Approximating the Worst-Case Execution of Soft Real-Time Applications

Matteo Corti

Doctoral Thesis ETH No. 15927

Approximating the Worst-Case Execution of Soft Real-Time Applications

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH ZÜRICH)

for the degree of
Doctor of Technical Sciences

presented by
Matteo Corti
Dipl. Informatik-Ing. ETH
born September 12, 1974
citizen of Aranno, TI

accepted on the recommendation of
Prof. Dr. Thomas Gross, examiner
Prof. Dr. Peter Puschner, co-examiner

2005

It is the mark of an instructed mind to rest satisfied with the degree of precision which the nature of the subject admits and not to seek exactness when only an approximation of the truth is possible.

– Aristotle

Abstract

The soundness of real-time systems not only depends on the exactness of the results but also on their delivery according to given timing constraints. Real-time schedulers need an estimation of the worst-case execution time (i.e., the longest possible run time) of each process to guarantee that all the deadlines will be met. In addition to be necessary to guarantee the correctness of soft- and hard-real-time systems, the worst-case execution time is also useful in multimedia systems where it can be used to better distribute the computing resources among the processes by adapting the quality of service to the current system load.

This dissertation presents a set of static compile-time analyses to estimate the worst-case execution time of Java processes focusing on the approximation of the WCET of fairly large soft-real-time applications on modern hardware systems.

In a first phase the program is analyzed to extract semantic information and to determine the maximal (and minimal) number of iterations for each basic block, by possibly bounding every cyclic structure in the program. As a first step, the semantic analyzer, embedded in an ahead-of-time bytecode-to-native Java compiler, performs an abstract interpretation pass of linear code segments delimiting the set of possible values of local variables at a given program point and discovering a subset of the program's infeasible paths. This step is followed by a loop-bounding algorithm based on local induction variable analysis that, combined with structural analysis, is able to deliver fine-grained per-block bounds on the minimum and maximum number of iterations. To resolve the target of method calls the compiler performs a variable type based analysis reducing the call-graph size.

In a second phase the duration of the different program's instructions or basic blocks is computed with respect to the used hardware platform and the computational context where the instructions are executed. Modern systems with preemption and modern architectures with non-constant instruction duration (due to pipelining, branch prediction and different level of caches) hinder a fast and precise computation of a program's WCET. Instead of simulating the CPU behavior on all the possible paths we apply the principle of locality, limiting the effects of a given instruction to a restricted period in time and allowing us to analyze large applications in asymptotically linear time.

We describe the effectiveness in approximating the worst-case execution time for a number of programs from small kernels and soft-real-time applications.

Riassunto

La correttezza dei sistemi in tempo reale non dipende unicamente dall'esattezza dei risultati ma anche dalla consegna dei risultati entro dei limiti temporali prestabiliti. Per garantire che tutte le scadenze siano rispettate, gli scheduler in tempo reale necessitano della stima del tempo massimo di esecuzione (*worst-case execution time*) di ogni processo. Il tempo massimo di esecuzione, oltre a essere necessario per garantire la correttezza dei sistemi in tempo reale (*soft-* e *hard real-time*), è anche utile in sistemi multimediali, dove può essere usato per meglio distribuire le risorse tra i processi, adattando la qualità del servizio offerto, al carico momentaneo del sistema.

Questa dissertazione presenta una serie di analisi statiche eseguite durante la compilazione per stimare la durata massima di esecuzione di processi Java concentrandosi sulla approsimazione del tempo di esecuzione di applicazioni in tempo reale sufficientemente grandi su piattaforme hardware moderne.

In una prima fase, la semantica del programma è analizzata in modo da determinare il numero massimo (e minimo) di iterazioni per ogni blocco trovando il limite del numero di iterazioni delle strutture cicliche. Questa analisi, implementata come modulo di un compilatore Java *ahead-of-time* (da bytecode a codice nativo), esegue un'interpretazione astratta di segmenti lineari di codice delimitando i possibili valori che le variabili locali possono assumere in un determinato punto nel programma, e scoprendo un sottoinsieme dei cammini impossibili nel grafo del controllo di flusso. In seguito, usando un algoritmo per la delimitazione del numero di iterazioni dei cicli, basato sull'analisi delle variabili induttive combinato con un'analisi strutturale del programma, ricaviamo dei limiti accurati sul numero di iterazioni a livello dei singoli blocchi. Per definire le possibili destinazioni delle chiamate dei metodi il compilatore esegue un analisi basata sui tipi delle variabili, in grado di ridurre la dimensione del grafo delle chiamate.

In una seconda fase, viene calcolata la durata delle singole istruzioni (o dei singoli blocchi) in relazione alla piattaforma hardware usata e al contesto computazionale nel quale le istruzioni sono eseguite. La presenza di diritto di prelazione e la durata non costante delle varie istruzioni sulle architetture moderne (dovuta alla presenza di pipeline, predizione delle diramazioni e diversi livelli di cache) impedisce un calcolo veloce e preciso del tempo massimo di esecuzione di un programma. Invece di simulare il comportamento del processore su tutti i possibili cammini applichiamo il principio di località, limitando gli effetti di una data istruzione a un periodo definito di tempo, permettendoci di analizzare grosse applicazioni in un tempo asintoticamente lineare.

Descriviamo infine l'efficacia dell'approssimazione del tempo massimo di esecuzione per una serie di programmi: da piccoli benchmark ad applicazioni *soft-real-time*.

Acknowledgments

This thesis would have not been possible without the support of many persons. First of all, I want to thank my advisor Thomas Gross who gave me the opportunity to work with him and who guided me during my research. He encouraged me during the years at ETH and taught me how to engage in independent research by being a continuous source of support, encouragement, and example.

I would also to thank Peter Puschner for agreeing to be my co-examiner and for his helpful comments and suggestions.

I am grateful to Roberto Brega who inspired this work by convincing me to work on the XO/2 embedded system and investigate the worst-case execution time estimation problem.

My warmest thanks to my parents, Bruna and Luciano Corti, who supported me financially and morally during my studies and to my brother Paolo Corti for his continuous support and encouragement.

It's a pleasure to thank Cristoph von Praun and Florian Schneider for their contributions and commitment to the Java bytecode-to-native compiler. Our common infrastructure, shared among various projects, helped me to discuss several ideas and problems, and to find synergies among different aspects of compiler construction.

A number of students contributed with term projects and master thesis to both the compiler and worst-case execution time estimations tools. Daniel Augsburger, Bernhard Egger, Damon Fenacci, Stephen Jones, Andrea Rezzonico, Daniel Steiner and Marco Terzer helped to improve and enhance the compiler. Fabiano Ghisla, Felix Pfrunder, Robert Simons, and Michael Szediwy worked on the hardware-level analysis and finally Nathalie Kellenberger and Stefan Heimann who worked on the evaluation infrastructure.

The friendly and collaborative environment at the Computer Systems Institute and especially in Thomas Gross' group made my time at ETH an enjoyable and interesting experience. I am therefore extremely thankful to all my past and present colleagues: Cristian Tuduce, my office mate for many years, Jürg Bolliger, Susanne Cech, Irina Chihaia, Hans Domjan, Roger Karrer, Peter Muller, Valery Naumov, Yang Su, Patrik Reali, and Oliver Trachsel.

A special thanks goes to Andrea Lombardoni, Andreas Hubeli and Luca Previtali, our daily coffee and lunch breaks in addition to be pleasant moments were a constant source of interesting discussions.

A final thanks to Patrick Schaufelberger and Robertino Bevacqua, my home mates during my stay in Zurich, and to the many friends for the encouragement and support in these past years.

Contents

1 Introduction

1.1 Worst-case execution time

A real-time system is a system whose correctness does not only depend on the soundness of the results (functional correctness) but also on the timeliness of their delivery. We normally distinguish between *hard-real-time systems*, where the each task has to finish before its deadline under any condition (e.g., high load and possible faults) and *soft-real-time systems*, where the system is occasionally allowed to miss a deadline. Soft-real-time systems, in contrast to hard-real-time systems, generally have longer response time requirements and they are asynchronous with the state of the environment. The driving characteristic in the design of a soft-real-time system is the average performance and no catastrophe occurs if a single deadline is missed. In hard-real-time systems, on the other hand, the focus is not on the overall system efficiency but on the deadlines of the single tasks which have to be respected for the system to be in synchrony with the environment (e.g., external sensors or actuators).

A hard-real-time scheduler has therefore to guarantee that all the concurrent tasks will meet the specified deadlines by distributing limited system resources. Hard-real-time schedulers usually schedule processes according to their fixed (e.g., rate monotonic schedulers [64] and deadline monotonic schedulers [58]) or dynamic priorities (e.g., earliest deadline first schedulers [64]). The reader is referred to [17] for a comprehensive introduction to hard-real-time systems and scheduling algorithms. If the worst-case execution time (WCET), which corresponds to the longest possible run time, and the deadline of each process is known, these schedulers, thanks to on-line or off-line admission testing, are able to guarantee that all the deadlines will be met [90, 64].

Soft-real-time systems instead are allowed to miss a deadline from time to time without catastrophic consequences. They also often relax the requirements for the scheduled applications, as the specification of a precise and known tight WCET (e.g., by allowing some imprecision), to accommodate dynamic and aperiodic tasks [37, 28, 52]. In these systems the WCET, in addition to be used for admission control, can also be used to adjust the quality of service by dynamically changing the type and quantity of running processes based on their longest possible execution time. As an example, a multimedia server could adjust the quality of the delivered streams, and therefore the CPU load, by a careful compile-time analysis of the encoding and decoding algorithms and their timing requirements.

In both hard- and soft-real-time systems, if the WCET of a process is overestimated the system will reserve more resources then needed and will be underutilized but the safety of the system will be guaranteed. If, instead, the WCET is underestimated, some processes will not be able to meet their deadlines resulting in a likely system crash, especially for hard-real-time systems.

For these reasons, before accepting a task the system performs a schedulability test (or analysis) to determine if a set of processes can be scheduled, ensuring that every task will meet its deadline. An exact schedulability test is an NP complete problem [35] and the systems usually performs a sufficient schedulability test (which could return a negative result for some acceptable configurations). The possibility to perform a correct schedulability analysis depends on the precision of its inputs: While the deadline and the starting time of a process depend on the system design its computation time depends solely on its implementation. The importance of the WCET analysis of real-time processes is therefore clear since the correctness of the schedulability analyses depends on it.

The simplest and most common way to estimate the WCET of an application is by experimentation: The WCET corresponds to the highest measured execution time of several runs of the analyzed process with different input sets (the problem in this case consist in choosing the inputs that will generate the longest program duration). Although many systems used to estimate the WCET in this way this solution is clearly not always satisfactory since it is possible that some other input will generate a higher running time. For this reason, despite the fact that a general automated worst-case execution time analysis is impossible due to the undecidability of the halting problem, the efforts in automatic WCET computation have rapidly grown in the past years attracting many research groups [85, 95, 84, 40, 31, 44]. These efforts differ in the programming language that is supported, the restrictions that are imposed on programs admitted for real-time scheduling, the (real-time) runtime system that controls the execution, the hardware platforms that are used and the amount of approximation that is allowed (there are even approaches that use probabilistic models to estimate the WCET [5]). But all of them try to compute or estimate the WCET, while reducing the need of programmers intervention.

The WCET of a process usually depends on two main aspects: the semantic of the program, which includes aspects as the maximum number of loop iterations and the procedure or method call sequences, and the actual time needed to execute the instructions on the given hardware. The first aspect, the semantic analysis, deals with the meaning of the code, trying to understand the role of the expressions and structures in the execution of the program with particular attention to cyclic structures that can greatly influence the duration of the process (see Section 1.2). After all the possible execution sequences have been computed, an hardware-level analysis computes the maximum duration of the program on the given hardware (see Section 1.3). Section 1.4 describes our thesis and summarizes the contributions of this work.

1.2 Semantic analysis

Real-time, or predictable programs, must have an upper bound on their duration. This implies that every loop has a bound on the maximum number of iterations and that there is a bound on the depth of recursive procedure calls.

The simplest way for a WCET analyzer to compute these bounds is to require the users to manually specify them in the source code with special language constructs, but these annotations require expert programmers—which is not always the case for embedded systems developers—since complex control sequences could obfuscate some nonfrequent paths. But the main problem of annotations lies in their correctness and their maintenance: the compiler and the system cannot guarantee that the programmer made no mistake specifying the bounds, and moreover every time the code is changed the affected annotations have to be updated. These issues are aggravated by the fact that, due to the increasing speed of embedded systems, real-time programs are getting larger and more and more complex, hindering a simple manual analysis of the code (e.g., modern programs make an extensive use of dynamic structures and dynamic method calls).

For this reasons in the past years, the efforts in WCET analysis automatization tried to build tools able to compute or approximate a process's longest execution time at compile time by, for example, bounding the maximum number of loop iterations or by detecting the presence of infeasible paths.

1.2.1 Goals

The main goal of (automatic) WCET analysis is therefore to find a bound on the cyclic structures of the program. All the loops must have a finite and known maximum number of iterations, and for each recursive call sequence there must be a finite and known maximum recursion depth.

These bounds must be as small as possible to avoid wasting runtime resources: If the WCET is overestimated the scheduler will allocate more time than needed for the application.

In addition, to further reduce the WCET estimation, a WCET analyzer normally tries to identify false or infeasible paths (paths that cannot be executed with any input, see Section 2.3.5), which are then excluded from the set of possible program traces.

1.2.2 Manual annotations and special purpose languages

The first automatic attempts to compute the longest possible duration of real-time processes made use of programmer's annotations and can be divided in two major groups: special purpose languages and extensions to common general purpose languages.

Special purpose languages

Instead of designing tools to compute the WCET for real-time programs developed with general purpose languages, several groups designed new languages focusing on predictability and timing.

One example is Real-Time Euclid [55, 95] which restricts the control-flow by prohibiting recursion and by allowing only for-like loops with fixed bounds. Another example are synchronous programming languages such as Esterel [6]. Synchronous languages take a different approach: they are designed as reactive systems waiting for events and acting in a deterministically determined amount of time (event handling is divided in a finite number of steps).

These languages, having strict timing limitations, allow a precise and simple analysis and provide a nice and elegant solution to the WCET computation of hard-real time programs.

These languages are very useful for hard-real-time projects as they either have strict requirements and feature limitations or they enforce a particular style of programming directly related to the timing constraints.

This, on the other hand, makes them not ideal for soft-real-time programming or to multimedia environments where the WCET could be of interest but where timing constraints are not the driving application design factor.

Annotations for general purpose languages

A different approach is the modification of existing programming languages to ease the WCET analysis or to enforce a certain timing behavior. In this case some language constructs are introduced to allow the user to annotate the program and manually specify iteration bounds.

One example, *object-code analysis*, is given by the work of Aloysius Mok and his group: they introduced a technique to automatically produce template annotations for C code which are then edited by the programmer to specify the duration of loops [70]. Another example is the *Timing Schema* [92] which uses, as annotations, intervals of iterations in a set of formulas derived by a recursive traversal of the program. The method was later extended by the *information description language* (IDL), which allows to specify path constraints identifying infeasible paths [78].

