scope is executed.

Figure 5.13 gives the core code of the function duff. It contains an unstructured code part with multiple entry points. For the loop scope, the definition of a header node allows *any* set of nodes to be header nodes as long as node 1: is included. In the example we have chosen the node 1: to be the only header node. Allowing an iteration "0" we can for example add a flow fact like: loop : <0> : #0: = 0, specifying that the execution can not enter loop at node 0:.

The semantics of flow facts and ranges given in Section 5.5 does not allow us to specify constraints valid for such an iteration "0", and is therefore not supported by the current implementation (but is planned for future work). However, for a scope with unstructured code, we can provide flow information using total flow facts valid for all its iterations, as illustrated for scope loop in Figure 5.13.

Chapter 6

Low-level Analysis

In this chapter we present our low-level analysis. The purpose of the low-level analysis is to determine the timing behaviour of instructions executing on the target hardware. For modern processors it is especially important to study the effects of various performance enhancing features, like caches and pipelines.

Our low-level analysis is divided into two distinct phases. In the *global low-level analysis* phase the effects of caches, branch predictors, etc. are determined. In the *local low-level analysis* phase the effects of pipelining are determined and execution times for program parts are generated.

Sections 6.1-6.4 present our view of global low-level analysis. We do not present any new global low-level analysis, but show how to extend the scope graph structure to incorporate results from various types of global low-level analyses.

Sections 6.5-6.9 present our local low-level analysis. It consists of a *pipeline timing analysis* and a resulting *timing model*. The pipeline analysis can make use of a trace-driven cycle-accurate processor model to perform WCET analysis, making it both easier to retarget the WCET analysis to new processors, and to validate the model against real target hardware. Previous research has required the construction of special purpose hardware models to safely capture the same type of timing.

The resulting *timing model* allows us to calculate a program WCET estimate from the timing of smaller program parts, still capturing all relevant hardware timing effects. The timing model is independent from the hardware model used in the pipeline analysis, thereby further enhancing retargetability.

6.1 Global low-level analysis

The global low-level analysis phase considers the effects of performance enhancing machine features that must be analysed over the *entire program* to be safely captured. Examples of such features are *instruction caches*, *data caches*, *branch*

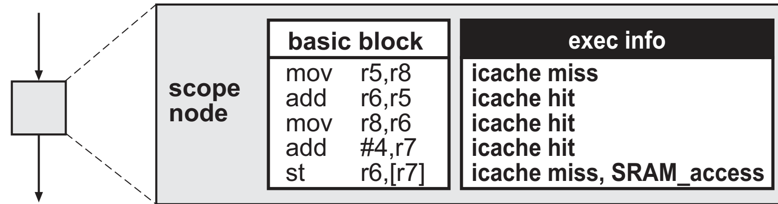


Figure 6.1: Scope node with basic block and execution scenario

predictors, and *translation lookaside buffers (TLBs)*. To determine how such features will affect the execution of an instruction it is not enough to consider a few neighbouring instructions. Instead, global-low level analysis must take a *global* program view, and consider interactions between instructions located arbitrarily remote from each other.

A global low-level analysis only determines how the investigated feature(s) affect the execution of instructions, but does not generate actual execution times. The resulting information should be a safe estimate on how the investigated feature will affect the execution of instructions. The features to analyse in global low-level analysis are target-dependent, and consequently, the collected information will be different for different CPUs. The complexity of the global-low level analysis depends on the type of features used by the target hardware and how much they interact.

Global low-level analysis can sometimes benefit from flow analysis information to produce better estimates. For example, if we have information that a certain basic block never will be executed, it is safe to assume that its instructions will not be loaded and therefore not interfere with other instructions in the cache.

A large number of global-low level analyses have been presented in the literature, including analyses for instruction caches [FMW97, HAM+99, LBJ+95, SA00], multi-level caches [Mül97], unified caches [FHL+01], data caches [KMH96, SA00, WMH+97], and branch predictors [CP00]. Our working philosophy has been to avoid reinventing the wheel. Instead, we have focussed on providing a modular framework which allows us to express and use the results from such previously presented analyses.

6.2 Execution scenarios

The information generated in the global analysis phase needs to be communicated to the local analysis phase. This includes safe information on cache and branch prediction hits and misses. Other type of information needed to determine the execution time of an instructions is information on the type of memory being accessed, and bounds on data for instructions whose execution time is data-dependent.

We have solved this problem by attaching such requisite information to the

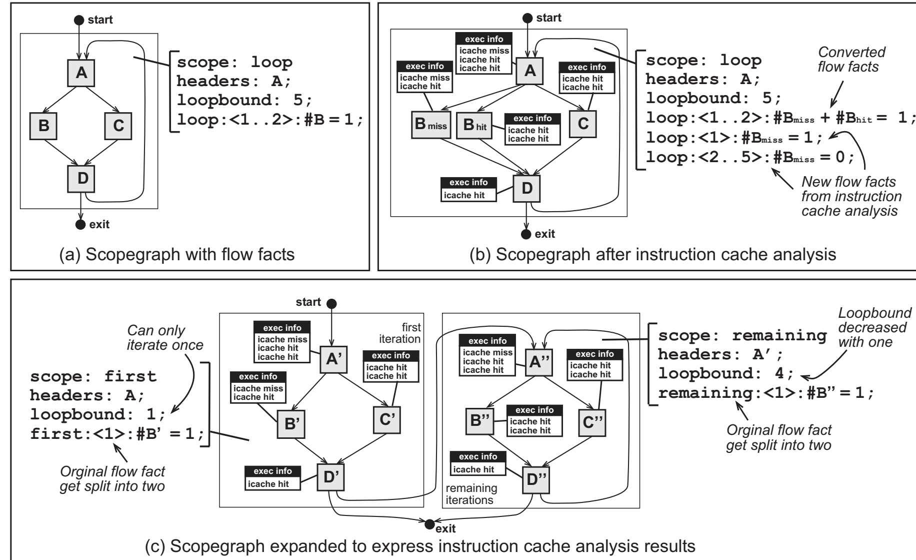
nodes in the scope graph [EES+03, EE99]. Every instruction in a basic block can have zero or more associated *execution facts*, used for providing information about the execution of the instruction. The execution facts for all the instructions in a basic block together form an *execution scenario.* Figure 6.1 shows an example of a scope node with associated basic block and execution scenario. The execution scenario contains information from memory access and instruction cache analysis.

The information in the execution scenarios is used in the low-level analysis to determine the execution time of the instructions in the associated basic block. It is possible to have several execution scenarios for each basic block in the program, each represented by a different node in the scope graph. However, each execution fact should have a constant effect on the execution of the associated instruction. Cache behaviour characterisations like `first-miss` and `first-hit` proposed by Healy et al. [HAM+99] are not valid as execution facts. In this case, two execution scenarios should be set up, one with a cache hit, and one with a cache miss scenario. Also, the scope graph should be modified to include a scope node for each alternative.

6.3 Expressing global low-level analysis results

The combination of execution scenarios and flow facts is powerful enough to represent and combine the results from various types of global-low level analyses. For example, Theiling et al. [TF98, FMW97] outline a cache analysis using abstract interpretation over abstract cache states. The result is a categorisation of cache references for instructions as `always-hit`, `always-miss`, `persistent` and `undefined`. The `always-hit` or `always-miss` categorisations are used when all references to an instruction can be safely categorised as always generating a hit, respectively miss in the cache. `persistent` means that all references of the instruction after the first one will result in a hit and `undefined` is used when a reference does not fit into any of the other categorisations. To increase the precision in the analysis, loops are unrolled to distinguish between the first and remaining iterations (maybe over several loop-levels), using the fact that within loops the first reference to an instruction is more likely to result in a cache miss than the following references.

Figure 6.2(b) illustrates how the scope graph in Figure 6.2(a) gets modified to represent results from such *iteration-based* instruction cache analysis. In this example, the analysis has derived that during the first iteration of `loop` a reference to `B` will result in a cache miss, but references in the remaining iterations will all result in cache hits. To account for the analysis result, `B` gets expanded into two different execution scenarios, where $\texttt{B}_{\texttt{miss}}$ holds the cache miss scenario and $\texttt{B}_{\texttt{hit}}$ holds the cache hit scenario. Two flow facts are created to specify during which iterations $\texttt{B}_{\texttt{miss}}$ and $\texttt{B}_{\texttt{hit}}$ can be executed. Furthermore, the original `loop : <1..2> : #B` $= 1$ flow fact is modified to reflect that `B` exists in two new versions.

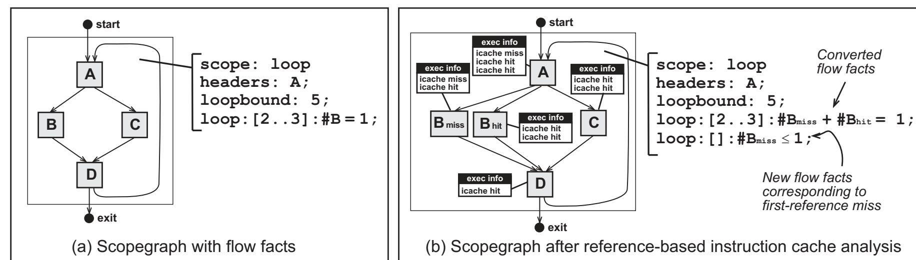


(a) Scopegraph with flow facts
(b) Scopegraph after instruction cache analysis
(c) Scopegraph expanded to express instruction cache analysis results

Figure 6.2: Representing results from iteration-based instruction cache analysis

An alternative is to unroll the scope graph to explicitly differentiate between the first and remaining iterations. Figure 6.2(c) depicts the scope graph in Figure 6.2(a) expanded into two scopes `first` and `remaining`, each with its own copy of the nodes from the original scope. The execution scenarios for `B'` and `B''` holds the cache analysis results for the first and remaining iterations respectively. Furthermore, the original loop bound and flow fact are modified to reflect the graph transformation performed.

Healy et al. [HAM+99] perform a static cache simulation. The result is a categorisation of each instruction cache access as one of **`always-miss`**, **`always-hit`**, **`first-miss`** and **`first-hit`**. The categorisation is *reference-based*, i.e., a **`first-miss`** categorisation should be interpreted as the first reference to the instruction (independent of iteration number) will result in a cache miss, while the remaining references all will be cache hits.



(a) Scopegraph with flow facts
(b) Scopegraph after reference-based instruction cache analysis

Figure 6.3: Representing reference-based instruction cache analysis

Figure 6.3(b) illustrates how the scope graph in Figure 6.3(a) gets modified to represent the result of such reference-based instruction cache analysis. In this example, the analysis has derived a first-miss categorisation for one instruction in the `B` node. This results in two execution scenarios: $\texttt{B}_{\texttt{miss}}$, where the instruction miss the cache, and $\texttt{B}_{\texttt{hit}}$, where the instruction hit the cache. The flow fact `loop : [] :` $\texttt{B}_{\texttt{miss}} \leq 1$ does not explicitly specify that a $\texttt{B}_{\texttt{miss}}$ must be taken before $\texttt{B}_{\texttt{hit}}$, but limits the number of executions of $\texttt{B}_{\texttt{miss}}$ to be less than one per entry of the loop. Additionally, the original `loop : [2..3] : #B` $= 1$ flow fact is modified to reflect that `B` exists in two versions in the resulting scope graph.

The combination of execution scenarios and flow facts can also be used to express data cache analysis results. White et al. [WMH+97, Whi97] perform analysis resulting in an upper bound on the number of data cache misses an instruction can cause in relation to surrounding loops. For example, a categorisation of `c 25 2500` for an instruction i means that the number of misses for an entry of the innermost surrounding loop is bounded by 25 and the number of misses for an entry of the next surrounding loop is bounded by 2500. Such categorisation can easily be expressed using two different flow facts as: `inner : [] :` $i_{\texttt{dmiss}} \leq 25$ and `outer : [] :` $i_{\texttt{dmiss}} \leq 2500$ where $i_{\texttt{dmiss}}$ is the execution scenario with a data cache miss.

6.4 Safe removal of scenarios

When the global-low level analysis cannot safely determine a certain execution scenario for an instruction, nodes for all scenarios have to be created. If we can guarantee that, for one original node, one scenario will always give worse timing than all the others, and if we are only interested in the WCET (and not the BCET), we can safely remove the best-case scenario. For example, for many types of processors it is safe to assume that a cache hit will always result in a total execution time that is less than a miss at the same place. However, removal of scenarios might increase the WCET estimate pessimism, especially if there exists flow information constraining the number of times the worst case scenario can be executed.

For more complex hardware, including a lot of performance enhancing features, it might be hard to safely determine the feature combination that will result in the overall worst case program timing. One single scope node will then be cloned into multiple versions, each referencing the same basic block, but with different execution scenarios. In the local-low level analysis timing for all the execution scenarios will be extracted and the combination of scenarios that produce the worst case timing will be determined.

Finally, note that if the header node in a scope gets copied due to multiple execution scenarios, the resulting scope might become unstructured, i.e., a loop with multiple in-nodes. Figure 6.4(b) illustrates how the scope graph in Figure 6.4(a) would be modified to reflect that both `A` and `B` gets copied into two versions due to different execution scenarios. This indirectly requires that the

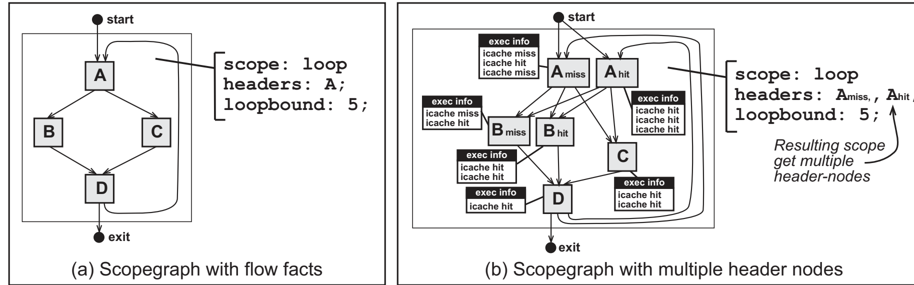


Figure 6.4: Execution scenarios causing multiple headers

local low-level analysis and the calculation method should be able to handle unstructured code.

6.5 Local low-level analysis

The purpose of the local low-level analysis is to determine the effects of pipelining and to generate execution times for program parts. This includes determining the effects of machine-dependent factors that can be analyzed locally for one or a few neighbouring instructions, such as pipeline and memory access timing.

In the following sections we will present a *pipeline timing analysis* able to safely capture the effects of dependencies and pipeline overlap that exist between instructions both within and between basic blocks. The outlined pipeline analysis is able to take execution facts into account, maybe derived by one or several different global low-level analyses.

The pipeline analysis makes use of a *hardware model* to extract time for sequences of instructions annotated with execution facts. The analysis treats the hardware model as a black box, i.e., it does not need to have access to its internal state. In principle, the weak requirements on the hardware model allow us to use any trace-driven cycle-accurate processor model (or even the hardware itself) to extract timing for instructions.

The result of the analysis is a *timing model* which allows a worst case execution time estimate of a program to be calculated from the timing of smaller program parts. We only store times for node and timing effects for sequences of nodes in the model, (and not any internal state related to the hardware model), thereby clearly decoupling the low-level analysis from the calculation phase.

The pipeline analysis and the resulting timing model have been previously described in Engblom's Ph.D. thesis [Eng02].

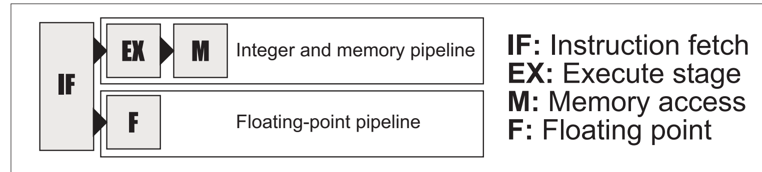


Figure 6.5: Scalar pipeline with parallel units

6.6 The problem of pipeline analysis

The purpose of pipeline analysis is to model the execution time effects of the overlap between instructions in pipelined processors. For simple non-pipelined CPUs it is possible to assign a fixed execution time to an instruction (or a basic block), but for pipelined processors this is no longer sufficient since instructions (and basic blocks) can overlap in the pipeline.

The overlap between instructions makes the execution time for a sequence of instructions (or basic blocks) less than the sum of the execution times of the individual instructions (or basic blocks). Furthermore, not all instructions need to use all pipeline stages, and instructions can be *stalled*, waiting for pipeline stages or data of other instructions to become available. A pipeline analysis suitable for WCET analysis should be able to safely capture the effects of such pipeline interference between instructions. The resulting timing model should model the timing effects due to such pipeline interference in a way that an overall program WCET estimate can be calculated.

For an illustration of the problems involved in performing such pipeline analysis and constructing a resulting timing model, we use an example pipeline with the layout as depicted in Figure 6.5. The pipeline is an in-order scalar pipeline (see Section 2.3.2) containing an instruction fetch (`IF`), an execute (`EX`), and a memory access (`M`) stage. It also contains a separate floating point (`F`) stage, which can be used instead of the (`EX`) and (`M`) stages. Instructions can execute in the (`F`) stage for several clock cycles.

Figure 6.6(a) shows a scope graph fragment consisting of three nodes `Q`, `R` and `S`, each referring to a basic block. Figure 6.6(b) shows the pipeline layout of the three basic blocks, when executed on the pipeline depicted in Figure 6.5. Nodes `Q` and `S` have instructions using the (`F`) stage, while `R` has not. The execution time for a sequence of instructions can be given as the time from when the first instruction enters the pipeline until the last instruction exists the pipeline. Using this definition the pipeline overlap between instructions within the same basic block is captured, as illustrated in Figure 6.6(b), and the execution times for the nodes become 8 cycles for `Q`, 6 cycles for `R` and 7 cycles for `S`, respectively.

When executing basic blocks in sequence, the total execution time for the sequence is usually smaller than the sum of the times for the individual basic blocks, reflecting that the instructions of the different basic blocks are overlapping in the pipeline. For example, the timing for the sequence `QR` is 11 cycles,

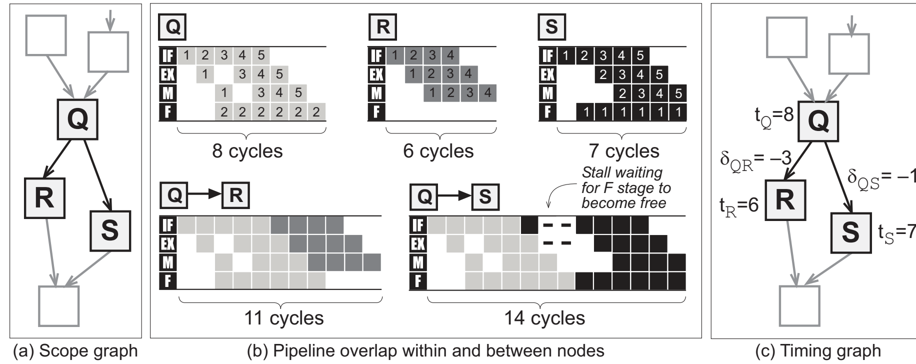


Figure 6.6: Pipeline overlap over pair of nodes

while the sum of the execution times for `Q` and `R` is $8 + 6 = 14$ cycles. Similarly, the time for executing sequence `QS` is 14 cycles, while the execution time sum of `Q` and `S` is $8 + 7 = 15$ cycles.

Our way to model such pipeline overlap between basic blocks is to assign execution times to nodes, corresponding to the time for the node when executed in isolation, and timing effects to edges, for the overlap effect between basic blocks. For example, the pipeline timing effect for edge `Q`$\rightarrow$`R` would be minus three $(11 - 14 = -3)$ cycles. The effect is negative to indicate a speedup of the overall execution time. Similarly, the pipeline timing effect for edge `Q`$\rightarrow$`S` would be only minus one $(14 - 15 = -1)$ cycle. Note that the timing effect for `Q`$\rightarrow$`R` is different from the timing effect for `Q`$\rightarrow$`S`, indicating that the pipeline overlap is different for the two sequences.

Figure 6.6(c) illustrates how the graph fragment could be annotated with timing and timing effects. In Section 6.8 we will describe how such timing and timing effects are derived. The total timing for executing a sequence of nodes is obtained by summing the time and timing effects for all nodes and edges included in the sequence. For example, the timing for sequence `QR` is then $8 - 3 + 6 = 11$ cycles.

Unfortunately, just considering pairs of basic blocks might be insufficient, since in many cases instruction overlap and interference can reach across more than two basic blocks. Such effects can for example be caused by instructions completing their execution long after the other instructions in the basic block they belong to. For example, in the pipeline outlined in Figure 6.6(a) an instruction might occupy the `(F)` stage for many cycles, while the rest of the instructions in the block have finished executing.

An example of such a long-reaching overlap between instructions in non-adjacent blocks is illustrated in Figure 6.7. The execution of node `C` is partially overlapped by the node `A`. This results in an additional execution time speedup which is not captured by just considering the pipeline overlap in neighbouring

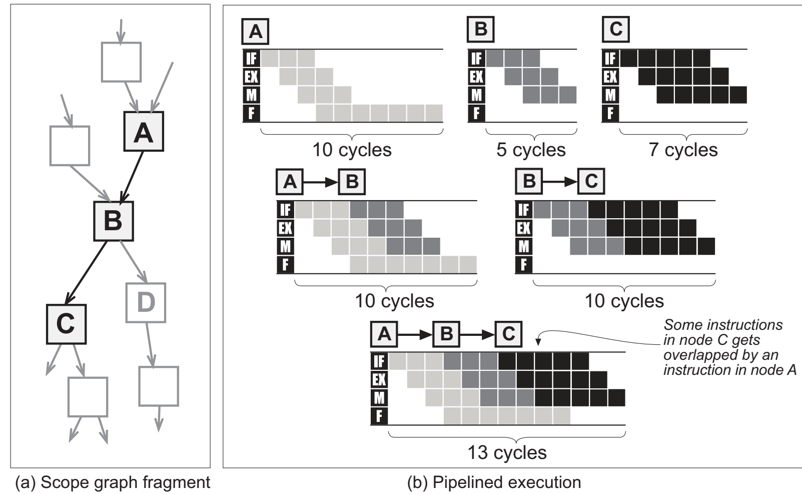


Figure 6.7: Pipeline overlap over three nodes

nodes. The execution time for the sequence `ABC` is 13 cycles, while the sum of block times and pairwise timing effects are $10 + 5 + 7 - 5 - 2 = 15$ cycles. Ignoring this extra speedup would lead to a safe, but potentially pessimistic, program WCET estimate.

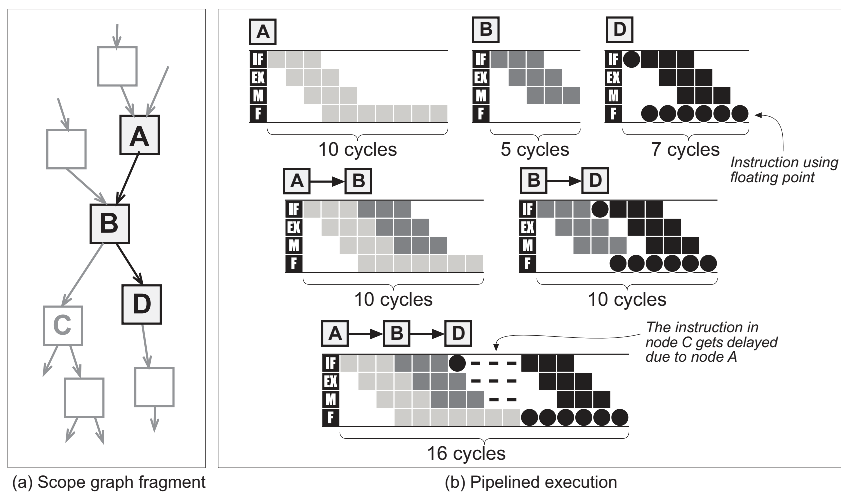


Figure 6.8: Pipeline interference over three nodes

More critically, not all long reaching instruction interferences are speedups, some interferences can cause slowdowns. Figure 6.8 shows an example of such an interference between a long-running instruction and an instruction in a non-neighbouring basic block. Node `A` and `B` are the same nodes as in Figure 6.7, but we here focus on the `D` successor node. The execution of `D` gets stalled due to `A`'s

use of the floating point stage. This is despite the presence of node B between them. The execution time for sequence ABD is 16 cycles, while the sum of block times and pairwise timing effects is $10 + 5 + 7 - 5 - 2 = 15$ cycles. Ignoring this type of execution time slowdown would lead to a potentially unsafe WCET estimate.

6.7 Pipeline timing analysis

We have constructed a *pipeline timing analysis* able to capture instruction interference and overlap both between adjacent and non-adjacent basic blocks. Our pipeline analysis makes use of a *hardware model*, or timing function T, that given a sequence of nodes in the scope graph (containing instructions and execution facts) returns its execution time. The pipeline analysis do not need to access to the internal state of the hardware model, but instead consider the hardware model as a block box. The only requirement on the hardware model is that it should safely enact the effects of the execution facts, and that the model is deterministic, i.e., the same sequence of instructions and execution facts should always give the same execution time.

The exact definition of the execution time reported by T for an instruction sequence depends on the hardware model used. For a typical model of a pipelined processor, the execution time will be from the time the first instruction enters the pipeline until the last instruction exits the pipeline as illustrated in Figure 6.6, Figure 6.7 and Figure 6.8.

In principle, the weak requirements on the hardware model allow us to use any *trace-driven* cycle-accurate processor model (or even the hardware itself) to perform WCET analysis, making it easier to retarget the WCET analysis to new processors. The simulator does not need to simulate the semantics of the code executed (like the results of arithmetic operations or the contents of memory locations), it need only to provide execution times for a given sequence of instructions. Such simulators are commonly used by embedded systems developers to evaluate new features and design tradeoffs [BC98], and we can leverage existing simulators to port our technique to a variety of hardware platforms.

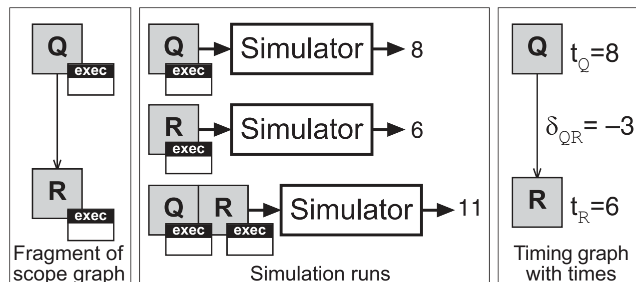


Figure 6.9: Timing analysis using a simulator

The input to our pipeline timing analysis is a scope graph, containing nodes referencing basic blocks with attached execution scenarios. The flow information and the hierarchical scope structure of the scope graph is of no interest for the analysis and therefore removed. The resulting input is a flattened graph consisting of just nodes and edges.

The analysis works by running individual nodes (i.e., basic blocks with execution scenarios) and sequences of nodes through the hardware model. The reported execution times are used to construct a *timing model*, with timing for nodes and timing effects for sequences of nodes (see Section 6.8). Figure 6.9 provides an illustration of a pipeline analysis where a cycle-accurate simulator is used to extract timing for nodes `Q` and `R` and the `QR` node sequence.

Note that we are only interested in the sequences of nodes that can actually occur in the program; that is, only sequences where the successive nodes are linked by edges in the scope graph. Also, no sequence needs to be run more than once in the hardware model, thereby reducing the run time of the analysis when using a complex hardware model with potential high execution time cost.

A central part of the analysis is the *extension condition*, a function that determines the sequences of nodes that we need to investigate to safely capture all timing effects of pipeline overlap and hardware dependencies. The extension condition to be used depends on the properties of the target processor and corresponding hardware model. For hardware models not subject to long timing effects, only nodes and sequences of nodes up to the length of two need to be run. For other type of hardware models the extension condition becomes more complicated, (see the Ph.D. thesis of Jakob Engblom [Eng02] for more details).

It should be noted that most WCET researchers do not support the use of cycle-accurate simulators. Instead, they construct a special-purpose pipeline model of their target processors, relying on the fact that even if a certain hardware processor is subject to long timing effects, it might be possible to construct a hardware model which is not. Our pipeline timing analysis can make use of such special purpose hardware model, just like a simulator.

Note that using a special purpose hardware model for WCET analysis might have the drawback that another hardware model has to be created for BCET analysis. A model which is safe for WCET analysis is by necessity unsafe for BCET analysis, since the model will overestimate the execution time in cases of uncertainty, while a model for BCET analysis would need to underestimate the time in such cases.

6.8 Timing model

The result of our pipeline timing analysis is a *timing model*, orginally introduced in [EE99] and extended in [Eng02]. The model represents the times reported by the timing function T, using *node times*, denoted by t_{node}, and *timing effects* for sequences of nodes, denoted by δ_{seq}. Figure 6.10(a) illustrates a scope graph fragment and Figure 6.10(b) its resulting timing model.

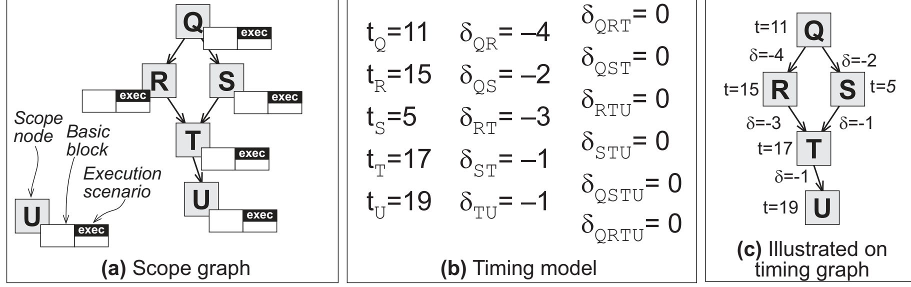


Figure 6.10: Scope graph, timing model and timing graph

A node time t_{node} represents the *time* it takes to execute a node in isolation on the hardware model. A timing effect δ_{seq} represents the *change* in execution time encountered when a node sequence *seq* is executed compared to the timing of nodes and timing effects of subsequences included in the sequence. Timing effects are negative to indicate a speedup over a sequence of nodes, and positive to indicate a slowdown. In the following we will outline how the node times and timing effects are related to the node sequence times reported by the hardware model.

We use the notation $T(seq)$ for the execution time returned for sequence *seq* when executed on the hardware model. The time for an empty sequence is zero. The node time t_{node} for a node *node* is equal to $T(node)$, i.e., the execution time for the node when running in isolation on the target hardware:

$$t_{node} = T(node) \tag{6.1}$$

The execution time $T(seq)$ for an arbitrary node sequence *seq* is defined to be equal to the sum of times for all the nodes in the sequence and all the timing effects of all subsequences of the sequence:

$$T(seq) = \sum_{\forall n \in \texttt{nodes}(seq)} t_n + \sum_{\forall s \in \texttt{subsequences}(seq)} \delta_s \tag{6.2}$$

For example, using a hardware model which gives the execution time for a sequence as the time from when the first instruction enters the pipeline until the last instruction exists the pipeline, the execution time for the `ABC` sequence in Figure 6.7 becomes: $T(\texttt{ABC}) = t_{\texttt{A}} + t_{\texttt{B}} + t_{\texttt{C}} + \delta_{\texttt{AB}} + \delta_{\texttt{BC}} + \delta_{\texttt{ABC}} = 10+5+7-5-2-2 =$ 13 cycles.

Figure 6.11 illustrates how times for successively longer sequences are constructed from Equation 6.2. The execution time $T(seq)$ for a sequence *seq* is obtained by summing all the t and δ variables within the corresponding triangle.

Using some algebraic manipulation of Equation 6.2, we can express the timing effect for a sequence in terms of the execution times of just a few subsequences. Using l to denote the length of a sequence *seq* and the notation

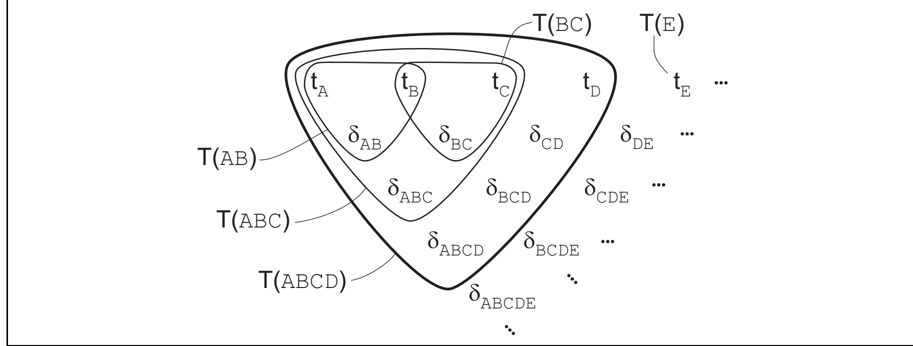


Figure 6.11: Timing triangle

$seq[i..j]$ to denote the subsequence starting at the i:th element and ending at the j:th element of seq we get:

$$\delta_{seq} = T(seq) - T(seq[2..l]) - T(seq[1..l-1]) + T(seq[2..l-1]) \qquad (6.3)$$

For example, $\delta_{\texttt{ABC}} = T(\texttt{ABC}) - T(\texttt{AB}) - T(\texttt{BC}) + T(\texttt{B})$. Note that for sequences of length two, the last term becomes zero, e.g., $\delta_{\texttt{AB}} = T(\texttt{AB}) - T(\texttt{A}) - T(\texttt{B})$. In Figure 6.7 we get a timing effect for sequence AB of $\delta_{\texttt{AB}} = -5$, reflecting that A completely overlaps the execution of B. For the sequence ABC, we get the *negative* timing effect of $\delta_{\texttt{ABC}} = -2$, reflecting that the execution of A overlaps both B and C. For the example in Figure 6.8 the pipeline interference results in a *positive* timing effect $\delta_{\texttt{ABD}} = +1$ for sequence ABC.

Engblom showed in his Ph.D. thesis [Eng02] that both positive and negative long timing effects can occur over sequences longer than three nodes, and even more problematic: for certain architectures *arbitrary* long timing effects can be constructed. For such architectures a special-purpose hardware model should preferably be used, making sure to make safe but potentially pessimistic overestimations when such long timing effects can occur.

The resulting timing model will consists of timing for nodes and timing effects for sequences of nodes. Figure 6.12(a) provides an illustration of the node times and timing effects generated for the scope graph in Figure 6.7 and Figure 6.8. The timing model can also be illustrated as a decoration of a *timing graph*, a flattened version of the scope graph with the same node and control-flow structure but with the scopes and flow facts removed. Figure 6.12(b) depicts this alternative represention. Note that the long timing effects $\delta_{\texttt{ABC}}$ and $\delta_{\texttt{ABD}}$ are represented by an explicit encapsulation of the nodes included in the sequence.

For the calculation, a timing effect for a node sequence should only be accounted for when the execution goes uninterrupted via all the nodes in the sequence. For example, $\delta_{\texttt{ABC}}$ should not be accounted for if the execution leaves the sequence before node C or if the execution enters the sequence at a node different from A. The calculation method used must be able to safely capture

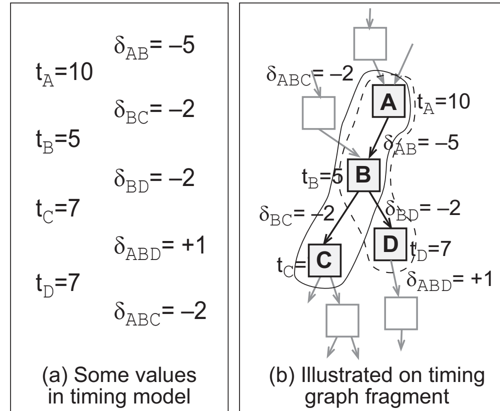


Figure 6.12: Representing long timing effects

such timing effect execution constraints, otherwise it may end up with an unsafe WCET estimate.

6.9 Alternative timing analyses

For many types of WCET analysis the way to extract and represent timing is closely connected to the calculation method used. Our approach to low-level analysis clearly separates the hardware modelling, the timing model, and the calculation. This enhance retargetability since different low-level analysis can be used together with the same calculation method.

For example, some approaches do not make an explicit decoupling between the hardware analysis, the resulting timing-model and the calculation. For example, the path-based calculation methods proposed in [HAM+99, SA00, RGL02, ZWR+01] form paths across multiple basic blocks when calculating program WCET, simultaneously extracting instruction pipeline behaviour and program timing.

Most IPET-based calculation methods, e.g., [PS95, FMW97, HLS00b] use a separation similar to ours between the low-level analysis and the resulting timing model. The potential pipeline overlaps are calculated using this hardware model, resulting in a timing model with timing typically on either nodes or edges (but not both). Since none of our calculation methods are dependent on timing effects for sequences of nodes to be present, they can be used with the timing models producied such low level analysis.

Modelling pair-wise pipeline overlaps using timing effects on edges was introduced by Ottosson and Sjödin [OS97]. They derive the potential pipeline overlap between a pair of nodes using pipeline reservation diagrams. By only considering a simple scalar pipeline, they avoid the problem of long timing effects [Eng02].

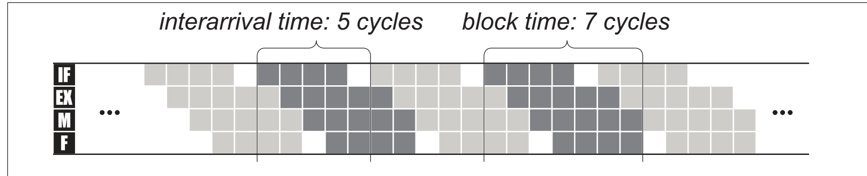


Figure 6.13: Block times and inter-arrival times

Its should be noted that our proposed timing model is not limited to our modular type of timing analysis with our clear separation between the local- and global low-level analysis phases. As mentioned in Section 3.5, for more complex hardware with a lot of inter-dependencies between processor components, a more integerated and potentially more complex low-level analysis is preferably used to reduce the pessimism in the resulting WCET estimate. The results from such analysis can in many cases be formulated as a classical IPET [HLTW03] (i.e., just having timing on nodes or edges), something which allow our calculation methods to use the timing models resulting from such more integrated types of low-level analyses.

The timing model we propose is also independent on how the hardware model (i.e., the timing function T) defines the execution time for a piece of code. The pipeline diagrams presented in this chapter assigns an execution time to a basic block, corresponding to the number of cycles its instructions are occupying stages in the pipeline. Other definitions are possible, for example, it is possible to describe the execution time of a pipelined processor by ignoring the overlap between instructions which "always" occurs, and rather use the *interarrival time*, which is done in many CPU manuals (see Figure 6.13 for an illustration).

Processor manuals written in this style give a number of cycles for each instruction rather than a detailed pipeline diagram, together with some rules for when the execution gets slowed down due to resource contention [ARM00]. Atanassov et al. [AKP01] have built a model for the pipelined Infineon C167 processor using constant execution times for each instruction, plus formulas that account for the interference between neighbouring instructions, the effect of memory access times, etc. We use a similar model for the ARM9 processor (see Section 10).

For a more detailed discussion of our pipeline analysis, timing model and the occurrence and causes of long timing effects we refer to Engblom's Ph.D. thesis [Eng02].

Chapter 7

Efficient Path-based Calculation

This chapter presents a fast and effective WCET calculation method which takes account of low-level machine aspects like pipelining and caches, and high-level program flow like loops and infeasible paths. The method is *path-based* in that the possible execution paths of a program or piece of a program are *explicitly* explored to find the overall program path with the worst case timing.

7.1 Introduction

There has been some other approaches to path-based calculation presented in the WCET literature e.g., [HAM$^+$99, SA00, RGL02] (see Section 2.4.2 for more details). The basic principle between all path-based methods are the same, extracting explicit longest paths over some type of graph structure in a bottom-up manner, replacing code subparts with a safe timing and hardware model abstraction.

The main difference between our approach and previous work is our clear separation of pipeline analysis and calculation. This allows us to construct a simpler calculation phase just working over a graph annotated with times and timing effects. Previous approaches [HAM$^+$99, SA00, RGL02] have explicitly integrated a hardware model state, including pipeline and sometimes cache information, in their path analysis. Thereby, our path-based calculation eliminates the need for an explicit enumeration of all paths to handle pipeline effects and becomes more efficient than previous presented path-based approaches, especially in the presence of many potential execution paths.

We further extend the presented path-based calculation method to handle complex program flow information such as dependent conditional statements and implication. The flow information are expressed using a subset of our flow fact language presented in Chapter 5. We extend the calculation method

further to use graph rewriting to handle pipeline timing effects between non-neighbouring basic blocks. We also show how to handle such long timing effects going over the calculation borders dictated by the calculation method. Our experiments demonstrate that speed does not sacrifice precision, and that programs with extreme numbers of potential execution paths can be analyzed quickly (see Chapter 10).

7.2 Method overview

Our path-based calculation module takes two inputs, a description of possible program flows, represented as a scope graph with flow facts (see Chapter 5), and a timing model, represented as a global timing graph generated by our low-level analysis (see Chapter 6).

7.2.1 Program flow

The description of possible program flows is represented by a scope graph annotated with flow facts, but the method can only handle a subset of the flow fact constructs presented in Chapter 5. The only flow facts allowed in our path-based calculation are *foreach* facts, i.e., facts with <*min..max*> or < > context specifiers. Furthermore, facts are only allowed to use count variables referring to nodes located in the defining scope of the fact.

These kinds of flow facts have a natural relation to path-based calculations, since they talk about what happens on a *single iteration* of a scope. Such flow facts can express complex flow information extracted using flow analysis method as presented in [Gus00, HAM^{+}99]. Flow information stretching over several scopes or expressed using total facts, i.e., facts using [*min..max*] or [ ] context specifiers, can not be handled.

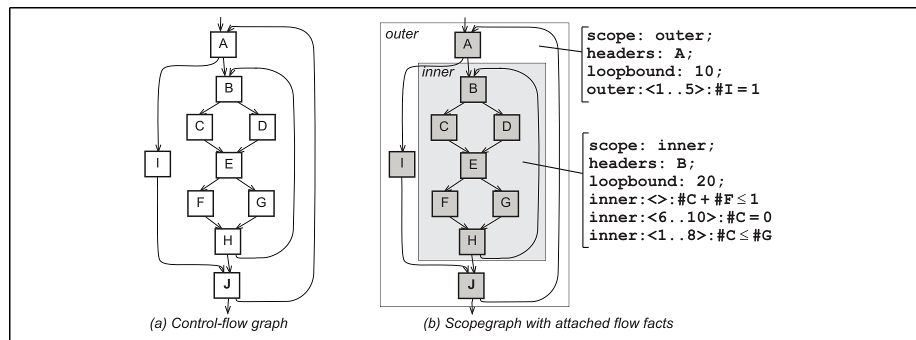


Figure 7.1: Control-flow graph and scope-graph with flow facts

Each scope in the scope graph is structurally restricted to only contain a single header node, thereby limiting the calculation method to only handle well-

formed loops. A scope might have several out-edges, allowing for non-local jumps out of loops and multiple return sites for functions. In Section 7.7 we outline possible method extensions to manage more advanced types of flow information and graph structures.

Figure 7.1(a) shows an example of a control-flow graph containing two nested loops. Figure 7.1(b) shows the corresponding scope graph with some attached flow facts. The fact inner : <> : #C + #F ≤ 1 gives that the nodes C and F can never execute on the same iteration of the scope (an infeasible path), while the fact inner : <6..10> : #C = 0 gives that for each entry of the inner loop, node C can not be executed during iteration 6 to 10. The fact inner : <1..8> : #C ≤ #G gives that, for each entry of inner, during the first eight iterations, an execution of C implies that G must also be executed. The fact outer : <1..5> : #I = 1 gives that for each entry of outer, during the first five iterations of outer, the execution is forced to take the path passing node I, (and can therefore not enter the inner scope during those iterations).

7.2.2 Program timing

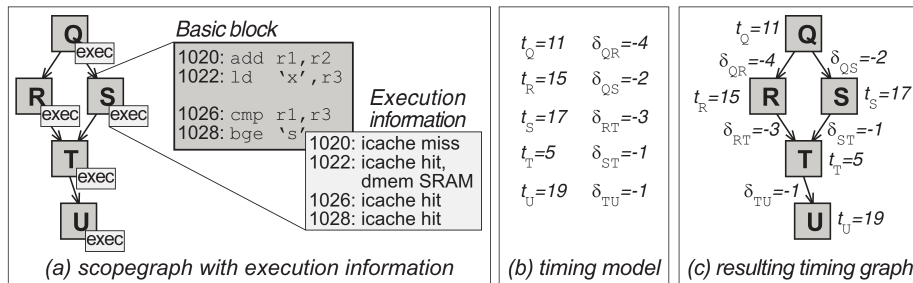


Figure 7.2: Scope graph fragment, timing model and timing graph

As presented in Chapter 6, the global low-level analysis annotates the nodes in the scope graph with execution information, and the local-low level analysis extracts timing for entities using a hardware model. The second input to the path-based calculation method is the result of the low-level analysis, a *timing model* represented as a *global timing graph*. The timing graph is a flattened version of the scope graph with timing on nodes and timing effects on sequences of nodes.

Figure 7.2(a) shows a fragment of an example scope graph annotated with execution information. Figure 7.2(b) shows the time model generated from the low-level analysis and Figure 7.2(c) shows the corresponding timing graph fragment. In the timing graph, times for nodes correspond to the execution times of nodes in isolation, (e.g., $t_{\mathtt{Q}}$ in Figure 7.2), and timing effect for sequences, (e.g., $\delta_{\mathtt{QR}}$ in Figure 7.2), to the pipeline effect when the two successive nodes are

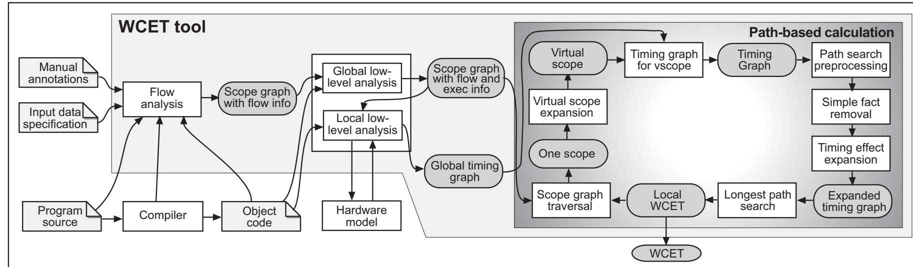


Figure 7.3: WCET analysis using path-based calculation

executed in sequence. The timing graph is generated once for the entire scope graph.

As explained in Chapter 6 there is a potential for timing effects along longer sequences of nodes than just two, usually caused by a node using some CPU resource that is used by a later node in the sequence, but not by the nodes in between. Such timing effects should only be included in the final WCET estimate if all the nodes in the sequence are executed in sequence. For example, in Figure 7.16(a) on page 114, we have a timing effect for the sequence `CDE`.

7.2.3 Calculation overview

Figure 7.3 gives an overview of our WCET analysis system, when using longest path-based search as described in this chapter. The path-based calculation module is shown in detail.

Our path-based method works bottom-up over the scopes in the scope graph, extracting worst case timing of descendant scopes before their ancestors (box "Scope Graph Traversal" in Figure 7.3). Within each scope we extract a number of worst case paths each valid for one single iteration. The extracted paths will be combined to form a worst case execution for the whole scope. The WCET time derived for the root scope in the scope graph becomes the WCTE estimate for the whole program.

To account for flow facts which make execution paths infeasible for all or some iterations, we unroll a scope into a number of *virtual scopes*, (box "Virtual Scope Expansion"). Fragments of the global timing graph are used to create a local timing graph for each such virtual scope (box "Timing Graph for VScope").

We apply an acyclic longest-path search algorithm over the local timing graph to extract a worst-case iteration path. Before the algorithm can be applied all cycles in the timing graph are removed, (box "Path Search Preprocessing"). Also, some nodes and edges in the timing graph are removed to account for certain types of simpler flow facts, such as facts specifying that a certain entity must or can't be taken (box "Simple Fact Removal"). Paths not fulfilling given flow information are removed until a feasible longest path is found, (box "Local Path Search").

```
Dijkstra's(timinggraph TG):
  // Initialization
  for each node u in TG do
    predecessor[u] := nil
    time_sum[u] := 0
  end for
  // Breadth-first-search
  for each node u in TG in breadth-first order do
    for each outgoing edge e = u → v in G do
      d := time_sum[u] + t_u + δ_e
      // Is u on the longest path to v
      if time_sum[v] < d then
        predecessor[v] := u
        time_sum[v] := d
    end for
  end for
  return TG
```

Figure 7.4: Longest path search algorithm

The timing graph is expanded to handle pipeline effects across sequences of nodes longer than two, (box "Timing Effect Expansion"). Special care has to be taken to handle long timing effects going over our analysis borders, e.g., effects over nodes which belong to different scopes.

The following sections will describe the algorithm steps in more detail. Section 7.3 present the basic longest path search algorithm used. Section 7.4 and Section 7.5 presents how the algorithms and data structures are modified to handle complex flow information and long pipeline effects.

7.3 Basic path search algorithm

The classical approach to longest executable path search in path-based calculation is to generate all possible paths for a certain program segment (function, loop body, or other unit), run all the paths through some kind of hardware model, and select the path with the longest execution time. The unit of analysis is the complete path, and the number of paths to explore is up to 2^n, where n is the number of decisions in the program segment being analyzed. The need to handle many complete paths arises from the use of pipelining in modern processors: to get a tight timing estimate, one must account for the overlap between basic blocks, and this can only be done by analyzing all the basic blocks in a path in a continuous sequence.[1]

However, since our pipeline analysis allows the timing of a path to be composed from smaller components, it is possible to reformulate the longest path search problem for a program segment as finding the longest path in a directed acyclic graph, eliminating the need for an explicit enumeration of all paths to

[1] To keep complexity under control while losing some precision, it is possible to cut a program segment into smaller pieces with a lower number of decisions in each [HAM+99].

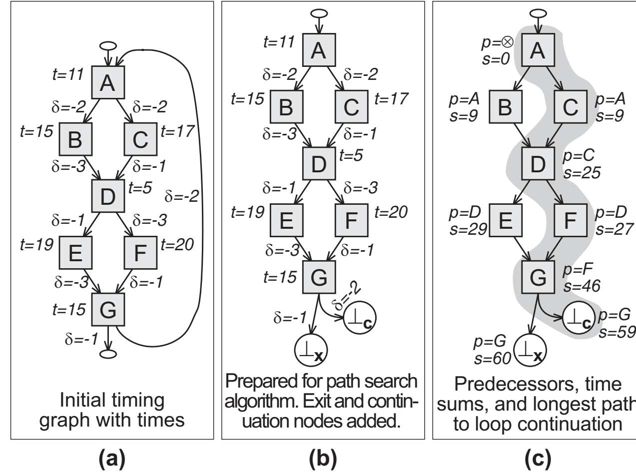


Figure 7.5: Longest path search

handle pipeline effects.

We base our efficient path search on Dijkstra's algorithm (shown in Figure 7.4) [CLR90]. The algorithm computes the longest path in $O(m+n)$ time where m is the number of edges and n is the number of nodes, i.e., it is linear in the size of the graph.

In order to be able to apply the algorithm on the timing graph TG for a scope s, we must first remove all cycles from the graph, since in general the longest path in a graph is undefined if there are cycles in the graph. Therefore, within each scope, we replace all back-edges (i.e., edges back to the header node of the scope) with edges to a special *continuation node* $\perp_c$. Furthermore, all edges leading to nodes outside the scope are redirected to a special *exit node* $\perp_x$ (see Figure 7.5(b)). This is the "Path Search Preprocessing" stage in Figure 7.3.

After this preprocessing, the algorithm works by breadth-first search. For each node, it computes the predecessor with the greatest total time on the longest path from the start node of the graph (called *time_sum* in the algorithm). If a node is not reachable from the header node, due to the removal of certain paths (Section 7.4), the corresponding *time_sum* is zero. Figure 7.5(c) illustrates the result of the algorithm, showing the predecessor and *time_sum* for each node. This is the "Longest Path Search" stage in Figure 7.3.

After each run, for each node n, *predecessor*[n] defines the predecessor of node n on the longest path from the start node to n. Thus, a path can easily be constructed backwards by following the *predecessor* chain.

The extracted worst case paths to $\perp_c$ and $\perp_x$, respectively, both corresponds to a single iteration of s. When computing the local WCET for a looping scope s, we have to give the last iteration a special treatment, since when exiting a scope, a different path is usually taken, which may be longer or shorter than the

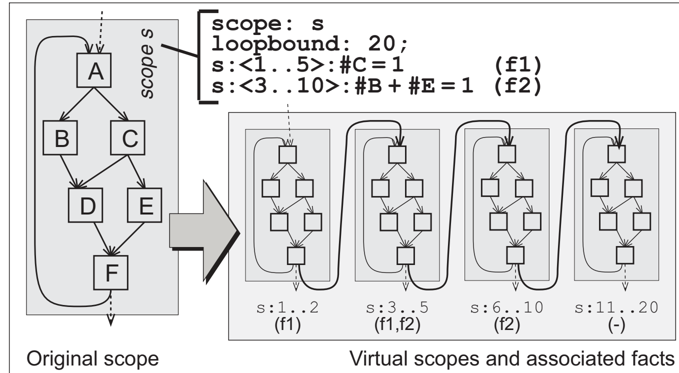


Figure 7.6: Virtual scope expansion

repeating path. Therefore, we calculate *two* longest executable paths in each scope: one to the special node $\bot_c$ and one to the special node $\bot_x$. If there is no executable path to $\bot_c$, s does not iterate at all, in which case the WCET for the scope is the longest executable path to $\bot_x$.

If there is a path going to $\bot_c$ the final WCET for scope s becomes: $\mathit{time_sum}(\bot_c) * (\mathit{loopbound}(s) - 1) + \mathit{time_sum}(\bot_x)$.

7.4 Path search with flow facts

In this section, we show how flow facts can be used to remove infeasible paths in the path-based calculation, and thus obtain more precise WCET estimates.

7.4.1 Ranges and Virtual Scopes

In order to account for flow facts with ranges, i.e., facts with a <*min..max*> context specifier, we expand a scope into a number of *virtual scopes*. A virtual scope corresponds to a certain range of iterations of a scope, and the virtual scope expansion will, creates a copy of the original scope for each virtual scope. Each virtual scope has a corresponding *virtual scope specifier*, `s:range`, specifying which scope and range the virtual scope is valid for.

The purpose of the virtual scope expansion is to make sure that each fact attached to a virtual scope has a range covering the *entire* iteration range of a virtual scope, as illustrated in Figure 7.6. Here, the two facts `s : <1..5> : #C` $=$ 1 and `s : <3..10> : #B` $+$ `E` $= 1$ are specified for the scope `s`. Both facts hold for the iterations 3..5. Only fact `f1` holds in iterations 1..2, and `f2` in iterations 6..10. In iterations 11..20, none of the facts hold. Thus, the scope is split into the virtual scopes `s:1..2`, `s:3..5`, `s:6..10`, and `s:11..20`. After the expansion, we note which facts are valid for each virtual scope.

```
VirtualScopeSpecCreation(scope s):
  VSS := ∅, min := 1
  F := facts in scope graph having s as defining scope
  F_current := facts in F spanning first iteration in s
  // Loop over all iterations in the scope
  for each iteration iter between 1 and loopbound(s) do
    F_iter := facts in F spanning iteration iter in s
    // Has set of spanning facts changed
    if F_current ≠ F_iter then
      max := iter - 1
      VSS := add virtual scope spec s : min..max to VSS
      min := iter
      F_current := F_iter
  end for
  // Return set of created virtual scope specifications
  return VSS
```

Figure 7.7: Virtual scope generation

```
SimpleFactsRemoval(timinggraph TG):
  // Handle 'forbidden' nodes, (#node = 0)
  for each forbidden node n in TG do
    delete n from TG
    remove resulting dead paths
  end for
  // Handle 'must-have' nodes, (#node = 1):
  for each must-have node n in TG
    mark all transitive predecessors of n
    mark all transitive successors of n
    for each node m in TG
      if m not marked
        delete node m from TG
    end for
  end for
  return TG
```

Figure 7.8: Simple facts removal

An algorithm for finding virtual scope specifications over a scope is given in Figure 7.7. For each created virtual scope specification a corresponding virtual scope will be created.

7.4.2 Simple facts removal

To make the path search with facts more efficient, certain facts can be handled in a *preprocessing* stage (the "Simple Fact Removal" stage in Figure 7.3). The algorithm for this step is shown in Figure 7.8.

A fact with a constraint expression of the form $\#node = 0$, stating that *node* must not be taken in the covered iterations, can be handled by simply removing *node* from the corresponding virtual scope. A fact with a constraint expression of the form $\#node = 1$, stating that *node* must be taken on each iteration, can be handled by removing all paths from the graph that do not include *node*.

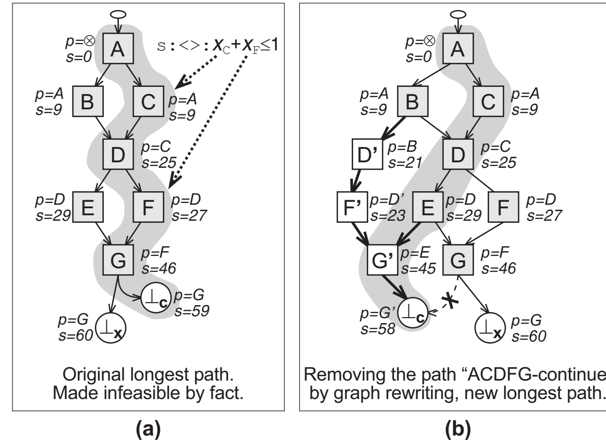


Figure 7.9: Infeasible path removal

```
LongestFeasiblePathSearch(timinggraph TG, factset F, node n_end):
  // Extract longest path p in TG
  TG = Dijkstra's(TG)
  begin loop
    // Get timing and longest extracted path
    t_p := time_sum(n_end)
    p := longest path from startnode(TG) to n_end
    // Return if p feasible against flow facts
    if Feasible(p, F) then return t_p
    // Return if there was no path to n_end
    if t_p == 0 then return t_p
    // Else, remove p from TG and redo extraction
    TG := DeletePathFromGraph(TG, p)
  end loop
```

Figure 7.10: Longest feasible path search

The paths can be found in time linear to the size of the graph by first marking all transitive predecessors and successors of *node*, and then removing all nodes that are not marked. Any such node would not be on a path from start to end involving *node*, and is thus infeasible.

7.4.3 Longest path search with flow facts

The longest path found by Dijkstra's algorithm is then checked for feasibility against the flow facts not removed in the preprocessing. The checking is done by counting the occurrence of entities in the path, and comparing this to the constraints specified in the facts. For example, for a fact like "`inner : <> : #C` + `#F` ≤ 1", we will check that the path does not contain both node `F` and node `C`.

If the path is not feasible, it is removed from the graph and the search begins again, now finding the second-longest path. The path is removed using

an algorithm by Martins and Santos [MS00], and the effect is illustrated in Figure 7.9. The idea is to create a deviation around the path to be removed. This is achieved by adding copies of already existing nodes and edges to the graph as shown in Figure 7.9, and removing the end of the original path. Note that the modified graph still contains all paths of the original graph, except precisely the removed one. All new nodes and edges have the same timing as their originals in the timing graph. During the path removal, the algorithm also computes the next longest path in the graph by updating the path information $time_sum[v]$ and $predecessor[v]$ (see Section 7.3 above) for all affected nodes v (thereby avoiding another pass of Dijkstra's algorithm).

The process of longest path search and infeasible path detection and removal is repeated until a feasible path is found. The first feasible path found is the longest executable path in the virtual scope. The algorithm for longest feasible path search is given in Figure 7.10.

The removal of a path and finding the next longest one runs in $O(m)$ time, where m is the number of edges in the graph. This comes from the fact that for each new node the *time_sum* and *predecessor* are computed by traversing all incoming edges of the corresponding original node (see [MS00] for details). However, this complexity is obtained only if the next longest path is found after scanning the entire set of edges, which is the case only when a path passes all nodes. For a typical flow graph, this is not realistic. Thus the actual complexity can be assumed to be much lower.

After one initial run of Dijkstra's Algorithm, the path search algorithm runs in $O(K * m)$ time, where m is the number of edges in the graph and K is the number of paths removed. As the number of paths in a flow graph grows exponentially with the number of decisions (see Section 7.3 above), the whole path search might take exponential time, since in the worst case all paths have to be examined (if the shortest path is the only feasible path). However, for typical programs this is very unlikely and may only happen when we have many complex flow facts covering the same virtual scope. It was not a noticeable problem for any of our benchmark programs (see Section 10.8). Thus, in general, the number of paths examined should be low compared to the total number of possible paths. Also note that since the calculation successively improves the WCET estimate in each step, it can be interrupted at any time, still yielding a safe, but probably pessimistic result.

7.4.4 WCET calculation algorithm

After applying the modifications to the graph as described above, we start searching for the longest executable path allowed by the remaining facts. We have a fragment of the global timing graph corresponding to a virtual scope, with back-edges removed and special nodes added, and a set of remaining facts.

Figure 7.11 shows the top-level algorithm, used for each scope. It performs WCET analysis recursively for the subscopes, divides the scope into virtual scopes, retrieves a piece of the timing graph, and removes the paths corre-

```
WCETCalculation(scope s, timedatabase s):
  // Initialize timing variables for scopes and virtual scopes
  t_{s,cont} = t_{s,cont} = t_{cont} = t_{vs,exit} = t_{vs,exit} = t_{exit} = 0
  // Extract timing for subscopes
  for each child scope sub to scope s do
    t_sub := WCETCalculation(sub)
    s := in s replace call to sub with new call node n_sub
    add time t_sub for node n_sub to tdb
  end for
  // Divide scope s into virtual scopes
  VS := VirtualScopeCreation(s)
  // Calculate times for virtual scopes
  for each virtual vs in VS in increasing order do
    // Get and convert timing graph for virtual scope
    TG := TimingGraphFragment(s, vs)
    TG := PathSearchPreprocessing(TG)
    TG := SimpleFactRemoval(TG)
    TG := LongTimingEffectExpansion(TG)
    // Get time for longest continuation and exit paths
    t_cont := LongestFeasiblePathSearch(TG, F, ⊥_c)
    t_exit := LongestFeasiblePathSearch(TG, F, ⊥_x)
    // Does there exist a feasible continuation path?
    if valid(t_cont) then
      // Update longest time time to back-edges for scope
      t_{s,cont} := t_{s,cont}+ span(vs) *t_cont
      // Update longest time time to out-edges for scope
      if valid(t_exit) then
        t_{s,exit} := t_{s,cont} + (span(vs)-1) * t_cont + t_exit
    else // We must take the exit path
      if valid(t_exit) then
        t_{s,exit} := t_{s,cont} + t_exit
      return t_{s,exit}
  end for
      // Return the longest time to out-edges
  return t_{s,exit}
```

Figure 7.11: Path-based WCET algorithm

sponding to the simple facts. By calling the algorithm with the root scope of the scope graph, a WCET for the complete program is generated.

The algorithm creates a number of virtual scopes for each scope as given by the flow facts defined on the scope. The algorithm then iterates over the created virtual scopes in increasing range order. For each virtual scope, two timing values are calculated, one time t_{cont} to the continuation node $\perp_c$ and one time t_{exit} to the exit-node $\perp_x$. The longest path in each virtual scope is generated as described in Section 7.3 above.

The times extracted for the virtual scope are combined together to form a WCET for the complete scope. For each scope s we have one variable $t_{s,exit}$ holding the worst case timing accumulated for scope s to $\perp_x$, and one variable $t_{s,cont}$ holding the worst case timing accumulated to $\perp_c$. The final $t_{s,exit}$ value extracted is the worst case timing for the scope, and is returned.

A potential optimization reducing the path search needed is to compare the

```
LongTimingEffectExpansion(timinggraph TG):
  // Breadth-first-search
  for each node v in TG in breadth-first order do
    if in_degree[v] > 1 and v in long timing effect then
      for each incoming edge u → v inside a sequence do
        // Copy v and add and redirect edges
        add node v' to TG
        add edge u → v' to TG
        remove edge u → v from TG
        for each outgoing edge e = v → w in TG do
          add edge v' → w to TG
          // Add long timing effect to edge
          if e is last in a timing sequence s then
            add δ_s to weight of e
        end for
      end for
  end for
```

Figure 7.12: Long timing effects expansion

set of facts covering the different virtual scopes. If the set of facts covering a virtual scope vs_i is a subset or equal to the set of facts covering a virtual scope vs_j then all paths removed from vs_i can be safely removed also from vs_j. This means that when doing a longest path search over vs_j we can instead start with the resulting graph after the longest-path search over vs_i.

7.5 Handling long pipeline effects

The basic longest path-search algorithm presented in Section 7.3 manages timing effects up to the length of two. But, if there are pipeline effects across sequences of nodes longer than two, they must be considered during the path search since they might affect the longest path. Path-based calculation methods have previously required complete paths to be executed to capture such effects [HAM$^+$99, SA00, RGL02], while here we show how to capture the effects locally by graph rewriting.

To account for such effects, we need to know when we have taken a path containing the sequence of nodes corresponding to the timing effect. Since the longest path search only looks at the predecessors for a node, a preprocessing algorithm, given in Figure 7.12 and corresponding to the box "Timing Effect Expansion" in Figure 7.3, is used.

The idea behind the preprocessing algorithm is to make each path that contains a long timing effect separate in the graph, and to add the time of the timing effect to the last edge in the sequence. This is obtained by traversing the nodes in the graph in topological order, and for each node visited, if it is inside a timing effect sequence (i.e., not the first or last of a sequence), and has more than one incoming edge, it is copied together with its outgoing edges and the incoming edge that is part of the sequence. The original incoming edge

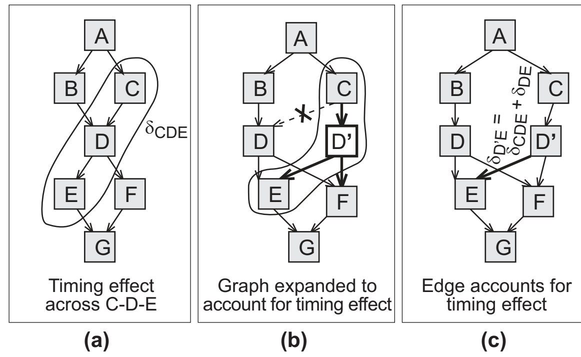


Figure 7.13: Simple timing effect

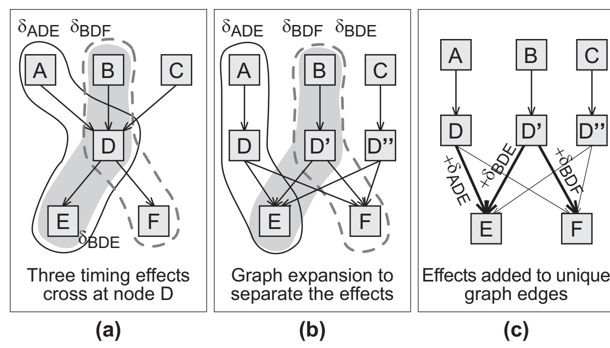


Figure 7.14: Crossing timing effects

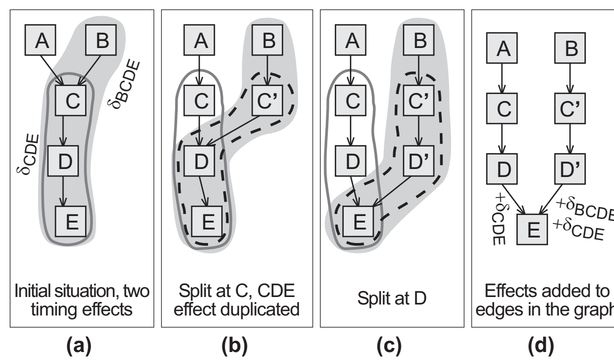


Figure 7.15: Overlapping timing effects

is removed. The copied nodes and edges have the same timing effects as the original nodes and edges.

In Figure 7.13 the expansion process for a timing effect $\delta_{\mathtt{CDE}}$ on a graph fragment is shown in more detail. The node `D` is copied to `D'` and the $\delta_{\mathtt{CDE}}$ timing effect is added to the $\delta_{\mathtt{DE}}$ timing effect, to create the new timing effect $\delta_{\mathtt{D'E}}$.

Figure 7.14 shows what happens when several time effects cross each other. A number of new nodes is added, and three edges are rewritten. Figure 7.15 shows the case where two timing effects overlap. The graph must be expanded in such a way that we count both effects only if the sequence `BCDE` is taken, and only the effect over `CDE` if only that path is taken. The end result is a graph where the effect $\delta_{\mathtt{CDE}}$ is added to two edges.

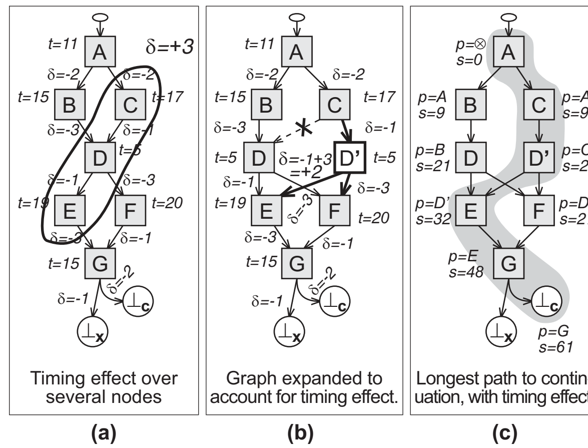


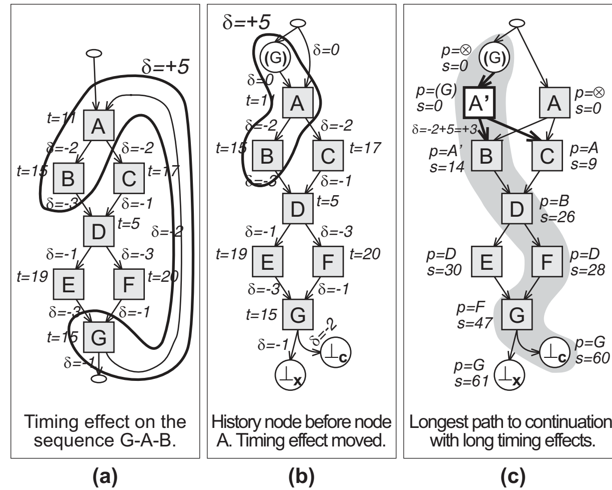
Figure 7.16: Path search with timing effects

An example showing that the long timing effects might affect the longest path is shown in Figure 7.16, where the timing effect on the sequence `CDE` increases the execution time of that path by 3 cycles, making the longest path different from the one shown in Figure 7.5.