The first approach presented by the MAintainable Real-time System (MARS) project from the Vienna University of Technology [82] is a further extension to the annotations for general purpose languages including the handling of incorrect specifications. The method handles MARS-C programs, a C dialect that supports timing annotations and constraints. The timing schema was also successfully adapted to an object-oriented language (RealTimeTalk) supporting polymorphism, inheritance, and message passing [39].

Some of these methods use timing trees [79, 103], a data structure created to decouple the semantic information from the source. These syntax trees contain all the relevant timing information and can be used by all the tools involved in the WCET analysis as they work as a mapping from the source to the timing properties of the code. The information contained in timing trees can also be adapted to support simple code changes performed by an optimizing compiler.

Other approaches, as a graph-based technique from the MARS project [84] or the Cindarella method [87], use integer linear programming (ILP) to solve sets of user specified inequalities that specify the number of loop iterations (additional inequalities can be introduced to model infeasible paths). Both approaches present an elegant formulation and are based on clear and well-known mathematical foundations.

But although all these methods can produce excellent results with correct annotations, they are very prone to errors since the responsibility for the correctness of the annotations lies in the hands of the developers. Specified constraints can always be checked at run-time but again there is no guarantee that eventual faults will be discovered.

This problem motivated the growth of tools to automatically compute semantic annotations to minimize the needed user intervention. These tools can be broadly characterized in two major groups by the type of analysis they are based on: tools based on abstract interpretation (see Section 1.2.3) and tools based on data-flow analyses (see Section 1.2.3).

1.2.3 Abstract interpretation

Abstract interpretation [23] is a technique to automatically analyze programs by executing them in an abstract domain keeping track of semantic information. This method can be used to compute the WCET of a program automatically, without user annotations, by keeping track of all the possible values for each variable on every path.

Although this approach can, in theory, deliver perfect results, the number of possible paths grows exponentially for every conditional jump and every loop iteration. To avoid the explosion of paths that have to be considered (keep in mind that to have an exact knowledge of the program we should store information for each possible path) the intermediate data is merged at certain program points (such as the end of loops) to safely approximate the results.

This approach was successfully implemented by Jan Gustafsson for RealTimeTalk [40], an object-oriented language for distributed hard real-time systems and used by the group of Reinhard Wilhelm for their WCET computation framework [31].

Section 2.3 presents a more in-depth description of abstract interpretation in relation to this thesis.

1.2.4 Data-flow analysis

To avoid the twin problems of path explosion and information loss of the approaches based on abstract interpretation (see Section 1.2.3), David Whalley and his group at the Florida State University presented a technique to bound loops using only static information on the local variables involved in loop termination [42, 44, 43]. Loops are therefore analyzed in isolation looking at the conditional branches that could provoke an exit from the loop, at the variables used in the conditional checks, and at how these variables are changed in the loop.

This method, although less general than approaches using abstract interpretation, is able to handle a large number of loops found in real-time programs in a quick and simple way (the running time is in the asymptotic case quadratic in the number of branches).

Section 2.2 shows how this method works in detail and presents how we enhanced it to precisely compute bounds for every program's block.

1.3 Hardware analysis

The second important issue in WCET computation, after the analysis of the program's semantics, is the estimation of the time needed to execute the different instructions on the given hardware.

On simple processors where the duration of the execution of each instruction is known and constant, this task is trivial: The maximal number of iterations for each basic block is known from the semantic analysis, and duration of each block corresponds to the sum of each instruction's duration. But with the introduction of modern pipelined processors in the embedded world, the computation of the instruction duration became more complex. Because of pipelines, caches and branch prediction, the duration of an instruction is not constant but depends on the previously executed instructions making the computation of a tight WCET a hard problem.

A very conservative solution (not considering timing anomalies [66]), for the instruction duration computation, would be to consider each memory access a cache miss, every branch as wrongly predicted, and each instruction in the pipeline as dependent an all the previous ones. The computed WCET would be formally correct but orders of magnitude higher than any practical value, leading a severe and unacceptable waste of resources (e.g., a cache miss normally requires several orders of magnitude more time than a cache hit). The goal of a good hardware WCET estimator is therefore to conservatively minimize these pessimistic assumptions by a careful code and architecture analysis.

Similarly to the semantic analysis there are two main groups of approaches to estimate the instruction duration on the hardware level: it is possible to execute the assembler code in an abstract domain keeping track of the CPU state [31], or it is possible to deduce the CPU behavior from sophisticated data-flow analyses.

If the program is small and only contains simple for-like loops it is possible to simulate the cyclic structures as an atomic block: the maximum number of iterations is easily computable and it is possible to simulate (or measure) the block's WCET since there longest path is implicitly known [94]. As the complexity of the analyzed programs grows this approach suffers from the fact that loops cannot be considered as independent blocks with a fix duration but their execution time could depend on the program or method input.

Since abstract interpretation suffers from the same problems that are present in the semantic analysis (see Section 1.2.3), several research groups analyze the pipeline status with a data-flow method to determine the WCET [41, 60, 63, 107]; these approaches can be extended to the analysis of the instruction cache, since the behavior of this unit is, as the pipeline, independent from the system's memory allocation [41, 60, 98]. Data caches, instead, pose an additional problem since memory allocation does not only depend on the analyzed program but also on the underlying system and input set [53, 60].

The difficulties in computing the instruction duration are worsened by modern embedded operating systems which support preemption [48, 37] and the dynamic loading and unloading of processes [15, 89, 77]. The possible interferences caused by unpredictable context switches make a precise static analysis of the hardware behavior almost impossible. The phenomenon is even more significant in soft-real-time environments where the analyzed tasks often have to share the system and hardware with other non-real-time processes. Techniques exist to take into account the effects of fixed-priority preemptive scheduling on caches [57, 16], but the nature and flexibility of dynamic systems prevents an off-line analysis with acceptable results.

Another possible approach to the data cache problem focuses on the active management of the caches: If we can prevent the invalidation of the cache content, then execution time of memory accesses is predictable. One option is to partition the cache [54, 61] so that one or more partitions of the cache are dedicated to each real-time task. These approaches help the analysis of the cache access time since the influence of multitasking is eliminated, but on the other hand, cache partitioning limits the system's performance [62].

1.4 Thesis statement

All the approaches in Sections 1.2 and 1.3 are pragmatic solution for the computation or estimation of the worst-case execution time addressing various application domains.

The WCET of a program is an important concept that is not only useful in hard-real-time scheduling analysis—to ensure that every process will have enough resources—but can be applied in soft-real-time and in adaptive systems guaranteeing different quality of service. An example could be a multimedia system where the encoders or decoders of the live streams are chosen depending on the available timing resources.

In this dissertation, I show that

> *"Compile-time semantic analysis can detect loop bounds and gather control-flow information to delimit the longest execution path of Java programs. It is also possible to delimit the size of the call graph of programs using method ovverriding by a whole programs worst-case execution time analysis of object-oriented code.*
>
> *The worst-case execution time can be approximated on modern execution platforms combining compile time analyzes and run-time information.*
>
> *It is possible to perform the WCET estimation of large programs without relying on path enumeration."*

The dissertation establishes this thesis statement as follows:

1. It presents a set of fast compile time analyses to bound the maximum number of iterations for each source language semantic construct. These algorithms present a linear or quadratic asymptotic behavior (on the size of the source code) allowing us to handle large applications.

   It presents an implementation of the semantic analyzer for Java programs embedded in a general purpose bytecode-to-native ahead of time compiler.

2. It shows how to approximate the instruction duration on modern complex processors with several levels of caches, pipelines and branch prediction in asymptotically linear time.

   It describes two different implementations, one for the XO/2 system on PowerPC and the other for Linux running on Intel processors, of a hardware-level WCET analyzer which makes use of some run-time data to approximate the duration of the analyzed program.

3. Using various synthetic benchmarks as well as some real applications the evaluation shows that it is possible to compute WCET approximations in an unpredictable scheduling environment.

1.5 Road-map for this dissertation

This dissertation is organized as follows: Chapter 2 describes the semantic analysis performed by our WCET approximation technique. We present a local analysis which automatically bounds the number of loop iterations, reduces the call graph by reducing the set of targets for dynamic method calls and detects a subset of the false paths present in the program. Chapter 3 describes the analysis on the hardware level showing how we approximate the instruction durations without performing path enumeration. Chapter 4 discusses implementation issues related to the chosen programming language (Java) and the used hardware platform (PowerPC and Intel). Chapter 5 evaluates the results obtained testing benchmarks and real applications while Chapter 6 addresses possibilities for further research and concludes this dissertation.

2 Semantic Analysis

The semantics of a program are a crucial point that must be understood to bound the program's worst-case execution time: In particular a WCET analyzer must identify every cyclic semantic structure of the control-flow graph and find a bound on the maximum number of iterations. Loops and recursive subroutine calls are therefore the main focus of an automatic WCET semantic analysis (see Section 2.2).

In addition to identify loop bounds, particular attention is usually paid to infeasible (also called false) paths. These are part of control-flow paths that are impossible for every potential execution. These paths can be then excluded from the WCET computation since they will never be possibly executed (see Section 2.3).

Object-oriented languages, with dynamic method calls, pose an additional difficulty: For every dynamic call site, there are many different possible jump targets, and it is therefore essential to perform a precise type analysis to minimize the set of possible targets reducing an eventual WCET overestimation (see Section 2.8).

This chapter describes how we extract the semantic information from single threaded Java programs and compute a safe bound on the maximum number of iterations for each program's statement.

2.1 Structural analysis

Our WCET analyzer is implemented as a module in a bytecode to native compiler. Working with the class bytecodes as input has several advantages as it provides the ability to work with several compilers and even different languages (e.g., there exist Java bytecode backends for Lisp, Eiffel, SmallTalk and Ada). Java bytecode, in addition to normal object-code binaries, has also the advantage that it retains some useful information about the program's symbols such as variables and types. The drawback is that we have no access to the source code, and the semantic structure of the programs is obfuscated (i.e., Java bytecode stores no information on the type of jumps, which are simply *gotos*).

For this reason the first step in our semantic analyzer is a precise structural analysis of the program: we divide the control-flow graph into sets of regions that describe the program's semantic structures. This kind of structural analysis was first presented by Sharir [91, 72] and employed in several decompilers and optimizing compilers. Sharir's structural analysis is a powerful interval analysis able to decompose the control-flow graph into a set of basic semantic structures like loops and various kind of conditional structures. It proves particularly useful when the source code is not available as in the case of decompilers [19], or, as in our case, in a bytecode to native compiler [68].

In addition to precisely classify the generated intervals, the individual semantic regions are smaller and simpler than in a classic interval analysis (although the control tree formed by the semantic regions is typically larger). Another positive aspect is that every reducible region has a unique entry point (the region header) facilitating many analyses.

The algorithm builds a depth-first spanning tree of the control-flow graph, which is then traversed in postorder. For every node that is traversed the compiler tries to match it with a series of predefined structural language-dependent patterns (see Figure 2.1 for the list of patterns we recognize). Note that we do not consider any pattern to represent *Case–switch* structures since they are translated to a sequence of *If–then* and *If–then–else* structures before building the control-flow graph. This simplification limits the efficiency of the produced code (we cannot generate jump tables) but helps in many analyses since for every block we have at most two successors which are easily identifiable (i.e., branch and fail).

These patterns are categorized into cyclic, noncyclic, proper, and improper regions. Improper (or nonreducible) regions are cycles with more than one entry point (these constructs are not allowed in Java and many other languages) while proper regions are noncyclic structures which do not follow a standard pattern (they are normally generated by complex *If* statements containing complex boolean expressions). We use proper regions to define arbitrary noncyclic regions that cannot be reduced to any known pattern as in [72] and not to describe regions that are non improper in a general way.

While *Block*, *If–then*, *If–then–else*, *While–loop*, *Repeat–loop* and *Improper* patterns are static in the number of nodes and edges (see Figure 2.1.a), *Natural–loop* and *Proper* can have an undefined number of nodes keeping the general structure (see Figure 2.1.b). E.g., a *Natural–loop* can have an unlimited number of exit edges).

During the traversal every node is first compared to the header of each known pattern: if the node and its immediate successors match with the predefined structure they are reduced to an abstract node representing the semantic region—similarly to a classical interval analysis.

In addition, before reduction, for every cyclic region (i.e., loop) we add a special empty *pre-header* node which acts as the unique header predecessor outside the loop and a special empty *continue* node which acts as the tail of the unique back-edge (see Figure 2.2): Every edge from a header predecessor that is not part of the loop is redirected to the pre-header, and every back-edge is redirected to the continue node. This ensures that every loop has a unique entry point and a unique back-edge.

Figure 2.3 shows an example of the pattern matching process: When node n_0 is analyzed the compiler checks if the successors (n_1 and n_2 in our case) have a common unique successor (n_3) and no other predecessors other then n_0 itself. If all the conditions apply the gray nodes are reduced to an abstract node representing an *If–then–else* region, if not, the compiler tries to match n_0 with the header of the next patterns from the list shown in Figure 2.1.

The result is a hierarchical tree of subgraphs embedded into each other representing the semantic constructs of the program. Figure 2.4 shows an example decomposition of a simple Java program: the program on the left represented by the control-flow graph on the right is decomposed into semantic regions (dotted lines); the table on the bottom summarizes the composition of the different regions.

Structural analysis, in contrast to a normal analysis based on the dominator relation, can

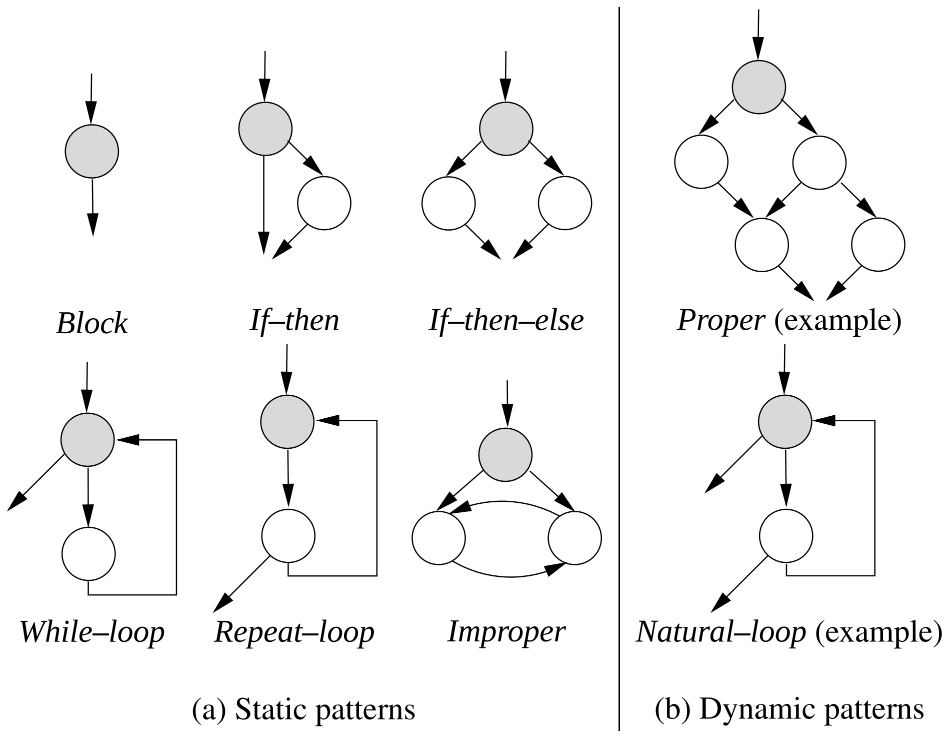
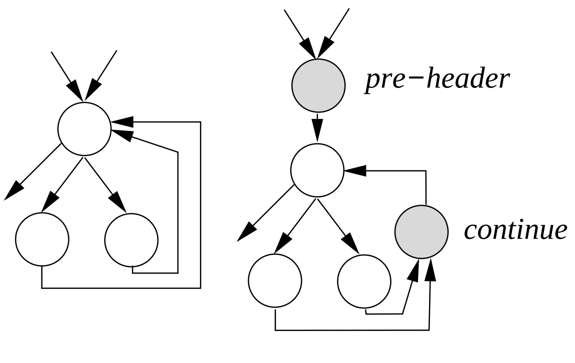


Figure 2.1: Structural patterns (the header is shown shades).