7.5.1 Long timing effects over analysis boundaries

One problem with a bottom-up WCET calculation such as the one presented in this chapter is that there might be long timing effects across the boundaries between successive paths or scopes, which have to be handled in order to ensure a safe WCET estimate. For pipeline effects across scope boundaries, all timing effects of length two are accounted for at the scope where the edges begin. For timing effects of longer length crossing calculation borders, a safe but potentially pessimistic solution is needed.

The simplest way to handle such long border crossing timing effects is to



Timing effect on the sequence G-A-B. History node before node A. Timing effect moved. Longest path to continuation, with long timing effects.

(a) (b) (c)

Figure 7.17: Timing effect across back-edge

ignore all pipeline effects between nodes in different scopes and loop iterations, i.e., pessimistic but safely assume that we always have an empty pipeline when we start a new iteration and exiting and entering scopes[2].

Our less pessimistic (but still not perfect) solution is to add special *history nodes* to the timing graph. The history nodes represent the potential paths taken *before* the beginning of a path search and therefore have no weight associated with them. Long timing effects will then begin at history nodes and end at a node in the current scope (as shown for the timing effect from "`(G)`" to `B` in Figure 7.17(b)).

For each node v in the graph that is the target of a back-edge, return from a child scope call or an entry to the scope, we collect all timing effects with a length greater than two (those of length two are already handled as regular edge times). For each such timing effect we create a history node labelled with the sequence of nodes up to v, and insert it between the start node and the node v. For example, the sequence `GAB` in Figure 7.17, makes us insert the history node "`(G)`". This changes the longest path from `ACDFG` as shown in Figure 7.5 to `ABDFG` as shown in Figure 7.16(c).

The insertion of history nodes gives a safe but possibly pessimistic estimate of the execution times, since we will always use the worst incoming timing effect. This remaining pessimism is the price we have to pay for the convenience and efficiency of extracting WCET times for scopes and virtual scopes in isolation. In Chapter 9 more details on how to handle border crossing timing effects will be given.

---

[2]This is safe if we assume that a two basic blocks can never have a positive timing effect between them, i.e., in the worst case, we assume that two basic blocks executed on end in a pipeline will have no overlap.

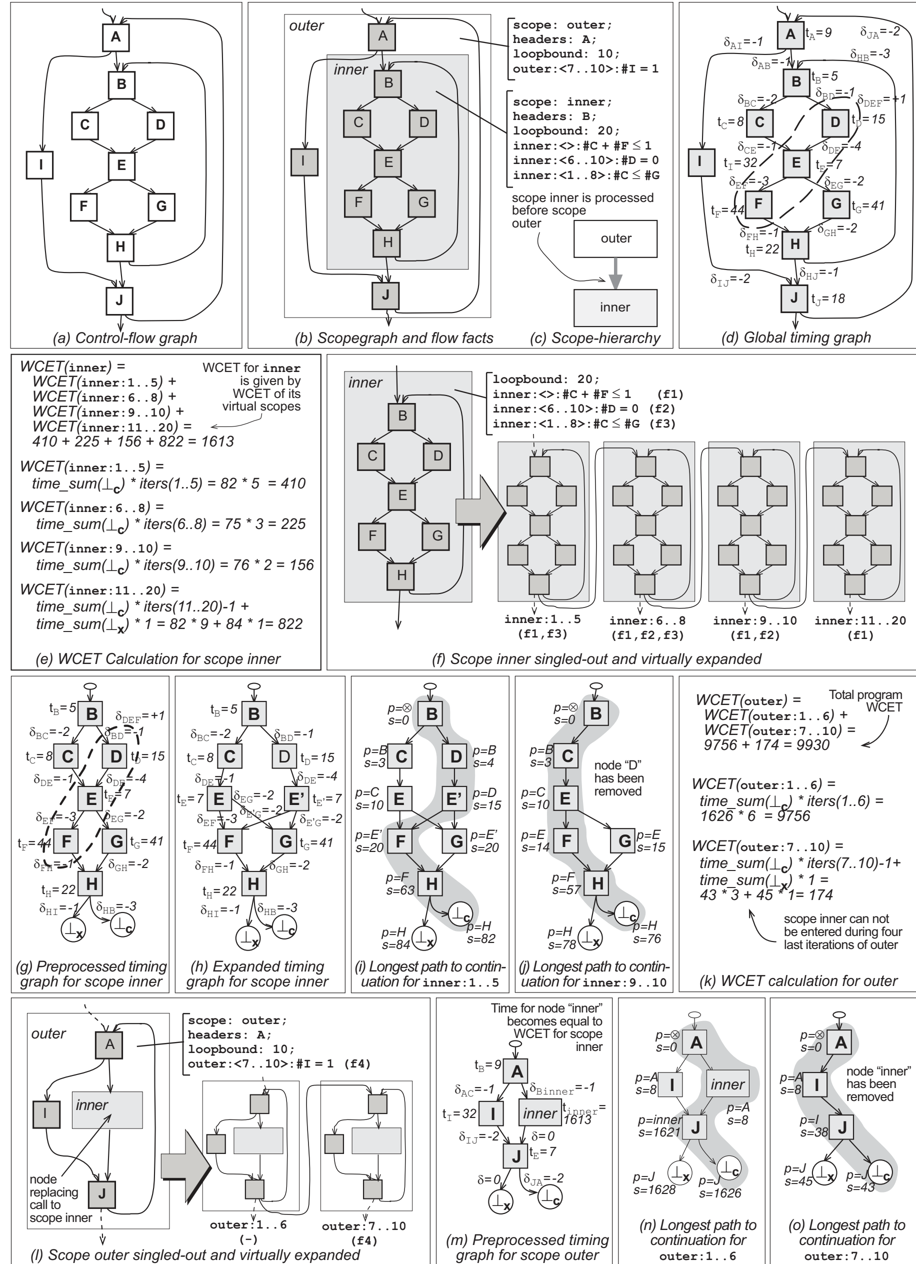


Figure 7.18: Complete path-based WCET calculation example

7.6 Complete example

In Figure 7.18(a)-(m) we give a compact illustration of the steps involved in our path-based calculation method.

Figure 7.18(a) shows an example control-flow graph consisting of two nested loops. Figure 7.18(b) shows the corresponding scope graph with the scopes `inner` and `outer` and some attached flow facts. Figure 7.18(c) shows the corresponding scope-hierarchy. Figure 7.18(d) illustrates the global timing graph generated from the scope graph by our low-level analysis.

The path-based calculation is performed bottom-up over the scopes in the scope-hierarchy, starting with the `inner` scope. The flow facts attached to `inner` will generate four different virtual scopes, `inner:1..5`, `inner:6..8`, `inner:9..10` and `inner:11..20` as illustrated in Figure 7.18(f). A timing graph fragment corresponding to `inner` is singled out, made acyclic and expanded for long timing effects as given in Figure 7.18(g) and Figure 7.18(h). The longest path-search method is then applied to the timing graph for each of the virtual scopes. Figure 7.18(i) illustrates the longest path extracted for virtual scope `inner:1..5` fulfilling the `f1` and `f3` flow facts. Figure 7.18(j) shows the longest path extracted for virtual scope `inner:9..10` fulfilling the `f1` and `f2` flow facts. Note that node `D` has been removed due to fact `f2`.

The times for extracted paths over the different virtual scopes are combined to form a WCET for each virtual scope. The virtual scope WCETs are combined to form the WCET of 1613 clock cycles for the whole `inner` scope as illustrated in Figure 7.18(e).

After calculating a WCET for the `inner` scope the calculation proceed with its parent scope `outer`. A special node is created in the `outer` scope corresponding to the call to `inner`. Two virtual scopes `outer:1..6` and `outer:7..10` are created for the `outer` scope as illustrated in Figure 7.18(l). The timing graph fragment corresponding to `outer` is extracted and the `inner` call node is given the timing extracted for the `inner` scope as illustrated in Figure 7.18(m). For virtual scope `outer:1..6` the longest feasible path will include this call node, as illustrated Figure 7.18(n), while for virtual scope `outer:7..10` the execution must go over the `I` node, as illustrated in Figure 7.18(o).

The times for the two virtual scopes are combined to form a WCET of 9930 clock cycles for the whole program as illustrated in Figure 7.18(k).

7.7 Possible method extensions

7.7.1 Managing more advanced flow information

One limitation with the current version of our path-based calculation method is the inability to handle flow facts which are giving flow information over collections of paths, i.e., total facts or facts referring to the execution of entities in descendant scopes. One key observation is that for such facts it is often not

enough to extract a single path, but instead we need to combine several paths that together satisfy given flow facts.

A small illustrating example is the following: Assume that a total fact s : [] : #A = #B + 2 is given over a scope s with a loop bound of 10. Both node A and B are located in scope s. We can not extract a single path that fulfils the given fact when executed on all ten iterations. Instead, we need to extract a number of longest paths which together form the WCET of s.

We here outline a possible extension of our path-based method which allows us to handle non-overlapping total flow facts. The algorithm works in two steps. In the first step we extract a number of longest paths. In the second step we create a small constraint system to extract the worst case path combination which together satisfies the given flow fact. We will use the given fact to illustrate the different steps in more detail.

We first create a virtual scope s:1..10 and corresponding local timing graph for our path-based method. To extract longest paths for virtual scope s:1..10 we note that only two different nodes, A and B are referred by the fact. We are therefore interested in the longest paths that include respectively not include these nodes. Each extracted path must be structurally feasible and not contradicting any foreach facts which overlap the virtual scope. For s:1..10 the longest paths of interest are:

- $p_{\mathtt{A},\mathtt{B}}$: the worst case path passing both A and B.
- $p_{\mathtt{A},\neg\mathtt{B}}$: the worst case path passing A but not B.
- $p_{\neg\mathtt{A},\mathtt{B}}$: the worst case path not passing A but B.
- $p_{\neg\mathtt{A},\neg\mathtt{B}}$: the worst case path neither passing A or B.

The set P holds all the extracted paths. Note that no other paths need to be extracted, since any other path will have lower execution time and have the same properties in terms of passing / not passing A and B as one of the already extracted paths.

In the second step we extract the worst case path combination fulfilling the given flow fact by creating a small constraint system. Let t_i hold the timing of path p_i and let count variables x_i hold the number of times path p_i is executed. For example, $t_{\mathtt{A},\mathtt{B}}$ holds the timing and $x_{\mathtt{A},\mathtt{B}}$ is the count variable of path $p_{\mathtt{A},\mathtt{B}}$. The additional count variables, $x_{\mathtt{A}}$ and $x_{\mathtt{B}}$, are created to hold the number of times nodes A and B are executed. A number of constraints are created to constrain path executions and relate their executions to the given flow fact.

Figure 7.19 shows the constraint system for this example. The resulting maximization problem will extract a WCET for s:1..10 with the given constraints. A solver would also provide an instantiation of the given count variables, i.e., the combination of paths that produces the worst case timing.

The method can be further extended to handle facts with overlapping ranges in a fashion similar to the IPET-based calculation method presented in Chapter 8. Similarly, to handle facts with count variables referring to entities located in descendant scopes we can no longer extract WCET for a single scope in isolation but need to combine paths in several different scopes.

// *Original flow fact*
s : [] : #A = #B + 2
// *Paths of interest*
$p_{\mathtt{A,B}}$, $p_{\mathtt{A},\neg\mathtt{B}}$, $p_{\neg\mathtt{A},\mathtt{B}}$, $p_{\neg\mathtt{A},\neg\mathtt{B}}$
// *Limit the number of times the different paths can be taken*
$x_{\mathtt{A,B}} + x_{\mathtt{A},\neg\mathtt{B}} + x_{\neg\mathtt{A},\mathtt{B}} + x_{\neg\mathtt{A},\neg\mathtt{B}} \leq iters(1..10) = 10$
// *Collect all paths executing A respectively B*
$x_{\mathtt{A}} = x_{\mathtt{A,B}} + x_{\mathtt{A},\neg\mathtt{B}}$
$x_{\mathtt{B}} = x_{\mathtt{A,B}} + x_{\neg\mathtt{A},\mathtt{B}}$
// *Flow fact converted to constraint*
$x_{\mathtt{A}} = x_{\mathtt{B}} + 2$
// *Problem to maximize*
$WCET = max(\sum_{p_i \in P} x_i * t_i) =$
$max(x_{\mathtt{A,B}} * t_{\mathtt{A,B}} + x_{\mathtt{A},\neg\mathtt{B}} * t_{\mathtt{A},\neg\mathtt{B}} + x_{\neg\mathtt{A},\mathtt{B}} * t_{\neg\mathtt{A},\mathtt{B}} + x_{\neg\mathtt{A},\neg\mathtt{B}} * t_{\neg\mathtt{A},\neg\mathtt{B}})$

Figure 7.19: Paths and constraint system

A potential advantage of a combined path- and constraint-based approach is that the resulting constraint system becomes very small, i.e., a lot of preprocessing is made in the longest path search part. The disadvantage is that the number of paths to extract within a scope might rapidly grow with the number of virtual scopes and referred count variables. The method has not been implemented and will therefore not be further elaborated upon.

7.7.2 Differentiating between out-edges

When calculating a WCET estimate for a scope s the least costly alternative is to calculate a WCET estimate for all out-edges s simultaneously. This will result in a safe but sometimes pessimistic estimate.

One cause of pessimism is that flow facts might sometimes prohibit the execution to leave a scope at a certain out-edge or that the timing for a scope becomes different depending on where the execution left the scope. The pessimism can be resolved by calculating a separate WCET for each individual out-edge of the scope, i.e., having several $\perp_x$ nodes in the timing graph. The cost to pay for the reduced pessimism would then be a more costly calculation.

In Section 9.4.2 we will elaborate further on calculating timing for program fragments with several entry and exit locations.

7.7.3 Handling unstructured code

Another limitation of the current version of our path-based method is its inability to handle unstructured code. For example, a scope can only have one in-node which must also be its only header-node. Since unstructured code exists in embedded systems and might be produced by compiler optimizations, this is a potentially severe limitation [Eng99a]. We will here outline some possible method extensions for handling this problem.

To capture that we can enter a scope at one or several nodes not being the header node of the scope we can create a separate virtual scope for the zero:th

iteration. During this iteration the execution can take nodes in the scope which are not the header node. The execution must not be able to leave the scope before a header node has been taken.

For scopes having multiple header nodes the calculation becomes more complicated, but we can still extract one or several worst-case paths corresponding to a single iteration. Remember that in the case of multiple header-nodes an iteration is defined to start at any of the header nodes and end at a back- or out-edge. The calculation must then guarantee that the final extracted longest path becomes continuous, i.e., if an iteration i ends at a back-edge with header h as target then iteration $i + 1$ must start at header h.

Chapter 8

Extended IPET Calculation

In this chapter we present a WCET calculation method using integer linear programming (ILP) or constraint programming (CP) techniques to determine a WCET estimate. In the WCET literature this type of calculation technique is called *implicit path enumeration technique, (IPET)*, reflecting that the longest path no longer is explicit but implicitly represented [PS95, FMW97, HLS00b, LM95]. Our method is able to handle more complex flow and timing information than previously presented IPET calculation methods, thereby allowing for tighter WCET estimates to be obtained.

Our calculation method takes two inputs: a scope graph with flow facts, representing possible program flows (see Chapter 5), and a timing model with timing for nodes and timing effects for sequences of nodes, representing hardware timing (see Chapter 6). The use of our timing model allows the calculation method to handle the effects of hardware without using a detailed hardware model in the calculation.

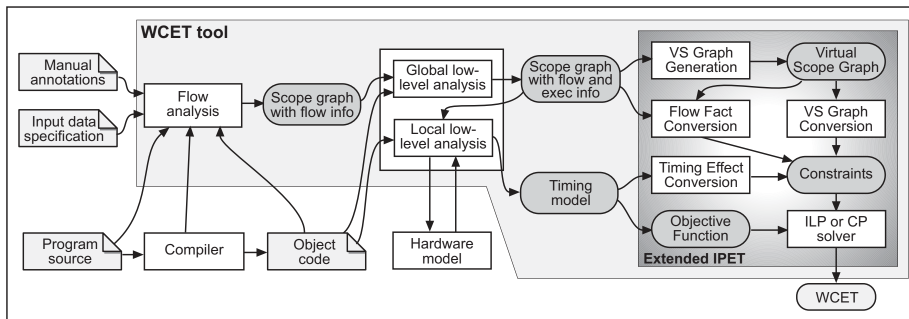


Figure 8.1: WCET tool architecture for IPET calculation

Figure 8.1 gives an overview picture of our WCET tool when using extended IPET. All components of the system, except the calculation phase, remain un-

changed, demonstrating the modular structure of the tool.

The first section of this chapter presents the basics of constraint-based calculation. The succeeding sections describe our constraint-based calculation method in more detail. Section 8.2 presents how the input scope graph is expanded into a *virtual scope graph*, to correctly account for given flow facts. Section 8.3 presents how constraints are generated from the virtual scope graph and the input flow facts. Section 8.4 presents how to generate an objective function, whose maximization, subject to generated constraints, will produce a WCET estimate. The section also shows how to safely constrain the impact of long timing effects. Finally, Section 8.5 gives the overall WCET calculation algorithm and a complete example of our constraint-based calculation method.

8.1 IPET calculation basics

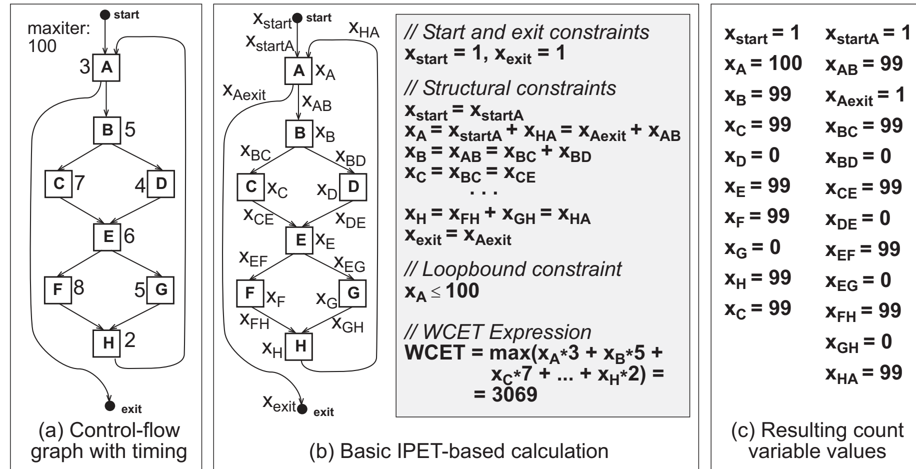


Figure 8.2: Basic IPET calculation

An IPET calculation method builds upon reformulating the WCET calculation problem into a maximization problem (or minimization for BCET), where the program structure, restrictions on program flow and execution times are given as algebraic and/or logical constraints. Each basic block and/or program flow edge in the program is given a time variable (t_{entity}), denoting the execution time of the node or edge, and a count variable (x_{entity}), denoting the number of times the node or edge is executed. For example, node B in Figure 8.2(b) has timing $t_{\texttt{B}} = 5$ and count variable $x_{\texttt{B}}$.

The count variables are considered *global* for the program (part) for which WCET is calculated, and their values reflect the total number of executions of entities for the complete execution of the program (part). For example, $x_{\texttt{B}}$ holds the total number of times that node B is executed over the complete program

(part) execution.

The possible flows through the program structure are modelled using *structural constraints*. For a node to be executed a number of times the execution must have entered the node through its incoming edges and exited the node through its outgoing edges the same number of times. For each node, two constraints are generated: One constraint sets the sum of count variables for in-edges to be equal to the node count variable, and one constraint sets the node count variable to be equal to the sum of count variables for out-edges. For example, in Figure 8.2(b) the constraints $x_{\texttt{CE}} + x_{\texttt{DE}} = x_{\texttt{E}}$ and $x_{\texttt{E}} = x_{\texttt{EF}} + x_{\texttt{EG}}$ are generated for node `E`. The structural constraints are given by the program graph layout and need not be described by any additional flow information.

Constraints are also needed to guarantee the finiteness of the program and the termination of the calculation. The finiteness is ensured by giving an upper bound on the number of times each looping construct in the code can iterate. This is done by bounding the count variable of the header node of the loop. For example, for the program in Figure 8.2(b) the constraint $x_{\texttt{A}} \leq 100$ gives a total bound on the number of iterations of the inner loop.

Additionally, constraints are needed to make sure that the program starts and exits once. This is done by setting the execution count of the program start node and the program exit node to one, (i.e., $x_{start} = x_{exit} = 1$).

To constrain the possible flows through the program additional constraints can be given on the count variables, limiting the total number of times the corresponding entity can be executed. The type of constraints allowed depends on the type of solver used, e.g., using a linear solver limits the flow constraints to be linearly formulated.

The final WCET estimate is generated by maximising an objective function, i.e., a sum of products of execution count variables and timing, subject to the constraints reflecting the structure of the program and possible flows: $WCET = max(\sum_{i \in entities} x_i * t_i)$. The maximisation problem can be solved using a constraint programming [OS97] (CP) or integer linear programming (ILP) [PS95, FMW97, HLS00b, LM95]. Constraint solvers allows for more complex constraints to be expressed, but with a potential risk of longer calculation times.

The result of the maximisation is a WCET estimate as well as a worst case count for each entity count variable, and not an explicit path like in path-based calculation. Figure 8.2(c) illustrates generated worst case count values for entities found in Figure 8.2(a). Note that there is no information about the resulting execution order, only information on the total amount of executions made for each entity in the graph.

The constraint-based calculation method presented in the following sections extends the basic IPET calculation method. By allowing the full expressive power of our flow fact language, the type of flow information possible to handle using IPET is extended. By generating constraints to safely capture the impact of timing effects for longer node sequences, the method also extends the type of timing information that can be taken into account.

8.2 Expanding the scope graph

The first input to our extended IPET calculation method is a scope graph with flow facts that together represent possible program flows. Both total and foreach facts, valid for all or certain ranges, are allowed. Flow facts can refer to count variables corresponding to entities located in the defining or descendent scopes of the fact. Scopes are allowed to contain one or more header nodes, and to have several in-nodes and out-edges, something which allow us to handle most types of unstructured code. To simplify the presentation all in-nodes are assumed to also be header-nodes.

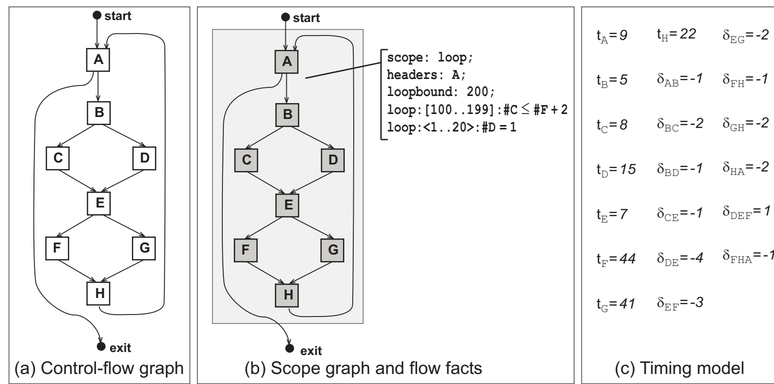


Figure 8.3: Example scope graph and timing model

The type of flow constraints allowed by basic IPET can be said to have the same expressive power as total/all flow facts with a defining scope equal to the root scope in the scope graph. However, there are many types of flows which we are able to express in our flow fact language which can not be formulated using constraints in classical IPET. For example, `loop : <1..20> : #D` = 1 illustrated in Figure 8.3, gives flow constraints in a local `loop` scope context and valid only for `loop`'s first 20 iterations.

Comparing flow facts to the IPET flow constraints in more detail, we note that our flow fact language uses *scope-local* semantics, while IPET uses a *global* semantics. Therefore, to use IPET for calculating a WCET estimate we need to convert the local flow information into a consistent set of constraints reasoning about variable values valid over the complete program execution.

Furthermore, flow facts with ranged context specifiers only provides *partial* information on the executions of referred entities. This means that such flow facts cannot be converted to constraints valid for all executions of an entity. Also, flow facts can interact with each other, e.g., by having overlapping ranges or referring to the same entity count variable, and we need to make sure that all facts together get safely accounted for in the final WCET estimate.

The following subsections will outline how the scope graph and flow facts are converted to constraints to account for the flow information provided by flow

facts. The basic approach is as follows: we use flow facts to create specifications of execution environments that need to be differentiated to correctly account for flow information provided by flow facts. The specifications are used to extend the input scope scope into a *virtual scope graph*, consisting of one or more *virtual scopes*. Each virtual scope is a physical representation of a specific execution environment but also a copy of a scope in the original scope graph, including copies of nodes and edges.

In Section 8.3 we show how to convert the flow facts to constraints over executions of entities in the virtual scope graph. Additional constraints are generated from the virtual scope graph reflecting possible program flows and the structure of the graph.

8.2.1 Iteration space partitioning

We start by showing how flow facts are used to create specifications of execution environments that should be differentiated. The need to differentiate between execution environments comes from flow facts with ranged context specifiers, only valid for one or a few iterations of a particular scope. For example, a flow fact like inner : [5..10] : #A ≤ 3 only constrains the execution of A when the iteration count of scope inner is between 5 and 10. This means that we cannot convert such a flow fact to one single constraint valid for all executions of A.

Instead, we note that we can partition the executions of A into two sets; the ones taking place when the iteration count of inner is between 5 and 10 and the ones taking place when the iteration counter of inner is something else. Only the executions taking place when the iteration count of inner is between 5 and 10 should be constrained. The observation is used to break down the iteration space of a scope into non-overlapping subranges, where each flow fact is valid for all iterations of a subrange.

The iteration space partitioning of a scope creates one or more *scope:range* pairs. For example, inner:5..10 specifies executions made in inner when the iteration count of inner is between 5 and 10.

If several ranged flow facts have the same defining scope and overlapping ranges, the iteration space of the scope needs to be split into disjoint subranges. For example, the facts inner : <1..7> : #B $=$ 1 (f1) and inner : [5..10] : #A $\leq$ 3 (f2) produce the following pairs: inner:1..4, inner:5..7 and inner:8..10. Both facts cover inner:5..7, while only f1 covers inner:1..2 and only f2 covers inner:6..7.

If the created pairs not cover the complete iteration space of the scope, extra pairs are created to fill the gaps. Therefore, each iteration from the first iteration to the loop bound of the scope gets covered by one scope:range pair. For example, assuming scope inner has an upper bound of 20 and flow facts f1 and f2, we get the scope:range pairs: inner:1..4, inner:5..7, inner:8..10 and inner:11..20.

It would be possible to make a partitioning that differentiate between each iteration of a scope, e.g., inner:1..1, inner:2..2, ..., inner:20..20. How-

```
CreateVirtualScopeSpecs(factset FS, scope s):
// Create scope:range pairs for scope s
SR := ∅
b := 1 // Get first iteration of scope s
FS_b := facts in FS with range covering iteration b in s
// Loop over all iterations of s
for each iteration i between b + 1 and loopbound(s) do
  FS_i := facts in fs with range covering iteration i in s
  // Check if set of covering facts has changed
  if FS_b ≠ FS_i or i = loopbound(s) then
    // Create new scope:range pair
    e := i - 1
    sr := s : b..e
    SR := SR ∪ {sr}
    // Restart search
    b := i
    FS_b := FS_i
end for
// Create virtual scope specifications for scope s
VSS_sub := ∅, VSS := ∅
// Get virtual scope specifications from descendant scopes
for each child c to scope s do
  VSS_sub := VSS_sub ∪ CreateVirtualScopeSpec(fs, c)
end for
// Create virtual scope specifications with s as root scope
for each scope range pair sr in SR do
  for each virtual scope specification vss_sub in VSS_sub do
    // Append specifications from descendant scopes to sr
    vss := append(sr,vss_sub)
    VSS := VSS ∪ {vss}
  end for
  // Create specification with sr as only element
  VSS := VSS ∪ {sr}
end for
// Return set of virtual scope specifications
return VSS
```

Figure 8.4: Virtual scope specification extraction

ever, this would result in a very large virtual scope graph, with each iteration explicitly represented by a copy of the original scope. Instead, we note that we only need to differentiate between executions in different iterations of the same scope if there are flow facts which constrain execution within the iterations differently.

8.2.2 Virtual scope specifications

Another key observation is that a flow fact can constrain executions in scopes differently from its defining scope by referring to entities located in descendant scopes. This means that the iteration space partitioning of a scope can not be made in isolation, but needs to be made in relation to the partitioning of iteration spaces of ancestral scopes.

The algorithm given in Figure 8.4 does iteration space partitioning over several scopes. The algorithm works in two steps; first partitioning the iterations space of each individual scope into a set of scope:range pairs, and then extending the partitioning in relation to the partitioning of ancestral scopes.

The algorithm creates a number of specifications of execution environments, each consists of a list of scope:range pairs. We call such execution environment specification a *virtual scope specification*. The scope in the last (right-most) pair is called the *anchor scope*, and is the scope that the virtual scope specification will be used to constrain executions. The scope:range pairs in the list goes via a sequential branch in the scope-hierarchy, i.e., the last range corresponds to the anchor scope for the fact, the second last range corresponds to the parent scope of the anchor scope, etc.

An example of the generation of virtual scope specifications can be found in Figure 8.5(a). First we partition the iteration spaces of scopes according to attached flow facts. Flow fact `f3` partitions the iteration space of `q` into scope range pairs `q:1..4` and `q:5..10`. Flow facts `f1` and `f2` are both defined in `p` and have different ranges. The iteration space partition of scope `p` result in the scope:range pairs `p:1..15` and `p:16..20`.

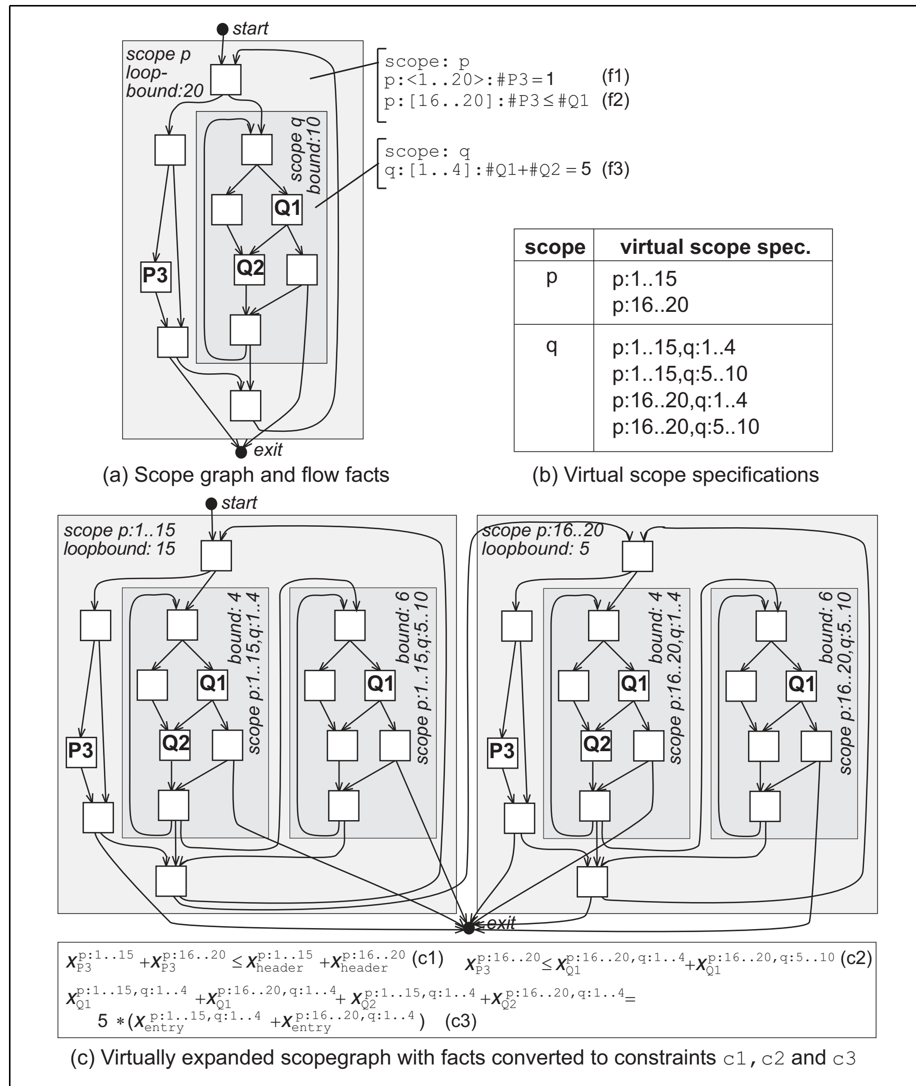
Flow fact `f2` reaches down into the `q` scope by constraining executions of node `Q1`, and the iteration space partitioning of `q` should therefore be made in relation to the partitioning of `p`. As a result, four virtual scope specifications are created for `q`, as depicted in Figure 8.5(b). A specification such as `p:16..20,q:5..10` has `q` as anchor scope, and specifies the execution environment of the last six iterations of the `q` scope when the iteration count of scope `p` is between 16 and 20. Since `p` has no parent in the scope-hierarchy, only two virtual scope specifications are created for `p`, as shown in Figure 8.5(b).

8.2.3 Virtual scope graph creation

The virtual scope specifications are used to expand the input scope graph into a *virtual scope graph*, consisting of one or more *virtual scopes*. Each virtual scope is a copy the anchor scope of its specification, including nodes and edges.

Figure 8.5(c) illustrates the virtual scope graph created from the scope graph in Figure 8.5(a) and virtual scope specifications in Figure 8.5(b). Two virtual scopes are created from `p` and four virtual scopes are created from `q`. The algorithm for the graph expansion is given in Figure 8.6. It takes a scope graph and a set of virtual scope specifications as input and produces a virtual scope graph.

The algorithm starts by creating a virtual scope for each virtual scope specification. A virtual scope will contain copies of the nodes and internal edges of the anchor scope of the specification. For example, the virtual scope specifications `p:1..15` and `p:16..20` generate two copies of the `p` scope. Each virtual scope gets the name of its virtual scope specification. We denote the node created from a virtual scope specification *vss* and original node n with n^{vss}.



$$x_{P3}^{p:1..15} + x_{P3}^{p:16..20} \leq x_{header}^{p:1..15} + x_{header}^{p:16..20} \quad (c1) \qquad x_{P3}^{p:16..20} \leq x_{Q1}^{p:16..20,q:1..4} + x_{Q1}^{p:16..20,q:5..10} \quad (c2)$$

$$x_{Q1}^{p:1..15,q:1..4} + x_{Q1}^{p:16..20,q:1..4} + x_{Q2}^{p:1..15,q:1..4} + x_{Q2}^{p:16..20,q:1..4} = 5 * (x_{entry}^{p:1..15,q:1..4} + x_{entry}^{p:16..20,q:1..4}) \quad (c3)$$

(c) Virtually expanded scopegraph with facts converted to constraints `c1`, `c2` and `c3`

Figure 8.5: Virtual scope graph generation

```
CreateVirtualScopeGraph(scopegraph sg, vsspecset VSS):
// Create virtual scopes
vsg := create empty scope graph
for each vsspec vss in VSS do
  vscope := create scope with name vss
  // Copy nodes and internal edges
  s := anchor scope of vss
  for each node n in s do
    add node n^vss to vscope
  end for
  for each edge u → v in s with u, v ∈ nodes(s) do
    u^vss := node in vscope generated from u
    v^vss := node in vscope generated from v
    add edge u^vss → v^vss to vscope
  end for
  // Set loop bound and add virtual scope to graph
  r := range of scope s in vss
  loopbound(vscope) := max(r) - min(r) + 1
  add scope vscope to vsg
end for
// Create hierarchy between virtual scopes
for each vsspec vss_child in VSS do
  vscope_child := scope with name vss_child in vsg
  vss_parent := remove last scope:range pair from vss_child
  vscope_parent := scope with name vss_parent in vsg
  set vscope_parent to be parent of vscope_child in vsg
end for
// Create edges between nodes in different virtual scopes
for each edge u → v in sg such that scope(u) ≠ scope(v) do
  // Get ancestral scopes of source and target nodes
  A_u := ancestors(scope(u)) ∪ {scope(u)},  A_v := ancestors(scope(v)) ∪ {scope(t)}
  // Get common ancestral scopes and ancestral scopes unique for node v
  A_both := A_u ∩ A_v ,    A_v_only := A_v − A_both
  for each vsspec vss_u in VSS with scope(u) as anchor scope do
    for each vsspec vss_v in VSS with scope(v) as anchor scope do
      if (∀s in A_both exists identical s : r pair in both vss_u and vss_v) and
        (∀s in A_v_only exists s : r pair in vss_t such that min(r) = 1) then
        vscope_u := scope with name vss_u in vsg
        vscope_v := scope with name vss_v in vsg
        u^vss_u := node in vscope_u generated from u
        v^vss_v := node in vscope_t generated from v
        add edge u^vss_u → v^vss_v to vsg
    end for
  end for
end for
// Copy and redirect back-edges to successor scopes
for each back-edge u → v in sg do
  for each vsspec vss_v in VSS with scope(v) as anchor scope do
    r_v := range of scope(v) in vss_v
    // Skip virtual scope covering last iteration of scope(v)
    if max(r_v) ≠ loopbound(scope(v)) then
      // Get virtual scope covering following iteration
      for each vsspec vss_n in VSS with the same length as vss_v do
        if (∀s in ancestors(scope(v)) exists identical s : r pair in both vss_n and vss_v)
          and (∃ range r_n in vss_n for scope(v) such that min(r_n) = max(r_v)+1) then
          vscope_n := scope with name vss_n in vsg
          v^vss_n := node in vscope_n created from v
          // Get source nodes in descendant virtual scopes
          for each vsspec vss_u in VSS with prefix vss_v and scope(u) as anchor scope do
            vscope_u := scope with name vss_u in vsg
            u^vss_u := node in vscope_u generated from u
            add edge u^vss_u → v^vss_n to vsg
          end for
      end for
  end for
end for
// Return created virtual scope graph
return vsg
```

Figure 8.6: Virtual scope graph creation

The loop bound of a virtual scope is set to be the number of iterations covered by the range of its anchor scope. For example, virtual scope `p:1..15` gets a loop bound of 15 and the virtual scope `p:16..20,q:5..10` gets a loop bound of 6.

The next step is to create a hierarchy of virtual scopes. The parent of a scope c with a specification *vss* is the scope p corresponding to *vss* with the anchor scope:range pair removed. For example, virtual scope `p:16..20` is the parent of virtual scopes `p:16..20,q:1..4` and `p:16..20,q:5..10`. The resulting scope-hierarchy forms a tree.

Next, edges in and out of virtual scopes are created. The edges reflect the potential program flows through the virtual scope graph. We only create an edge between virtual scopes if there exists a corresponding edge in the original scope graph. However, the scope-crossing edges should reflect the potential flows going between their target and source nodes specific execution environments. Therefore, an edge $u^{vss_i} \rightarrow v^{vss_j}$ between nodes u^{vss_i} and v^{vss_j} is only created if there exists an original edge $u \rightarrow v$ with `scope(`u`)` $\neq$ `scope(`v`)`, all scope:range pair for all common ancestors of `scope(`u`)` and `scope(`v`)` are the same in vss_i and vss_j, and all the remaining scope:range pairs in vss_j cover the first iteration.

For example, scope `q` in Figure 8.5(a) has one in-edge and two out-edges. In the resulting virtual scope graph, in Figure 8.5(c) scope `q` exists in four different versions. In the original graph the execution can enter `q` from `p` using one in-edge. In the new graph the original in-edge has been copied into two new in-edges entering virtual scopes `p:1..15,q:1..4` and `p:16..20,q:1..4` respectively. The remaining virtual scopes originating from `q` will have no in-edges from virtual scopes originating from `p`, reflecting that the execution must take the first iterations of a scope before the following ones.

Since the execution should be able to leave the original `p` scope during any iteration of `p`, its original out-edges are copied to all the generated virtual scopes. For example, in Figure 8.5 the original out-edge of `q` going to *exit*, will exist in four different versions in the virtual scope graph.

Finally, back-edges are copied and redirected to reflect that the execution should continue to the next virtual scope after all iterations have been executed in the current. For example, the back edge in virtual scope `p:1..15,q:1..4` points to the to the header-node of virtual scope `p:1..15,q:5..10`. For virtual scopes with a range that cover the last iteration of the anchor scope, (i.e., the iteration corresponding to the loop-bound), no back-edges need to be redirected. Note that back-edges can have source nodes in descendant scopes.

8.3 Constraint generation

The next step is to convert the virtual scope graph and flow facts to constraints. The virtual scope graph is converted to constraints reflecting possible program flows and graph structure, and the flow facts are converted to constraints over executions of entities in the virtual scope graph.

```
VSGraphToConstraints(vsspecset VSS, scopegraph vsg):
CS := ∅ // Empty constraint set
// Start and exit constraints
add x_start = x_exit = 1 to CS
// Structural constraints
for each node n located in virtual scope tree vsg do
  E_source := edges having n as source node
  add x_n = Σ_{e∈E_source} x_e to CS
  E_target := edges having n as target node n
  add x_n = Σ_{e∈E_target} x_e to CS
end for
// Relate header variables to headers
for each virtual scope vs in vsg do
  add x^vs_header = Σ_{h∈headers(vs)} x_h to CS
end for
// Relate entry variables to in-edges
for each virtual scope vs in vsg do
  add x^vs_entry = Σ_{e∈in_edges(vs)} x_e to CS
end for
// Relate entries and iterations of virtual scopes
for each virtual scope vs in vsg do
  add x^vs_header ≥ x^vs_entry to CS
  add x^vs_header ≤ loopbound(vs) * x^vs_entry to CS
end for
// Relate entries of different virtual scopes
for each vsspecs vss_i and vss_j in VSS do
  if immediate_predecessor(vss_i,vss_j) do
    add x^vss_i_entry ≥ x^vss_j_entry to CS
    add (loopbound(vss_i) − 1) * x^vss_j_entry ≤ x^vss_i_header − x^vss_i_entry to CS
end for
// Return updated constraint set
return CS
```

Figure 8.7: Virtual scope graph constraint generation

8.3.1 Constraints from the virtual scope graph

We with the algorithm for generating constraints from the structure of the virtual scope graph. The constraints are similar to the ones generated by basic IPET (see Section 8.1), but extra constraints are needed for relating entries and iterations of virtual scopes. The algorithm is given in Figure 8.7.

For each *entity* (node or edge) in the virtual scope graph a corresponding *virtual scope variable*, x^{vss}_{entity}, is created. The variable holds the number of times the referred entity is executed within the execution environment given by the virtual scope. Each scope in the virtual scope graph is also given a header count variable (x^{vss}_{header}), and an entry count variable (x^{vss}_{entry}). For example, for a scope `p:1..15` and node `P3` we generate a virtual scope variable $x^{\texttt{p:1..15}}_{\texttt{P3}}$, an entry variable $x^{\texttt{p:1..15}}_{entry}$ and a header variable $x^{\texttt{p:1..15}}_{header}$. The resulting constraints will all be formulated over such virtual scope variables.

The start and exit constraints give that the execution must be able to enter

```
ForeachToTotalFacts(factset FS):
// Convert foreach facts to total facts
for each fact f = scope : context : constr in FS do
  if is_foreach(f) then
    // Multiply all constants with header of scope
    for each constant c in constr not part in multiplication or division do
      constr := in constr replace c with c * #header(scope)
      context := '[' range(context) ']'
    end for
end for
// Return updated set of facts
return FS
```

Figure 8.8: Conversion of foreach to total flow facts

and exit the virtual scope graph precisely once. Next, structural constraints are generated from the structure of the virtual scope graph. These constraints are identical to the constraints generated by basic IPET.

We need to further relate and constrain the executions within and between virtual scopes. For this we use the virtual scope header variables, x^{vss}_{header}, and virtual scope entry variables, x^{vss}_{entry}. We first make sure that these variables have a correspondence in the virtual scope graph. For each virtual scope, its header variable is set equal to the sum of header node variables of the virtual scope and its entry count variable is set equal to the sum of its in-edge variables. For example, $x^{\texttt{p:1..15,q:1..4}}_{entry}$ is set equal to the count variable of the in-edge originating from `p:1..15`.

Additional constraints are generated to relate the number of iterations in a virtual scope to the number of entries of a virtual scope. Constraint $x^{vscope}_{header} \geq x^{vscope}_{entry}$ gives that the virtual scope must be iterated once for each entry of the virtual scope. Constraint $x^{vscope}_{header} \leq \texttt{loopbound}(vscope) * x^{vscope}_{entry}$ are generated to specify the loop bound of a virtual scope that should be valid for each entry of the virtual scope. The last constraint also serves as a basic finiteness bound for each loop found in virtual scope graph. For example, for virtual scope `p:1..15,q:1..4` we generate the constraints $x^{\texttt{p:1..15,q:1..4}}_{header} \geq x^{\texttt{p:1..15,q:1..4}}_{entry}$ and $x^{\texttt{p:1..15,q:1..4}}_{header} \leq 4 * x^{\texttt{p:1..15,q:1..4}}_{entry}$.

The executions and entries between different virtual scopes are also related. A virtual scope specification *prev* is defined to be an *immediate predecessor* of another specification *next* if the two specifications have the same anchor scope and identical lists of scope:range pairs except for the anchor scope. Also, the last iteration in the range of the anchor scope for *prev* should be one smaller than the first iteration of the range for the anchor scope of *next*. For example, `p:1..15` is an immediate predecessor of `p:16..20`. Virtual scope specification `p:1..15,q:1..4` is an immediate predecessor of `p:1..15,q:5..10`, but not of `p:16..20,q:5..10`.

The constraint $x^{prev}_{entry} \geq x^{next}_{entry}$ ensures that no virtual scope can be entered more often than its immediately preceding virtual scope. To guarantee that a

virtual scope only can be entered if the preceding virtual scope has executed all its iterations, the constraint $(\texttt{loopbound}(prev) - 1) * x^{next}_{entry} \leq x^{prev}_{header} - x^{prev}_{entry}$ is generated. For example, virtual scope `p:16..20` can not be entered unless its immediate predecessor `p:1..15` has been entered and all 15 its iterations has been executed. This generates the constraint $14 * x^{\texttt{p:16..20}}_{entry} \leq x^{\texttt{p:1..15}}_{header} - x^{\texttt{p:1..15}}_{entry}$.

8.3.2 Fact conversion

The next step is to convert the locally defined flow facts to globally valid constraints over virtual scope variables. This is made by the two algorithms outlined in Figure 8.8 and Figure 8.9.

All foreach facts are first converted to total facts by the algorithm given in Figure 8.8. Basically, a foreach fact gets converted from being for *each* separate iteration of a given range to be for *all* iterations within the given range. This is done by multiplying each constant not part in any multiplication or division by the header count variable of its defining scope, and by substituting the foreach context specifier (`<>`) with a total context specifier (`[]`). This is necessary since IPET works across *all* iterations and not just small parts.

For example, flow fact `p : <1..20> : #P3` $= 1$ gets converted to flow fact `p : [1..20] : #P3` $= 1 *$ `#`*header*`(p)`. While the first fact specifies that node `P3` should be executed each iteration of the scope `p`, the resulting fact specifies that `P3` should be executed as many times as `p` is iterated. Basically, we make sure that the constant gets counted as many times as the scope is iterated within the range. It should be noted that this conversion might introduce some pessimism since more executions might get allowed by the converted total fact than for the original foreach fact.

The next algorithm, given in Figure 8.9, converts locally defined (total) flow facts, valid for *each* entry to a given scope and range, to global constraints, valid for *all* entries of the given scope and range. Simultaneously, the facts are converted to constraints over virtual scope variables.

First, all constants not part in any multiplication or division, i.e., terms without variables, gets multiplied by the entry count variable of the defining scope. This ensures that the constant gets counted as many times as the scope and range is entered.

Next, each execution count variable gets replaced by a sum of virtual scope variables. Only count variables originating from the same entity as the original count variable and are located in virtual scopes covered by the range of the fact should be included in the sum. For example in Figure 8.5(a) flow fact `p : [16..20] : #P3` $\leq$ `#Q1` gets converted to $x^{\texttt{p:16..20}}_{\texttt{P3}} \leq x^{\texttt{p:16..20,q:1..4}}_{\texttt{Q1}} + x^{\texttt{p:16..20,q:5..10}}_{\texttt{Q1}}$. Variable $x^{\texttt{p:16..20}}_{\texttt{P3}}$ holds the number of times node `P3` gets executed when the iteration counter of scope `p` is between 16 and 20. Since `Q1` is located in the `q` scope we get two resulting variables $x^{\texttt{p:16..20,q:1..4}}_{\texttt{Q1}} + x^{\texttt{p:16..20,q:5..10}}_{\texttt{Q1}}$, together holding the executions of `Q1` when the iteration counter of scope `p` is between 16 and 20.

```
FactsToConstraints(scopegraph vsg, factset FS, constraintset CS):
// Multiply all constants in fact with entry variable of scope
for each fact f = scope : context : constr in FS do
  for each constant c in constr not part in multiplication or division do
    constr := in constr replace c with c * #entry(s)
  end for
end for
CS := ∅ // Empty constraint set
// Convert fact to constraint on virtual count variables
for each fact f = scope : context : constr in FS do
  VSS_f := vsspecs in VSS including s : r pair such that s = scope and
    min(range(context)) ≤ min(r) and max(range(context)) ≥ max(r)
  // Replace each entity count variable
  for each entity count variable #e in constr do
    s_e := scope where entity #e is located
    VSS_e := all vsspecs in VSS_f with s_e as anchor scope
    if #e = #entry(s_e) then
      // Treat scope entry variables specially
      VSS_e := all vspecs in VSS_e ending with s_e:r_e such that min(r_e) =
        smallest iteration of s_e covered by any scope-range pair in VSS_e
    // Replace original variable with sum of virtual scope variables
    constr := in constr replace #e with Σ_{vss∈VSS_e} x_e^vss
  end for
  // Add generated constraint to constraint set
  add constr to CS
end for
// Return updated constraint set
return CS
```

Figure 8.9: Flow facts to constraints conversion

Substitution of header variables is performed in the same way as normal entity count variables. However, substitution of entry count variables needs to be given special treatment. This is because an entry of a subrange of a range indirectly implies that the whole range is entered. An entry count variable should therefore be replaced with a sum of virtual scope entry variables with a specification whose anchor scope is the same as the flow facts defining scope and includes the first iteration in the range of the flow fact. For example, if flow fact `p : [1..20] : #P4` $= 7$ gets added to Figure 8.5(a) it would result in the constraint $x_{\texttt{P4}}^{\texttt{p:1..15}} + x_{\texttt{P4}}^{\texttt{p:16..20}} = 7 * x_{entry}^{\texttt{p:1..15}}$. An entry of range `p:1..20` is implied by an entry of virtual scope `p:1..15` and $x_{entry}^{\texttt{p:16..20}}$ should therefore not be included in the sum.

It should be noted that the conversion to global constraints might potentially introduce some pessmism in the WCET estimate. Basically, the program execution paths allowed by the local flow facts is a subset of the execution paths allowed by the global flow facts. However, as long as no execution paths allowed are removed, the calculated WCET estimate are safe.

8.3.3 Mapping to global count variables

The generated constraints are sufficient to, together with timing information from the timing model, (see Section 8.4), generate a WCET estimate. However, we can create additional constraints that relate virtual scope variables to the count variables used in basic IPET. This is useful for giving simplified execution information on the total number of times each scope graph entity can be executed. However, the virtual scope variables provide a more detailed view than the global variables. Also, only virtual scope variables and no global variables should be included in the objective function (see Section 8.4.1).

The mapping to global IPET variables is made by creating count variables (x_{entity}) for the different entities (nodes and edges) found in the original scope graph. The value of each such entity count variable is set equal to the sum of the virtual scope variables created from the entity. For example, in Figure 8.5 we create the count variable $x_{\texttt{Q1}}$ for node `Q1` and generate the formula $x_{\texttt{Q1}} = x_{\texttt{Q1}}^{\texttt{p:1..15,q:1..4}} + x_{\texttt{Q1}}^{\texttt{p:1..15,q:5..10}} + x_{\texttt{Q1}}^{\texttt{p:16..20,q:1..4}} + x_{\texttt{Q1}}^{\texttt{p:16..20,q:5..10}}$.

We can also create constraints to relate entries of scopes to entries of virtual scopes. However, we can not just set a global entry count scope variable to be equal to the sum of entry count variables of virtual scopes generated from the scope. An entry of the scope is the same as an entry of the virtual scopes which covers the first iteration of the scope. This means that for a scope s only entry count variables with a specification with s as anchor scope and a range covering the first iteration of s can contribute to the sum. For example, for scope `p` in Figure 8.5 we get the constraints $x_{entry(\texttt{p})} = x_{entry}^{\texttt{p:1..15}}$ and for scope `q` we get the constraint $x_{entry(\texttt{q})} = x_{entry}^{\texttt{p:1..15,q:1..4}} + x_{entry}^{\texttt{p:16..20,q:1..4}}$.

8.3.4 Constraint generation example

For an example on how the algorithms works, consider the code given in Figure 8.10. It shows the flow-relevant parts of the kernel of a DSP algorithm. To simplify the presentation, the code in the `fir` function and the `inner` loop is not shown in detail. Also, the structure of the source code and object code are assumed to be the same (i.e., no flow-changing compiler optimizations are performed).

Analysis of the code has given flow information specifying that node `A` is executed during the last 18 iterations of the `outer` loop and node `B` is executed during first 17 iterations of the outer loop. Also, node `C` is taken the first 682 iteration of the `outer` loop. This is given as flow facts `f1`–`f3`.

The inner loop executes $(18, 19 \ldots 34, 35 \ldots 35, 34 \ldots 19, 18)$ times across the iterations of the outer loop. This gives a basic finiteness bound of 35, and the more precise loop bounds given in flow facts `f4`–`f6`. The ramp-up and ramp-down each contain a total of $\sum_{i=18}^{34} i = 442$ iterations, and are given as one fact each of total style (`f4` and `f6`).

The derived flow information results in the flow facts `f1`–`f6`. Note that we use both foreach and total flow facts to model the possible flows, illustrating

```
acc_length = 18, len = 35, // initialization
ptr = start + 17, end = start + 700;
for(i=0;i<700;i++) { // outer loop
   for(j=1;j<acc_length;j++) // inner loop
      { // calculation }
   if(ptr == end)
      acc_length--; // node A
   else {
      if(acc_length < len)
         acc_length++; // node B
      ptr++; // node C
      }
} // end of loop
```

Basic finiteness	loopbound(`outer`) = 701
	loopbound(`inner`) = 35
node A taken last 18	`outer:<683..700>` : `#A` = 1 `(f1)`
node B taken first 17	`outer:<1..17>` : `#B` = 1 `(f2)`
node C taken first 682	`outer:<1..682>` : `#C` = 1 `(f3)`
Bounds on inner	`outer:[1..17]` : $\#header$(`inner`) = 442 `(f4)`
	`outer:<18..683>` : $\#header$(`inner`) = 35 `(f5)`
	`outer:[684..700]` : $\#header$(`inner`) = 442 `(f6)`

Figure 8.10: `fir` kernel with flow facts

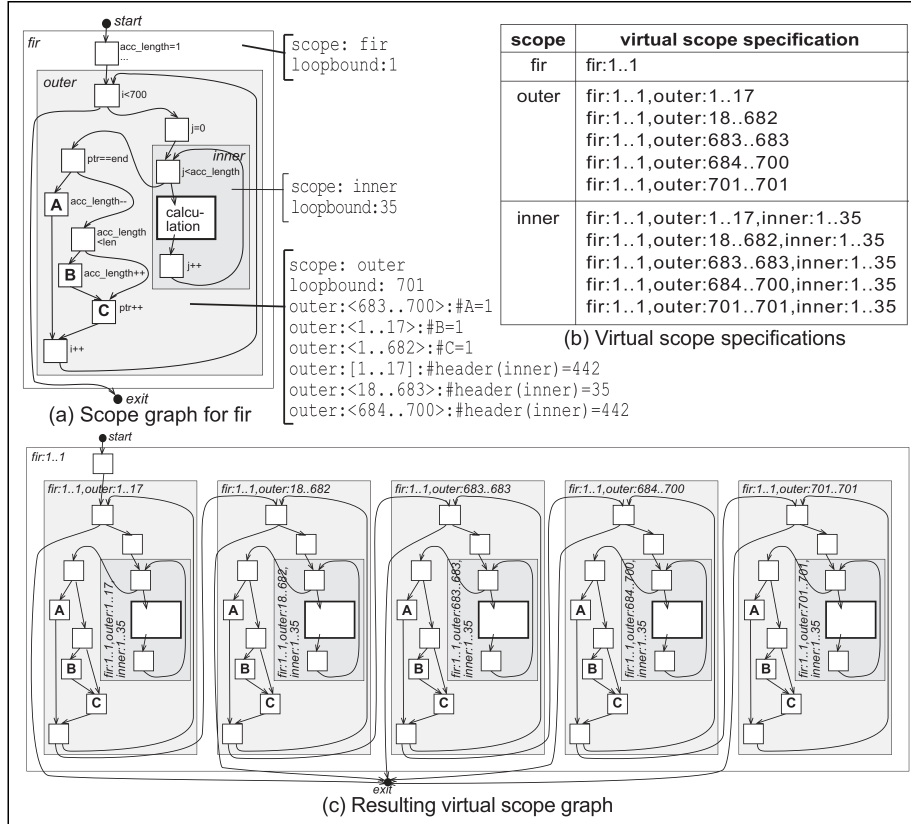
the convenience of having both types. Flow facts `f4`–`f6` have `outer` as defining scope but refers to the header node of `inner`, illustrating the need to allow flow information to reach over the borders dictated by loops or functions.

Figure 8.11(a) illustrates the scope graph generated from the `fir` code and the derived flow information. Loop bounds for all loops have also been added to the scope graph. Figure 8.11(b) illustrates the virtual scope specifications derived from the scope graph and the corresponding flow information. Figure 8.11(c) illustrates how the resulting virtual scope graph would look like. The virtual scope specifications have five distinguished execution environments for the `outer` scope, and accordingly five copies of scope `outer` are created, each with an own copy of `inner` as a child scope.

Some of the resulting constraints are illustrated in Figure 8.12. (A) shows the constraints over virtual scope variables generated from the original facts. (B) shows constraints generated between the entry variables and header variables of virtual scopes. Finally, (C) shows examples of how global IPET variables gets related to virtual scope variables. The constraints generated from the structure of the virtual scope graph and the start and exit constraints are not shown in the example.

8.4 Converting the timing model

The constraints generated so far are not sufficient to produce a WCET estimate. To construct an objective function to maximize, we also need information on timing for entities. This is provided by the timing model tm, which has been



scope	virtual scope specification
fir	fir:1..1
outer	fir:1..1,outer:1..17 fir:1..1,outer:18..682 fir:1..1,outer:683..683 fir:1..1,outer:684..700 fir:1..1,outer:701..701
inner	fir:1..1,outer:1..17,inner:1..35 fir:1..1,outer:18..682,inner:1..35 fir:1..1,outer:683..683,inner:1..35 fir:1..1,outer:684..700,inner:1..35 fir:1..1,outer:701..701,inner:1..35

Figure 8.11: Virtual scope graph generation for `fir` kernel

generated by our low-level analysis, as outlined in Chapter 6.

8.4.1 Objective function

The low-level analysis is performed once for the complete scope graph. The resulting timing model consists of timing for nodes t_{node} and timing effects δ_{seq} for sequences of nodes (see Chapter 6).

After expanding a scope graph into a virtual scope graph, each original node or node sequence exists in one or more versions in the virtual scope graph. From the original timing model tm, we therefore create a new timing model tm_{vsg}, containing timing and timing effects for the entities found in the virtual scope graph. This allows us to perform the pipeline timing analysis over the original scope graph, but to use the derived timing information for the entities found in the virtual scope graph, avoiding a reanalysis. The resulting objective therefore

A: Converted flow facts

$$x_{\mathtt{A}}^{\mathtt{fir:1..1,outer:683..683}} + x_{\mathtt{A}}^{\mathtt{fir:1..1,outer:684..700}} = 1 * (x_{header}^{\mathtt{fir:1..1,outer:683..683}} + x_{header}^{\mathtt{fir:1..1,outer:684..700}}) \quad \mathtt{(f1)}$$

$$x_{\mathtt{B}}^{\mathtt{fir:1..1,outer:1..17}} = 1 * x_{header}^{\mathtt{fir:1..1,outer:1..17}} \quad \mathtt{(f2)}$$

$$x_{\mathtt{C}}^{\mathtt{fir:1..1,outer:1..17}} + x_{\mathtt{C}}^{\mathtt{fir:1..1,outer:18..682}} = 1 * (x_{header}^{\mathtt{fir:1..1,outer:1..17}} + x_{header}^{\mathtt{fir:1..1,outer:18..682}}) \quad \mathtt{(f3)}$$

$$x_{header}^{\mathtt{fir:1..1,outer:1..17:inner:1..35}} = 442 * x_{entry}^{\mathtt{fir:1..1,outer:1..17}} \quad \mathtt{(f4)}$$

$$x_{header}^{\mathtt{fir:1..1,outer:18..682:inner:1..35}} + x_{header}^{\mathtt{fir:1..1,outer:683..683:inner:1..35}} = 35 * (x_{header}^{\mathtt{fir:1..1,outer:18..682}} + x_{header}^{\mathtt{fir:1..1,outer:683..683}}) \quad \mathtt{(f5)}$$

$$x_{header}^{\mathtt{fir:1..1,outer:684..700:inner:1..35}} = 442 * x_{entry}^{\mathtt{fir:1..1,outer:684..700}} \quad \mathtt{(f6)}$$

B: Relation between subheader and subentry variables for `outer`

$$x_{header}^{\mathtt{fir:1..1,outer:1..17}} \leq 17 * x_{entry}^{\mathtt{fir:1..1,outer:1..17}} \qquad x_{header}^{\mathtt{fir:1..1,outer:1..17}} \geq x_{entry}^{\mathtt{fir:1..1,outer:1..17}}$$

$$x_{header}^{\mathtt{fir:1..1,outer:18..682}} \leq 665 * x_{entry}^{\mathtt{fir:1..1,outer:18..682}} \qquad x_{header}^{\mathtt{fir:1..1,outer:18..682}} \geq x_{entry}^{\mathtt{fir:1..1,outer:18..682}}$$

$$x_{header}^{\mathtt{fir:1..1,outer:683..683}} \leq 1 * x_{entry}^{\mathtt{fir:1..1,outer:683..683}} \qquad x_{header}^{\mathtt{fir:1..1,outer:683..683}} \geq x_{entry}^{\mathtt{fir:1..1,outer:683..683}}$$

$$x_{header}^{\mathtt{fir:1..1,outer:684..700}} \leq 17 * x_{entry}^{\mathtt{fir:1..1,outer:684..700}} \qquad x_{header}^{\mathtt{fir:1..1,outer:684..700}} \geq x_{entry}^{\mathtt{fir:1..1,outer:684..700}}$$

$$x_{header}^{\mathtt{fir:1..1,outer:701..701}} \leq 1 * x_{entry}^{\mathtt{fir:1..1,outer:701..701}} \qquad x_{header}^{\mathtt{fir:1..1,outer:701..701}} \geq x_{entry}^{\mathtt{fir:1..1,outer:701..701}}$$

$$x_{entry}^{\mathtt{fir:1..1,outer:1..17}} \geq x_{entry}^{\mathtt{fir:1..1,outer:18..682}} \qquad x_{entry}^{\mathtt{fir:1..1,outer:18..682}} \geq x_{entry}^{\mathtt{fir:1..1,outer:683..683}}$$

$$x_{entry}^{\mathtt{fir:1..1,outer:683..683}} \geq x_{entry}^{\mathtt{fir:1..1,outer:684..700}} \qquad x_{entry}^{\mathtt{fir:1..1,outer:684..700}} \geq x_{entry}^{\mathtt{fir:1..1,outer:701..701}}$$

$$(17-1) * x_{entry}^{\mathtt{fir:1..1,outer:18..682}} \leq x_{header}^{\mathtt{fir:1..1,outer:1..17}} - x_{entry}^{\mathtt{fir:1..1,outer:1..17}}$$

$$(665-1) * x_{entry}^{\mathtt{fir:1..1,outer:683..683}} \leq x_{header}^{\mathtt{fir:1..1,outer:18..682}} - x_{entry}^{\mathtt{fir:1..1,outer:18..682}}$$

$$(1-1) * x_{entry}^{\mathtt{fir:1..1,outer:684..700}} \leq x_{header}^{\mathtt{fir:1..1,outer:683..683}} - x_{entry}^{\mathtt{fir:1..1,outer:683..683}}$$

$$(17-1) * x_{entry}^{\mathtt{fir:1..1,outer:701..701}} \leq x_{header}^{\mathtt{fir:1..1,outer:684..700}} - x_{entry}^{\mathtt{fir:1..1,outer:684..700}}$$

C: Examples of mapping to global variables

$$\begin{aligned}
x_{header(outer)} &= x_{header}^{\mathtt{fir:1..1,outer:1..17}} + x_{header}^{\mathtt{fir:1..1,outer:18..682}} + x_{header}^{\mathtt{fir:1..1,outer:683..683}} + \\
&\quad x_{header}^{\mathtt{fir:1..1,outer:684..700}} + x_{header}^{\mathtt{fir:1..1,outer:701..701}} \\
x_{header(inner)} &= x_{header}^{\mathtt{fir:1..1,outer:1..17,inner:1..35}} + x_{header}^{\mathtt{fir:1..1,outer:18..682,inner:1..35}} + \\
&\quad x_{header}^{\mathtt{fir:1..1,outer:683..683,inner:1..35}} + x_{header}^{\mathtt{fir:1..1,outer:684..700,inner:1..35}} + \\
&\quad x_{header}^{\mathtt{fir:1..1,outer:701..701,inner:1..35}} \\
x_{entry(outer)} &= x_{entry}^{\mathtt{fir:1..1,outer:1..17}} \\
x_{entry(inner)} &= x_{entry}^{\mathtt{fir:1..1,outer:1..17:inner:1..35}} + x_{entry}^{\mathtt{fir:1..1,outer:18..682:inner:1..35}} + \\
&\quad x_{entry}^{\mathtt{fir:1..1,outer:683..683:inner:1..35}} + x_{entry}^{\mathtt{fir:1..1,outer:684..700:inner:1..35}} + \\
&\quad x_{entry}^{\mathtt{fir:1..1,outer:684..701:inner:1..35}} \\
x_{\mathtt{C}} &= x_{\mathtt{C}}^{\mathtt{fir:1..1,outer:1..17}} + x_{\mathtt{C}}^{\mathtt{fir:1..1,outer:18..682}} + x_{\mathtt{C}}^{\mathtt{fir:1..1,outer:683..683}} + \\
&\quad x_{\mathtt{C}}^{\mathtt{fir:1..1,outer:684..700}} + x_{\mathtt{C}}^{\mathtt{fir:1..1,outer:701..701}}
\end{aligned}$$

Figure 8.12: Example of constraints generated for `fir`

function becomes:

$$WCET = max(\sum_{n \in \mathtt{nodes}(tm_{vsg})} x_n * t_n + \sum_{s \in \mathtt{sequences}(tm_{vsg})} x_s * \delta_s) \tag{8.1}$$

For each node with a timing t_{node} the product $x_{node} * t_{node}$ is added to the objective function, where x_{node} is the count variable of the node. Similarly, each node sequence seq, with a timing effect δ_{seq}, gets an associated execution count variable x_{seq}, holding the number of the times the sequence is executed. The resulting outcome will be an instantiation of virtual scope count variables

giving the number of times each entity is taken in the worst case, as well as a total worst case execution time for the program.

8.4.2 Constraints for node sequences

To be sure that get a safe and tight WCET estimate we need to produce constraints to constrain the impact of long timing effects on the total execution time. Basically, a timing effect for a sequence should only be accounted for when the execution goes uninterrupted via all the nodes in the sequence.

The count variable for a sequences of length two is defined to be the count variable for the edge going between the two nodes, and number of executions of an edge is constrained by structural constraints from the scope graph. For example, $x_{\mathtt{AB}}$ is the count variable for the $\mathtt{A} \rightarrow \mathtt{B}$ edge but also the count variable for the number of times $\delta_{\mathtt{AB}}$ should be accounted for in the final WCET.

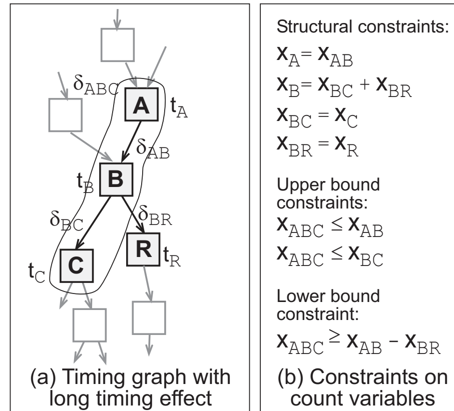


Figure 8.13: Example of constraints on node sequences

For sequences longer than two we do not have such direct correspondence in the scope graph, and we need to derive safe upper and lower bounds on their executions. The bounds are expressed relative to the executions of shorter sequences, where the structural constraints represent the base case. The upper bound is needed when the timing effect is positive (to bound the program execution time), and the lower bound makes sure that the execution of a negative timing effect is not ignored by the objective function.

The upper bound constraint is derived from the observation that a sequence *seq* of length l is executed only if its two subsequences of length $l-1$ are executed. This is expressed using two inequalities, using $s[i..j]$ to denote the subsequence of s starting at node i and ending at node j:

$$x_s \leq x_{s[1..l-1]} \tag{8.2}$$

$$x_s \leq x_{s[2..l]} \tag{8.3}$$

For example, for the timing effect $\delta_{\texttt{ABC}}$ over a sequence `ABC` in Figure 8.13 we get the following upper bound constraints: $x_{\texttt{ABC}} \leq x_{\texttt{AB}}$ and $x_{\texttt{ABC}} \leq x_{\texttt{BC}}$.

Note that this means that count variables for all subsequences of a sequence have to be generated, even if those subsequences have no timing effects associated with them (for a sequence `ABCD`, we will need constraints over the count variables for sequences `ABC` and `BCD`, etc.).