(a) (b)

Figure 2.2: Loop transformations: the loop (a) in enhanced by the addition of a pre-header and a continue node (b).

easily recognize cyclic structures with more than one back-edge as a single loop (see Figure 2.5) and handle improper (or irreducible) regions (see the *Improper* pattern in Figure 2.1).

Unfortunately the comparison of small graph regions with a series of predefined patterns, as initially presented by Sharir, is not sufficient to handle every reducible control-flow graph, as some noncyclic structures generated by complex boolean expressions in conditional statements do not follow a general scheme (proper regions and natural loops, see Figure 2.1). Complex boolean expressions regions are not reducible to any known pattern: They are noncyclic and are connected to the rest of the control-flow graph by two *joint nodes* (also called *articulation points*). Joint nodes are a set of nodes, normally two, which connect a strongly connected region

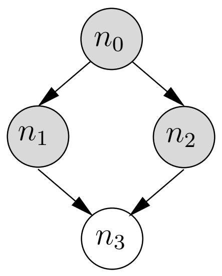


Figure 2.3: Pattern selection.

of the graph to the rest of it, removing these joint nodes, this region remains disconnected. Since such a region has a unique entry point, every node is dominated by the region header. Figure 2.6 shows such a region (dashed line) for a composite *If–then–else* statement (the two joint nodes are shown in gray).

To properly identify these commonly found regions, we developed the graph traversal algorithm shown in Figure 2.7 that is performed on the partially reduced graph: Starting from the node that is supposed to be the region header ($header$), after having tried to match it with a known pattern, we iteratively traverse the graph looking for the second joint node. We stop, and return an empty region, if we find a node that has a successor or a predecessor that is not dominated by the header. Otherwise we create a new region (*proper region*) containing the subgraph between the two joint nodes.

The complete algorithm has a worst-case running time asymptotically quadratic in the number of basic blocks but has shown experimentally to be asymptotically linear in the average case.

Figure 2.8 shows an example with a snapshot of the traversal algorithm. Starting from node K (the possible header of the region) we traverse all the successors and mark them as visited (gray nodes). For each node we check if all its successors and predecessors are dominated by the header. In this case node L is part of the region while node M is not, since node N or node O are not dominated by the header (K). This means that K is not a *proper region* header and we return the empty set.

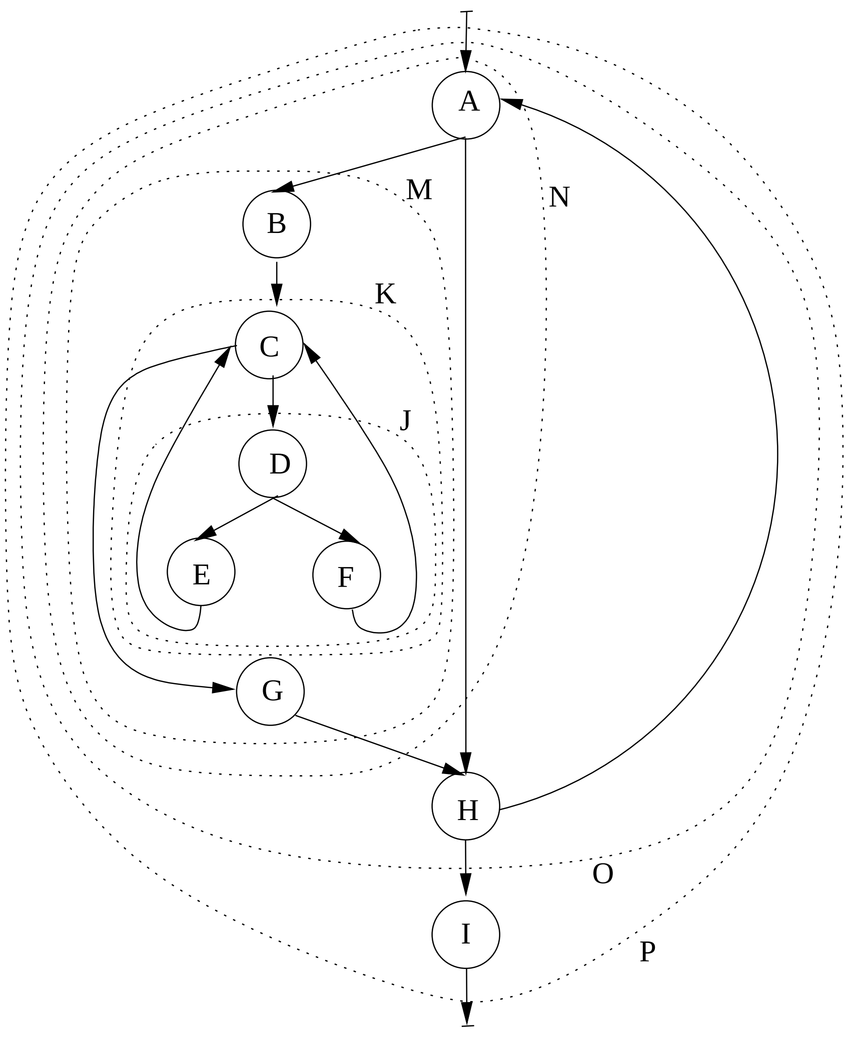
2.2 Loop bounding

Cyclic structures as loops or recursive procedures are of central interest to compute an upper bound on the execution time of any program. Being one of the major factor that contributes to the duration of a process, loops and their analysis techniques are the main concern in the design and implementation of a WCET analyzer since the choice of the analysis methodology influences the whole framework and determines which information we will be able to retrieve.

As described in Section 1.2 there are two main approaches: techniques as abstract interpretation based on symbolic execution that try to simulate the program behavior for all possible executions or techniques that try to analyze the loops in isolation with a set of static semantic analyses.

Both approaches have advantages and drawbacks: Approaches such as abstract interpretation allow, in theory, a complete knowledge of the program behavior since all the possible

```
do {
  if (A) {
    B;
    while (C) {
      if (D)
        E;
      else
        F;
    }
    G;
  }
} while (H);
I;
```



```
J:=If-then-else(D, E, F)  N:=If-then(A, M)
K:=While(C, J)            O:=Repeat(N, H)
M:=Block(B, K, G)         P:=Block(O, I)
```

Figure 2.4: Structural regions for a simple Java program.

executions are simulated.

In practice, due to the number of paths to be analyzed (exponential in the number of executed conditional branches), the precision of the results is artificially reduced to maintain the size of the problem manageable. In addition to the maximum number of loop iterations, abstract interpretation methods are able to detect false or infeasible paths.

Static analyses instead do not suffer from the path explosion problem since they analyze loops in isolation without differentiating different incoming paths in asymptotically polynomial analysis times. On the other hand the local character of these techniques does not allow them to consider information regarding more than one loop (or semantic structure) like loop dependences or infeasible paths.

The choice of the type of analysis for our WCET estimator was driven by the type of applications that we want to analyze: We target large soft real-time applications and therefore we

```
while (A) {
  if (B)
    C;
  else
    D;
}
```

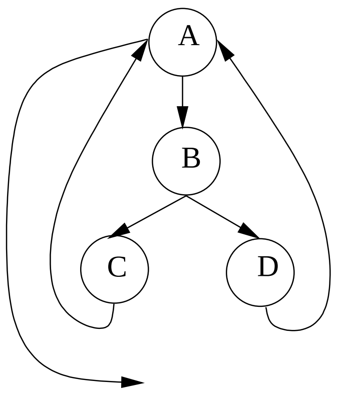


Figure 2.5: Loop with more than a back-edge.

```
if (A || (B && C)) {
    D;
} else {
    E;
}
F;
```

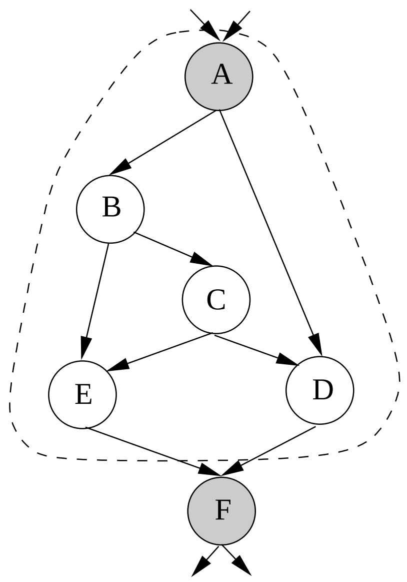


Figure 2.6: Region that cannot be represented by a predefined pattern.

need a fast and very scalable analysis that requires a minimal user intervention whereas some overestimation in the WCET can be tolerated.

To that end, to analyze a loop we choose a local and static approach (described in Section 2.4) but as shown in Section 2.3, we do not completely discard the ideas behind abstract interpretation, which we used, in a simplified way, to find infeasible program paths.

2.3 Partial abstract interpretation

Although our loop bounding technique analyzes the loops in isolation and does not perform any path enumeration (see Section 2.4) we use a limited form of abstract interpretation to detect infeasible paths on acyclic code segments.

The main problem of abstract interpretation is that every possible execution trace[1] has to

[1]We define a *trace* as a sequence of instructions, including branches and loops, that is executed for some input

```
$W := successors(header)$
$R := \emptyset$
mark the $header$ as visited
**while** $W \neq \emptyset$ **do**
   $node := \diamond W$
   $R := R \cup node$
   mark $node$ as visited
   **if** a predecessor of $node$ is not dominated by the header **then**
      (* this means that the header is not a joint node *)
      return $\emptyset$
   **end if**
   **for all** successors $s$ of $node$ **do**
      **if** $s$ is not dominated by the header **then**
         (* we do not follow back edges *)
         return $\emptyset$
      **else**
         **if** $s$ is not visited **then**
            $W := W \cup s$
         **end if**
      **end if**
   **end for**
   return $R$
**end while**
```

Figure 2.7: Algorithm to detect proper regions.

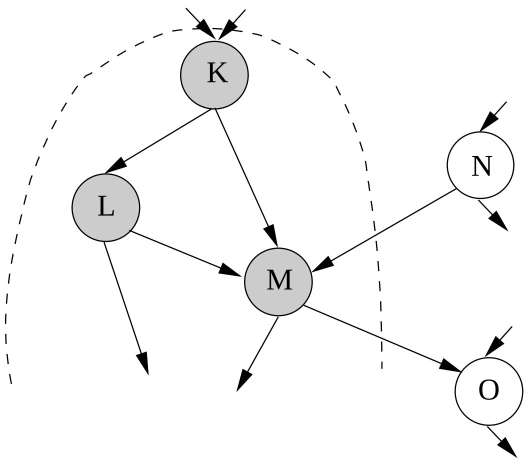


Figure 2.8: Snapshot of the proper region detection algorithm.

be considered with the consequence that the number of paths to be analyzed is exponential in the number of conditional branches that will be executed. In fact, at every loop iteration the number of possible paths is doubled increasing the necessity to merge and reduce the gathered data.

information.

To avoid the path explosion problem we do not iterate over loops and we limit the interpretation to linear code segments. In this way we are clearly not able to bound loops and we are not able to store any information relative to a path or partial path that crosses the loop boundaries. Nevertheless we are able to detect a number of false paths (see Section 2.3.5) and to bound the possible values an induction variable can assume at runtime (see Section 2.3.6).

2.3.1 Partial linear paths

We define a *partial linear path* as a part of a complete method trace that does not cross a loop boundary. This may include linear code snippets as well as loop bodies and does not exclude paths containing a loop header. If we consider Figure 2.4 both partial paths `ABCGH` and `DE` are valid partial linear paths while `BCD` is not a valid linear path since it crosses the border of the loop `While(C,D,E,F)` (`ABCGH` includes the loop header `C` but does not include blocks inside and outside the loop at the same time).

We define the *border* of a loop as the set of basic blocks that have both edges connecting them to blocks in the loop and edges connecting them to blocks outside the loop (the loop border is composed by the loop header and all blocks ending in a conditional branch that could cause the loop to terminate).

The choice to focus on linear paths instead of working with general program traces (as for a classical abstract interpretation pass) allows us to perform the analysis in exponential time over the number of conditional jumps present in the program whith a small known constant. In this case we only consider the presence of a conditional jump and not how many times it could be executed since we do not consider cyclic partial linear paths. Table 2.1 lists the number of analyzed partial linear paths for a number of benchmark applications, showing the limited number of paths that have to be executed in the abstract domain. The table also lists the total number of sets of integer intervals needed for the abstract interpretation and loop bounding passes along with their maximum and average length (i.e., the number of intervals in the set).

These benchmarks demonstrate that the restriction of the analysis to partial lineal paths and the confinement of the environment representation to integers (see Section 2.3.2) keep the size of the set of intervals small and manageable.

Table 2.1: Size of the abstract domain representation.

Benchmark	Size (LOC)	Partial linear paths	Set of intervals		
			Number	Max. length	Av. length
_201_compress	574	2 891	3 773	7	1.2
_205_raytrace	1 978	26	138	3	0.9
_209_db	664	313	1 854	7	0.8
JavaLayer	5 816	8 050	12 141	5	1.5
Scimark	756	233	1 860	3	0.7
Whetstone	128	12	465	1	0.7

2.3.2 Representation of the abstract domain

The aim of abstract interpretation is to observe the program behavior by executing it, or part of it in our case, in an abstract domain. The representation of the values of the elements (variables) in the abstract domain and the operations that we can perform on them must be defined so that they are as precise as possible without leading to incorrect conclusions about the analyzed program.

The abstract representation of the concrete values must be safe, i.e., it must not include values that are not possible in the concrete domain and that could lead to a wrong interpretation of the program behavior. This loose definition allows abstract representation to be a safe approximation of the concrete domain (approximations are introduced to reduce the calculations needed at each step and to reduce the number of paths that have to be considered).

To further simplify our approach we decided to consider integer local variables only, for this analysis. This includes all the Java integer types plus the boolean type but not references. We do not consider floating point types since their representation would require too much space and floating point types are rarely used as variables responsible for loop termination. Table 2.2 shows the type of the control variables used for loop termination in a number of benchmarks: the last column holds the total number of conditional branches with a loop's control variable as operand while the third, fourth and fifth columns categorize the type of the variables into integers, floating point types and object fields (note that these categories are not mutually exclusive).