The lower bound constraint is derived by considering the stepwise construction of a sequence from prefix sequences. Intuitively, a lower bound can be derived by the possible flow through a sequence: a certain number of executions enter the start of the sequence, and at each node in the sequence, one edge leads the way to the next node in the sequence. At each such node there might be other edges leading to nodes not part of the sequence, thus diminishing the total number of executions of the sequence.

The result is that a sequence can be executed as many times as its prefix of length $l-1$ is executed, minus the sum of executions of edges leaving the sequence at the second-last node. Using $s[i]$ to pick a single node from a sequence s, and $\texttt{succ}(n)$ to return all successor nodes to node n in the scope graph we get:

$$x_s \geq x_{s[1..l-1]} - \sum_{\forall n \in \texttt{succ}(s[l-1]) \neq s[l]} x_{s[l-1] \circ n} \tag{8.4}$$

For example, for a timing effect $\delta_{\texttt{ABC}}$ in Figure 8.13 we generate the following constraint: $x_{\texttt{ABC}} \geq x_{\texttt{AB}} - x_{\texttt{BR}}$.

8.4.3 Tightening constraints

IPET has no concept of "execution paths", only the values of the execution count variables matter for the maximisation of Equation 8.1, but not the precise execution order of the entities. Therefore, it is sufficient to model infeasible paths as constraints on single nodes, as long as there are no timing effects for a sequence along the infeasible path.

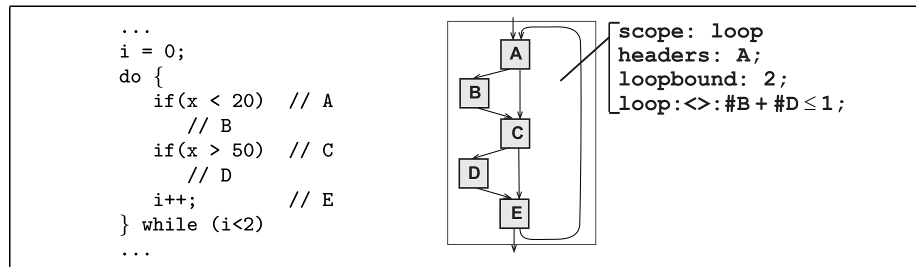


Figure 8.14: Example of infeasible path

For example, consider Figure 8.14, where the sequence `BCD` can never be taken during any iteration. The infeasible path is expressed using the flow

fact $\texttt{loop} : \texttt{<>} : \texttt{\#B} + \texttt{\#D} \leq 1$. The flow fact is converted to the constraint $x_{\texttt{B}}^{\texttt{loop:1..2}} + x_{\texttt{D}}^{\texttt{loop:1..2}} \leq 1 * x_{\texttt{A}}^{\texttt{loop:1..2}}$ using the fact that A is the header node of the scope and the loop has an upper bound of 2. This constraint is sufficient for a tight WCET estimate, despite the fact that we do not directly state that the specific sequence ABCDE is infeasible. E.g., assuming that the loop is iterated twice, it is possible that the first iteration took sequence ABCDE and the second iteration took sequence ACE. As the timing of a node or edge is independent of when the node or edge is taken, the obtained answer is correct.

However, the infeasibility of the specific sequence BCD would matter if there was a timing effect for the sequence (i.e., $\delta_{\texttt{BCD}} \neq 0$). To make sure that the timing effect does not contribute to the total WCET sum an additional constraint like $x_{\texttt{BCD}}^{\texttt{loop:1..2}} = 0$ has to be added. The constraints generated for $x_{\texttt{BCD}}^{\texttt{loop:1.2}}$ by Equations 8.2, 8.3, and 8.4 are not strong enough to express this fact. This extra constraining of timing effect variables are not part of any of the algorithms presented and has not been implemented.

8.5 Main algorithm and complete example

```
CalculateWCET(scopegraph sg, factset FS, timingmodel tm, constraintset CS):
  // Create virtual scope specifications
  VSS := CreateVirtualScopesSpecs(FS, root(sg))
  // Create virtual scope graph from specifications
  vsg := CreateVirtualScopeGraph(sg, VSS)
  // Convert flow facts to constraints
  FS := ForeachToTotalFacts(FS)
  CS := FactsToConstraints(vsg, FS)
  // Convert virtual scope tree to constraints
  CS := CS ∪ VSGraphToConstraints(VSS, vsg)
  // Create timing model for virtual scope graph
  tm_vsg := CreateVSTimingModel(tm)
  // Generate objective function
  sum := CreateObjectiveFunction(tm, sum)
  // Greate constraints for long timing effects
  CS := CS ∪ TimingEffectsToConstraints(tm_vsg, vsg)
  // Generate WCET estimate
  WCET := maximize(sum) given constraints in CS
  // Return obtained WCET estimate
  return WCET
```

Figure 8.15: Extended IPET WCET calculation

We are now ready to give the complete algorithm for our extended IPET calculation method. The algorithm is illustrated in Figure 8.15 and uses all the algorithms previously presented in this chapter. To solve the obtained maximization problem an integer integer linear programming (ILP) solver or constraint programming (CP) can be used.

In Figure 8.16(a)-(n) we give a compact illustration of the steps involved in our extended IPET calculation method.

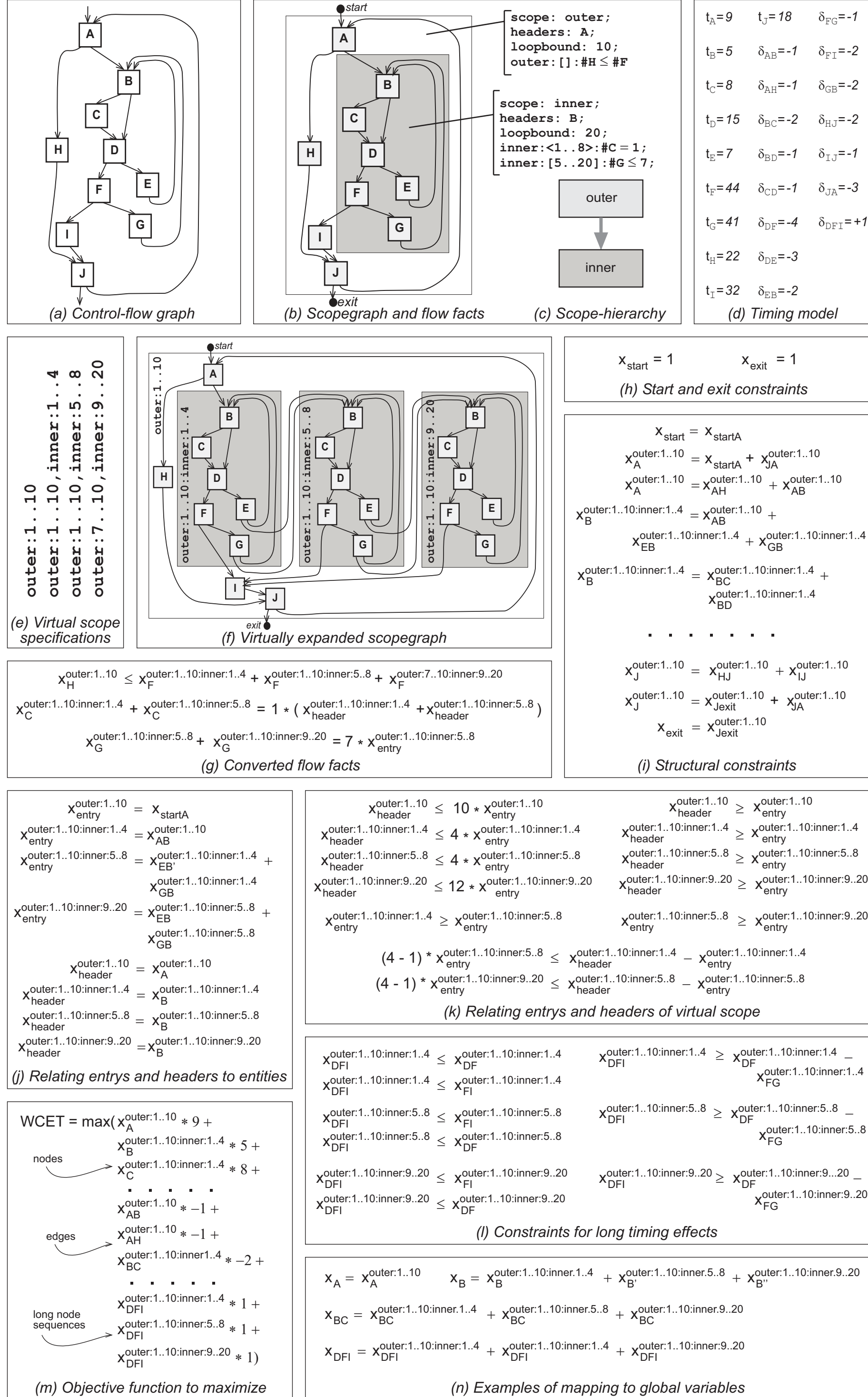


Figure 8.16: Complete IPET-Based WCET Calculation Example

Figure 8.16(a) shows an example control-flow graph consisting of two nested loops. Figure 8.16(b) shows the corresponding scope graph with scopes `inner` and `outer`. Each scope has a loop bound and some flow facts attached. Figure 8.16(c) shows the resulting scope-hierarchy. Figure 8.16(d) illustrates the timing model generated from the scope graph by our low-level analysis. Observe the long timing effect $\delta_{\mathtt{DFI}}$ over the `DFI` node sequence.

Figure 8.16(f) gives the virtual scope graph generated from the virtual scope specifications in Figure 8.16(e). Scope `inner` gets cloned into three virtual scopes. Figure 8.16(g) shows the flow facts after being converted to be constraints over virtual count variables. Figure 8.16(h) gives the start and exist constraints for the virtual scope graph. Figure 8.16(i) gives some example constraints generated from the virtual scope graph structure.

Figure 8.16(j) shows how entry and header variables get related to some entity count variables found in the virtual scope graph. Figure 8.16(k) shows constraints generated to relate the executions and entries of different virtual scopes. Figure 8.16(l) shows upper and lower bound constraints generated on the count variables for copies of the `DFI` node sequence. Figure 8.16(m) shows the objective function to maximize, generated for all nodes and sequences of nodes in the timing model. Finally, Figure 8.16(n) shows some examples of mappings between count variables of the original scope-graph and count variables found in the new expanded graph (not really needed to produce a WCET estimate).

Chapter 9

Clustered Calculation

A critical part of WCET analysis is the calculation, which combines the flow information and low-level timing information in order to derive a program WCET estimate. The type of flow information which a calculation method can handle is to a great extent determining the WCET estimate precision that can be achieved. Traditionally, we have had a choice between precise methods that perform global calculations with a risk of high computational complexity, and local methods that are fast but cannot take into account all types of flow information.

In this chapter we present an innovative hybrid method to handle complex flows with low computational complexity, and still generates safe and tight WCET estimates. The method uses flow information to find the smallest possible parts of a program that have to be handled as a unit to ensure precision. These units are used to calculate a program WCET estimate in a demand-driven bottom-up manner. The calculation method to use for a particular unit is not fixed, but could depend on the characteristics of the included flow information and program structure.

9.1 Introduction

A correct WCET calculation method must take into account possible program flow, like loop iterations and function calls, as well as effects of hardware features, like caches and pipelines. The possible program flow can be described by a set of flow facts, each providing a certain piece of information about the program (like loop bounds, infeasible paths, execution dependencies, etc.).

A key observation is that program flow information is usually local in its nature, expressing facts that only affect a small region of a program. However, these regions might be larger than the units used in local calculation schemes (a loop nest rather than a loop, or an entire function rather than just a loop inside that function). In general, the boundaries for a flow fact might not agree with the boundaries of a calculation scheme based on the structure of a program.

When flow facts cross structural boundaries, and thus calculation boundaries, they cannot be accounted for, which leads to a lower precision.

The obvious solution to the boundary problem is to work globally on the entire program at once. However, this has a potentially high complexity, and makes scaling to large programs risky. Almost all techniques for performing global calculations are based on integer linear programming (ILP) or constraint programming (CP) techniques, thus having a complexity potentially exponential in the program size. Also, as discussed in Section 8.3, some flow facts get less precise when being converted to a global program level.

However, by structuring local calculations after the boundaries dictated by the flow facts, it is possible to achieve both efficient local calculation and high precision, since all flow information can be accounted for, while avoiding the need for performing global calculation (unless there are actual flow facts that make this necessary).

Another key observation is that in many cases it is not sufficient to consider each provided flow fact in isolation. For a WCET tool using several types of automatic flow analyses and manual annotations, many different type of flow facts might be generated for the same program. Such flow facts can *interact* and together constrain program flow in a manner not possible by single flow facts. A WCET calculation method working over smaller program part must therefore, to achieve maximum precision, find interacting flow facts and treat them as a unit.

9.2 Method overview

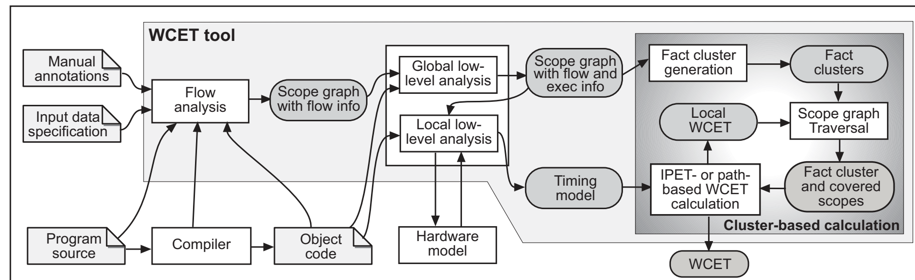


Figure 9.1: WCET Tool architecture when using fact clustering

The *clustered WCET calculation* method presented in this chapter builds upon the reasoning above. Figure 9.1 gives an overview of our WCET tool when using a clustered calculation module (as discussed in this chapter). All components of the system except the calculation phase remain unchanged, demonstrating the modular structure of the tool. As for our path-based and IPET-based calculation methods, the cluster-based calculation method builds upon a clear

separation of the calculation from the rest of the WCET analysis stages, thereby enhancing reusability and retargetability.

The possible program flow is represented by a scope graph with flow facts. The calculation method uses the given flow information to construct sets of flow facts, where all the included flow facts have to be considered together to ensure precision (box "Fact cluster generation" in Figure 9.1). Such *fact clusters* are caused by flow facts sharing application area with some other flow facts, by reaching down into subscopes and by having overlapping range specifications. The facts in a fact cluster indirectly specify the scopes in the scope graph that needs to be evaluated as a unit together with the flow facts.

The flow facts and covered scopes are used together to perform a local WCET calculation (box "IPET- or path-based WCET calculation" in Figure 9.1). The WCET estimates extracted for different fact clusters together form a WCET estimate for the whole program. The calculation is performed *bottom-up*, letting extracted WCET estimates for clusters be represented as a node when calculating a WCET estimate for clusters higher up in the scope-hierarchy, (box "Scope graph traversal"). The calculation is also *demand-driven*, and calculates a WCET estimate for a cluster only when it is needed in the WCET calculation of other clusters. Sections 9.3-9.4 will present the details of our cluster-based WCET calculation method.

For a given set of flow facts it is possible to construct various kind of fact clusters. In Section 9.3.2 we outline several potential fact clustering algorithms, each with the property of putting interacting flow facts in the same resulting cluster. The presented fact clustering algorithms will each generate different amount of fact clusters. Having many small fact clusters results in more local WCET calculations, but the calculation cost for each cluster decreases. Furthermore, the scope graph fragment specified by a fact cluster might have multiple entry and exit points. In Section 9.4.2 we present different calculation alternatives for such fact clusters and graph fragments, showing that WCET estimate precision can be traded for calculation speed.

Furthermore, not only flow information can reach over the borders dictated by local calculation schemes. As discussed in Chapter 6 some hardware dependencies might reach over larger program parts of a program. When performing WCET calculation over smaller program parts the calculation must guarantee to safely capture such long reaching hardware dependencies.

We identify such long reaching dependencies in our low-level analysis, i.e., in a stage clearly separated from the calculation. In our resulting timing model, which is given as input to the calculation, the dependencies are represented using timing effects over sequences of nodes (see Chapter 6). In Section 9.5 we outline how to safely capture the impact of long timing effects when they reach over calculation borders dictated by fact clusters.

Finally, in Section 9.6 we give a complete example of our clustered calculation method.

9.3 Clustering of flow facts

Our clustered calculation method takes two inputs: (1) a description of possible program flows, represented as a scope graph with flow facts (see Chapter 5), and (2) a timing model, including timing for nodes and timing effects for sequences of nodes (see Chapter 6).

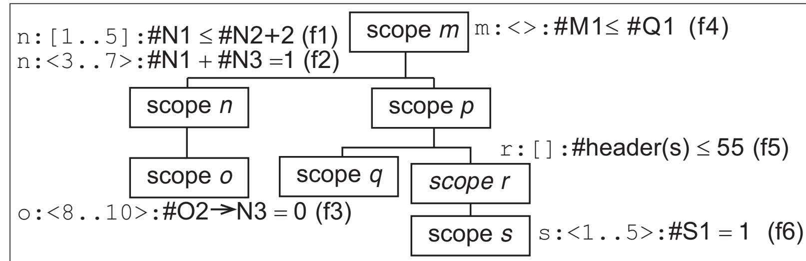
The scope graph structure and flow facts allowed depends on the final calculation methods used within the extracted clusters, but basically we allow the full expressiveness of the flow fact language as presented in Chapter 5. Both total and foreach facts, valid for all or certain ranges, are allowed. Scopes are allowed to contain one or more header nodes, and to have several in-nodes and out-edges, something which allow us to handle most types of unstructured code.

To understand how we construct fact clusters with dependent flow facts we need some extra definitions. For a flow fact we define all scopes between the defining scope and the scopes containing referred count variables to be *covered* by the fact. As mentioned in Chapter 5, a flow fact is only allowed to refer to count variables corresponding to entities in the complete subtree of the flow fact. Thus, the scopes covered by a fact form a subtree in the scope-hierarchy with the defining scope as root (see Section 5.2.3 on page 64). We define `cover(`f`)` to return the set of scopes covered by a facts f. Furthermore, for each scope covered by a fact the fact *spans* a number of iterations. For the defining scope the span is the number of iterations specified by the context specifier. For all other covered scopes the span is all iterations of the scope.

We define a *fact cluster* to be a set of flow facts. The *defining scope* of a fact cluster is defined to be the first common ancestor of all the facts in the cluster. The *cover* of a fact cluster is all scopes between the defining scope and the scopes containing count variables referred to by a flow fact in the scope. Thus, the covered scopes forms a subtree in the scope-hierarchy with the defining scope as root. We define `cover(`c`)` to return the set of scopes covered by a fact cluster c. The scopes in a covered subtree with no children are called *leaf-scopes*. For the defining scope s of a cluster the *span* is all iterations between the minimum and maximum iteration of s spanned by any of the facts in the cluster. For all other covered scopes the span is equal to all iterations of the scope.

In Figure 9.2(a) an example scope-hierarchy with associated flow facts is given. In Figure 9.2(b) we show the defining scopes, defining scope spans, and cover of each given flow fact. The name of a referred count variable gives the scope in which the corresponding entity is located, e.g., `#N1` refers to executions of node `N1` located in scope `n`. The function `lb(`i`)` returns the loop bound for a scope i.

The fact clusters generated from the facts are given in Figure 9.2(c). For each generated fact cluster we show its defining scope, its defining scope span, and the scopes covered by the cluster. Note that the same flow fact can be present in several clusters, and that not all flow facts in a cluster need to have the same defining scope.



(a) *Example scope-hierarchy with associated facts*

Fact	Defining scope	Span def scope	Covered scopes
f1	*n*	*1..5*	{*n*}
f2	*n*	*3..7*	{*n*}
f3	*o*	*8..10*	{*o*}
f4	*m*	*1..lb(m)*	{*m,p,q*}
f5	*r*	*1..lb(r)*	{*r,s*}
f6	*s*	*1..5*	{*s*}

(b) *Information about facts*

Fact cluster	Defining scope	Span def scope	Covered scopes
{**f1,f2**}	*n*	*1..7*	{*n*}
{**f3**}	*o*	*8..10*	{*o*}
{**f4**}	*m*	*1..lb(m)*	{*m,p,q*}
{**f5,f6**}	*r*	*1..lb(r)*	{*r,s*}
{**f6**}	*s*	*1..5*	{*s*}

(c) *Information about fact clusters*

Figure 9.2: Fact clustering example

9.3.1 Flows causing clusters

```
for(i=0; i<10; i++)      /* Bound:  10, (scope r) */
   for(j=i; j<10; j++)   /* Local bound:  10, (scope s) */
      { ...  }           /* executed at most 55 times */
```

Figure 9.3: Example 1: Triangular loop

Program flows causing fact clusters and reaching over several scopes are actually quite common. The simplest example is illustrated in Figure 9.3. It is the classical "triangular" loop, i.e., a nested loop where the number of iterations of the inner loop depends on the current iteration number of the outer loop (cf. scopes `r` and `s` in Figure 9.2(a)).

The inner loop considered in isolation will have an iteration bound of 10, and so will the outer loop. If WCET calculation is performed locally, the WCET calculation for the inner loop will assume 10 iterations, and the WCET calculation for the outer loop will use 10 executions of the inner loop, leading to the body of the inner loop being counted 100 times, when it is actually never executed more than 55 times. This requires that we handle the inner and outer loop together. Flow fact `f5` in Figure 9.2(a) shows how this type of triangular loop dependency can be captured.

Flows in nested scopes can be related in other ways, for example if the outcome of a decision in a scope determines the paths taken in a loop (maybe deeply) nested in the scope (with varying outcome), like e.g., for the scopes

```
void foo(bool x) {        /* Function foo(), (scope m) */
   if( cond )
      x = true;           /* Block M1 */
   for( ...  )            /* Outer loop, (scope p) */
      for (...  )         /* Inner loop, (scope q) */
         if( x )
            Q1            /* Block Q1, execution is implied by M1 */
}
```

Figure 9.4: Example 2: Long reaching dependency

m, p and q in Figure 9.2(a). Figure 9.4 gives example code with such long reaching dependency. Flow fact f4 in Figure 9.2(a) captures this type of nested dependency. It gives that an execution of M1 *implies* an execution of Q1, (node Q1 can still be executed on its own).

In the next example, (shown in Figure 9.5), block N3 does not belong to the loop, (due to the break statement), and the way the loop is exited will determine whether it should be counted or not. Thus, N3 depends on the decision cond in the loop body, but N3 is a node in the parent scope of o (scope n).

```
   for( ...  )  {            /* Bound:  10, (scope o) */
      if( cond ) {           /* False during last 3 iterations */
         N3                  /* Block N3, big chunk of work */
         break;
      }
   ...
   }
```

Figure 9.5: Example 3: Condition dependent dependency

Fact f3 in Figure 9.2 captures this dependency by specifying that the edge from O2 to N3 can't be taken during the last three iterations of the o scope.

Another case of flow information causing clusters is when information from different types of flow analysis methods or manual annotations interact, and therefore need to be considered together in the WCET calculation. An example of such overlapping flow information is shown in Figure 9.2(a) with flow facts f1 and f2. Both flow facts have the same defining scope n and they overlaps in the ranges of their context specifications.

9.3.2 Fact clustering algorithm

An algorithm to create the clusters of flow facts is given in Figure 9.6. The algorithm makes a bottom-up traversal of the scopes in the scope-hierarchy, i.e., all clusters for a subscope are generated before its parent scope is processed.

For each scope s, we look at the facts defined in the scope, and partition the facts based on their range specifications. Two facts with ranges that overlap, i.e., have some iteration numbers in common, should belong to the same set:

```
ClusterFacts(scopegraph sg):
  FC := ∅ // To hold set of fact clusters
  // Traverse scope-hierarchy bottom up
  for each scope s in sg in bottom-up order do
    F_s := facts in sg with s as defining scope
    FC_s := ∅
    // Partition facts in F_s into overlapping sets of facts
    for each fact f in F_s do
      FC_f := all fact clusters in FC_s with span of s overlapped by f
      if FC_f = ∅ then
        // No overlapping facts, create and add cluster for f
        FC_s := FC_s ∪ {f}
      else // Merge all overlapping facts into one fact cluster
        FC_s := FC_s - FC_f
        fc_f := merge(FC_f) ∪ f
        FC_s := FC_s ∪ fc_f
    end for
    // Add fact clusters defined in descendant scopes
    for each fact cluster fc_s in FC_s do
      Sub := scopes covered by fc_s in sg
      Sub := Sub - s
      for each fact cluster fc_sub in FC with defining scope in Sub do
        fc_s := fc_s ∪ fc_sub
      end for
    // Update set of fact clusters
    FC := FC ∪ {fc_s}
    end for
  end for
  // Return set of fact clusters
  return FC
```

Figure 9.6: Minimal fact clustering algorithm

$\forall f_i, f_j \in$ `facts`(s) : $((\exists i \in$ `range(context(`f_i`))` $\wedge\, i \in$ `range(context(`f_j`))` $\wedge$ $\wedge f_i \in F) \Rightarrow f_j \in F$. This creates sets of facts where the range expression of each fact overlaps one or more of the facts in the same set.

For example, fact `f1` and `f2` in Figure 9.2 have overlapping ranges and the same defining scope `n`, and should therefore be put in the same set. Note that if there are any “all iterations” facts (using context specification `[]` or `<>`), there will only be one fact set for this scope since these facts include all iterations, and thus overlap with all other facts defined on the scope.

We also need to consider interactions of flow facts located in different scopes. For each extracted fact-cluster we check if it covers any descendant scopes. For all covered descendant scopes all facts in clusters defined on these scopes are added to the cluster, together covering a set of scopes that have to be considered jointly. Note that this means that a fact can be part of several fact clusters. For example, fact `f5` in Figure 9.2 covers both scope `r` and `s` and should therefore be clustered together with fact `f6`, resulting in fact cluster {`f5`,`f6`} with `r` as its defining scope. Fact `f6` is at the same time the only fact in the cluster having `s` as defining scope.

The algorithm given in Figure 9.6 generates minimal fact clusters, i.e., sets of

facts where all included facts need to be considered together, but includes as few facts as possible. We call this clustering algorithm for *minimal fact clustering*. It is also possible to form larger clusters, (note that any clustering has to put all interacting facts in the same unit), and natural examples of such clusterings are:

- *Scope-based clustering*: All facts defined in a scope are put in the same cluster, together with all the facts in fact clusters defined in covered subscopes.
- *Maximum clustering*: All flow facts in the scope graph are put into one big cluster with the first common ancestor scope as its defining scope. Scopes not covered by the resulting fact cluster will be calculated separately from the scopes in the cluster.
- *Global clustering*: All flow facts in the scope graph are put into one big cluster with the root scope of the scope graph as its defining scope. All scopes in the scope-graph are part of the cluster. This is identical to the global calculation view used by our IPET-based calculation method (see Chapter 8).

Furthermore, we can construct even smaller clusters by subdividing foreach facts into facts valid for smaller ranges. A foreach fact gives flow information valid for *each individual* iteration and therefore do not need to force overlapping subranges to the same cluster. Instead, we apply the algorithm given in Figure 9.6 only to total facts. The remaining foreach fact are *split* into new foreach facts across the ranges of the resulting clusters. E.g., in Figure 9.2(a) the total fact `f1` does not overlap `f2` completely, so we split `f2` into the facts `n : <3..5> : #N1` + `#N3` = 1 (`f2`') and `n : <6..7> : #N1` + `#N3` = 1 (`f2`"). The resulting fact clusters becomes {`f1,f2`'} and {`f2`"}. We call such clustering *split-foreach-fact minimal clustering*. Compared to the *minimal clustering* algorithm the split of foreach facts will result more facts but in smaller clusters covering a smaller program parts.

In Chapter 10 we evaluate the WCET estimate precision and calculation speed due to the usage of different fact clusterings.

9.4 WCET calculation using fact clusters

The algorithm for calculating a WCET estimate using fact clusters is shown in Figure 9.7. The algorithm performs a demand-driven traversal of the scopes in the scope-hierarchy. For each scope we find the fact clusters defined on the scope, and for each fact cluster the scopes covered by the cluster are extracted as a subgraph over which a local WCET calculation is made (the *CutOutScopeGraph*`(...)` algorithm, given in Figure 9.9). This means that if there are fact clusters that cover more than one scope, a WCET calculation is performed over all covered scopes as a unit.

The WCET estimate for a scope s is obtained by iterating over the clusters having s as defining scope in range order, i.e., fact clusters spanning the first iterations of s are processed before fact clusters spanning later iterations of s.

```
ScopeWCET(scope s, scopegraph sg, factclusterset FC, timedatabase tdb):
// Initialize timing variables for scopes and clusters
t_{s,back} = t_{s,out} = t_{fc,back} = t_{fc,out} = 0
// Get fact clusters for scope s
FC_e := create empty clusters with a cover of s for each range
  of s not spanned by fact clusters in FC
FC_s := FC_e ∪ fact clusters in FC with s as defining scope
// Make WCET calculation over clusters
for each fact cluster fc in FC_s in increasing range order over s do
  // Create scope graph for cluster
  sg_cut := CutOutScopeGraph(sg,FC,fc,tdb)
  // Get begin nodes for cluster
  if fc spans first iteration of s then b := in_nodes(s)
  else b := headers(s)
  // Calculate time to out-edges for cluster
  t_{fc,out} := ClusterWCET(fc, b, out_edges(s), sg_cut, tdb)
  // Update time to out-edges for scope
  if valid(t_{fc,out}) then
    t_{s,out} := max(t_{s,back} + t_{fc,out}, t_{s,out})
  // Calculate time to back-edges for cluster
  if fc does not span last iteration of s then
    t_{fc,back} := ClusterWCET(fc, b, back_edges(s), sg_cut, tdb)
    // Update time to back-edges for scope
    if valid(t_{fc,back}) then
      t_{s,back} := t_{s,back} + t_{fc,back}
    // Break loop if execution can't continue
    else break
end for
// Update timing database and return
add time t_{s,out} for scope s to tdb
return tdb
```

Figure 9.7: Cluster-based WCET calculation algorithm

If some scope range is not spanned by any fact cluster, an *empty fact cluster* is created. Such empty cluster covers just the current scope and spans only consecutive iterations not spanned by any fact. For a scope not covered by any flow fact, an empty cluster is created, spanning all iterations of the scope.

A timing estimate for a program fragment should be calculated from where the execution can enter the fragment to where the execution can exit the fragment. A calculation for a cluster is therefore performed from some *begin-nodes*, where the execution can enter the covered scopes, to some *end-edges*, where the execution can exit the covered scopes. The cluster spanning the first iteration of a scope has begin-nodes equal to the in-nodes of the scope. For the remaining clusters the begin-nodes are equal to the header-nodes of the scope, since this defines the start of a new iteration.

Similarly, for all clusters except the one including the last iteration, we make two distinct calculations, one ending at an out-edge and one ending at a back-edge. This is because the execution path taken to exit a scope might be different from the path taken when continuing to the next cluster. If a WCET estimate for the back-edges cannot be calculated, e.g., due to some contradicting flow

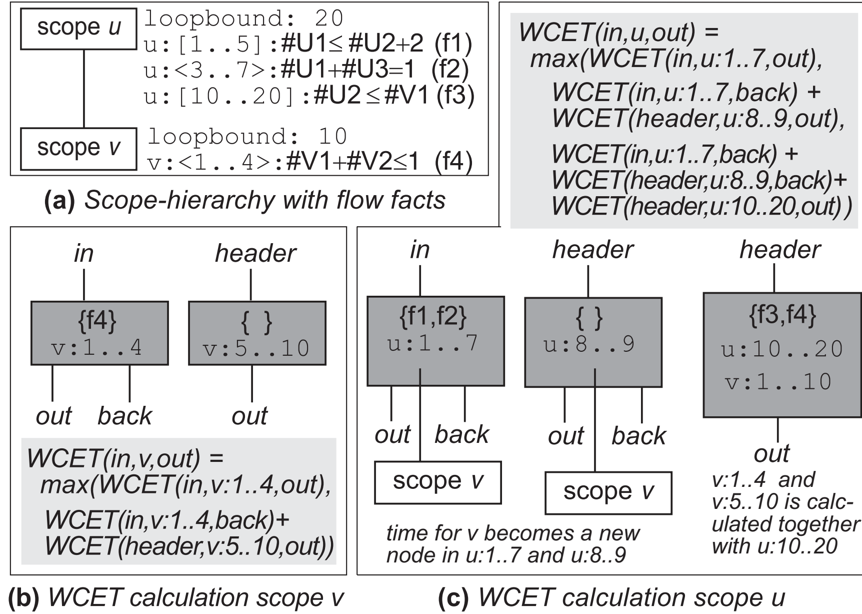


Figure 9.8: Calculation over clusters

information in the cluster, the execution can not continue further. If so, we stop iterating over the ranges and return the total time accumulated for the scope.

Figure 9.8 shows an example of a WCET calculation working over clusters. The algorithm starts at scope `u` where there are several facts covering the iteration range. The name of a referred count variable gives the scope in which the corresponding entity is located, e.g., `#V1` refers to executions of node `V1` located in scope `v`. The facts `f1` and `f2` together form a cluster, {`f1,f2`}, spanning range 1..7 of `u`. Since neither `f1` nor `f2` cover `v`, a local calculation is made for `v` by a recursive call to the algorithm.

The local WCET calculation for `v` only needs to consider facts and fact clusters defined on `v`. Fact `f4` creates a fact cluster on its own, {`f4`}. Two calculations are made for the {`f4`} cluster: one to the out-edges and one to the back-edges. The remaining iterations (5..10) of `v` are not spanned by any fact and an empty fact cluster { } is therefore created for these iterations. Since the fact cluster covers the last iteration of `v`, a WCET estimate is only made to the out-edges, and not to the back-edges as in the previous clusters.

After calculating a WCET estimate for `v`, the calculation restarts at `u` with the fact cluster {`f1,f2`}. There are no facts spanning range 8..9, and an empty fact cluster { } (covering just `u`) is created. For both these calculations, the call to scope `v` is represented by a call node with the timing of the extracted WCET estimate for `v`, i.e., no details of `v` except its timing are included in the

```
CutOutScopeGraph(scopegraph sg, factclusterset FC, factcluster fc, timedatabase tdb):
// Create new scope graph, copy covered scopes, nodes and edges
sg_cut := create empty scope graph
add root scope with nodes start_cut and exit_cut to sg_cut
for each scope s in cover(fc) do
  s_cut := create new scope
  // Copy nodes
  for each node n in s do
    add node n_cut to s_cut
  end for
  loopbound(s_cut) := iterations of s spanned by fc
  add scope s_cut to sg_cut
end for
// Copy edges
for each edge u → v in sg with scope(u),scope(v) ∈ cover(fc) do
  u_cut := node in sg_cut generated from u
  v_cut := node in sg_cut generated from v
  add edge u_cut → v_cut to sg_cut
end for
// Copy hierarchy between scopes
for each scope pair c,p in sg such that p = parent(c) do
  c_cut := scope in sg_cut generated from c
  p_cut := scope in sg_cut generated from p
  set p_cut to be parent of c_cut in sg_cut
end for
// Add edges to enter and exit scope graph
d := defining scope of fact cluster fc
// Create edges to in-nodes and header-nodes
for each edge u → v within (in_edges(d) ∪ back_edges(d)) do
  v_cut := node in sg_cut generated from v
  add edge start → v_cut to sg_cut
  sg_cut := CreateHistoryNodes(u,v,start_cut,v_cut,tdb,sg_cut)
end for
// Copy and redirect out-edges and back-edges
for each edge e = u → v within (out_edges(d) ∪ back_edges(d)) do
  u_cut := node in sg_cut generated from u
  add edge u_cut → exit_cut to sg_cut
end for
// Add call-nodes for non-covered subscopes
for each child scope sub to leaf scopes of cover(c) do
  // Do demand-driven analysis of subscopes
  if time for sub not in tdb then
    tdb := ScopeWCET(sub, sg, FC, tdb)
  // Replace calls to subscopes with call nodes
  t_sub := time for scope sub in tdb
  add node n_sub to sg_cut
  add timing t_sub for node n_sub to tdb
  // Create edges going to call node
  for each edge u → v in sg with scope(u) ∈ complete_subtree(sub) and
    scope(v) ∈ cover(fc) do
    u_cut := node in sg_cut g generated from u
    add edge u_cut → n_sub to sg_cut
  end for
  // Create edges leaving call node
  for each edge u → v in sg with scope(u) ∈ scope(u) ∈ cover(fc) and
    scope(v) ∈ complete_subtree(sub) do
    v_cut := node in sg_cut generated from v
    add edge n_sub → v_cut to sg_cut
    sg_cut := CreateHistoryNodes(u,v,n_sub,v_cut,tdb,sg_cut)
  end for
end for
// Return extracted scope graph
return sg_cut
```

Figure 9.9: Algorithm for creating scope graph for fact cluster

calculation.

Fact `f3`, however, covers both scope `u` and `v` and will be clustered together with the `f4` fact as {`f3,f4`}. This means that when calculating a WCET estimate for scope `u` over the range 10..20 we cannot use the previously generated time for scope `v`, but must do the calculation over *both* `u` and `v`. Observe, that the cluster covers the last iteration of `u` so no calculation for the back-edges is needed.

9.4.1 Extracting a scope graph from covered scope

The scopes covered by a fact cluster are used to extract a new scope graph over which a local WCET calculation is made. The algorithm for extracting the scope graph is given in Figure 9.9. The algorithm starts by generating copies of all the scopes, nodes and edges covered by a fact cluster. A root scope with a start and exit node is also created.

Edges are added to let the execution enter and exit the scopes covered by a fact cluster. Edges are created from the start node to the nodes corresponding to in-nodes and header-nodes of the defining scope of the fact cluster, i.e., the nodes where the execution can enter the scopes covered by the fact cluster. Similarly, to be able to exit the extracted scope graph, edges corresponding to out-edges and back-edges of the defining scope of the fact cluster are copied and redirected to go to the program exit node.

WCET estimates for subscopes not covered by the fact cluster are generated by a recursive call to the algorithm given in Figure 9.7 (this is the demand-driven aspect of the algorithm) and each non-covered subscope is replaced with a *call-node*. A call node is a place-holder for a call to a subscope with a timing of the WCET estimate extracted for the subscope. Edges are added to let the execution enter and exit the call-node in the same way as the execution can call and return from the subscope. To safely encapsulate the effect of border crossing timing effects, extra history nodes are added when the execution enters the scopes covered by the fact cluster or returns from a call to a non-covered subscope (see Section 9.5 for more details). Finally, the extracted scope graph is returned.

9.4.2 Calculation alternatives

A timing estimate for a program fragment should be calculated from where the execution can enter the fragment to where the execution can exit the fragment. For a scope with several in-nodes and out-edges we have the possibility to make a separate WCET calculation for each in-node and out-edge pair, or to make one single WCET calculation for all pairs simultaneously. The first alternative is the least costly in terms of needed CPU resources. However, such calculation will result in a safe but sometimes pessimistic program WCET estimate, since a program fragment WCET estimate sometimes differs depending on where it can be entered or exited.

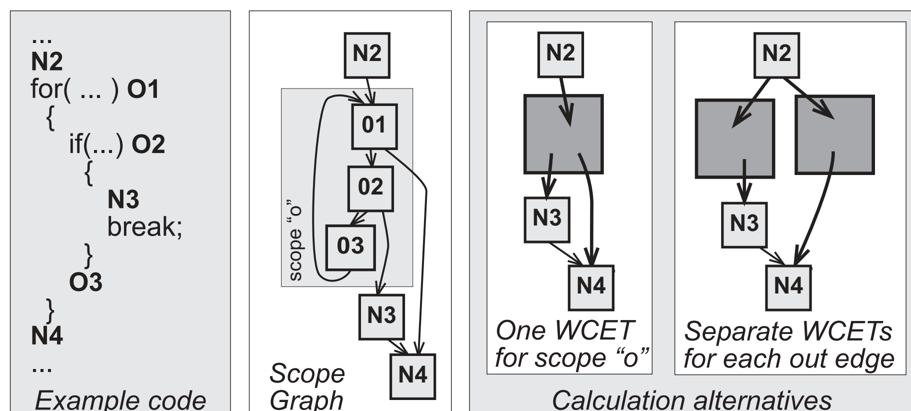


Figure 9.10: Calculation alternatives

As an example of the need for the calculation to differentiate between different out-edges of a scope, consider Figure 9.10. The code corresponds to the third code example in Section 9.3.1, where flow fact `f3` specifies that the edge between `O2` and `N3` can not be taken during the last three iterations of scope `o`.

If only one calculation is made for scope `o` for both its out-edges it will result in a timing estimate for `o` which gives that the loop is iterated 10 full iterations. Later, when doing a WCET calculation for scope `n` the worst case path will be passing the call node for scope `o` together with the nodes `N3` and `N4`. This gives a safe but pessimistic WCET estimate, since the extracted worst case path could not be taken in an actual execution.

The other calculation alternative, which is to make a separate WCET calculation for each out-edge of scope `o`, will discover that the out-edge to node `N3` can not be taken during the last three iterations of `o`. The WCET estimate for scope `o` will therefore be different depending on the execution out-edge used. In the calculation of scope `n` this will result in two separate call nodes for scope `o`, each with different timing. Thus, by making separate calculations for different in-nodes and out-edges the calculation cost increases, but more precise WCET estimates can be achieved.

The same reasoning can be applied for WCET calculation of fact clusters. As discussed in Section 9.4 each fact cluster has a set of begin-nodes and end-edges, and a timing estimate for a cluster is calculated from its begin-nodes to its end-edges. The most costly calculation alternative for a cluster is to make a separate calculation for each pair of begin-nodes and end-edges and the least costly alternative is to calculate a WCET estimate for all pairs of begin-nodes and end-edges simultaneously. The algorithm presented in Figure 9.7 is somewhere in between, not differing between begin-nodes, but making two separate calculations for each cluster, one ending at out-edges and one ending at back-edges. This alternative will keep the calculation complexity at a reasonable level, but allows us to differentiate between executions continuing to the next

cluster in the scope and executions which exit the scope.

Note that we only extract a single scope graph for a fact cluster even though we perform separate calculations for its in-nodes and out-edges. By adding extra flow facts stating which begin-nodes and end-edges that are possible for each particular calculation, an extracted scope graph can be reused. For example, when performing separate calculations for scope `o` in Figure 9.10 we extract a scope graph covering just the `o` scope. For the first calculation of `o` we add a flow fact stating that the `O2` → `N4` edge is not possible to take. For the second calculation of `o` another flow fact is added, stating that the `O2` → `N3` edge is infeasible.

In Chapter 10 we evaluate the impact of different node and edge combinations on the WCET estimate precision and calculation speed.

9.4.3 WCET calculation of clusters

After constructing a WCET calculation working over fact clusters, the next step is to calculate a WCET estimate for each fact cluster and extracted program part in isolation. Since our clustered calculation framework does not specify any specific calculation method, different methods can be used for different clusters, depending on the characteristics of the flow information and covered program fragment.

For example, for fact clusters consisting of only foreach facts and covering only a single scope, our efficient path-based calculation method, (see Chapter 7), can be used. For fact clusters with more complicated flow information our IPET-based calculation method, (see Chapter 8), is preferably used.

9.5 Hardware timing and local calculations

The second input to our clustered calculation is the hardware timing represented by a timing model with timing for nodes (t_{node}) and timing effects (δ_{seq}) for sequences of nodes. As discussed in Section 6.5 there is a potential for timing effects along sequences of nodes longer than two, usually caused by a node using some CPU resource that is used by a later node in the sequence, but not by the nodes in between. A WCET calculation method must be able to safely handle such long reaching hardware timing effects.

Our basic timing model assumes that there is a graph for the entire program, meaning that there are no internal borders in the model. In previous chapters we have outlined how calculation methods can handle timing and timing effects for nodes in the same program graph fragment. This can be done either by graph rewriting, as in our path-based calculation method, (see Chapter 7), or by giving constraints for bounding the number of times a sequence can be executed, as in our IPET-based calculation method (see Chapter 8).

However, for efficiency reasons some calculation methods divides a program into smaller fragments, thereby introducing artificial calculation borders. For

the path-based calculation method the program fragment is a single iteration or a scope, while for the clustered calculation the program fragment can reach over a larger part of the graph. The IPET-based method does not introduce any calculation borders by itself, but need to handle the border problem when used as the final calculation method of our clustered calculation.

9.5.1 Border-crossing timing effects

Timing for nodes and node sequences of length two should be accounted for whenever the corresponding node or edge is executed. A timing effect for a longer node sequence should only be accounted for when the execution goes uninterrupted via all the nodes in the sequence. This means that timing effects have to be accounted for where they end, since only at the end of a sequence do we know that it was actually executed.

This means that when timing effects cross one or more calculation boundaries, they should be accounted in the fragment containing the end of the sequence. However, when calculating the timing for a graph fragment in isolation no knowledge of the worst case execution path in the surrounding fragments can be assumed, i.e., we do not know if the prefix nodes of the sequence was taken or not. Therefore, at the boundaries between fragments, rules are needed to make sure to maintain a safe timing approximation and (hopefully) precision, basically by using a pessimistic but safe assumptions on the possible executions made in the surrounding fragments.

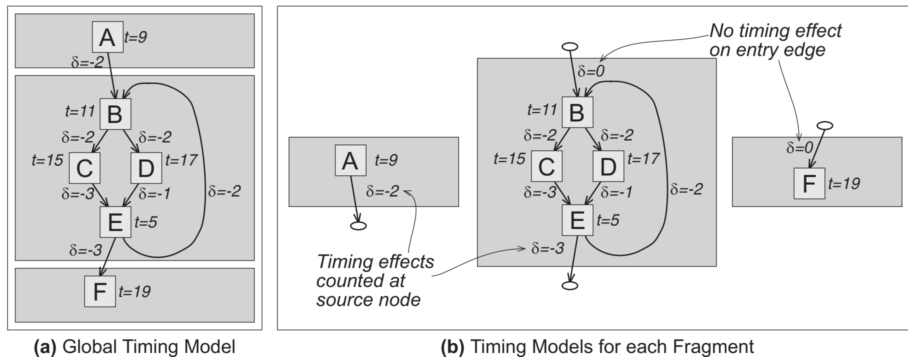


Figure 9.11: Managing timing effects at fragment borders

Using this reasoning, timing effects over pairs of nodes can be handled by including the edge in the same calculation fragment as its source node. This means that edges going from one fragment into another will be counted with the "from" fragment, as shown in Figure 9.11. It is not safe to count the timing effect at both the source and target nodes, since this will count the timing effect of the edge (at least) twice.

Long timing effects are harder to handle, since they lie on a specific sequence

of nodes in the graph. If that sequence is divided by a boundary, it is in the general case not possible to know whether to include the timing effect of the sequence or not. This is handled by adding special *history nodes*, representing the potential paths taken *before* entering the given graph fragment. In a fragment containing incoming timing effects, such history nodes are added at the start of the fragment.

History nodes have an execution time of zero, and their only purpose is to provide support for the specification of long timing effects. A history node is added for each node belonging to a surrounding fragment also included in a long timing effect which ends in a node in the fragment. Edges are added to connect the history nodes, with the result that all incoming timing effects get a corresponding history node sequence. Each added edge has a timing effect of zero.

If there are incoming execution paths not included in any timing effect sequence, extra edges should be added from the start node to the entry of the fragment. This allow the calculations to bypass incoming timing effects if this would result in a larger WCET estimate for the fragment.

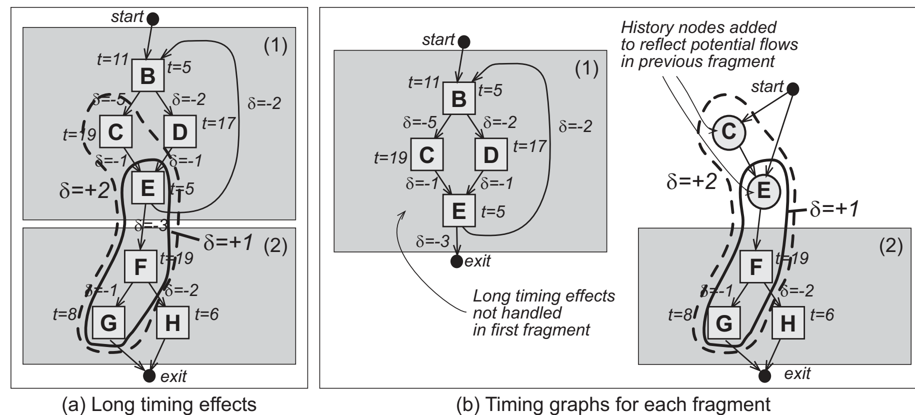


Figure 9.12: Managing long timing effects at fragment borders

Figure 9.12 shows an example of the handling of long timing effects. Due to the two (overlapping) long timing effects over the sequences `CEFG` and `EFG`, two history nodes corresponding to nodes `C` and `E` are added in the timing model for fragment (2), together with the long timing effects. The calculation for fragment (2) will then find the local longest path including the long timing effects overlapping fragments (1) and (2).

The insertion of history nodes gives a safe but possibly pessimistic estimate of the execution times, since we will always use the worst incoming timing effect. For example, in Figure 9.12 the worst case execution path for fragment (1) will exit in sequence `DE`, giving that the long timing effects should not be used in the calculations in fragment (2). However, this cannot be modeled in a fragmented timing model, which will cause an overestimation of the final WCET estimate.

This imprecision is the price we have to pay for the convenience of using a subdivided calculation method.

Note, however, that there is a special case when long timing effects can be modeled precisely across fragment boundaries. If a long timing effect sequence has all its nodes but the last inside one fragment, its timing effect is modeled as part of the originating fragment. Since the exit edge leading to the last node is part of the first fragment, it can be deduced within the first fragment whether the long timing effect is activated or not.

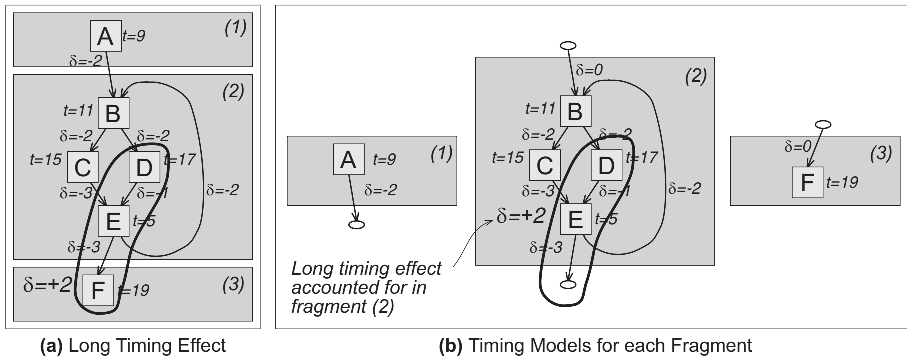


Figure 9.13: Special case of only last node across the boundary

Figure 9.13 illustrates the special case. When fragment (2) exits via the edge `EF`, we know that `F` will be the next node executed. Thus, the long timing effect `DEF` can be modeled within fragment (2) rather than by history nodes in fragment (3).

9.5.2 Generating history nodes

We have constructed an algorithm for generating history nodes for long timing effects reaching over calculation borders. The algorithm is given in Figure 9.14 and takes six arguments. The first two arguments, u_{org} and v_{org}, are the nodes that the original long timing sequence must be passing to generate new history nodes in the extracted graph. The third and fourth argument, u_{new} and v_{new} are the nodes that the new history node sequences will be going in between. tdb is the timing model for the original scope graph. For border crossing sequences from subscopes, the u_{new} node will be a newly created call-node, and for border-crossings over the begin-nodes of the cluster, u_{new} will be the root-scope start node of sg_{new}.

The algorithm starts by extracting border crossing time sequences with more than one node in the covered scopes. These are the long timing sequences including the u_{org},v_{org} subsequence and ending in a node in the fragment. For example, for the example graph in Figure 9.12 the `CEFG` and `EFG` sequences both pass the `EF` subsequence and has more than two nodes in fragment (2), and will therefore both generate history nodes.

```
CreateHistoryNodes(scopenode u_org, scopenode v_org, scopenode u_new,
                   scopenode v_new, timedatabase tdb, scopegraph sg_new):
// Collect long border crossing timing sequences
PreSeqs := ∅ // Set for holding prefixes of border crossing sequences
// Extract node sequences generating history nodes
for each timing effects δ_seq with length(seq) ≥ 3 in tdb do
  if seq includes subsequence u_org,v_org and last node in seq ≠ v_org then
    pre := prefix sequence of seq ending at u_org in last u_org,v_org subsequence
    PreSeqs := PreSeqs + pre
end for
// Create history nodes from prefix sequences
Proc := ∅ // Set for holding processed prefix sequences
for each prefix sequence pre in PreSeqs in decreasing length order do
  if pre is suffix of previously processed sequence in Proc do
    // The prefix sequence is suffix of previously created history nodes
    proc := previously processed sequence in Proc having pre as suffix
    n_proc := new history node in proc created from first node in pre
    create edge from node u_new to node n_proc
    add newly created edge to sg_new
  else
    // Create and connect history nodes
    pre_new := create new history node sequence from nodes in pre
    create edge from node u_new to first node in pre_new
    create edge from last node in pre_new to node v_new
    add newly created history nodes and edges to sg_new
  Proc := Proc + pre
end for
// Return updated scope graph
return sg_new
```

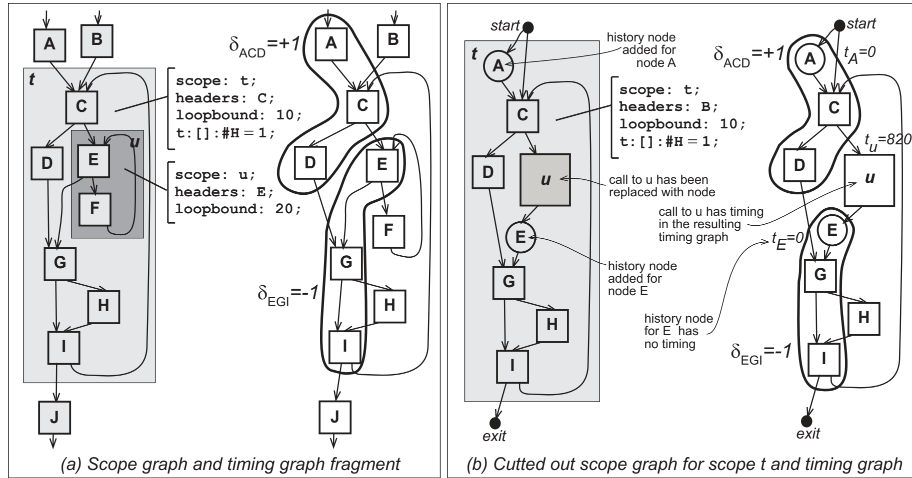
Figure 9.14: Algorithm for history node creation

Next, sequence prefixes are extracted to reflect that only the nodes before the last u_{org},v_{org} subsequence will generate history nodes. For example, for the border crossing sequences in Figure 9.12 the prefix sequences `CE` and `E` will be extracted.

In the next step the history nodes are generated and connected according to the extracted prefix sequences. The algorithm needs to consider that if a short prefix sequence is a suffix of another longer prefix sequence then an execution of the longer sequence implies an execution of the shorter sequence. For example, in the resulting graph fragment in Figure 9.12 an execution of history node sequence `CE` implies an execution of history node sequence `E`.

9.5.3 Example of timing effects over calculation borders

Figure 9.15 gives an example of long timing effects reaching over fact cluster calculation borders. Figure 9.15(a) shows a scope graph fragment with the scopes `t` and `u`. Scope `t` has a flow fact covering scope `t` but not scope `u`. This generates a fact cluster only consisting of this single fact covering only the `t` scope. The corresponding timing graph fragment contains timing for nodes and



(a) Scope graph and timing graph fragment (b) Cutted out scope graph for scope t and timing graph

Figure 9.15: Example of clustered calculation and long timing effects

edges, (not shown in the figure), as well as two long timing effects, illustrated as $\delta_{\texttt{ACD}}$ and $\delta_{\texttt{EGI}}$.

When making a clustered calculation the `u` scope will be calculated in isolation from the `t` scope. A worst case estimate for scope `u` is first calculated and given a timing. The extracted scope graph for scope `t` is illustrated in Figure 9.15(b). In the graph, the call to scope `u` has been replaced with a call-node. To be sure to capture the long timing effect $\delta_{\texttt{ACD}}$ a history node is created for the `A` node. Since we can enter `t` by two different edges, an extra edge from the start node is added to allow the calculation to bypass the timing effect sequence.

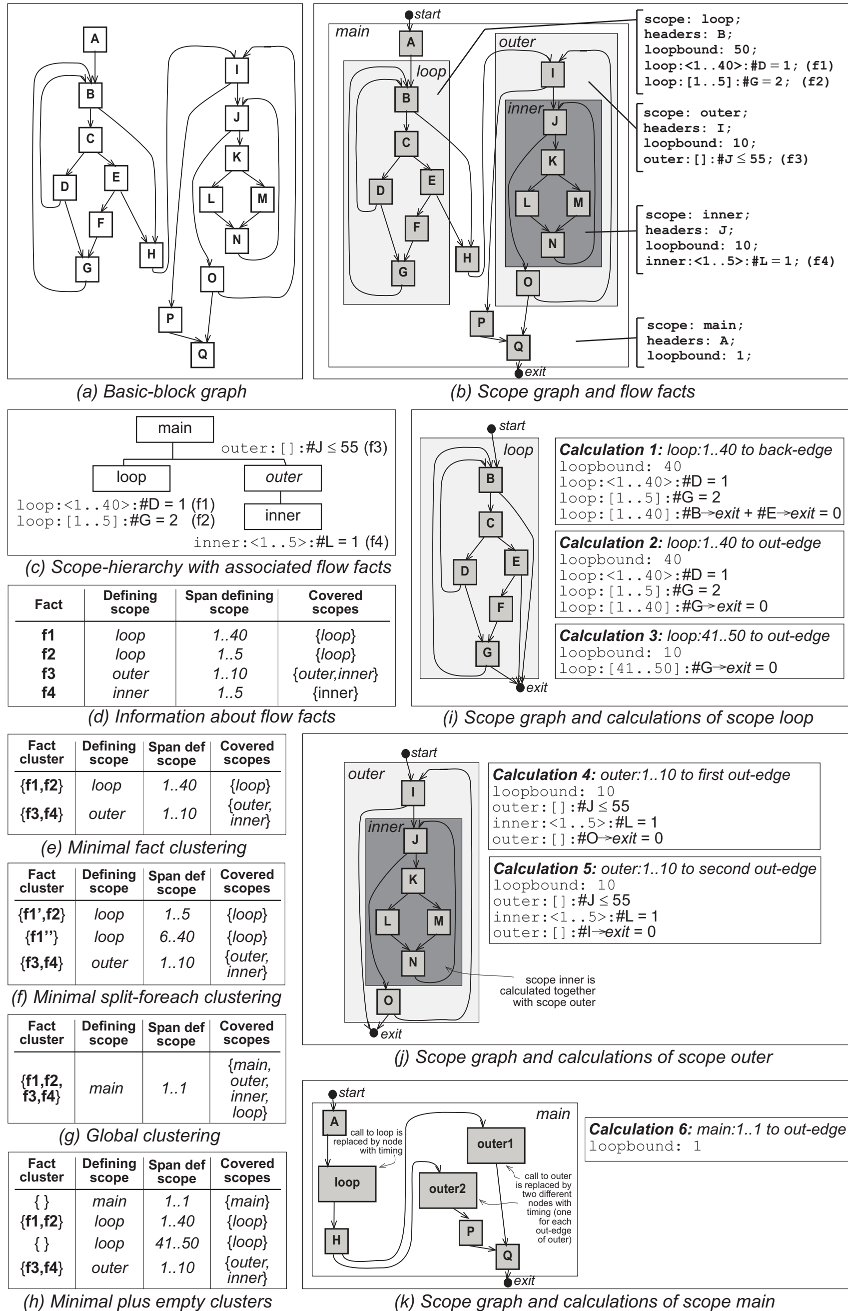
We also have a long timing effect $\delta_{\texttt{EGI}}$ originating in node `E` in the non-covered scope `u`. To be sure to capture this long timing effect, a history node is created for the `E` node. Since the execution is forced to pass the `E` node in order to leave scope `u`, all paths leaving the new `u` node must also pass the `E` history node.

In Chapter 10 we will evaluate the pessimism introduced by long timing effects over calculation borders dictated by fact clusters.

9.6 Complete example

In Figure 9.16(a)-(k) we give a compact illustration of the steps involved in our clustered calculation method. To simplify the presentation, no timing for entities is included in the example.

Figure 9.16(a) shows an example control-flow graph consisting of a single loop and a loop nest consisting of two loops. Figure 9.16(b) shows the corresponding scope graph with scopes `main`, `loop`, `outer` and `inner`. Each scope has a loop bound and some have flow facts attached. Note that both `loop` and `outer` have multiple out-edges. Figure 9.16(c) shows the corresponding scope-



Fact	Defining scope	Span defining scope	Covered scopes
f1	*loop*	*1..40*	{*loop*}
f2	*loop*	*1..5*	{*loop*}
f3	*outer*	*1..10*	{*outer,inner*}
f4	*inner*	*1..5*	{inner}

(d) Information about flow facts

Fact cluster	Defining scope	Span def scope	Covered scopes
{**f1,f2**}	*loop*	*1..40*	{*loop*}
{**f3,f4**}	*outer*	*1..10*	{*outer, inner*}

(e) Minimal fact clustering

Fact cluster	Defining scope	Span def scope	Covered scopes
{**f1',f2**}	*loop*	*1..5*	{*loop*}
{**f1''**}	*loop*	*6..40*	{*loop*}
{**f3,f4**}	*outer*	*1..10*	{*outer, inner*}

(f) Minimal split-foreach clustering

Fact cluster	Defining scope	Span def scope	Covered scopes
{**f1,f2, f3,f4**}	*main*	*1..1*	{*main, outer, inner, loop*}

(g) Global clustering

Fact cluster	Defining scope	Span def scope	Covered scopes
{ }	*main*	*1..1*	{*main*}
{**f1,f2**}	*loop*	*1..40*	{*loop*}
{ }	*loop*	*41..50*	{*loop*}
{**f3,f4**}	*outer*	*1..10*	{*outer, inner*}

(h) Minimal plus empty clusters

Figure 9.16: Complete example of clustered calculation

hierarchy, with flow facts attached to the different scopes. Figure 9.16(d) shows the defining scopes, defining scope spans, and cover of the different flow facts, as defined in Section 9.3.

Figure 9.16(e) shows the fact clusters generated when applying the minimal clustering algorithm given in Figure 9.6. Since fact `f1` and `f2` overlap in their ranges they will be put in the same cluster. Fact `f3` refers to the header node of scope `inner` and is therefore put in the same cluster as `f4`[1].

Figure 9.16(f) shows the fact clusters generated when applying the split-foreach-fact minimal clustering (see Section 9.3.2). Fact `f1` has been split into two new facts `loop : <1..5> : #D` = 1((`f1'`) and `loop : <6..40> : #D` = 1((`f1''`). The fact clusters {`f1',f2`} and {`f1''`} together span the same range as the {`f1,f2`} cluster given in Figure 9.16(e).

Figure 9.16(g) shows the fact cluster generated when applying the global clustering, (see Section 9.3.2). All facts are put into one cluster, with `main` as defined scope, and will all be consider together as a unit in the final calculation. This corresponds to the overall program view used in our extended IPET calculation (see Chapter 8).

For the rest of the example we use the clusters in Figure 9.16(e) as generated by the minimal clustering. Figure 9.16(h) shows the resulting set of clusters, after adding empty clusters for all ranges of scopes not covered by any cluster. This is done as part of the algorithm given in Figure 9.7. An empty cluster for range 1..1 of `main` and one empty cluster for range 41..50 of `loop` is created.

Our demand-driven WCET calculation algorithm given in Figure 9.7 starts at scope `main`. Since only `main` is covered by the empty fact cluster, recursive calls are made for scope `loop` and `outer`, before calculating the WCET of `main`.

Scope `loop` is covered by two fact clusters, {`f1,f2`} and an empty cluster. The calculation starts with cluster {`f1,f2`}, since it spans the first iteration of `loop`. Using the algorithm given in Figure 9.9 a new scope graph is extracted for the covered scope `loop`. The resulting scope graph is given in Figure 9.16(i). For the extracted scope graph and fact cluster {`f1,f2`}, two different calculations are made: one ending at the back-edge of `loop` (Calculation 1) and one ending at the out-edge of `loop` (Calculation 2). The same scope graph is used for both calculations, but some extra flow facts are added in each calculation to constrain where the execution should end (as outlined in Section 9.4.2).

The calculation continues with the empty cluster spanning range 41...50 of scope `loop`. When calculating a WCET estimate for this cluster we reuse the extracted scope graph for scope `loop`. Since the cluster is empty, no flow facts are included, except one specifying that the execution must end at the out-edge (Calculation 3). The three different WCET estimates extracted are used together, as given by the algorithm in Figure 9.7, to calculate a WCET estimate for scope `loop`.

The next step is to calculate a WCET estimate for scope `outer`. Since the

[1]Fact `f4` is also constituting a fact cluster on its own, {`f4`}, defined on scope `inner`, but this is not included in Figure 9.16(e), since `f4` will always be calculated together with `f3`.

fact cluster {f3,f4} covers both scope outer and inner, a WCET estimate will be extracted for both scopes together. Using the algorithm given in Figure 9.9 we first extracts a new scope graph for the scopes, as shown in Figure 9.16(j). For the extracted scope graph and fact cluster, two different calculations are made, one to the out-edge with node I as source (Calculation 4), and one to the out-edge with node O as source (Calculation 5). As discussed in Section 9.4.2, making just one calculation for both out-edges might result in a pessimistic WCET estimate.

After calculating WCET estimates for scope loop and outer a WCET estimate for scope main can be calculated. A scope graph for scope main is extracted (again using the algorithm in Figure 9.9), as shown in Figure 9.16(k). In the extracted graph, the calls to scope loop and outer are replaced with call nodes. Each node is given a timing equal to the WCET estimate extracted for the call to the corresponding scope. Note that scope outer gets replaced with two different call nodes. No fact is covering scope main, and only calculation needs to be made for this scope (Calculation 6).

Note that we do not put any demands on the calculation method to use when calculating a WCET estimate for a scope graph and fact cluster. For example, for the calculations of the main scope or the last range of loop, both our path-based (Chapter 7) and extended IPET (Chapter 8) methods can be used. For fact clusters with more complicated flow information, such as {f3,f4}, the extended IPET is preferably used to guarantee that all included flow facts can be accounted for.

Chapter 10

Prototype Tool and Experiments

In this chapter we present our implemented WCET analysis tool. We present results from a number of experimental runs to evaluate the correctness, precision and efficiency of our prototype. In particular, we evaluate the impact of timing and flow information on the WCET estimate precision and computation time, as well as individual analysis phases.

10.1 Prototype implementation

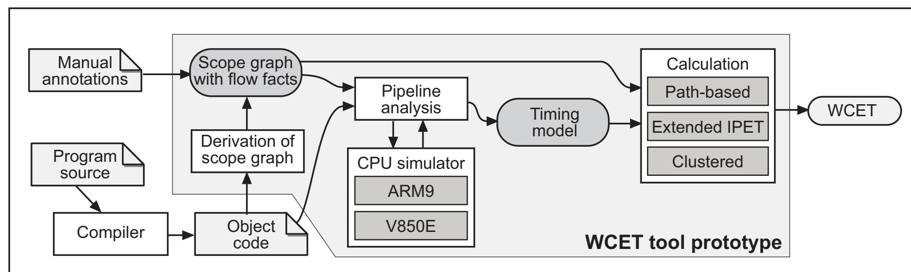


Figure 10.1: WCET tool prototype implementation

Based on the modular WCET tool architecture presented in Chapter 3 we have implemented a WCET analysis tool prototype. Figure 10.1 illustrates the current prototype implementation. The prototype incorporates the scope graph and flow facts flow representation (Chapter 5), the pipeline timing analysis and the resulting timing model (Chapter 6), and the three different calculation methods (Chapters 7, 8 and 9). The tool does not yet use automatic flow

analysis, and global low-level analysis has not been needed for the targets and programs we have studied. Thus, the current prototype is a subset of the WCET tool architecture outlined in Chapter 3.

The prototype runs under Solaris, Linux, and Windows 2000. The tool currently performs WCET analysis for two different target chips, NEC V850E [NEC99] and ARM9 [ARM00], both typical 32-bit RISC micro-controller architectures with pipelines.

A basic block graph and a scope graph (annotated with loop bounds and flow facts) are required as input to our prototype. For ARM9 programs, we directly parse object code to generate a basic block graph. For NEC V850E programs, we rely on a modified IAR V850/V850E C/Embedded C++ compiler [IAR99]. We have defined a textual format for scope graphs and flow facts and a special target-independent textual format for basic block graphs.

We have implemented an algorithm to construct a scope graph from a basic block graph by identifying loops and functions. The program does not currently handle all type of programming constructs found in normal code, such as function pointers and recursion. No automatic flow analysis is currently performed, and flow information, in the form of flow facts and loop-bounds, must be manually added. Work is currently under way to automate the generation of scope graphs and flow information [GLSB03].

The implementation of the scope graph and flow fact language supports all of the features presented in Chapter 5. However, the implementation does not support scopes with multiple headers, disallowing some type of unstructured code to be handled, but this is planned as future work.

We have implemented a cache analysis in the style of Ferdinand et al. [FMW97], (not shown in Figure 10.1), just to check that it is possible to deposit information about cache hits and misses in the scope graph. However, we are not using it in our experiments, since none of our target architectures use a cache.

We have implemented two rather different CPU models [Eng02], one for the NEC V850E [NEC99] and one for the ARM9 [ARM00]. Both models returns obtained execution times for a given sequence of instructions based on a trace-driven simulation.

To gain insight in how a simulator works and to facilitate experiments with complete control of the simulator, the V850E CPU model was implemented in a detailed manner. Engblom reports on experiences from the implementation in [Eng02]. The model includes a detailed simulation of CPU resource usage in terms of pipeline stages for instructions. The timing behaviour of the V850E model has been validated against an emulator for the NEC V850E [Mon00]. The validation indicates that the CPU model has a close timing correspondence to the real hardware.