Table 2.2: Loop control variable types.

Benchmark	Size (LOC)	Integer		Floating point		Field		Comparisons
_201_compress	574	50	(100%)	0	(0%)	10	(20%)	50
_205_raytrace	1 978	4	(100%)	0	(0%)	0	(0%)	4
_209_db	664	75	(84%)	0	(0%)	4	(4%)	89
JavaLayer	5 816	208	(95%)	2	(1%)	52	(24%)	218
Linpack	291	124	(69%)	55	(31%)	0	(0%)	179
Scimark	756	77	(93%)	4	(5%)	0	(0%)	83
Whetstone	128	26	(100%)	0	(0%)	4	(4%)	89

Limiting the abstract representation to local variables allow us to safely ignore aliasing problems. Aliased variables in addition to complicate the representation would unnecessarily increase the size of the abstract representation.

This restrictions are not limiting since the majority of the conditional branches that influence the program's execution flow are either integer or boolean comparisons or checks on dynamic data structures that could not be analyzed at compile time in any case since they are often dependent on the input set (see Table 2.2).

The simplest way to store the possible values of an integer variable is to represent them as a set of closed intervals obeying to the following criteria:

- All the variables that are undefined are represented as $v \in]-\infty..\infty[$ in the abstract domain. Since we limit the analysis to integer types, ∞ and $-\infty$ can be safely represented by the maximum and minimum value of the given type.

- The assignment of a constant c to a variable v is represented as $v \in [c]$ in the abstract domain.
- After a conditional jump that compares a variable v to a constant or another integer whose possible values are known, we split the interval of possible values in the abstract domain according to the two different outcomes of the test.

  This test can in case of the equality operators ($=$ and $\neq$) generate integer intervals with *holes*, as illustrated by the following example:

  ```
  {$v \in ]-\infty..\infty[$}
  **if** $v \neq c$ **then**
    {$v \in ]-\infty..c-1] \cup [c+1..\infty[$}
  **else**
    {$v \in [c]$}
  **end if**
  ```

  For this reason instead of representing variables values in the abstract domain with integer intervals we store them as sets of disjoint integer intervals.

  $$I = \bigcup_i I_i \text{ where } \bigcap_i I_i = \emptyset.$$

  In Java bytecode, boolean checks are simply mapped to integer equality tests with the constants 1 and 0. This contributes significantly to the presence of set of intervals in the abstract representation.

- The abstract representation is updated when arithmetic operations are performed on the concrete variables: For this purpose we defined the most common mathematical operations (addition, subtraction, multiplication and division) on the sets of intervals.

  The operations on two sets of integers are simply, and safely, defined as the set of integers that represent the result of applying the operation on every couple of elements.

  $$I_i \circ I_j = \bigcup_{v_i \in I_i, v_j \in I_j} v_i \circ v_j.$$

2.3.3 Safe approximations of the abstract domain representation

Even if we do not iterate over loops there are situations where it is advisable to reduce the size of the representation of the variables in the abstract domain.

The repetition of multiplications between two interval sets can, for example, generate very scattered sets of intervals. Such a representation can have a negative impact on the analysis performance.

When the number of paths to be considered or the number of intervals in a set becomes too large, the analyzer may decide to approximate the representation in the abstract domain by merging a set of intervals I_i to a unique interval I

$$I := \left[\min_i(I_i)..\max_i(I_i)\right].$$

In our case these approximations are performed when the number of elements in a set grows above a given threshold.

2.3.4 Abstract interpretation

To perform the partial abstract interpretation pass we first compute the topological ordering of the program's basic blocks. For this purpose we treat the back edges of all the loops as edges to every possible exit of the considered loop. In this way we ensure that all the nodes in the loop body come before all the loop exits in the topological ordering.

We then visit every block in topological order and update the variables' representation in the abstract domain. At conditional jumps we duplicate the abstract domain representation for each outgoing path and we continue the analysis on both paths recursively.

If the conditional jump is at the end of a loop's header block we follow only the edge that is not entering the loop (if there is one). Since some of the variables in our environment could be changed in the loop body we have to remove them from the abstract domain before continuing: All the variables that are not loop invariant are therefore set as unknown ($[-\infty..\infty]$) in the abstract domain at the beginning of the loop's header.

We call the representation of the variables in the abstract domain for a given path an *environment*. Such an environment contains a partial linear path and a series of set of intervals for all the local integer variables that are not undefined on the given partial linear path.

The same algorithm is also recursively applied to the loop bodies allowing us to analyze the loop local behavior of the considered variables.

2.3.5 False paths

The first important result that we can derive from the partial abstract interpretation analysis is that, using the abstract environment, we can automatically detect some infeasible or false path. A false or infeasible (partial) linear path is a sequence of basic blocks which cannot be executed under any circumstance.

Having the possible values for all the integer local variables on every path for some conditional branches, it is possible to determine if the tested condition is not satisfiable for some environments.

Figure 2.9 shows a synthetic short linear program with some false paths. When the analysis reaches block `F` the following abstract domain environments have been computed:

`ABD` $\{$`par` $\in [0]$, `pos` $\in [1]\}$
`ABE` $\{$`par` $\in [1..\infty[$, `pos` $\in [1]\}$
`AC` $\{$`par` $\in]-\infty..-1]$, `pos` $\in [0]\}$.

After analyzing block F we have:

`ABDFH` $\{$`par` $\in [0]$, `pos` $\in [1]\}$
`ABDFG` $\{$`par` $\in [0]$, `pos` $\in [1]$, `pos` $\in [0]\}$
`ABEFH` $\{$`par` $\in [1..\infty[$, `pos` $\in [1]\}$
`ABEFG` $\{$`par` $\in [1..\infty[$, `pos` $\in [1]$, `pos` $\in [0]\}$
`ACFH` $\{$`par` $\in]-\infty..-1]$, `pos` $\in [0]$, `pos` $\in [1]\}$
`ACFG` $\{$`par` $\in]-\infty..-1]$, `pos` $\in [0]\}$.

```
    public void foo(int par) {
A     boolean pos;
A     if (par < 0) {
C       pos = false;
B     } else if (par == 0) {
D       bar0();
D       pos = true;
      } else {
E       bar1();
E       pos = true;
      }
F     if (pos) {
H       bar2();
      } else {
G       bar3();
      }
I   }
```

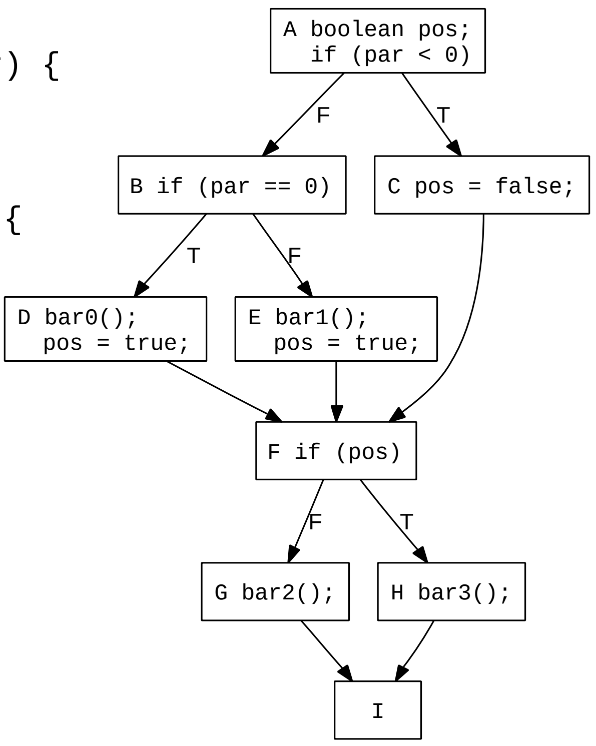


Figure 2.9: Example of a program with false paths.

The partial linear paths `ABDFG`, `ABEFG` and `ACFH` are clearly impossible (`pos` should be 1 and 0 at the same time) and can be marked as infeasible.

If a partial linear false path is composed of a series of blocks with only one successor and one predecessor the partial linear false path corresponds to dead code which can then be eliminated by the timing analyzer.

The main difference between this approach with what was proposed by Altenbernd [2] and later refined by Gustafsson [40] lies in the scope of the analysis: we do not aim to discover all the possible false paths and we therefore limit the checks to linear code segments. This important simplification has major implications in the implementation of our approach, which does not suffer from the path explosion problem and requires almost no approximation of the values in the abstract domain (see Table 5.1).

2.3.6 Variable ranges

In addition to the detection of partial false paths the abstract interpretation pass gathers useful control-flow aware information on the range of values a certain variable can assume.

This can not only be used for some classical compiler optimizations like the elimination of unnecessary array index checks or the reduction of the size of integer variables but can help to significantly reduce the estimated number of loop iterations—the more we can reduce the computed set of possible values of the iteration variables, the preciser the loop bounding algorithm can be (see Section 2.4)—or to introduce an elegant way to express manual code annotations (see Section 2.7).

2.4 Loop header bounding

Since we limit the abstract interpretation pass to linear code segments we have to rely on another technique to determine the maximum number of loop iterations (in our case a loop-local analysis). To bound the minimal and maximal number of executions of the loop header we choose an approach very similar to the analysis proposed by Christopher Healy at Florida State University [42, 44, 43].

We adapted the basic idea of this method to our language (Java) and integrated the same data structures (sets of intervals) that we use for the partial abstract interpretation pass (see Section 2.3.2).

The original method [42] basically consists of four main steps: First the program's control-flow graph (CFG) is built and the loops and nodes that could be responsible for loop termination are identified (they are called *iteration branches*). These nodes are then linked using a precedence relation representing the order in which these nodes could be executed in a loop iteration. The resulting directed acyclic graph is called *precedence graph*. In a second step, for each of the nodes of the precedence graph (i.e., the iteration branches), it is computed when the conditional jump for each iteration branch could change its result based on the number of iterations. I.e., the algorithm computes, if possible, the iteration at which the control-flow could change at such a node. In the next step, we determine the range of possible iterations when each of the outgoing edges is reached. In a fourth and final step, the maximum (and minimum) number of loop iterations is computed.

2.4.1 Precedence graph

An *iteration branch* is, as defined by Healy, a basic block with a conditional jump where the choice between the two outgoing edges could influence the number of loop iterations. In other words iteration branches have an outgoing edge to a node outside the analyzed loop, an edge to the loop header, or an edge to a block which is postdominated by the loop header.

The original algorithm to compute the set of iteration branches I for a loop L with header H is shown in Figure 2.10. The idea is to iteratively include (1) all the conditional branches with an edge exiting the loop which have a successor that is postdominated by the header (and therefore is on all the paths from the header to the exit of loop) or (2) that can conditionally lead to other iteration branches.

In addition to the set of iteration branches found by the original algorithm, to be able to handle general loops with complex bodies, we have to include the set of nodes that could conditionally lead to a single iteration branch:

$((S_1 \in I) \wedge (S_1 \neq S_2) \wedge \neg\, (S_1 \text{ pdom } S_2)) \vee$
$((S_2 \in I) \wedge (S_1 \neq S_2) \wedge \neg\, (S_2 \text{ pdom } S_2))$

S_n represents, as in Figure 2.10, one of the successors of block B and I is the set of identified iteration branches.

The simple example in Figure 2.11 shows a loop body where a node (block 3, `if (i < 3)`), which could clearly be responsible for loop termination, is not included in the set of iteration branches I by the original algorithm.

A loop's *precedence graph* is a directed acyclic graph composed by the loop's iteration

```
$I = \emptyset$ {set of iteration branches}
**repeat**
   {$H$ is the header node of the loop $L$}
   **for all** blocks $B \in L$ **do**
      **if** ($B$ has two successors $S_1$ and $S_2$) $\wedge$ ($B \notin I$) **then**
         **if** ($S_1 \notin L$) $\vee$
         ($S_2 \notin L$) $\vee$
         ($H$ pdom $S_1$) $\vee$
         ($H$ pdom $S_2$) $\vee$
         ($\exists\, J, K \in I \mid J \neq K \wedge J$ pdom $S_1 \wedge K$ pdom $S_2$) **then**
            $I := I \cup B$
         **end if**
      **end if**
   **end for**
**until** any change to $I$
```

Figure 2.10: Finding iteration branches.

```
     public void foo(boolean a, b) {
 2     for(int i=1; i<10; i++) {
 3       if(i < 3) {
 6         if(a) {
 7           break;
15         }
        } else {
 4         if(i == 3) {
 5           break;
14         }
 8       }
 9     }
     }
```

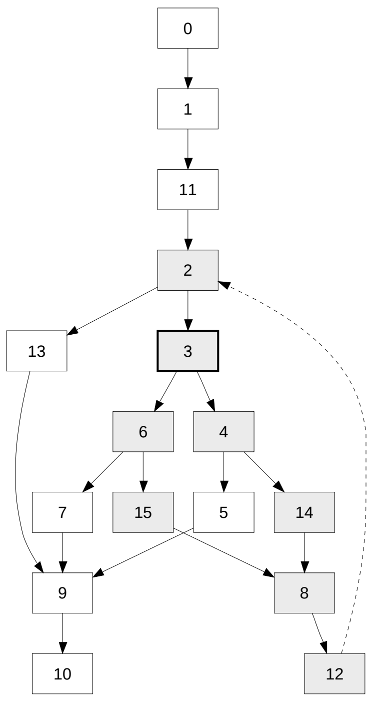


Figure 2.11: Additional iteration branches.

branches and two special nodes: the *continue* node and the *break* node representing the back-edge and the exit of the loop respectively. The graph is constructed collapsing all the nodes that are not iteration branches, substituting all the back-edges with edges to the continue node and substituting all the loop exits with edges to the break node as described in [42].

2.4.2 Iteration branches

The next step is the computation of the point in time, in terms of loop iterations, when the conditional branch of an iteration branch will change its direction. Our implementation is based again on the algorithm presented by Healy [42] with the introduction of ranges of integers in the algorithm's description and implementation. Due to the integration of sets of integer ranges and to some refinements to handle a broader set of loops than the loops handled by the original paper, we redefine some of the terms and equations as explained in the following paragraphs.

Each iteration branch is analyzed in isolation by detecting the variables that are compared in the conditional branch and by understanding how they change before and during the loop execution.

We only consider comparisons where an integer variable, the *iteration variable*, is compared to a constant or to a known range of possible constants that do not change in the loop body (the majority of loops are conditioned by integer induction variables, see Table 2.2). Once the iteration variable is determined we try to compute its value before entering the loop and how it is changed by in the loop body during every iteration (we distinguish between changes before the iteration branch and changes after the iteration branch).