Our ARM9 CPU model was developed to support a case study for doing WCET analysis on interrupt disable regions of the OSE Delta RTOS [CEE+02]. The model has mainly been developed from the execution time information found in ARM9 user's manual [ARM00]. The model uses a table of execution

counts for each instruction, based on type of operands and operations performed, with additional rules for handling data dependencies between instructions, i.e., a much simpler model than for the V850E. The ARM9 model has not been validated against real hardware. We will not use the ARM9 model in any of our experiments.

As illustrated in Figure 10.1, we have implemented three different calculation modules: a path-based (Chapter 7), an Extended IPET (Chapter 8) and a clustered calculation method (Chapter 9). Each calculation module takes the same inputs: a scope graph with flow facts (representing possible program flows) and a timing model (representing hardware timing). Each calculation module can be chosen independently from the target processor, indicating that our modular design makes it easy to port the tool to new targets and exchange components and analyses.

Our extended IPET calculation method and the clustered calculation method both rely on integer linear programming (ILP) or constraint programming (CP) to solve generated constraint systems. We have integrated the mixed ILP solver `lp_solve` [Ber03] into our prototype. We also support the possibility to export generated constraints in a format suitable as input to Sicstus Prolog CP [Int95]. The Prolog solver is much slower than `lp_solve`, but allows more complicated constraints in our flow facts, e.g., multiplication between variables. We have also done some testing with the commercial mixed integer and constraint solver `CPLEX` [CPL03]. It was very fast for most of our benchmarks. We have used `lp_solve` in all our experiments, mainly because it is much faster than the Prolog solver, and it is freely available (which `CPLEX` is not).

The current implementation does not support the possibility to perform path-based calculation within clusters, i.e., only our extended IPET can be used within clusters. Support for allowing both calculation methods inside clusters is planned as future work.

The extended IPET calculation can produce both WCET and BCET estimates. A BCET estimate is calculated by minimizing (instead of maximizing) the objective function. Flow facts should be valid for *all* possible program executions, i.e., not just the worst case, and can therefore be used in both WCET and BCET calculations. Also, the timing information produced by our pipeline timing analysis should be valid both for WCET and BCET analysis as long as we do not use a hardware model specially constructed for WCTE analysis. However, when using global low-level analysis we must be sure that all possible execution scenarios are included, not just the worst cases.

10.2 User interaction and feedback

Our prototype WCET analysis tool is currently command-line driven, and supports a number of different command-line options. For example, it is possible to select the CPU model by using a `--cpu` option, and the `--mode` option specifies the input file and which calculation method to use.

For each calculation method there are additional options to choose from, specific for the particular method used. For example, for our clustered calculation method we can choose between five different algorithms for clustering facts (see Section 9.3.2 on page 150). Also, there is a number of debug options used for giving statistics of program runs, something used heavily for the measurements presented in this chapter.

Both the scope graph and the basic block graph can be visualized graphically. We use `dot` [Dot97] from AT&T Bell laboratories to illustrate generated graphs. Obtained results, in form of generated counts for nodes and edges, are displayed in the graphs.

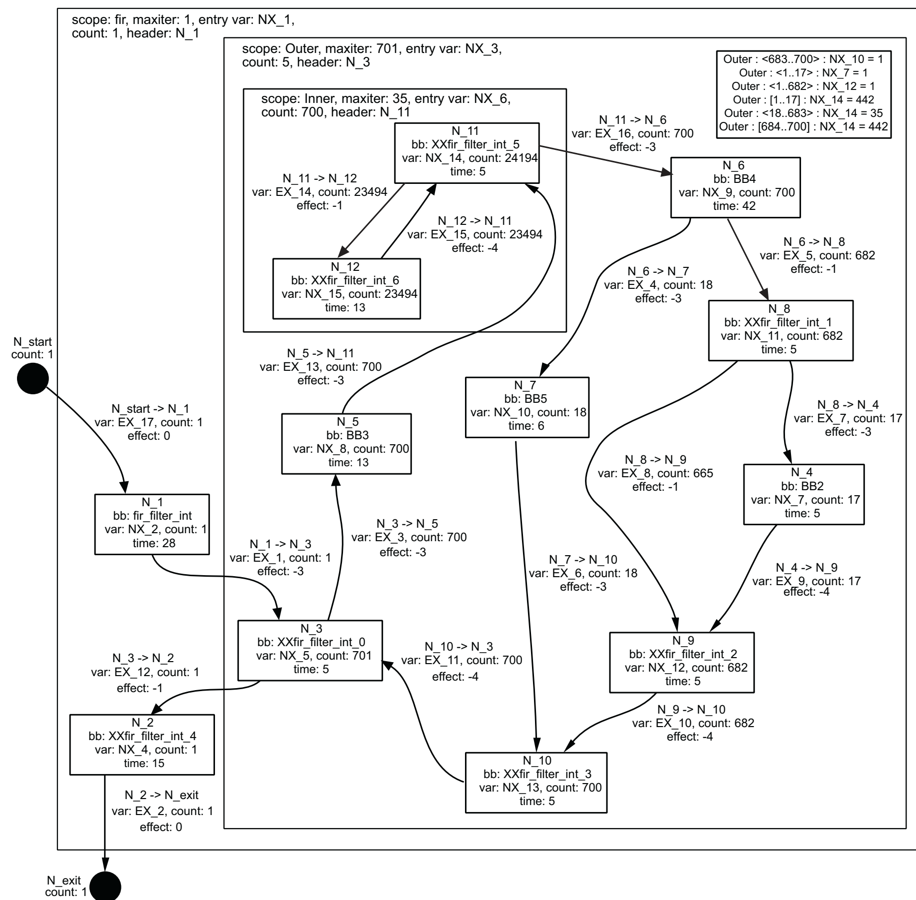


Figure 10.2: Scope graph for `fir` generated by prototype tool

Figure 10.2 illustrates the scope graph generated for a fragment of `fir` using our WCET analysis tool. The scope graph corresponds to the scope graph

depicted in Figure 8.11 on page 137. Note that each node and edge contains the execution time and timing effect, as well as an execution count of the entity in the WCET execution.

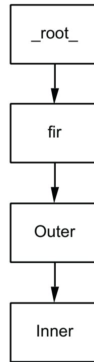


Figure 10.3: Scope hierarchy for `fir` generated by prototype tool

Figure 10.3 illustrates the corresponding scope hierarchy as generated by our prototype tool. Note that the root scope is explicity represented as `_root_`.

10.3 Benchmark programs

"Benchmarks are like sex, everyone wants it, everybody is sure they know how to do it, but nobody knows how to compare performance" [Tur02]. This citation is particularly true when evaluating WCET analyses. All WCET research groups use different target architectures. Two obtained WCET estimates can only be directly compared, if the estimates are derived for same target architecture using the same configuration and for the same object code (different compilers might generate very different executables from the same source code). This is never the case.

However, by comparing obtained WCET estimates with actual worst-case executions (assuming they are available) we can get an idea of the *precision* of a particular WCET analysis method. Furthermore, to get an indication of computational complexity of a particular WCET analysis method and its included components, we can investigate its *computation time* characteristics as we scale the workload. For example, having three different calculation methods and using an identical setup for the remaining tool components, we can compare their precision and computation time.

We have performed a number of experiments using the collection of benchmark programs shown in Figure 10.4. The benchmarks are a diverse collection of test programs differing in types of flows and structures, intended to thoroughly test all aspects of the flow representation, pipeline timing analysis and calculation. Most of the programs have been used by other WCET research

Program	Description	Properties
`compress`	Compression using lzw.	Nested loops, goto-loop, function calls.
`crc`	Cyclic redundancy check computation on 40 bytes of data.	Complex loops, lots of decisions, loop bounds depend on function arguments, function that executes differently the first time it is called.
`duff`	Using "Duff's device" [Ray00] to copy 43 byte array.	Unstructured loop with known bound, `switch` statement.
`expint`	Series expansion for computing an exponential integral function	Inner loop that only runs once, structural WCET estimate gives heavy overestimate.
`fibcall`	Simple iterative Fibonacci calculation, used to calculate fib(30).	Parameter-dependent function, single-nested loop.
`fir`	Finite impulse response filter (signal processing algorithms) over a 700 items long sample.	Inner loop with varying number of iterations, loop-iteration dependent decisions.
`insertsort`	Insertion sort on a reversed array of size 10.	Input-data dependent nested loop with worst-case of $n^2/2$ iterations.
`jfdctint`	Discrete-cosine transformation on a 8x8 pixel block.	Long calculation sequences (i.e., long basic blocks), single-nested loops.
`lcdnum`	Read ten values, output half to LCD	Loop with iteration-dependent flow.
`matmult`	Matrix multiplication of two 20x20 matrices.	Multiple calls to the same function, nested function calls, triple-nested loops.
`ns`	Search in a multi-dimensional array	Return from the middle of a loop nest, deep loop nesting.
`nsichneu`	Simulate an extended Petri Net	Automatically generated code containing massive amounts of `if`-statements ($\gg$ 250)

Figure 10.4: Benchmark programs used in experiments

groups, while some have been specially crafted to test a particular aspect of our WCET analysis.

Figure 10.5 shows some properties of interest for our WCET analysis phases. All properties are given for the executable program when compiled for the V850E processor. The `Basic block` column gives the number of basic blocks in the object code and is an indication of the program size. The `Scopes` and `Scope nodes` give the number of scopes and scope nodes in the scope graph. The number of scopes is an indication of the number of loops and functions found in the source code program.

For most programs the number of scope nodes is just two plus the number of basic blocks. In these scope graphs each basic block is referenced by a single scope node, and then we add start and exit nodes. However, for some programs, such as `crc` and `matmult`, the number of scope nodes are higher. This indicates that some basic blocks are referenced by several different scope nodes. This is typically due to a function being called from two different places in the code (see Section 5.6). The `Flow facts` column gives the number of flow facts (excluding

Program	Basic blocks	Scopes	Scope nodes	Flow facts
`compress`	91	24	108	9
`crc`	29	12	57	11
`duff`	20	6	22	2
`expint`	24	7	26	4
`fibcall`	7	4	9	0
`fir`	14	5	16	7
`insertsort`	7	4	9	1
`jfdctint`	14	6	16	0
`lcdnum`	26	4	28	2
`matmult`	27	16	38	0
`ns`	18	7	20	1
`nsichneu`	754	3	756	129

Figure 10.5: Benchmark properties

	Basic		With Flow		With Pipe		Flow & Pipe		Actual
Program	Cycles	+%	Cycles	+%	Cycles	+%	Cycles	+%	Cycles
`compress`	126242	+1357	10388	+20	92482	+967	8672	+0.12	8662
`crc`	61624	+104	61340	+103	30389	+0.39	30271	0	30271
`duff`	1823	+86.3	1775	+63.9	1104	+1.9	1083	0	1083
`expint`	68077	+693	10062	+17.2	41359	+382	8588	0	8588
`fibcall`	559	+78.6	559	+78.6	313	0	313	0	313
`fir`	487970	+40.2	487808	+40.1	352162	+1.2	352073	+1.1	348095
`insertsort`	2328	+117	1428	+33.0	1794	+67.0	1074	0	1074
`jfdctint`	5388	+9.4	5388	+9.4	4925	0	4925	0	4925
`lcdnum`	501	+153	341	+72.2	283	+42.9	198	0	198
`matmult`	275879	+24.4	275859	+24.4	221824	0	221824	0	221824
`ns`	25741	+84.8	25713	+84.6	17373	+24.7	17353	+24.6	13928
`nsichneu`	150841	+195	87193	+70.6	97645		51116	0	51116

Figure 10.6: WCET estimates with and without timing effects and flow facts

basic loop-bounds) added to provide additional flow information.

For each program we have derived an *actual WCET* value by running a worst-case trace of the program through the CPU simulator used by the WCET analysis. Thereby, the experiments only deal with the effectiveness and precision of the WCET analysis phases, and do not need to take into account any differences between the simulator and the actual hardware. The traces were obtained by manual analysis of the program source and object codes, and extensive testing was undertaken to make sure that they really correspond to the actual WCETs.

10.4 WCET estimate precision

The need for both correct low-level timing and flow information in deriving safe and tight WCET estimates is illustrated in Figure 10.6. The column **Basic** gives the WCET estimate when ignoring pipeline overlap between basic blocks,

(but including the pipeline overlap within basic blocks), and using only basic loop bounds as flow information. The columns including **Flow** gives WCET estimates resulting from adding flow facts to the program. Columns including **Pipe** gives WCET estimates where timing effects between nodes have been accounted for. Column **Actual** gives the actual WCET of the program, as given by a simulation of the target platform. The numbers in the **+%** columns give the pessimism of each WCET estimate in percent, relative the actual WCET.

The need for analyzing the pipeline timing is illustrated by the values in the **With Pipe** column. The WCET overestimation is clearly reduced compared to the **Basic** column. In general, modelling pipeline overlaps between basic blocks seems to tighten the WCET estimates by at least 20 percent, (ignoring pipeline effects *within* basic block would create WCET estimates about five times higher, since V850E has a five-stage pipeline). The benefit is greatest for programs with many small basic blocks (like `crc`, `fibcall` and `nsichneu`), and least for programs with large basic blocks (like `jfdctint`).

The need for correct flow information is illustrated by the values in the **With Flow** column. The improvements in WCET estimate precison due to flow information vary much more for the different benchmarks than the effect of pipeline analysis. Some benchmarks, like `compress` and `nsichneu`, show large decrease in obtained WCET overestimations, while for other programs the improvement is much smaller.

The good results in the **Flow & Pipe** column indicates that to obtain WCET estimates both high quality flow and timing information must be obtained. All WCET estimates are safe, i.e., larger than or equal to the actual WCET. The remaining execution time overestimation is mostly due to the problem of correctly model the program flow.

Compared to other work in WCET analysis, the WCET estimate precision we achieve is competitive. Healy et al. report 0% overestimation (perfect agreement) for predictable programs like `matmult`, and up to 100% overestimation for a sort program similar to `insertsort`, considering only the pipeline of the MicroSPARC 1 [HAM+99]. Stappert and Altenbernd report overestimations between 4% and 13% for straight-line code on a superscalar PowerPC 604 processor with caches [SA00]. Ernst and Ye report overestimates ranging from 0% to 10%, analyzing a SPARC pipeline similar in complexity to our V850E, using IPET calculation [EY97]. Colin and Puaut report overestimates between 1% and 265% for a simplified integer-only Pentium CPU with caches and branch prediction [CP00]. Lundqvist and Stenström report overestimates between 0% for programs with predictable flow, up to 1600% for programs with very irregular flow [LS00].

Note that not all published work include measurements comparing the generated WCET estimates with actual execution times; in many cases, the effectiveness of analysis methods are evaluated by comparing the estimated and actual number of cache misses etc., and not actual execution times.

Program	Path-based		Local Facts		Clustered		Ext. IPET		Actual	Lte
	Cycles	+%	Cycles	+%	Cycles	+%	Cycles	+%	WCET	
compress	8670	+0.09	8670	+0.09	8670	+0.09	8670	+0.09	8662	3
crc	30275	+0.01	30271	0	30271	0	30271	0	30271	0
duff	-	-	1083	0	1083	0	1083	0	1083	0
expint	8588	0	8588	0	8588	0	8588	0	8588	0
fibcall	313	0	313	0	313	0	313	0	313	0
fir	352073	+1.14	352073	+1.14	348095	0	348095	0	348095	0
insertsort	1794	+67.04	1794	+67.04	1074	0	1074	0	1074	0
jfdctint	4942	+0.3	4926	+0.02	4926	+0.02	4925	0	4925	3
lcdnum	198	0	198	0	198	0	198	0	198	0
matmult	221824	0	221824	0	221824	0	221824	0	221824	0
ns	23746	+70.49	17353	+24.59	17353	+24.59	17353	+24.59	13928	0
nsichneu	51133	+0.03	51116	0	51116	0	51116	0	51116	1

Figure 10.7: WCET estimate precision for calculation methods using V850 model

10.5 Flow facts and WCET precision

Not all calculation methods can make use of all type of flow information. In order to evaluate how this will affect the WCET estimate precision we calculated WCET estimates for all benchmarks using our different calculation methods. Indirectly we also test how different types of flow facts will affect the WCET estimate precision.

The table in Figure 10.7 shows that the WCET estimate precision computation time depends on the calculation method used. The **Path-based** column gives results for our path-based implementation, using only foreach facts valid within single scopes. The **Local Facts** column gives results for a clustered calculation using facts (both foreach and total) valid within single scopes. The **Clustered** column gives results using minimal clustering and **Ext. IPET** gives results for the global extended IPET calculation method.

The **Cycles** columns gives the WCET estimates extracted (in clock cycles) and the +% columns gives the pessimism of each WCET estimate in percent. The numbers in the **Actual WCET** gives the actual WCET of the program. The path-based approach does not work with the `duff` benchmark, since it contains an unstructured loop.

Comparing the **With Pipe** column in Figure 10.6 with the columns in Figure 10.7, we first conclude that for most programs loop bounds alone are not sufficient to achieve tight WCET estimates. The path-based and local-facts WCET estimate precision is of the same quality as the clustered and extended IPET for most programs. This also indicates that using foreach facts to model possible program flows are for many programs sufficient for achieving precise WCET estimates. However, some programs like `insertsort` and `fir` need extra flow facts covering *several scopes* to reduce the overestimation. This indicates that the local calculation schemes are not sufficient to handle the flow of all programs. The almost identical values in columns **Clustered** and **Ext. IPET** tell us that the clustered calculation performs as well as extended IPET, the

Program	Path-based		Local Facts		Clustered		Ext. IPET		Actual	Lte
	Cycles	+%	Cycles	+%	Cycles	+%	Cycles	+%	WCET	
compress	15457	+1.10	15306	+0.11	15306	+0.10	15304	+0.10	15289	26
crc	34432	+1.53	33917	+0.01	33917	+0.01	33917	+0.01	33914	8
duff	-	-	1468	0	1468	0	1468	0	1467	2
expint	8720	0	8720	0	8720	0	8720	0	8720	0
fibcall	332	0	332	0	332	0	332	0	332	0
fir	597206	+1.19	597206	+1.19	590168	0	590168	0	590168	0
insertsort	3963	+40.55	3981	+68.97	2361	+0.21	2361	+0.21	2356	3
jfdctint	6626	+3.77	6393	+0.13	6393	+0.13	6392	+0.12	6385	3
lcdnum	268	+12.61	238	+12.61	238	0	238	0	238	5
matmult	312550	+9.53	287470	+0.74	287470	+0.74	287468	+0.74	285348	13
ns	29243	+70.56	21350	+24.53	21350	+24.53	21349	+24.52	17145	1
nsichneu	135793	+0.01	135776	0	135776	0	135776	0	135776	3

Figure 10.8: WCET precision using V850 model with 6 cycle memory

current method with highest precision.

10.6 Long timing effects and WCET precision

A long timing effect is an effect reaching over a sequence of three or more nodes (see Section 6.5). The number of long timing effects varies with the processor architecture and the properties of the program code. For the V850E, only a few of our benchmark programs contained long timing effects. The **Lte** column in Figure 10.7 gives the number of long timing effect generated for each program.

We have also used a model of the NEC V850E with a data memory latency of 6 clock-cycles. This is an artificial model not corresponding to any real setup of the V850E, but very useful in provoking long timing effects. The impact for the pipeline timing analysis computation time and precision with the appearance of many long timing effects is discussed in more detail in [Eng02].

We use the model to investigate how long timing effects affect the WCET estimate precision. This is particularly relevant for calculations that partition the program into smaller parts to increase efficiency, such as our path-based and clustered calculation. When long timing effects reach over calculation boundaries they might introduce pessimism in the WCET estimate calculation, as discussed in Section 9.5. For example, in Figure 10.7 the one clock cycle difference of WCET estimates for `jfdctint` of the clustered and the extended IPET calculation, is caused by a border-crossing long timing effect.

Figure 10.8 gives the WCET precision achieved for our calculation methods when using the V850E model with 6 cycle data memory. We use the same scope graphs, flow facts and basic block graphs as for the runs in Figure 10.7. For some programs, like `compress` and `matmult`, many long timing effects appeared, while for others there was no impact.

More long timing effects makes the precision of the clustered WCET estimates for `compress`, `jfdctint`, `matmult` and `ns` to be a little worse than for extended IPET. Since the WCET estimates for these programs were identical

Program	IPET, Lte		IPET, no Lte		Actual
	Cycles	+%	Cycles	+%	WCET
compress	15304	+0.10	15461	+1.12	15289
crc	33917	+0.01	34427	+1.50	33914
duff	1468	0	1765	+20.31	1467
insertsort	2361	+0.21	2361	+0.21	2356
jfdctint	6392	+0.12	6626	+3.77	6385
lcdnum	238	0	268	+12.61	238
matmult	287468	+0.74	312550	+9.53	285348
ns	21349	+24.52	21350	+24.53	17145
nsichneu	135776	0	135793	+0.01	135776

Figure 10.9: Effect on WCET estimate precision of ignoring long timing effects

when using the basic V850E model, as given Figure 10.7, we conclude that the obtained pessimism is due to long timing effects across calculation boundaries.

For all our benchmarks only negative long timing effects where observed, even though the V850E CPU has a potential of exhibiting positive long timing effects [Eng02]. This means that for the tested programs, a calculation method would obtain pessimistic but safe estimates by ignoring the long timing effects.

To evaluate the effect of ignoring long timing effects, we calculated WCET estimate for programs using the slow memory and extended IPET calculation method, while ignoring all long timing effects. The result is shown in the **IPET, no Lte** column in Figure 10.9. For all programs except `insertsort`, ignoring long timing effects lead to an overestimation of the WCET estimate. For some programs, such as `duff` and `lcdnum`, the consequences of ignoring long timing effect are quite substantial. Altogether, we conclude that long timing effects must be modelled in order to generate safe and tight WCET estimates.

10.7 Computation time

In order to test the computational complexity and find potential costly "hot spots" we measured the time spent in different analysis stages for our benchmark programs. The result is shown in Figure 10.10. All times are given in seconds, obtained on a Pentium III 500 MHz processor with 256 MB RAM. Column **Load Time** shows the time required to read the input files into the tool. The column **Pipe Time** shows the time for the pipeline analysis using our V850E model. The columns **Path**, **IPET** and **Clustered** give the computation time required for each calculation method.

The time taken for the pipeline analysis is roughly linear to the program size, as measured by the time to load the code. For all our benchmarks, except `nsichneu`, the time spent in the calculation stages is almost negligible. In general, the path-based calculation method is much faster than the other two calculation methods. The computation time for the clustered calculation method is comparable to extended IPET.

Program	Load Time	Pipe Time	Calculation Time		
			Path	IPET	Clustered
compress	1.2	1.6	0.1	1.2	1.5
crc	0.3	0.7	0.03	0.17	0.22
duff	0.15	0.3	0.01	0.05	0.27
expint	0.22	0.3	0.02	0.13	0.3
fibcall	0.05	0.05	0.01	0.02	0.07
fir	0.15	0.25	0.02	0.18	0.39
insertsort	0.1	0.15	0.01	0.02	0.02
jfdctint	0.7	0.7	0.02	0.04	0.1
lcdnum	0.12	0.35	0.03	0.23	0.9
matmult	0.23	0.54	0.03	0.13	0.31
ns	0.12	0.26	0.02	0.07	0.25
nsichneu	28.0	15.5	1.07	7.9	8.7

Figure 10.10: Computation times of WCET analysis stages

Program	Facts/ign.	Scopes	VScopes	Paths	Explored	+/−
compress	9 / 0	24	27	244	39	-15.98%
crc	11 / 4	12	11	44	16	-36.36%
expint	4 / 0	7	8	25	13	-52%
fibcall	0 / 0	4	3	6	4	-33%
fir	7 / 2	5	8	34	15	-56%
insertsort	1 / 1	4	3	6	5	-17%
jfdctint	0 / 0	6	5	10	8	-20%
lcdnum	2 / 0	4	5	30	7	-77%
matmult	0 / 0	16	15	25	22	-12%
ns	1 / 1	7	7	19	11	-57.89%
nsichneu	129 / 0	3	2	3.73E97	3	≈ -100%

Figure 10.11: Complexity measures for path-based calculation

10.8 Path-based calculation evaluation

This section evaluates our path-based calculation method in more detail. The path-based calculation method is very efficient, and for most programs it is much faster than the other two calculation methods.

Figure 10.11 shows more detailed measurents for the path-based calculation method. The **Facts/ign.** column gives the number of flow facts and the number of flow facts not used by the calculation method. Flow facts ignored were total facts and facts referring to entities not located in the defining scope of the fact. This explains most of the WCET estimate precision pessimism obtained for the path-based method, as given in Figure 10.7.

The **Scopes** column lists the number of scopes required to model the program, and **VScopes** the number of virtual scopes created to account for ranged flow facts. Note that no virtual scope is created for the root scope, and thus the number of virtual scopes might be smaller than the number of original scopes. **Paths** shows the number of possible execution paths (summed over all virtual

Extra	Path-based			Ext. IPET				Clustered		
facts	time	expl.	paths	time	lptime	constr.	vars.	time	lptime	calcs
0	0.09	3	3.73E97	1.78	0.66	1651	2139	2.153	0.72	2
1	0.2	5	1.11E98	14.45	5.02	3417	4528	8.64	1.85	4
2	0.31	7	1.87E98	32.84	11.71	4931	6665	13.28	3.06	6
3	0.46	9	2.61E98	58.38	21.80	6445	8802	19.98	4.24	8
4	0.58	11	3.36E98	92.43	35.32	7959	10939	27.04	5.48	10
5	0.74	13	4.11E98	135.90	51.94	9473	13076	36.13	6.73	12
6	0.92	15	4.85E98	187.86	71.35	10987	15213	49.33	7.99	14
7	1.05	17	5.59E98	248.87	95.06	12501	17350	54.35	9.14	16
8	1.19	19	6.34E98	319.51	120.81	14015	19487	59.44	10.34	18
9	1.32	21	7.09E98	390.73	149.54	15529	21624	66.80	11.52	20
10	1.34	23	7.83E98	476.26	182.84	17043	23761	78.20	12.82	22

Figure 10.12: Scaling measures of calculation methods

scopes, before applying facts to the graph), and **Explored** the number of paths that the search actually explored. The last column shows how **Explored** relates to **Paths**.

In every case, the path-based calculation method explores only a subset of the total number of paths, and the more complex the program gets (many paths compared to the number of virtual scopes), the proportion of paths explored goes down. For `nischneu`, which contains a massive amounts of `if`-statements, the number of possible paths is very large, but only three paths are actually explored. Compared to path-based calculation methods which explicitly explores each execution path, our path-based method will execute much faster.

10.9 Scalability of calculation methods

Most of the benchmarks programs given in Figure 10.4 are quite small and do not really stress our prototype tool and the calculation methods. To evaluate how the different calculation methods scale with added flow facts and the program size, we used an altered version of the `nischneu` benchmark.

The original scope graph generated for `nsichneu` consists of three scopes (see Figure 10.5). The innermost scope is very large, containing 752 scope nodes. By adding extra dummy flow facts, spanning a particular iteration of the inner scope and not actually removing any possible execution paths, we increase the computational load. For example, adding one dummy fact will create a virtual scope graph for our extended IPET method consisting of 1508 scope nodes (752 + 752 + 4). The IPET method will create a constraint system over the whole graph while the clustered and path-based calculation methods will partition the problem into smaller subproblems. For each calculation run all WCET estimates achieved were exactly the same as reported in Figure 10.6.

Figure 10.12 gives computation times obtained for our calculation methods when adding dummy flow facts. The **Extra facts** column gives the number

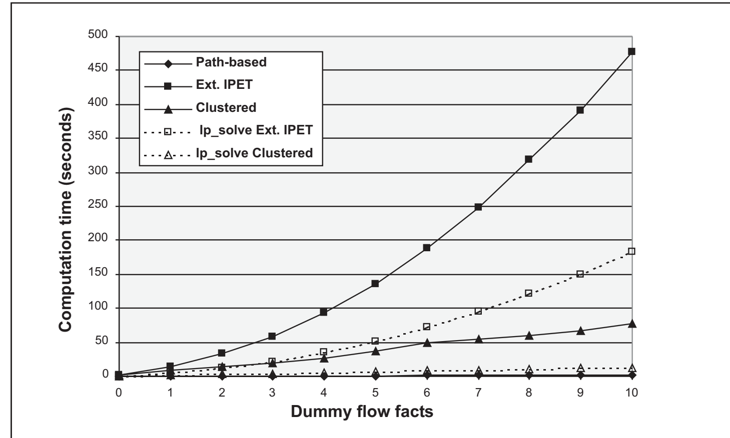


Figure 10.13: Computation time scaling

of added dummy facts. For each calculation method we give some values of interest for understanding its particular execution time properties. For the path-based calculation the computation time (**time**), the number of explored graphs (**expl.**) and the number of potential paths found in the virtual scope graph (**paths**) are given. For the Extended IPET calculation, the computation time (**time**), the time spent in the linear programming solver `lp_solve` (**lptime**), and the number of constraints (**constr.**) and variables (**vars.**) generated are given. For the clustered calculation the computation time (**time**), the number of `lp_solve` calls made and the total time spent in `lp_solve` (**lptime**) are given. All computation times are given in seconds. The computation time for Extended IPET and Clustered calculation includes the time spent in `lp_solve`.

The measurements were performed on a AMD Athlon 1800+ with 512 MB RAM. From the measurements it is clear that the path-based calculation method is always the fastest. Extended IPET is faster than the clustered calculation on the original version of `nsichneu`, but not when problem size increases.

Figure 10.13 shows computation times of each calculation method plotted against the number of added dummy flow facts. We note that the computation time seems to be linearly increasing with the problem size both for the path-based and clustered calculation, while the extended IPET has a more than linear increase. Furthermore, both the extended IPET and the clustered calculation method spend most of the calculation time in constructing graphs and generating constraint systems.

The graph also plots the time spent in `lp_solve` for the extended IPET and clustered calculation. For the extended IPET a single call to `lp_solve` is made for each calculation, with constraints and variables for the complete virtual scope graph. For the clustered calculation, the number of `lp_solve` calls increases with the number of added dummy scope, but not the size of each

Program	min-split			minimum			scope			max			global		
	cl	**call**	**time**	**cl**	**call**	**time**	**cl**	**call**	**time**	**cl**	**call**	**time**	**cl**	**call**	**time**
`compress`	6	34	0.29	4	34	0.28	4	30	0.3	1	23	0.34	1	1	0.28
`crc`	6	6	0.04	6	6	0.01	6	6	0.04	1	1	0.01	1	1	0.02
`duff`	2	12	0.04	2	12	0.04	2	12	0.03	1	4	0.02	1	1	0.01
`expint`	4	11	0.04	4	11	0.04	2	7	0.03	1	6	0.03	1	1	0.01
`fibcall`	0	4	0.01	0	4	0.01	0	4	0.01	0	4	0.01	0	1	0.01
`fir`	5	12	0.05	2	6	0.04	2	3	0.03	1	1	0.02	1	1	0.02
`insertsort`	1	1	0.01	1	1	0.01	1	1	0.01	1	1	0.01	1	1	0.01
`jfdctint`	0	5	0.02	0	5	0.02	0	5	0.02	0	5	0.02	0	1	0.01
`lcdnum`	2	23	0.13	2	23	0.12	1	19	0.11	1	21	0.12	1	1	0.03
`matmult`	0	15	0.04	0	15	0.04	0	15	0.05	0	15	0.04	0	1	0.01
`ns`	1	13	0.03	1	13	0.03	1	10	0.02	1	13	0.03	1	1	0.01
`nsichneu`	1	2	2.18	1	2	2.17	1	2	2.06	1	2	2.17	1	1	1.72

Figure 10.14: Clustered calculation measures

generated constraint system. Each call to `lp_solve` by the clustered calculation of the innermost scope contained 2156 variables and 1660 constraints and took approximately 0.65 seconds. The total computation time spent in `lp_solve` seems to grow exponentially with the problem size for extended IPET, but only linearly for the clustered calculation.

We conclude that the extended IPET has quite bad scaling properties. This could be a general problem for calculation methods relying on global ILP solvers for calculating WCET estimates. Our path-based calculation method is very efficient, only exploring a few of the total number of possible paths, and seems to scale very well. However, it should be noted that `nsichneu` only contains foreach facts, making it especially suitable for the path-based calculation. The clustered calculation is somewhere in between in complexity, scaling reasonable well, while still being able to handle complex flow information. In general, the computation time needed for each particular calculation method depends of the structure of the program, and the properties of the provided flow facts.

10.10 Clustered calculation evaluation

In Section 9.3.2 we outlined a number of alternative fact clustering algorithms. The algorithms differ in how many flow facts that will be considered together, and consequently in how large part of the scope graph that will be covered by each local WCET calculation performed. Figure 10.14 shows the effect of applying different fact cluster algorithms to our benchmarks. Columns labelled **cl** gives the number of fact clusters generated (not including empty clusters). Columns labelled **call** gives the number of local WCET calculations performed, i.e., the number of calls to `lp_solve` and columns labelled **time** gives the computation time of the calculation.

We have implemented all five clustering algorithms outlined in Section 9.3.2. Column **min-split** gives the gives values obtained when applying the split-foreach-fact minimal clustering algorithm, i.e., minimum clustering but splitting

Program	minimum, diff in-out			minimum, no diff		
	clusters	calls	WCET	clusters	calls	WCET
compress	6	38	8670	6	31	8670
crc	6	6	30271	6	6	30271
duff	2	12	1083	2	5	1083
expint	4	11	8588	4	10	8588
fibcall	0	4	313	0	3	313
fir	5	12	348095	5	12	348095
insertsort	1	1	1074	1	1	1074
jfdctint	0	5	4926	0	5	4926
lcdnum	2	23	198	2	7	198
matmult	0	15	221824	0	15	221824
ns	1	13	17353	1	8	23746
nsichneu	1	2	51116	1	2	51116

Figure 10.15: Effect of differing between flows in and out of clusters

foreach facts when the span ranges of created clusters. Column **minimum** gives values for the minimal clustering algorithm, as given in Figure 9.6. Column **scope** gives values when all facts defined on the same scope are put into the same cluster, (plus all facts in fact clusters defined in covered subscopes). Column **max** gives values when putting all facts in the same cluster, but not including non-covered scopes in the cluster. Column **global** gives values when creating a cluster that covers all scopes and all flow facts in the scope graph. This is identical to the global calculation view used by the extended IPET calculation method.

Looking at Figure 10.14, we note that the left-most clustering algorithms (**min-split** and **minimum**) generate many small clusters, and result in many local WCET calculations. At the other extreme we have the global clustering (**global**) which performs one single WCET calculation for the whole program. Scopes not covered by any fact clusters are traversed bottom-up generating one or more local WCET calculations, explaining the different number of WCET calculation calls made for programs not using any flow facts, such as `jfdctint`.

Which clustering algorithm to use depends on the structure of the program and the nature of provided flow facts. For small programs with few flow facts the global clustering (i.e., extended IPET) should preferably be used. For larger programs with many flow facts the clustered calculation should be used, as indicated by Figure 10.13.

For all benchmarks, except `jfdctint`, all clustering algorithms gave the same WCET estimates as presented for the clustered calculation in Figure 10.7. For `jfdctint` the global clustering gave a WCET estimate of 4925 clock cycles while the other clusterings gave a WCET estimate of 4926 clock cycles. The difference is due to a long timing effect reaching over the calculation borders dictated by fact clusters.

As discussed in Section 9.4.2, each fact cluster defines a fragment of the scope graph evaluated as a unit together with the flow facts. Furthermore, some graph

fragments have several points where the execution can enter or exit fragment. For such fragments we have the calculation option to either make a separate WCET calculation for each pair of entry and exit points, or to make just one WCET calculation for all entry or exit points together. The option allows us to trade WCET estimate precision for calculation time.

Figure 10.15 presents measurements performed using the minimal clustering algorithm. The **minimum, diff in-out** measurements differentiate between entry and exit points, while the **minimum, no diff** measurements do not. The **clusters** columns give the amount of fact clusters generated and is identical for both algorithms. What is more interesting is the number of local WCET estimates performed for each program, given by the **calls** columns. We note that for many programs, the number of local WCET estimates decreases quite significantly when not differentiating between entry and exit points. For all programs, except `ns`, the precision of calculated WCET estimates is the same. Program `ns` contains a non-local return from a deeply nested loop, causing an overestimation in a fashion similar to the example presented in Section 9.4.2.

The usefulness and impact of the different clustered calculation options depends on the structure of the program, properties of the added flow facts and the WCET estimate precision that is required. All measurements presented in previous sections for the clustered calculation have used minimum clustering differentiating between entry and exit points.

Chapter 11

Conclusions and Future Work

This chapter summarizes the contributions of this thesis, discusses how the work can be extended, and gives directions for future work in WCET analysis.

11.1 Summary of contributions

Estimations of the Worst-Case Execution Time (WCET) are required in providing guarantees for timing of programs used in computer controlled products and other real-time computer systems. Static WCET analysis is a way to determine the worst-case execution time of a program, using models of program flow and hardware timing for deriving WCET estimates.

Due to the complexity of embedded systems hardware and software, automated tools are essential to make static WCET analysis applicable for analysis of real systems. This thesis have covered several techniques crucial to building such automated tools.

Five main contributions to the state-of-the-art in static WCET analysis have been presented:

1) A *tool architecture* for the modularization of WCET analysis (Chapter 3). The architecture divides the WCET analysis task into the following four steps: *flow analysis*, *global low-level analysis*, *local low-level analysis* and *calculation*. To guarantee smooth flow of information between these steps, well-defined interface data structures have been designed.
2) A *program flow representation* for WCET analysis, consisting of a scope graph format and a flow fact language (Chapter 5). Additionally, a detailed *discussion* of the desired characteristics of a program flow representation is given (Chapter 4).
3) A *path-based calculation* method (Chapter 7). The method extends the

type of flow and timing information that path-based calculation methods can handle and is more efficient than previously presented methods [HAM+99, SA00, RGL02], having a computational complexity close to linear in the size of the program.

4) An *IPET-based calculation* method (Chapter 8), using integer linear programming (ILP) or constraint programming (CP) techniques for calculating a WCET estimate. Compared to previous research [OS97, PS95, FMW97, HLS00b, LM95], our method extends the power of IPET to handle new types of flow and timing information.
5) A *clustered calculation* method (Chapter 9), using flow information to divide a program into smaller parts for which local WCET calculations can be made. This is a completely novel approach which avoids the potential complexity of global analysis, while being able to produce high precision WCET estimates, since all flow information will be accounted for.

Overall, our modular tool architecture, including presented interface structures and analysis stages, aims towards building retargetable, flexible, efficient and correct static WCET analysis tools (see Chapter 3).

11.2 Evaluation

As presented in Chapter 10, we have implemented a WCET tool prototype and performed extensive evaluations to asses the correctness, precision and efficiency of our prototype, as well as to evaluate the individual analyses and calculation modules. For all our benchmark programs the prototype calculates WCET estimates that are close or even equal to the actual WCET, indicating that our techniques are capable of producing high quality WCET estimates.

For many of our benchmark programs the quality of provided flow information is a great (if not the greatest) determiner for the WCET estimation precision achievable. The results indicate that the scope graph and the flow fact language together form a representation expressive enough to capture the type of program flows found in various types of programs, most notably those including unstructured code. The results also indicate that we can perform a clearly separated flow analysis, and still obtain high quality WCET estimates.

The second determiner of the WCET estimate precision achievable is the quality of the hardware timing information. We have built hardware models for two 32-bit embedded RISC micro-controllers, the ARM9 and the V850E. We extract hardware timing for a CPU by using its corresponding CPU model in the pipeline time analysis, while the rest of the analysis modules are kept unchanged. This indicates that our modular architecture provides for easy retargetability.

The results indicate that our pipeline analysis is able to capture the hardware timing dependencies between instructions in our targeted processors, and that our timing model is expressive enough to hold the timing information extracted by the pipeline analysis. The results also tell us that we can perform hardware

timing analysis in a stage clearly separated from the calculation, and still obtain high quality WCET estimates.

We have constructed three distinct calculation methods, all based on the same input structures: a scope graph with flow facts (representing possible program flow) and a timing model (representing the hardware timing). All three calculation methods are able to handle complex flow and timing information. For all our benchmark programs, all presented calculation methods produce WCET estimates that are safe.

By selecting different calculation modules we have a possibility to trade WCET estimate precision against WCET calculation computation time. Our experiments show that the path-based calculation method is very efficient, usually only exploring a few of the total number of possible execution paths. However, the path-based method does not always produce tight WCET estimates, mainly because it can not handle all type of flow information.

The extended IPET and clustered calculation methods handle more complex types of flow information, but have potentially higher computational complexity. For large programs with a lot of flow facts, experiments indicate that the clustered calculation scales better than extended IPET. However, the clustered calculation might produce slightly less tight WCET estimates due to its inability to properly handle long timing effects over calculation borders.

Overall, the results indicate that it is possible to perform WCET analysis in a modular fashion, keeping different analysis and calculation stages clearly separated, and still producing high quality WCET estimates.

11.3 Future work in WCET analysis

This overall target for this thesis has been to outline how a general purpose WCET analysis tool should be constructed. To support this we have presented analysis and methods solving some of the problems which are encountered. However, we have not targeted all problems that should be solved in the search for the ultimate general purpose WCET tool. This section summarize what is left to do and the different problems that we believe future work on static WCET analysis should be targeting.

11.3.1 Flow analysis

High precision flow information is necessary to achieve good precision in WCET estimates. None of the currently available automatic flow analysis methods come close to managing all types of constructs found in real code, such as function pointers, recursion or loops using non-arithmetic operations. Current methods are basically limited to well-written programs with well-structured loops. It is not likely that a single type of flow analysis will provide all information. Instead a combination of several flow analysis methods will have to be used, each handling a particular part of the overall problem.

However, even if automatic flow analyses are developed, there will always be complex programs for which it is not possible to derive the intended program execution behaviour. In those cases the user should be able to manually provide extra flow information. It would also be beneficial for the overall precision if the user can aid the analysis with extra guiding information, such as bounds for variables.

Since the source code level is the semantically clearest and most easily analyzed level, the problem of mapping flow information from the source code to the (maybe heavily optimized) object code has to be solved (see Section 2.2.3). In our view, the most practical solution to this is to integrate the WCET analysis into a compiler framework. This both solves the mapping problem, and gives the analysis access to all the information about a program that is available inside the compiler.

The alternative route, chosen by many WCET researchers today, is to try to derive the flow information directly from the object code. This is certainly possible for some programs and architectures, but it is in general hard to get good precision with this method, since it is much harder to identify the variables and data objects that affect the program flow in the object code.

For future work we would like to see WCET researchers agree upon a standardized format for describing possible program flows, either at the object-code or preferably, source-code level. This would allow the possibility for WCET researchers to exchange flow analysis results, compare methods, and simplify the construction of tool frameworks.

11.3.2 Low-level analysis

Most research in the WCET analysis area has been targeting the impact of different hardware features, such as pipelines, caches, branch predictors etc. We believe that this will be important also in the future, basically because new performance enhancing features will be invented and added to CPUs, and new CPUs will constantly be released on the market. However, this unfortunately means that WCET analysis research will always be a step behind hardware design, trying to analyze the impact of whatever features that has been added to a particular CPU.

We would like WCET researchers to take a more active role in the design of real-time embedded systems, trying to show real-time practitioners how to design systems that are predictable, rather than just chasing whatever hardware designers happen to put out on the market.

For engineers building real-time systems, selecting hardware which in itself is predictable and analyzable is of key importance to ensure the ultimate reliability of the system. Performance enhancing features, such as caches and branch predictors, are often designed to optimize the *average case* execution time. This indirectly means that they will potentially introduce more execution time variability, i.e., an increase in the difference between the worst and average case execution time. These features will also make WCET estimates more pes-

simistic, since conservative assumptions will have to be made in the situations when precise analysis is not possible. The literature on WCET analysis offers some guidelines on which types of hardware features that make systems less predictable [AP01, BA01, Lun02, Eng02, HLTW03, Eng03].

CPU manufacturers should design their hardware with predictability in mind, and provide CPU manuals with sufficient detail to allow for good hardware models to be constructed. Our ability to predict the execution time of a program is critically dependent on the ability to obtain a correct and sufficient detailed model of the processor used.

11.3.3 Calculation

The calculation methods presented in this thesis gives a number of choices for trading WCET estimate precision against computation time. In general, which calculation method to use is highly dependent on what type of flow and timing information that needs to be handled and the WCET estimation precision that is required. IPET-based calculation methods are able to handle the most complicated types of flows and unstructured programs, making them the most probable choice for future WCET tools.

We believe that keeping the calculation clearly separated from the other analysis stages is crucial for producing a WCET tool allowing for easy hardware retargeting and replacement of analyses.

Parametrized WCET calculation methods might help the user getting better insight in how different parameters affect the worst-case program execution. However, the usefulness of a parameterized WCET calculation method highly depends on the ability to reduce the WCET calculation to a simple understandable parametric formula. For complex hardware, the combined effects of caches, pipelines, and unknown program make this hard.

A future trend of interest for calculation is that automatic flow analyses are likely to produce a large number of flow facts. While a human user will provide a handful of facts for a typical program, preliminary experiments with automatic flow analysis indicates that many more facts are generated [GLSB03]. In this scenario, the clustered calculation method becomes important to keep calculation times down.

11.3.4 WCET analysis tools

As discussed in Chapter 2, a lot of research effort has been spent on static WCET analysis. However, it is not likely that WCET researchers can have an impact on the development of real-life real-time systems without providing a WCET analysis tool useful for practitioners. We believe that the WCET technology is mature enough, and that the main focus for future WCET research should be on the practical development of usable tools.

We see an encouraging trend in that some WCET analysis tools are being deployed in industry. For example, the first major use of WCET tools in a

commercial setting has recently been presented by Thesing et al. [TSH+03], with very encouraging results. However, practical experience of WCET analysis in industrial settings has so far been quite limited, used in a few case studies in space [HLS00b, HLS00a] and aerospace industry [FHL+01, HLTW03]. Also, the WCET tools are still too hard-to-use to reach a broader audience. For instance, we believe that automatic flow analysis is necessary to make WCET analysis useful to other than experts.

When a broader market develops, it is possible that the main use for WCET tools will be for other purposes than scheduling and schedulability analysis, since most embedded real-time systems are not built using operating systems that really support advanced schedulability analysis. For most embedded system developers, getting some type of timing estimates would be of great value in its own.

Today, real-time system designers are in general unaware of the potential benefits of static WCET analysis. Students and system developers should therefore be educated about the benefits of WCET analysis and WCET analysis tools. One possibility for doing this would be to integrate the WCET analysis in more general system development environments, such as those available for compilers and/or real-time operating systems.

Bibliography

[ABD+95] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Fixed priority preemptive scheduling: an historical perspective. *Real-Time Systems*, 8(2/3):129–154, 1995.

[Abs03] 2003. AbsInt company homepage: `http://www.absint.com`.

[AFM+02] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: A Tools for Modelling and Implementation of Embedded Systems. In *8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), LNCS 2280*, pages 460–464, April 2002.

[AKP01] P. Atanassov, R. Kirner, and P. Puschner. Using Real Hardware to Create an Accurate Timing Model for Execution-Time Analysis. In *IEEE Real-Time Embedded Systems Workshop, held in conjunction with RTSS2001*, 2001.

[Alt96] P. Altenbernd. On the false path problem in hard real-time programs. In *Proc. of the 8th Euromicro Workshop of Real-Time Systems*, 1996.

[AM95] M. Alt and F. Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *Proc. of SAS'95, Static Analysis Symposium*. Springer-Verlag, 1995. LNCS 983.

[AMWH94] R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding Worst-Case Instruction Cache Performance. In *Proc. 15th IEEE Real-Time Systems Symposium (RTSS'94)*, pages 172–181, 1994.

[AP01] P. Atanassov and P. Puschner. Impact of DRAM Refresh on the Execution Time of Real-Time Tasks. In *Proc. of the Interantionall Workshop on Application of Reliable Computing and Communication (WARCC), held along with the IEEE 2001 Pacific Rim International Symposium on Dependable Computing (PRDC'2001)*, 2001.

[ARM00] ARM Ltd. *ARM 9TDMI Technical Reference Manual*, 2000. Document no. DDI 0180A.

[ART00] Embedded systems and the Future of Swedish IT-reserach.
URL: `http://www.artes.uu.se/reports/Embedded-IT-000427.pdf`, April 2000.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986. Generally known as the "Dragon Book".

[AT96] A. Adl-Tabatabai. *Source-Level Debugging of Global Optimized Code.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. URL: `ftp://reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-133.ps`, and as technical report CMU-CS-96-133.

[AWVW96] J. Armstrong, M. Williams, R. Virding, and C. Wikström. *Concurrent Programming in Erlang.* Prentice-Hall, 2 edition, 1996.

[BA01] M. D. Bennet and N. C. Audsley. Predictable and Efficient Virtual Addressing for Safety-Critical Real-Time Systems. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*. IEEE Computer Society Press, 2001.

[BBMP00] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-level Analysis of a Portable Java Byte Code WCET Analysis Framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications (RTCSA'00)*, pages 39–48, 2000.

[BC98] P. Bose and T. M. Conte. Performance Analysis and Its Impact on Design. *IEEE Computer*, 31(5):41–49, May 1998.

[BCP02] G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proc. 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, 2002.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. KRONOS: A Model-Checking Tool for Real-Time Systems. In *Proc. of the 10th International Conference on Computer Aided Verification*, pages 546–550. Springer-Verlag, 1998. LNCS 1427.

[Ber03] M. Berkelaar. *lp_solve: (Mixed Integer) Linear Programming Problem Solver*, 2003.
URL: `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

[BMSO+96] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding Instruction Cache Effects to Schedulability Analysis of Preemptive Real-Time Systems. In *Proc. 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 204–212. IEEE Computer Society Press, June 1996.

[Bör95] H. Börjesson. Incorporating worst-case execution time in a commercial c-compiler. Master's thesis, Department of Computer Systems, Uppsala University, 1995. DoCS MSc Thesis 95/69.

[BS98] B. Black and J. Shen. Calibration of Microprocessor Performance Models. 31(5):59–65, May 1998.

[BW95] M. R. Boyd and D. B. Whalley. Graphical visualization of compiler optimizations. *Journal or Programming Languages*, pages 69–94, 1995.

[CB02] A. Colin and G. Bernat. Scope-tree: a program representation for symbolic worst-case execution t8me analysis. In *Proc. 14th Euromicro Conference of Real-Time Systems, (ECRTS'02)*, pages 50–59, 2002.

[CBW94] R. Chapman, A. Burns, and A. Wellings. Integrated Program Proof and Worst-Case Timing Analysis of SPARK Ada. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'94)*, 1994.

[CEE+02] M. Carlsson, J. Engblom, A. Ermedahl, J. Lindblad, and B. Lisper. Worst-case Execution Time Analysis of Disable Interrupt Regions in a Commercial Real-Time Operating System. 2002.

[Cha95] R. Chapman. *Static Timing Analysis and Program Proof.* PhD thesis, Department of Computer Science, University of York, England, 1995.

[Cin97] WWW homepage for the `cinderella` system, 1997.
URL: `http://www.ee.princeton.edu/~yauli/cinderella-3.0/`.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[Cou81] P. Cousot. Semantic foundations of program analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

[Cou96] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, 1996.

[CP99] A. Colin and I. Puaut. Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System. Technical Report Publication Interne No 1277, IRISA, 1999.

[CP00] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Journal of Real-Time Systems*, 18(2/3):249–274, May 2000.

[CP01a] A. Colin and I. Puaut. A Modular and Retargetable Framework for Tree-Based WCET Analysis. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*, 2001.

[CP01b] A. Colin and I. Puaut. Worst-Case Execution Time Analysis for the RTEMS Real-Time Operating System. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*, 2001.

[CPL03] ILOG CPLEX homepage, 2003.
URL: `http://www.ilog.com/products/cplex/`.

[CRTM98] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano – A Revolution in On-Board Communications. *Volvo Technology Report*, 1:9–19, 1998.

[Das00] M. Das. Unification-based pointer analysis with directoral assignment. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 35–46, 2000.

[DBK01] R. Desikan, D. Burger, and S. Keckler. Measuring Experimental Error in Microprocessor Simulation. In *Proc. of the 28th International Symposium on Computer Architecture (ISCA 2001)*. ACM Press, 2001.

[Dot97] WWW homepage for the graphics visualizations at AT&T Research, 1997.
URL: `http://www.research.att.com/sw/tools/graphviz/`.

[EAE98] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution times analysis for optimized code. In *Proc. of the 10th Euromicro Workshop of Real-Time Systems*, pages 146–153, June 1998.

[EE99] J. Engblom and A. Ermedahl. Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, 1999.

[EE00] J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proc. 21th IEEE Real-Time Systems Symposium (RTSS'00)*, 2000.

[EES^{+}99] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Towards industry-strength worst case execution time analysis. Technical Report ASTEC 99/02, Advanced Software Technology Center (ASTEC), 1999.

[EES00] J. Engblom, A. Ermedahl, and F. Stappert. Comparing Different Worst-Case Execution Time Analysis Methods. In *Proc. of the Work-in-progress Session at the 21st Real-Time System Symposium (RTSS/WIP'00)*, December 2000.

[EES01] J. Engblom, A. Ermedahl, and F. Stappert. Validating a Worst-case Execution Time Analysis Method for an Embedded Processor. Technical report, Dept. of Information Technology, Uppsala University, Uppsala, Sweden, 2001.

[EES^{+}03] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Worst-Case Execution-Time Analysis for Embedded Real-Time Systems. *Software Tools for Technology Transfer*, 2003. Accepted for publication.

[EG97] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer Verlag, 1997.

[EH94] M. Emami and R. Ghiyaand L. Hendren. Context-sensitive interprocedural points-to analysis in the precense of function pointers. *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, June 1994.

[Ene03] Enea WWW Homepage.
URL: `http://www.enea.com`, 2003.

[Eng97] J. Engblom. Worst-case execution time analysis for optimized code. Master's thesis, Department of Computer Systems, Uppsala University, 1997. DoCS MSc Thesis 97/94.

[Eng99a] J. Engblom. Static Properties of Embedded Real-Time Programs, and Their Implications for Worst-Case Execution Time Analysis. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*. IEEE Computer Society Press, 1999.

[Eng99b] J. Engblom. Why SpecInt95 Should Not Be Used to Benchmark Embedded Systems Tools. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, May 1999.

[Eng02] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden, April 2002.

[Eng03] Jakob Engblom. Analysis of the Execution Time Unpredictability caused by Dynamic Branch Prediction. In *Proc. 8th IEEE Real-Time/Embedded Technology and Applications Symposium (RTAS'03)*, May 2003.

[ESE00] J. Engblom, F. Stappert, and A. Ermedahl. Structured Testing of Worst-Case Execution Time Analysis Tools. In *Proc. of the Work-in-progress Session at the 21st Real-Time System Symposium (RTSS/WIP'00)*, December 2000.

[ESS99] E. Erpenbach, F. Stappert, and J. Stroop. Compilation and Timing Analysis of Statecharts Models for Embedded Systems. In *Proc. 2nd International Workshop on Compiler and Architecture Support for Embedded Systems, (CASES'99)*, 1999.

[EY97] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *International Conference on Computer-Aided Design (ICCAD '97)*, 1997.

[FHL$^+$01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proc. 1st International Workshop on Embedded Systems, (EMSOFT2000), LNCS 2211*. Springer-Verlag, 2001.

[FMW97] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[Fre00] U. Fredriksson. JAS 39 Gripen - an overview: The internal network, 2000. URL: `http://www.canit.se/ griffon/aviation/gripen/gripen-network.html`.

[Gan01] J. Ganssle. Really Real-Time Systems. In *Proceedings of the Embedded Systems Conference San Fransisco (ESC SF) 2001*, 2001.

[GLSB03] J. Gustafsson, B. Lisper, C. Sandberg, and N. Bermudo. A Tool for Automatic Flow Analysis of C-programs for WCET Calculation. In *8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003), Guadalajara, Mexico*, 2003.

[GSW95] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables, detecting and classifying sequences using a demand-driven ssa form. *AVM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

[Gus94] J. Gustafsson. *Calculation of Execution Times in Object-Oriented Real-Time Software*. Licentiate Thesis, department of Machine Elements, the Royal Institute of Technology, Sweden, 1994.

[Gus00] J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Department of Computer Systems, Information Technology, Uppsala University, May 2000. DoCS Report 00/115, `www.docs.uu.se/docs/research/reports/`.

[Gwe95] L. Gwennap. New Algorithm Improves Branch Prediction. *Microprocessor Report, January 17*, 9(4), December 1995.

[Hal00] T. R. Halfhill. Embedded Market Breaks New Ground. *Microprocessor Report, January 17*, 2000.

[HAM+99] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1), 1999.

[Hav97] P. Havlak. Nesting of Reducible and Irreducible Loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, July 1997.

[Hep03] WWW homepage for the `heptane` system, 2003.
URL: `http://www.irisa.fr/aces/work/heptane-demo/heptane.html`.

[HHWT97] T. A. Henzinger, P-H. Ho, and H. Wong-Toi. HYTECH: A Model Checker for Hybrid Systems. In *Proc. of the 9th International Conference on Computer Aided Verification*, pages 460–463, 1997. LNCS 1254.

[HLS00a] N. Holsti, T. Långbacka, and S. Saarinen. Using a Worst-Case Execution-Time Tool for Real-Time Verification of the DEBIE software. In *Proceedings of the DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457)*, 2000.

[HLS00b] N. Holsti, T. Långbacka, and S. Saarinen. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, 2000.

[HLTW03] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *IEEE Proceedings on Real-Time Systems*, 2003. Accepted for publication.

[Hol97] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 2nd edition, 1996. ISBN 1-55860-329-8.

[HP00] M. Hind and A. Pioli. Which pointer analysis should i use? *ACM SIGSOFT International Symposium on software Testing and Analysis (ISSTA 200)*, August 2000.

[HS02] N. Holsti and S. Saarinen. Status of the bound-t wcet tool. In *Proc. 2nd International Workshop on Worst-Case Execution Time Analysis, (WCET'2002)*, 2002.

[HSR+00] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, May 2000.

[HSRW98] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding Loop Iterations for Timing Analysis. In *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, June 1998.

[HW99] C. Healy and D. Whalley. Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, pages 79–88, June 1999.

[I-L03] I-Logix WWW Homepage.
URL: `http://www.ilogix.com`, 2003.

[IAR99] IAR Systems. *V850 C/EC++ Compiler Programming Guide*, 1st edition, 1999.

[IAR03] IAR Systems. Reference Applications of VisualSTATE, 2003.
`http://www.iar.dk/products/references.htm`.

[Int95] Intelligent Systems Laboratory. SICStus Prolog user's manual. ISBN 91-630-3648-7, Swedish Institute of Computer Science, 1995.

[Ive98] A. Ive. Presented at the 8th Nordic Workshop on Programming Enviroment Research, August 1998.
URL: `http://www.ifi.uib.no/konf/nwper98/proceedings.html`.

[KHR+96] L. Ko, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M.G. Harmon. Supporting the specification and analysis of timing constraints. In *Proc. 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 170–178, 1996.

[Kir02] R. Kirner. The Programming Language WcetC. Technical Report 2/2002, Technische Universität Wien, Institut f'ur Technische Informatik, January 2002.

[KMH96] S.-K. Kim, S. L. Min, and R. Ha. Efficient Worst Case Timing Analysis of Data Caching. In *Proc. 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 230–240. IEEE, 1996.

[KP01] R. Kirner and P. Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*. IEEE Computer Society Press, 2001.

[KP03] R. Kirner and P. Puschner. Timing analysis of optimised code. In *8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003), Guadalajara, Mexico*, 2003.

[KWH95] L. Ko, D. Whalley, and M. Harmon. Supporting User-Friendly Analysis of Timing Constraints. *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'95)*, pages 107–115, June 1995.

[LBJ+95] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An Accurate Worst-Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, 1995.

[LEMC02] S. Lee, A. Ermedahl, S.L. Min, and N. Chang. Statistical Derivation of an Accurate Energy Consumption Model for Embedded Processors. Technical report, Dept. of Information Technology, Uppsala University, Uppsala, Sweden, 2002.

[LFW02] M. Langenbach, C. Ferdinand, and R. Wilhelm. Worst case Execution Time Prediction. In *Proc. 2nd International Workshop on Worst-Case Execution Time Analysis, (WCET'2002)*, 2002.

[LG98] Y. A. Liu and G. Gomez. Automatic accurate time-bound analysis for high-level languages. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, pages 31–40, 1998.

[LHKM98] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Multiple-Issue Machines. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98)*, 1998.

[LHS+96] C. Lee, J. Han, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling. In *Proc. 17th IEEE Real-Time Systems Symposium (RTSS'96)*, December 1996.

[LHT00] M. Lindgren, H. Hansson, and H. Thane. Using Measurements to derive the Worst-case Execution Time. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications (RTCSA'00)*, pages 15–22, 200.

[Lin02] M. Lindgren. Measurement and simulation based techniques for real-time systems analysis. Licentiate Thesis, Uppsala University Printers, Uppsala, Sweden, December 2002.

[Lis03] B. Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis. Technical report, Mälardalen Real-Time Research Centre, Mälardalen University, Sweden, April 2003. MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-97/2003-1-SE.

[LKM98] S-S. Lim, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Optimized Programs. In *Proc. of the fifth International Conference on Real-Time Computing Systems and Applications (RTCSA); Hiroshima, Japan*, pages 151–157, October 1998.

[LM95] Y-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proc. of the 32:nd Design Automation Conference*, pages 456–461, 1995.

[LMW96] Y-T. S. Li, S. Malik, and A. Wolfe. Cache Modelling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proc. 17th IEEE Real-Time Systems Symposium (RTSS'96)*, pages 254–263. IEEE Computer Society Press, 1996.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell,. *Springer International Journal of Software Tools for Technology Transfer, (STTT), 1(1-2)*, pages 134–152, December 1997.

[LS98] T. Lundqvist and P. Stenström. Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, June 1998.

[LS99] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. Technical Report 99-5, Chalmers University of Technology, 1999.

[LS00] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Real-Time Systems*, May 2000.

[Lun02] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 2002.

[Mar99] F. Martin. Experimental comparison of call string and functional approaches to interprocedural analysis. In *Computational Complexity*, pages 63–75, 1999.

[Mel98] K. Melin. Volvo S80: Electrical system of the future. *Volvo Technology Report*, 1:3–7, 1998.

[Mon00] S. Montán. Validation of Cycle-Accurate CPU Simulator against Actual Hardware. Master's thesis, Dept. of Information Technology, Uppsala University, 2000. Technical Report 2001-007, `http://www.it.uu.se/research/reports/2001-007/`.

[MR01] T. Mitra and A. Roychoudhury. Effects of Branch Prediction on Worst Case Execution Time of Programs. Technical Report 11-01, National University of Singapore (NUS), 2001.

[MR02] T. Mitra and A. Roychoudhury. A framework to model branch prediction for wcet analysis. In *Proc. 2nd International Workshop on Worst-Case Execution Time Analysis, (WCET'2002)*, 2002.

[MS00] E. Martins and J. Santos. A New Shortest Paths Ranking Algorithm. *Investigacao Operational*, 20(1):47–62, 2000.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997. ISBN: 1-55860-320-4.

[Mül97] F. Müller. Timing Predictions for Multi-Level Caches. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, pages 29–36, June 1997.

[NEC99] NEC Corporation. *V850E/MS1 32/16-bit Single Chip Microcontroller: Architecture*, 3rd edition, 1999. Document no. U12197EJ3V0UM00.

[OS97] G. Ottosson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

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

[Pat95] J. Patterson. Accurate Static Branch Prediction by Value Range Propagation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, June 1995.

[PD02] I. Puaut and D. Decotigny. Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. Proc. of the 23rd IEEE International Real-Time Systems Symposium, Austin, TX, USA, December 2002. . In *Proc. 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, 2002.

[Pet00] S. Petters. Bounding the Execution Time of Real-Time Tasks on Modern Processors. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications (RTCSA'00)*. IEEE Computer Society Press, 2000.

[PF99] S. Petters and G. Färber. Making Worst-Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, 1999.

[PH98] P. Persson and G. Hedin. Interactive execution time predictions using reference attributed grammars. In *Proc. of the 2:nd Workshop on Attribute Grammars and their Applications (WAGA'99), Amsterdam, Netherlands*, pages 173–184, August 1998. URL: `http://www.dna.lth.se/home/Patrik_Persson`.

[PK89] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *The Journal of Real-Time Systems*, 1(1):159–176, 1989.

[PPVZ92] G. Pospischil, P. Puschner, A. Vrchoticky, and R. Zainlinger. Developing Real-Time Tasks With Predictable Timing. *IEEE Software*, 9(5), September 1992.

[PS90] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on a source-level timing schema. In *Proc. 11th IEEE Real-Time Systems Symposium (RTSS'90)*, pages 72–81, 1990.

[PS95] P. Puschner and A. Schedl. Computing Maximum Task Execution Times with Linear Programming Techniques. Technical report, Technische Universität Wien, Institut für Technische Informatik, 1995.

[Pus94] P. Puschner. *Zeitanalyze von Echtzeitprogrammmen*. PhD thesis, Technische Universität, Institut für Technische Informatik, Vienna, Austria, 1994.

[Ram00] G. Ramalingam. On Loops, Dominators, and Dominance Frontier. *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

[Rat03] Rational. UML Reseource Center, 2003.
`http://www.rational.com/uml/`.

[Ray00] E. Raymond. The jargon file, version 4.2.0.
URL: `http://www.tuxedo.org/~esr/jargon/html/index.html`, 2000.

[Rea03] Realogy Real-Time Architect Product Brochure.
URL: `http://www.realogy.com`, 2003.

[RGL02] D. Renaux, J. Góes, and R. Linhares. WCET Estimation from Object Code implemented in the PERF Environment. In *Proc. 2nd International Workshop on Worst-Case Execution Time Analysis, (WCET'2002)*, 2002.

[RK02] G. Freiberger P. Puschner R. Kirner, R. Lang. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In *Proc. 14th Euromicro Conference of Real-Time Systems, (ECRTS'02)*, pages 31–40, June 2002.

[SA00] F. Stappert and P. Altenbernd. Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[Sch00] J. Schneider. Cache and Pipeline Sensitive Fixed Priority Scheduling for Preemptive Real-Time Systems. In *Proc. 21th IEEE Real-Time Systems Symposium (RTSS'00)*, pages 195–204, November 2000.

[SEE01] F. Stappert, A. Ermedahl, and J. Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. Technical Report Report 2001-012, Dept. of Information Technology, Uppsala University, 2001.

[SF99] J. Schneider and C. Ferdinand. Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*. ACM Press, May 1999.

[SKO$^+$96] V. Seppänen, A-M Kähkönen, M. Oivo, H. Perunka, P. Isomursu, and P. Pulli. Strategic needs and future trends of embedded software. Technical Report Technology Review 48/96, TEKES Technology Development Center, Oulu, Finland, 1996.

[Sta97] F. Stappert. Predicting pipelining and caching behaviour of hard real-time programs. In *Proc. of the 9th Euromicro Workshop of Real-Time Systems*, 1997.

[Ste93] B. Steensgard. Sequentializing Program Dependence Graphs for Irreducible Pprograms. Technical report, Microsoft Reserach, Redmond, Wash, October 1993. Technical Report: MSR-TR-93-14.

[Ste96] B. Steensgard. Points-to analysis in almost linear time. *Proc. in 23^{r}d Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, january 1996.

[Ste01] D. B. Stewart. Introduction to real time, 2001.
URL: `http://www.embedded.com/story/OEG20011016S0120`.

[Ste02] D. B. Stewart. Measuring Execution Time and Real-Time Performance. In *Proceedings of the Embedded Systems Conference (ESC SF) 2002*, March 2002.

[Tel03] Telelogic WWW Homepage.
URL: `http://www.telelogic.com`, 2003.

[TF98] H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98)*, 1998.

[The01a] H. Theiling. Generating Decision Trees for Decoding Binaries. 2001.

[The01b] H. Theling. Generating Decison Trees for Decoding Binaries. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'01)*, pages 112–120, 2001.

[The02] H. Theiling. ILP-Based Interprocedural Path Analysis. In *Proc. 2nd International Workshop on Embedded Systems, (EMSOFT2001)*, pages 349–363. Springer-Verlag, 2002.

[TSH$^+$03] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretetation-Base Timing Validation of Hard Real-Time Avionics Software. In *Proc. of the IEEE International Conference on Dependable Systems and Networks (DSN-2003)*, 2003.

[Tur02] J. Turley. Embedded processors. In Extremetech.com, January 2002.
URL: `http://www.extremetech.com/article2/0,3973,18917,00.asp`.

[VHMW01] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'01)*, pages 88–93, June 2001.

[Vrc94] A. Vrchoticky. *The Basis for Static Execution Time Prediction*. PhD thesis, Institut für Technische Informatik, Technische Universität Wien, Treitlstraße 3/182.1, A-1040 Wien, Austria, 1994.

[VSL96] G.R. Gao V.C. Sreedhar and Y-F. Lee. Identifying Loops using DJ Graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, November 1996.

[WE01] F. Wolf and R. Ernst. Execution Cost Interval Refinement in Static Software Analysis. *Journal of Systems Architecture, The EUROMICRO Journal, Special Issue on Modern Methods and Tools in Digital System Design*, 47(3-4):339–356, 2001.

[Whi97] R. T. White. *Bounding Worst-Case Data Cache Performance*. PhD thesis, Florida State University, 1997.

[Wis94] R. Wismüller. Debugging of globally optimized programs using data flow analysis. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, pages 278–289. ACM, 1994.
URL: `http://wwwbode.informatik.tu-muenchen.de/~wismuell/`.

[WMH+97] R. White, F. Müller, C. Healy, D. Whalley, and M. Harmon. Timing Analysis for Data Caches and Set-Associative Caches. In *Proc. 3rd IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pages 192–202, 1997.

[YP93] T.Y. Yeh and Y.N. Patt. A Comparasion of Dynamic Branch Predictors that use Two Levels of Branch History. In *Proc. of the 20th International Symposium on Computer Architecture (ISCA 1993)*, pages 257–266, May 1993.

[Zel84] P. Zellweger. *Interactions between high-level debugging and optimised code*. PhD thesis, Computer Science Division, University of California, Berkeley, 1984. Published as Xerox PARC Technical Report CSL-84-5.

[ZWR+01] D. Ziegenbein, F. Wolf, K. Richter, M. Jersak, and R. Ernst. Interval-based analysis of software processes. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'01)*, June 2001.