For each iteration branch we then compute on which iteration N the result of the conditional branch will change:

$$N = \left\lceil \frac{limit - (initial + before) + adjust}{before + after} \right\rceil + 1$$

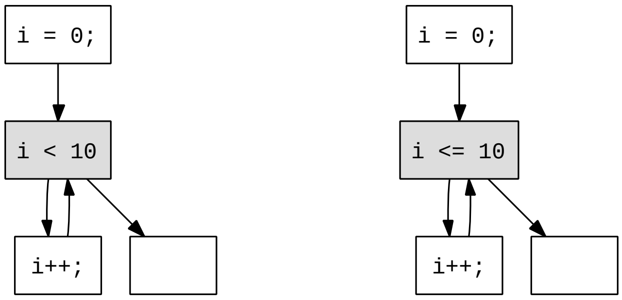
where $after$ and $before$ are the amount by which the iteration variable is changed during each loop iteration after and before the considered iteration branch (to be able to compute N, both $before$ and $after$ must remain constant during each iteration). The sum of $before$ and $after$ gives the $direction$ (positive or negative) of the induction variable increment. If this sum is 0, the result of the jump at the end of the iteration branch will never change, and N is set to a special value *unknown*. If $N > 0$ the result of the conditional branch will change after N iterations. If $N \leq 0$ the conditional branch could have changed before the first iteration (in this case we say that the change happens in the past).

$limit$ corresponds to the constant value used in the comparison. If $limit$ is not constant, and its possible values are represented by a set of integer intervals (see Section 2.3.6), we take its minimum (if $direction$ is positive) or its maximum (if $direction$ is negative). $initial$ is similar to $limit$ and represents the value of the induction variable when the iteration branch is reached for the first time, in other words it corresponds to the sum of the variable's value before entering the loop and $before$. In case limit is represented by a set of integer ranges we take its minimum if $direction$ is positive, or its maximum otherwise.

$adjust$ takes into account the type of the operators compensating the differences among them. If the induction variable is increased at every iteration, $adjust$ is set to 1 for $\leq$ and $>$, and if the induction variable is decreased at every iteration $adjust$ is set to -1 for $<$ and $\geq$. For all the other cases $adjust$ is set to 0 (see Figure 2.12 for an example).

If we are not able to compute one of these variables we assume that we have not enough information on the iteration branch behavior and we set the value of N to *unknown*.

An additional check is needed for equality operators ($=$ or $\neq$) since, in this case, it is possible that the checked condition will never be met although the variables are growing in the right



$$N = \left\lceil \frac{10-(0+0)+\mathbf{0}}{0+1} \right\rceil + 1 = 11 \quad N = \left\lceil \frac{10-(0+0)+\mathbf{1}}{0+1} \right\rceil + 1 = 12$$

Figure 2.12: Different adjustments for the computation of N of two sample loops (the loop's header and unique iteration branch is shown in gray).

direction (e.g., `for(i=0;i==5;i+=2)`). We therefore compute if

$$\frac{limit - (initial + before)}{before + after}$$

is an integral value, if not we set the value of N to *unknown*.

2.4.3 Reachability of iteration branches

Once we know when the behavior of each iteration branch might change (i.e., the iterations when it's result will be different from the one computed by the first loop iteration), we determine when the iteration branches might be executed.

We define $range(ib)$ as the iterations on which it will be possible to execute an iteration branch ib or $range(ib_0 \rightarrow ib_1)$ in the case of an edge form iteration branch ib_0 to iteration branch ib_1 ($ib_0 \rightarrow ib_1$). As before we adapted the approach presented by Healy to integrate sets of integer ranges and generalizing it to be able to easily handle infinite loops and loops that are never executed.

The precedence graph is traversed in a preorder fashion and for each node and edge we store the set of iterations during which it will be possible to traverse it.

The range of a node, or the set of iterations during which this node could be executed, is defined as the union of the ranges of all the incoming edges (the header of the loop is treated differently and we assign a range of $[1..\infty[$ to it):

$$range(ib) := \bigcup_{p \in pred(ib)} range(p \rightarrow ib).$$

The reason is obvious: every time an incoming edge is executed the node will also be executed. In the same way we can also define the ranges of the outgoing edges. If the point in time when the conditional branch of the considered iteration branch will change (N) is unknown, we have to assume that all the outgoing edges could be traversed in the same iterations as the iteration branch.

$$range(ib \rightarrow s) := range(ib) \quad \forall s \in succ(ib).$$

If, otherwise, N is known we split the set of all possible iterations between what happens before the change (before_N) and what happens after and during the change (after_N): this is done by splitting the entire iteration space and intersecting it with $\mathit{range}(\mathit{ib})$. If N is in the past (see Section 2.4.2) we set:

$$\begin{aligned}\mathit{before}_N(\mathit{ib}) &:= [1] \cap \mathit{range}(\mathit{ib}) \\ \mathit{after}_N(\mathit{ib}) &:= \emptyset.\end{aligned}$$

Otherwise the two sets are created according the equality operator:

$$\mathit{before}_N := \begin{cases} =, \neq & : \quad N \cap \mathit{range}(\mathit{ib}) \\ <, \leq, >, \geq & : \quad [1..N-1] \cap \mathit{range}(\mathit{ib}) \end{cases}$$

$$\mathit{after}_N := \begin{cases} =, \neq & : \quad ([1..N-1] \cup [N+1..\infty[) \cap \mathit{range}(\mathit{ib}) \\ <, \leq, >, \geq & : \quad [N..\infty[\cap \mathit{range}(\mathit{ib}) \end{cases}$$

Using the value of $\mathit{direction}$ we can determine when the iteration branch will be traversed for the first time: Given the direction of the loop, the value of the induction variable when it first reaches the iteration branch ($\mathit{initial} + \mathit{before}$) and the relational operator we assign before_N and after_N to the corresponding outgoing edges.

If at this point we discover that an edge is never executed (its range is empty) we can remove it from the graph (i.e., we detected an infeasible edge).

2.4.4 Loop header iterations

To determine the number of iterations for the loop header we traverse the precedence graph in postorder and for each node and edge we compute its *exit* value, i.e., when this node or edge could lead to a break (i.e., loop exit).

If an edge e is to a *break* node then $\mathit{exit}(e) := \mathit{range}(e)$: These are the only points in the control-flow graph where we can introduce a bound on the number of iterations since these edges ($n \xrightarrow{e} \mathit{break}$) are the only locations where a loop can exit. If the edge is to a *continue* node we do not have any useful information, and we set its exit value to unknown: $\mathit{exit}(e) := \emptyset$.

Otherwise the edge e is leading to an iteration branch ib inside the loop. In this case the exit value of e basically corresponds to the intersection between the range of the edge e and the exit value of the iteration branch ib ($\mathit{exit}(e) := \mathit{range}(e) \cap \mathit{exit}(\mathit{ib})$). Unfortunately there are a couple of exception that we have to consider. We distinguish three different cases:

- If the range of e is always smaller then the exit value of ib ($\max(\mathit{range}(e)) < \min(\mathit{exit}(\mathit{ib}))$), it means that the edge will be executed only once: $\mathit{exit}(e) := [\min(\mathit{range}(e))]$ (the value marked by the arrow in Figure 2.13.a). Even if the intersection $\mathit{range}(e) \cap \mathit{exit}(\mathit{ib})$ is empty the block will be executed once to perform the test on the first iteration.

  Example: $\mathit{exit}(\mathit{ib}) = [1..5], \mathit{range}(e) = [10..20]$. In this case the intersection between $\mathit{exit}(\mathit{ib})$ and $\mathit{range}(e)$ is empty: The edge will be traversed once (at the 10th iteration) and the conditional test at the end of ib will cause a loop exit ($\mathit{exit}(e) = [10]$).

- If the intersection of the exit values of the iteration branch and the range is not empty, it represents the iterations when this edge could lead to an exit: $exit(e) := range(e) \cap exit(ib)$ (the values marked by the gray area in Figures 2.13.b,c,f).

  Example: $exit(ib) = [5..15], range(e) = [10..20]$. In this case edge will be traversed from the 10th to the 15th iteration (the intersection between $exit(ib)$ and $range(e)$).

- If some exit values of the iteration branch are bigger than the maximum range of the edge it means that the loop is unbounded: $exit(e) := [\min(range(e) \cap exit(ib))..\infty[$ (the values marked by the gray area and the arrow in Figure 2.13.d,e). Note that we use infinity to denote unbounded ranges.

  Example: $exit(ib) = [15..30], range(e) = [10..20]$. The intersection between $exit(ib)$ and $range(e)$ is not empty and the edge will be traversed from the 15th iteration. Since some of the iterations in $exit(ib)$ are bigger than $\max(range(e))$ we are not able to tell when this edge will lead to a break and we therefore set $exit(e)$ as unbounded ($exit(e) = [15..\infty[$).

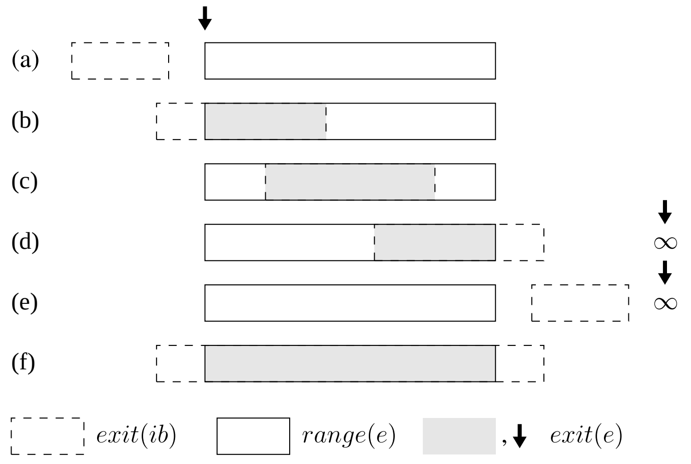


Figure 2.13: Exit value of ranges: the boxes represent the set of iterations in $range(e)$, the dashed boxes represent the set of iterations in $exit(ib)$ and the gray area corresponds to the resulting set of iterations in $exit(e)$. The down arrow represents the set of iterations in $exit(e)$ when the set is composed by a single element as in (a) or to include ∞ in the set when the sets are unbounded as in (d) end (e).

The exit value of an iteration branch is determined by the union of the exit values of its outgoing edges:

$$exit(ib) := \bigcup_{e_i \in out(ib)} exit(e_i)$$

where $out(ib)$ is the set of edges exiting from node ib.

At the end of the traversal the exit value of the loop header represents the number of iterations the loop header (and hence the loop) will be executed.

The main differences between the computation of the loop header iterations presented by Healy and Whalley and our implementation lies in the integration of the set on intervals which allows us to easily work in a uniform way with any operator (e.g., equality operators) and to cleanly define the algorithm behavior with conditional jumps that are always or never taken.

2.4.5 Nested loops

The described approach computes the number of loop iterations independently from other cyclic structures and only consider loops with a constant number of iterations. Inner loops with a variable number of iterations depending on the control variable of the outer loop (see Figure 2.14) are handled conservatively (i.e., the number of iterations for the inner loop is considered constant).

```
public void foo() {
  int i,j;
  for (i=0; i<100; i++) {
    for (j=i; j<100; j++) {
      bar();
    }
  }
}
```

Figure 2.14: Triangular loops: the number of iterations of the inner loop $(100 - i)$ depends on the control variable of the outer loop (i).

Healy showed that the technique can be extended (with some restrictions) to compute the number of iterations for triangular loops [42, 43].

2.5 Improving the precision of the loop bounder

A limitation of this method is that although the minimal and maximal number of iterations for the loop header are a safe bound for every block in the loop, this approximation does not take into account different paths inside the loop body.

To propagate bounding information from a loop header to each program block we need precise knowledge of the control-flow graph structure which can be derived from the semantic structures computed in Section 2.1.

The approach consists in propagating the loop bounds to the whole loop by adjusting the number of iterations in accordance with the different paths inside the loop body. In this way we compute a fine-grained per-block bounding information that allows the timing analyzer to handle infrequently executed blocks inside a loop body separately.

Figure 2.15 shows a simple example where a loop portion (block 4) in executed only once, and its duration should therefore not be included in each loop iteration. In this case, especially if the code in the region executed only once is computationally expensive the assumption that the header bounds are safe for each basic block in the loop is correct but leads to a WCET overestimation. This phenomenon can for example be observed in the _201_compress benchmark of the SPECjvm98 suite where a significant part of the main loop is executed only every 10,000 iterations.

For this reason our WCET estimator performs some additional passes to propagate the bounds of the loop header to all the blocks adapting them to the different path frequencies. Since the number of iterations for a block is not directly dependent on the block's predecessors

```
0   int i = 0;
2   while (i < 128) {
3     if (i == 16) {
4       // code executed once (block 4)
      }
5     // code executed at every iteration
5     i++;
6   }
```

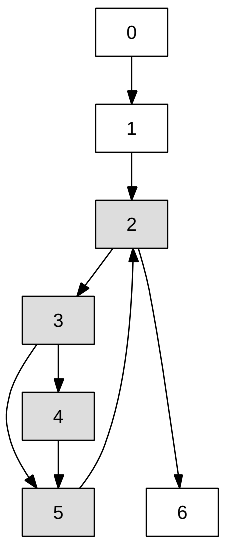


Figure 2.15: An example of a loop.

but is, instead, determined by the type of the enclosing semantic construct or structural region, along with the dominator relation, the bounds cannot be propagated easily to every basic block by analyzing the graph's nodes in isolation. Figure 2.16 shows three simple examples of structural and semantic constructs that influence the number of iterations (shown between brackets) of a given basic block (striped node). The number of iterations of a block cannot be derived from its direct predecessors, but only from the header (gray node) of the biggest enclosing structural region (a *While–loop* in Figure 2.16.a and an *If–then–else* in Figures 2.16.b and 2.16.c). In Figure 2.16.b we assume that we have no information on the conditional branch of the gray node and we have to conservatively assume that both white nodes could be executed 10 times.

To propagate the loop header bounding information to each block, we perform two main steps: In a first phase we propagate the minimal and maximal number of iterations along each loop's precedence graph (see Section 2.5.1) while in a second phase we spread this information to every basic block in the control-flow graph (see Section 2.5.2).

2.5.1 Execution counts for iteration branches

Loops are analyzed in isolation starting from the innermost one, allowing a simple handling of nested structures. Once the set of possible iterations for the loop header ($exit(header)$) has been computed (see Section 2.4.4) we propagate these values to the iteration branches with a top-down traversal of the precedence graph.

The first step consists in the computation of the *smallest enclosing cyclic region* for each iteration branch ($secr(ib)$). This structure is easily found by traversing the list of enclosing semantic regions stored in each block (see Section 2.1). The smallest enclosing cyclic region corresponds to the smallest loop that includes the considered iteration branch.

We then define, for each node and each edge in the precedence graph, a range $iter$ representing the minimal and maximal number of times the node or edge can be traversed. We set the number of iterations of the loop header to its exit value (i.e., the loop iterations):

$$iter(header) := exit(header).$$

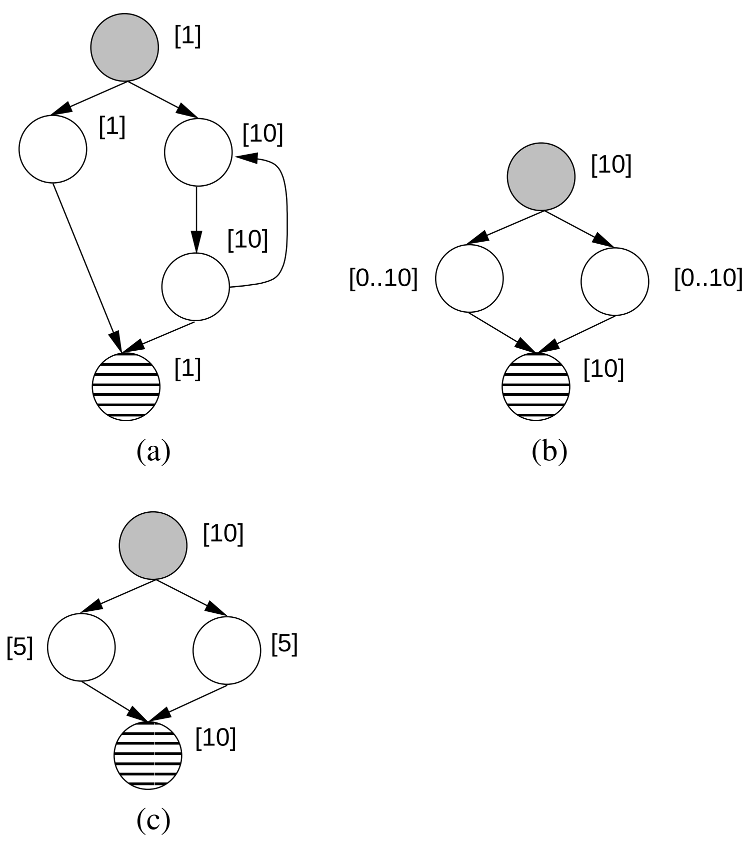


Figure 2.16: Examples of block iterations.

.

We then traverse the precedence graph inorder, and for each iteration branch ib and its outgoing edges e we compute the number of times they can be traversed: $iter(ib)$ and $iter(e)$ respectively.

The *iter* value of an edge e is determined by its range and the *iter* value of the iteration branch ib at the edge tail:

- If N, i.e., the iteration on which the result of the conditional branch at the end of ib will change direction, is known we distinguish three cases:
  - If the range of the edge is always smaller than the iter value of the iteration branch it means that the edge will always be taken:
$$iter(e) := [\max(range(e))].$$
  - If the range of the edge is always bigger than the iter value of the iteration branch it means that the edge will never be taken:
$$iter(e) := [0].$$
  - Otherwise the iter value corresponds to the intersection of $iter(ib)$ and $range(e)$ normalized to take into account the number of iterations when this edge will be traversed and not the actual iterations when this will happen:
$$iter(e) := (range(e) \cup iter(ib)) - (\min(range(e)) + 1).$$

- If N is unknown we do not have any information about when the jump result will change and we assign the minimal and maximal number of iterations $iter$ of the node i to the iter value of each outgoing edge:
$$iter(e) := iter(i).$$

In a similar way the iter value of an iteration branch ib is determined by the iter values of all the incoming edges. We determine the presence of inner loops by checking if the smallest enclosing cyclic region $secr(ib)$ corresponds to the loop we are analyzing.

- If the iteration branch is not in an inner loop (i.e., $secr(ib) = loop$) and each of its predecessors p has a known N value, the number of iterations (iter) corresponds then to the sum of the iterations of the incoming edges:
$$iter(ib) := \sum_{e \in pred \to ib} iter(e).$$
If one of the predecessors is *unknown* we don't have enough information and we have to conservatively assume that the iter value corresponds to the number of iterations of the header of the smallest region containing ib. This is given by the fact that given a region with a single entry point (the header) we can safely assume that every node in this region will not be executed more often that $\max(iter(header))$.

- If the iteration branch is the header of an inner loop (i.e., $secr(ib) \neq loop$) we have to multiply the iter value with the number of iterations of the inner loop. Note that the inner loop iterations are available since the loops are analyzed starting from the innermost one.
$$iter(ib) := \sum_{e \in pred \to ib} iter(e) \cdot iter(ib).$$
In addition, we store in the node ib a multiplication factor for the inner loop l_{inner} defined by the region $secs(ib)$ computed as follows:
$$mul(l_{inner}) := \sum_{e \in pred \to ib} iter(e).$$

- Otherwise, for inner loop nodes, we simply multiply the iterations of the inner loop node ib with the region multiplication factor stored in the header of each inner loop:
$$iter(ib) := iter(ib) \cdot mul(secs(ib)).$$

2.5.2 Execution counts for basic blocks

Now that the minimal number and maximal number of iterations of each basic block that could influence the flow of control of the program (i.e., the iteration branches) are known, we have to propagate the bounding information to every other block in the graph.

Once again we take advantage of the available structural information and we compute the number of iterations ($iter(b)$) for each basic block b. We treat the control-flow graph entry block differently, setting the minimal and maximal number of iterations to 1:
$$iter(root) := [1].$$

For each node b with a predecessor p we define $becr(b, p)$ as the *biggest enclosing cyclic region* containing p and not b. If $becr(b, p)$ exists for at least one predecessor p, it means that the node b is the exit point of one or more loops and that $bcr(b, p)$ corresponds to the outermost one.

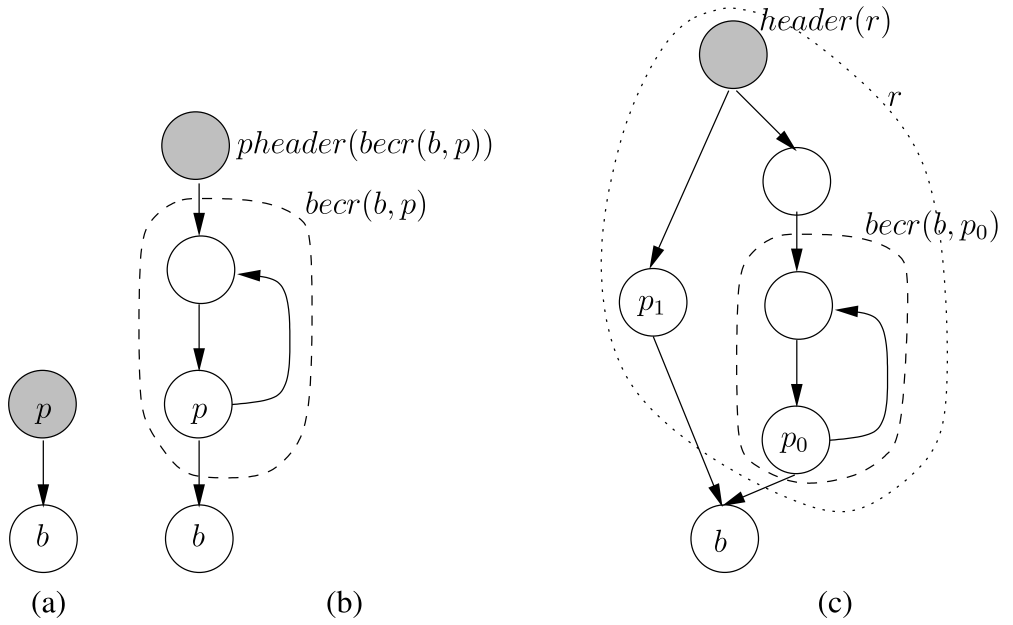


Figure 2.17: Biggest cyclic regions.

.

If p is the single predecessor of b, and $becr(b, p)$ is empty, we simply copy the range of iterations of the edge to the node (Figure 2.17.a):

$$iter(b) = iter(p).$$

If $becr(b, p_i)$ is the same region for all the predecessors p_i of b we have detected one or more loop breaking edges ($p_i \rightarrow b$), and the number of iterations of b is set equal to the number of iterations of the loop *pre-header* (see Figure 2.17.b). A loop pre-header ($pheader(l)$) is a special basic block artificially inserted by the compiler acting as a unique loop header predecessor (see Section 2.1).

$$iter(b) = iter(pheader(becr(b, p)))$$

Otherwise, if the predecessors belong to different enclosing regions, we look for the smallest region r that contains all the predecessors of b and not b itself (Figure 2.17.c). The iter value of the header of this region is a safe assumption since every path from the source to b will traverse it ($header(r)$ dom b).

The algorithm in Figure 2.18 summarizes the computation of $iter(b)$ for nodes that are not iteration branches.

At the end of this phase every basic block in the control-flow graph has a bound on its minimum and maximum number of iteration that takes into account the different path frequencies inside loops.

```
**if** $b = source$ **then**
  $iter(b) = [1]$
**else**
  $P :=$ predecessors$(b)$
  choose any$p \in P$
  **if** $|P| = 1 \vee becr(b, p) = \emptyset$ **then**
    $iter(b) := iter(b)$
  **else if** $\exists\ r \mid r = becr(b, p_i) \forall\ p_i \in P$ **then**
    $iter(b) := iter(pheader(r))$
  **else**
    $r :=$ smallestregion $\mid P \subset r \vee b \notin r$
    $iter(b) := iter(header(r))$
  **end if**
**end if**
```

Figure 2.18: Computation of $iter(b)$.

2.5.3 Complexity

One of the main advantages of the technique described here is that it is able to compute precise per block bounds in quadratic time over the number of iteration nodes: We are able to consider the effects of different path frequencies in the loop without the need to perform path enumeration.

Section 5.6 evaluates the effects of the increased granularity (from loops to blocks) of the bounding algorithm showing the improvements on the overall WCET estimation.

The computation of the iteration branches and the construction of the precedence graph are performed in $O(B^2)$, where B is the number of basic blocks in the considered loop. The original algorithm to compute the number of iterations [42] of the loop header and our enhancement to propagate the bounds to the single basic blocks are both performed in asymptotically linear time.

Healy and Whalley presented a different approach to enhance their loop bounder [42] (see Section 2.4) to compute fine grained block bounds using automatically detected value-dependent constraints [44]. They compute per block bounds analyzing the possible outcome of conditional branches by looking at the induction variable changes on all the possible paths. The algorithm has a complexity of $O(P)$ where P is the number of paths in a loop, which is exponential in B.

2.6 Example

This section presents a small synthetic example to summarize the analysis presented in this chapter. Figure 2.19 shows a small Java method with two nested loops: the inner loop is executed only after 50 iterations of the outer loop and the value of a parameter could determine the outer loop termination after 10 iterations.

Figure 2.20 shows the control-flow graph for our example with the list of semantic re-

```
public void foo(boolean somecond) {

  int i, j;

  for (i=0; i<100; i++) {
    if (i < 50) {
      if (i > 10 && somecond) {
        break;
      }
    } else {
      for (j=0; j<10;) {
        j++;
      }
    }
  }

}
```

Figure 2.19: Example Java program.

gions (dotted lines). The two program's loops are represented by `C = While(8,7,14)` and `F = While(10,2,3,4,9,6,13,8,7,14,16)`.

In the first phase similarly to Healy we identify the iteration branches (8 for loop `C` and 10, 2, 3, 4 and 8 for loop `F`) and we construct the respective precedence graphs. On these graphs as described by Section 2.4.3 we compute when it will be possible to execute the iteration branches ($range$). Figures 2.21.a and 2.21.b show the precedence graphs for the inner and outer loop respectively with the point in time when the conditional jump will change (N) and the ranges for each node and edge.

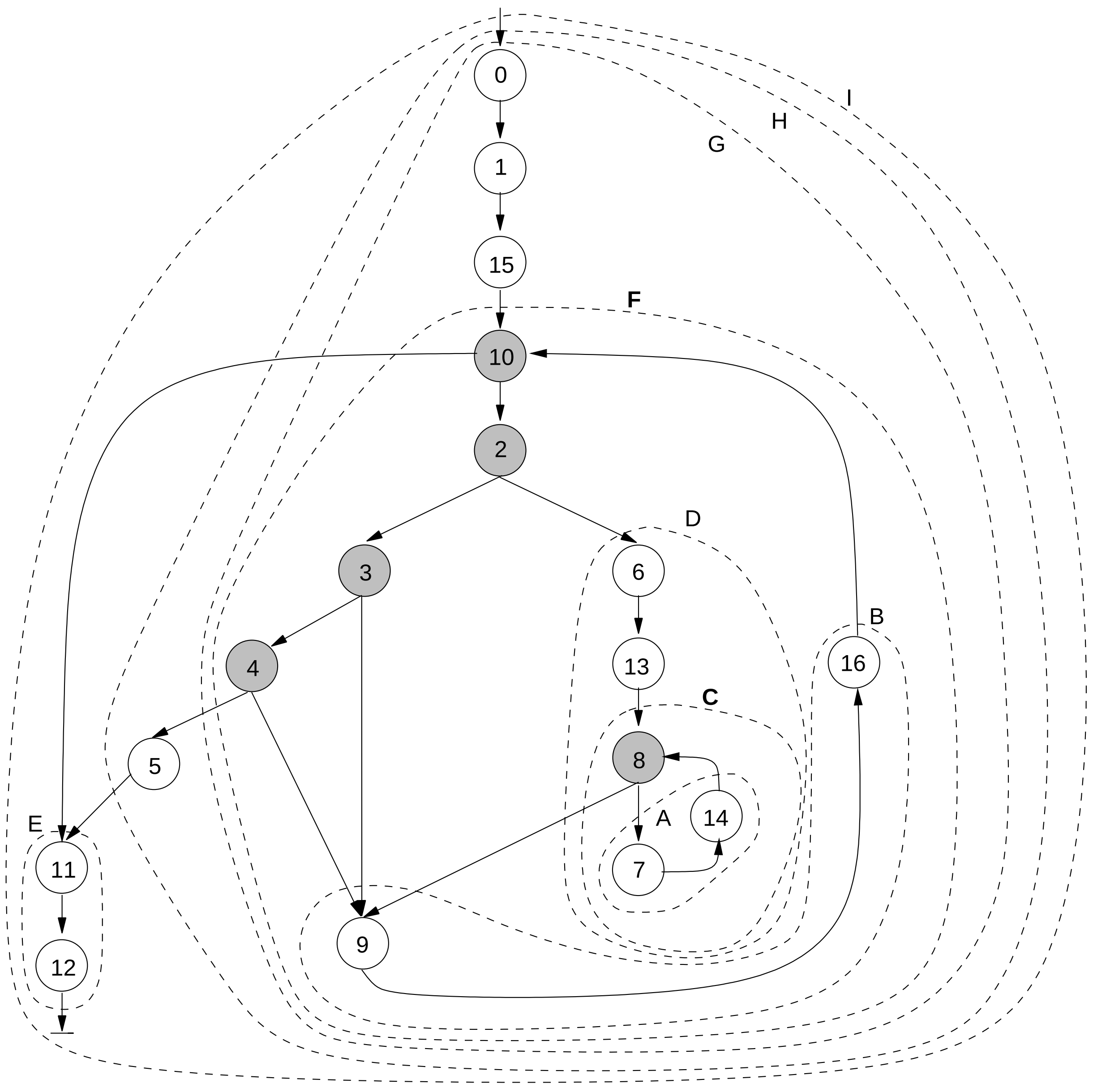
In the second phase we compute the minimal and maximal number of iterations for each loop header with the bottom up traversal described in Section 2.4.4. Figures 2.22.a and 2.22.b show the precedence graphs with the exit values for each node: The inner loop is bounded with 11 iterations while the outer loop can exit from iteration 12 to iteration 50 or exit at iteration 101.

Although these bounds are a safe assumption for the whole loop, using the algorithm described in Section 2.5.1 we compute the range of possible iterations for every iteration branch in the precedence graph as shown by Figure 2.23.

Figure 2.24 summarizes the propagation of the iteration branches bounds to the whole control-flow graph. We assign to every basic block a set of ranges representing the possible number of iterations.

2.7 Manual annotations

The goal of an automatic loop bounder is clearly to reduce or eliminate the need for the programmers to manually specify loop bounds. There are some cases, though, where the maximum

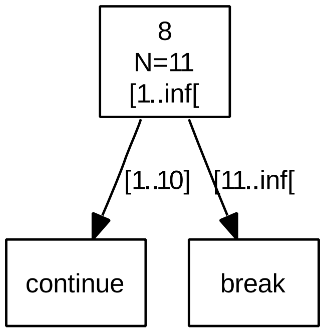


```
A := Block(14,7)           B := Block(16,9)
C := While(8, 17)          D := Block(19, 13, 6)
E := Block(12,11)          F := Natural-loop(10, 2, 3, 4, 18, 20)
G := Block(22, 15, 1, 0)   H := if-then(23, 5)
I := Block(24,21)
```

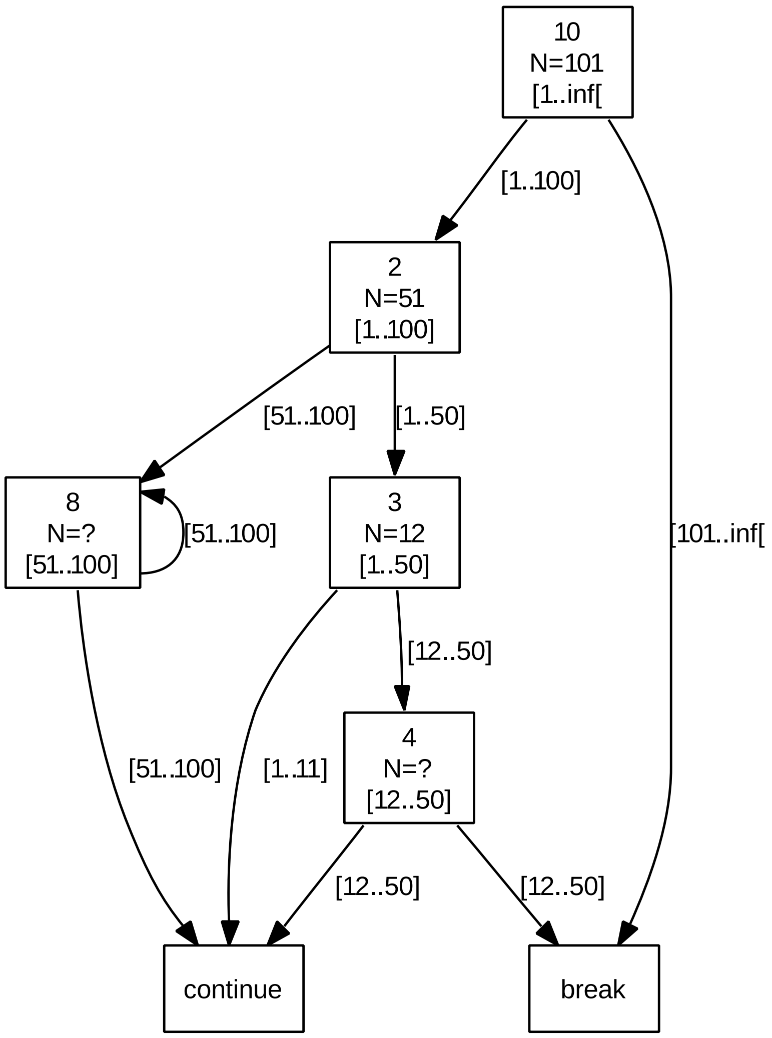
Figure 2.20: Control-flow graph.

number of loop iterations is dependent on the program input, or where the analyzer is not able to automatically compute them.

In these rare cases (the majority of the loops in real-time programs have a well defined and simple structure) the programmer must include comments in the code to manually specify the bounds.



(a) Inner loop C

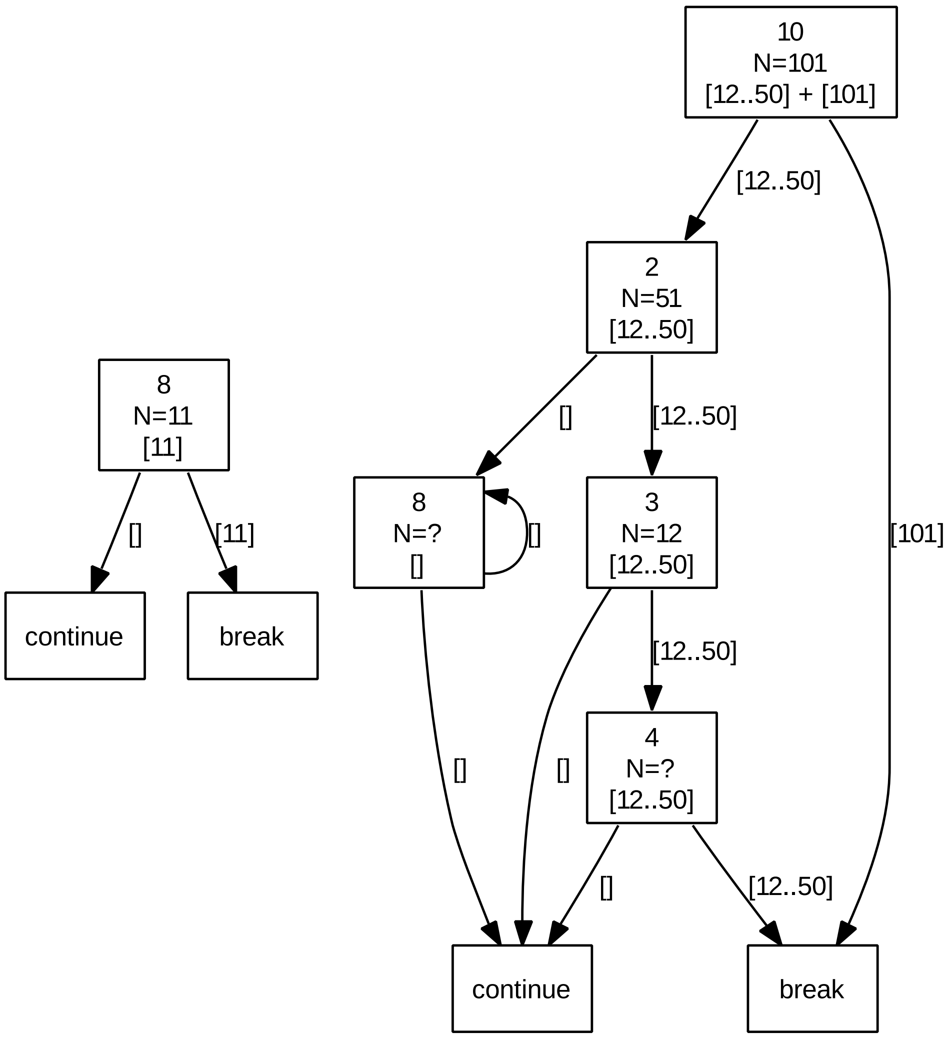


(b) Loop F

Figure 2.21: Precedence graphs with iteration ranges.

The simplest way to do this is to modify the code, inserting guards on the variables whose behavior is unknown to the timing analyzer. Figure 2.25.a shows a simple example where the timing analyzer is not able to compute the initial value of the variable `i`, which is input-dependent. In Figure 2.25.b a simple *If–then* construct can help the compiler to analyze the loop (after the partial abstract interpretation pass, see Section 2.3, `i` is known to have a value in the interval $[0..\infty[$) and at the same time ensures that the semantics of the annotation will be enforced at run time.

If the user considers the insertion of additional code an unnecessary overhead the analyzer allows him to comment the code with special library calls that do not produce any code (the dummy calls are safely removed before code generation since they do not have any functional meaning). These annotations are similar to a simplified version of the `WCETAn` system proposed by Bernat for their Java WCET analyzer [4]. In our case we only support the specification of bounds to cyclic structures since we do not need to define paths in the program as for `WCETAn` (e.g., to define infeasible paths).



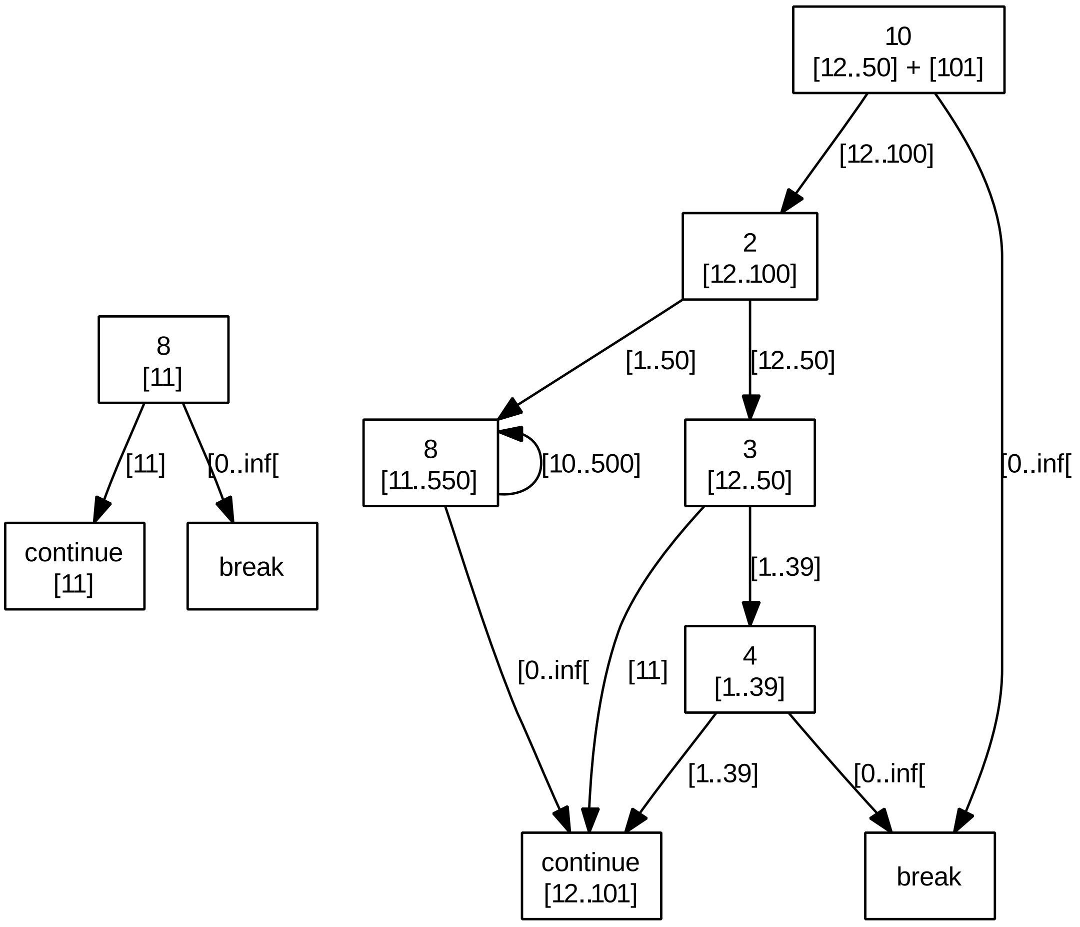
(a) Inner loop C (b) Loop F

Figure 2.22: Precedence graphs with exit values.

The `WCETAnnotations.maxLoopIterations(N)` defines the maximum number of iterations N of the first loop following the statement on all the paths leading to the exit of the control flow graph (i.e., the first cyclic semantic structure that is dominated by the annotation).

Figure 2.26 shows an example where manual annotations are used to specify the maximum number loop of iterations and the maximum recursion depth. The two method calls are used by the WCET semantic analyzer to bound the two cyclic structures but no checks are issued at run time and responsibility for correctness is left to the programmer.

Although annotations for loops are rarely needed, they are necessary for recursive procedures since we do not perform any automatic recursion depth analysis (see Section 2.8.3). The maximum recursion depth can be expressed with the `WCETAnnotations.maxRecursionDepth` dummy call. The `maxRecursionDepth` annotation works similarly to `maxLoopIterations` referring to the entry point of a cycle in the call graph.



(a) Inner loop C (b) Loop F

Figure 2.23: Precedence graphs iteration counts.

2.8 Method calls and object orientation

The additional level of dynamism in program execution, and therefore the reduced predictability of object-oriented programs, contributed to their slow acceptance in the real-time world, but in the last decade Smalltalk [40], Java [11, 38], Oberon-2 [15] and several other languages were successfully employed in hard and soft real-time projects.

One of the reasons for the introduction of modern languages into the real-time world, in addition to their popularity among programmers, is that they do not only allow to create more dynamic programs but often impose some restrictions on the program semantics. Strongly typed languages without pointer arithmetic like Java and Oberon [105] allow, as in our case, to reduce aliasing effects, making code analysis more effective.

The major problem of Java and object-oriented programs in general is that they allow the programmer to use object inheritance to override a method's implementations, so that the target of a method call depends on the dynamic type of the owning object. This can greatly enlarge the call graph and, if every possible target is considered, cause WCET overestimations.

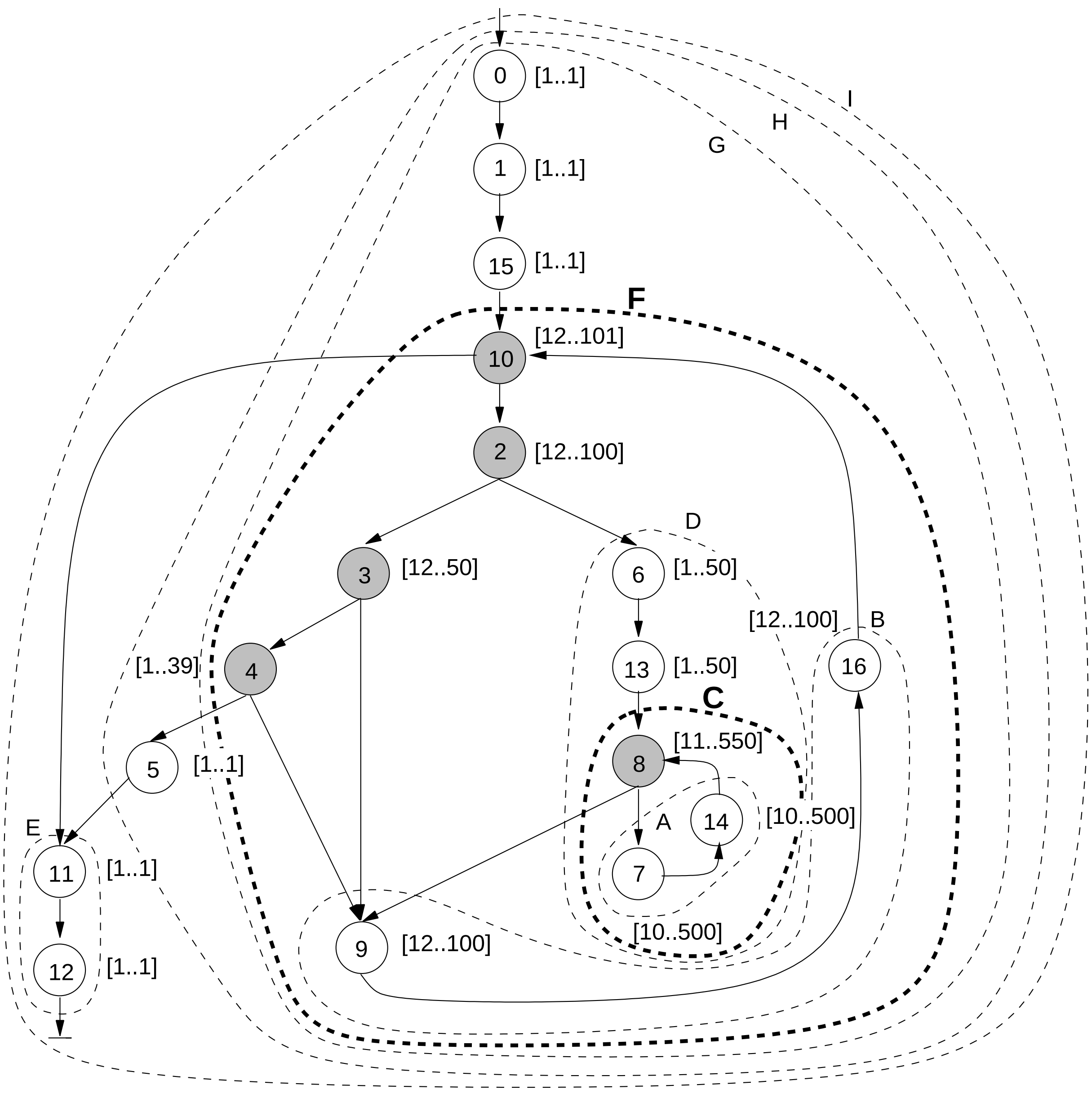


Figure 2.24: Basic block bounds.

2.8.1 Call graph reduction

Strong typing and a careful whole program analysis can help to determine at compile-time the set of possible targets for each dynamic call (i.e., virtual and interface calls).

Many techniques exist to approximate the call graph and reduce the set of possible targets for every method call. Our compiler resolves polymorphism at call sites employing a variable type based analysis (VTA) [96, 102], which computes the set of possible types of the object at each method call. The characteristic of VTA made a perfect choice for our environment: despite being flow-insensitive (but context-sensitive) it delivers good results and performs better than class type hierarchy [3, 26, 32] or rapid type analysis [3] and, more important, needs only one iteration so it scales linearly with the size of the program.

```
void foo(int i) {
  while(i < 10) {
    // loop body
    i++;
  }
}
```

(a)

```
void foo(int i) {
  if (i >= 0) {
    while(i < 10) {
      // loop body
      i++;
    }
  }
}
```

(b)

Figure 2.25: User inserted guards.

```
public void foo(int param) {

  /* manual loop bounds */
  WCETAnnotations.maxLoopIterations(64);
  while(param-- > 0) {
    foo();
  }

  /* manual recursion depth bound */
  WCETAnnotations.maxRecursionDepth(16);
  factorial(param/4);

}
```

Figure 2.26: Manual annotations.

2.8.2 Method specialization and inlining

It is often the case that one or more of the values determining the loop termination (e.g., $init$ or $limit$, see Section 2.4.2) are dependent on one of the method actual parameters.

Although this in is not a problem for approaches based on symbolic execution, since the method will be always simulated in the right context, the lack of local information hinders our loop bounder, which is not able to compute the bounds.

A complex context sensitive intra-procedural analysis could restrict the set of possible values of the input parameters given a certain context allowing a more precise loop bounding in the same way as partial abstract interpretation does (see Section 2.3.6). The drawback of this technique is that it is not always possible to derive a small set of values for the parameters that is valid for all the possible contexts in which the method is called. On the other hand, it is instead possible, to inline the method in the caller context. In this way the loop depending on the input parameter can be analyzed in a specialized version that may allow the compiler to compute the bounds automatically.

If the call site, where the method has to be inlined, has more than one possible target (i.e., the type of the object is not known) the compiler inserts guards to guarantee the correct execution (see Figure 2.27).

We inline and specialize a method in the caller context if one or more loops is dependent from an actual parameter. The maximum inlining depth is bound by a user specified parameter.

Figure 2.27 shows an example: the method `foo` of Figure 2.27.a is inlined in the caller context in Figure 2.27.b with an additional guard to check for the type of the object of the callee. The loop bound can then be easily computed since its upper bound is known in the caller context.

```
public class B extends A {

  public void foo(int limit) {
    int i;
    for(i = 1; i <= limit; i++) {
      // loop body
    }
  }

}

[...]
public bar(A o) {
  o.foo(42);
}
[...]
```

(a)

```
public class B extends A {

  public void foo(int limit) {
    int i;
    for(i = 1; i <= limit; i++) {
      // loop body
    }
  }

}

[...]
public bar() {
  if (o instanceof B) {
    int i;
    for(i = 1; i <= 42; i++) {
      // loop body
    }
  } else {
    o.foo(42);
  }
}
[...]
```

(b)

Figure 2.27: Method inlining: the method `foo` in (a) is inlined in the method `bar` using a guard to guarantee the correct execution depending on the type of the object `o`

2.8.3 Recursion

Recursive methods create loops in the call graph which must be bounded to be able to compute the WCET of the program, but the automatic computation of the maximum recursion depth is inherently more complex than loop analysis: Recursion can happen anywhere under any condition and is normally not restricted by semantic or syntactic structures as is the case for loops.

The easiest way to handle the problem is to forbid recursion in the language and force the programmers to convert their algorithm to iterative schemes (e.g., by applying program transformation rules [82]). Real-Time Euclid [55], Real-Time Concurrent C [36] and the MARS approaches [82, 84] are examples of languages explicitly forbidding recursive subroutine calls.

Blieberger showed that direct and even indirect recursion can be constrained and safely used in real-time systems [10, 9] and, as for call sites with more than one possible target, the problem can be addressed using abstract interpretation.

As our approach approximates the WCET without path enumeration and therefore avoids solutions based on abstract interpretation we currently forbid the use of recursion and the user is required to manually convert recursive algorithms to their iterative corresponding version or to annotate them (see Section 2.7).

2.9 Discussion

At the end of the different analyses, the semantic analyzer has the final task of structuring the computed semantic information that will be later used to estimate the WCET on the given hardware. The control flow graph and the semantic tree (list of semantic regions) are embedded in the generated assembler files along with the bounds on the number of iterations for each basic block and the list of partial linear false paths.

We decided to structure our WCET analyzer dividing it into two distinct tools (the semantic and the hardware level analyzers) to increase the flexibility and the portability to different systems and architectures. This is usually uncommon for approaches based on abstract interpretation because in this case the path enumeration should be done twice (once on the high level and one on the hardware level). For this reason abstract interpretation based approaches usually perform the analysis in one pass working directly on the object or assembler code.

The local character of our semantic analyzer has, on the other hand, the advantage to be coupled with the compiler from the first stages and to rely on high level information which is normally lost on the assembler level as variables and types.

It is worth noting that although our semantic analysis in based on a Java bytecode compiler the techniques presented in this chapter are language independent. The initial structural semantic analysis allows to easily work on binaries although without variable names and types and especially in presence of pointer arithmetic, the computation of the point in time when conditional branches change direction (N) could be more difficult.

3
Hardware-Level Analysis

In addition to computing the semantic behavior of a program a WCET analyzer must compute or estimate the duration of the produced code. The duration of the instructions is dependent on the given platform (hardware and system).

The duration of an instruction, which is trivial to compute for some simple architectures where it is constant and determined by the hardware specifications [13, 78, 75, 82], is becoming more and more challenging to estimate with the increase of the complexity of modern hardware. On modern CPUs the duration of an instruction depends on several factors such as stalls in the pipelines, cache hits or misses, and branch prediction. In other words, the duration of an instruction cannot always be computed offline but heavily depends on the context where the instruction is executed.

3.1 Related work

The most simple and conservative approach to compute the effects of modern hardware (notably caches and pipelines) would be to always consider the worst-case scenario: cache reads always resolve in misses, branch prediction is always wrong, and all the instructions are dependent on each other generating a maximum number of pipeline stalls. This resulting WCET estimation, computed considering the serial execution of the given instructions only, would be formally correct but overestimated by several orders of magnitude, making it useless [65]. Lundqvist has recently shown that serial execution does not exactly correspond to the WCET of the application because of timing anomalies [67]: a cache hit could in some cases generate a longer execution time than a cache miss.

Like for the semantic analysis, there are two main ways to attack the problem: methods based on path enumeration (i.e., based on symbolic execution) and methods relying on data flow analyses. Approaches using symbolic execution usually (but not always) integrate the semantic analysis to simulate the code only once [31], while data flow based WCET analyzer can more easily be divided into two distinct tools [41]. The latter approach is pursued here.

3.1.1 Caches and pipelines

Caches and pipelines are the major aspect of modern processors that influences the duration of an instruction, and several groups improved their WCET estimators to include analyses trying to reduce the overestimations caused by cache misses and pipeline stalls.

The semantic analyzer from Whalley's group at the Florida University (see Section 1.2.4) was complemented by a separate hardware analyzer for the MicroSPARC I processor [41]: this included an instruction cache simulator able to categorize each instruction by its probability to generate a cache hit. They also analyzed the different program paths to understand the pipeline behavior, avoiding to iterate over loops and applied a conservative approximation while merging the pipeline information for paths inside a loop.

The first MARS method (see Section 1.2.2), which used timing equations, was extended to support the two stage pipeline of the Intel 80C188 processor [107] and to support path abstractions including instruction reservation tables [62] or instruction dependence graphs [63] to handle processors with pipelines. Kim, using the timing schema along with data-flow analyses, was able to reduce the overestimation of dynamic load and store operations [53] (the global WCET overestimation was reduced in the average by 30% with peaks of 80%).

The third MARS approach [84] which used an integer linear programming (ILP) solver (see Section 1.2.2) was extended to handle the effects of a set associative instruction and data cache by adding new rules or constraints to their set of equations [60].

Wilhelm and his group combined their WCET tools based on abstract interpretation [31] (see Section 1.2.3) with ILP to support the analysis of instruction caches [98]. Abstract interpretation is used to model the architecture's behavior while ILP is used to find the longest path using the results from the abstract interpretation pass.

All these approaches assume that the program is executed without interruption or that preemption occurs at predictable points in time. Although this is the case for many hard-real-time systems, for applications running on some mechatronic operating systems supporting preemption [15] and for soft-real-time applications running on commodity systems a hardware-level analyzer has to take into account the effects of asynchronous context switches [37, 28, 52].

Active cache management

A completely different approach to the data cache problem focuses on active management of the cache. If we can prevent the invalidation of the cache content, then execution time is predictable. One option is to partition the cache [61] so that one or more partitions of the cache are dedicated to each real-time task. In this way no conflicts between different processes can occur, and each application can be analyzed offline regardless of scheduling strategies and preemption. In addition a careful compile time analysis can also help to prefetch certain memory locations and arrange them in memory to increase the whole system's predictability [54].

These approaches help the analysis of the cache access time since the influence of multitasking is eliminated, but on the other hand, partitioning limits the system's performance [62]. Moreover, cache partitioning requires severe changes to the operating systems and possibly to the applications, making this option unfeasible for soft-real-time multimedia applications running on commodity systems.

Static cache locking is another technique that allows to make the cache access times predictable by selectively locking cache portions [81] or by locking frequently used cache lines [54, 18].

Both cache partitioning and locking strategies are useful to improve cache predictability and efficiency but they do not solve the general problem, and they show performance degradation on large systems.

Preemption

Techniques exist to take into account the effects of fixed-priority preemptive scheduling on instruction caches when the set of running processes is known in advance [57, 16].

When the set of running processed is unknown it is still possible to conservatively take into account the preemption effects by multiplying the cache and pipeline related costs due to preemption by the maximum number of preemptions that could occur [56, 74, 100]. Experiments have shown that this assumption is pessimistic (i.e., the preemption cost is much smaller) and that the overestimation can be reduced by a cache related preemption delay analysis [94].

3.1.2 Branch prediction

Branch prediction schemes further complicate the WCET analysis since the fetched instructions not only depend on the trace that is considered, but also depend on past executions (of any other program trace) and the conditional branches included in them.

While this problem can be automatically solved in abstract interpretation based approaches (if not discarded to simplify the problem, the information on past executions of a given jump on a determined trace is stored in the abstract domain representation) different strategies were developed for control- and data-flow based analyses.

Colin and Puaut modeled the prediction table as a direct mapped cache integrating branch prediction (for local predictors only) in their WCET estimator [20], while Mitra and Roychoudhury studied the effects of dynamic branch prediction on WCET analysis and were able to integrate the effects of global branch prediction schemes in their ILP-based WCET estimator [69].

3.2 Statistical approaches

Hardware used in embedded systems is becoming more and more complex. Large mechatronic systems or soft-real-time multimedia systems are often developed using commodity processors (e.g., Pentium class or PowerPC processors) that are normally used in desktop systems.

Today's fast hardware and the increasingly dropping cost of memory contributes to the growth of the size and complexity of the real-time applications that are run today. These applications are composed of several thousands of line of code and are sometimes written in modern object-oriented languages making extensive use of dynamic data structures.

On the other hand the environment where these applications are deployed does not always require hard-real-time guarantees and therefore in the majority of the cases—especially for multimedia applications—the timing requirements are softened.

The XO/2 system [14, 15], developed at the Institute of Robotics at ETH Zürich for large mechatronic applications, is one example of this development. The system runs on PowerPC processors with two levels of (large) caches, pipelining with instruction reordering and two levels of branch prediction. The system's deadline-driven scheduler with admission testing is preemptive and supports the dynamic loading and unloading of tasks or processes at run time.

It was clear from the beginning that none of the available techniques to perform a WCET estimation are the ideal tools to be used in such an environment: the dynamism of the XO/2 software and hardware hinders a precise offline WCET analysis.

We therefore implemented a WCET estimator based on a mixed approach: the semantic attributes of the control-flow and call graphs were computed precisely while some components of the instruction duration as the effects of the caches were determined heuristically [21].

3.2.1 XO/2 WCET estimator

The XO/2 WCET estimator is implemented as a module for the XO/2 Oberon-2 PowerPC compiler [71]. It is able to perform a simple but effective loop analysis restricted to loops with a single exit point and a unique induction variable (the user has the additional possibility to manually annotate the code to specify the bounds).

The novel aspect of the approach is the hardware analysis (i.e., the estimation of the instructions durations). The technique is a trade-off between a precise approach, which requires a lot of resources and a complete knowledge of the scheduled tasks, and heuristic approaches completely based on measurements.

Given the precise computation of the maximum number of block iterations, the duration of an instruction is approximated by mixing the static characteristics of the processor, given by the manufacturer, and the dynamic components of the time needed to execute it, determined experimentally with an on-chip performance monitor (see Section 4.3).

To compute the duration of an instruction, normally expressed using the *cycles per instruction* (CPI) metric, we divide the instructions duration into its major components to achieve a better granularity.

As defined by Emma [30], the total CPI for a given architecture can be expressed as the sum of an infinite-cache performance and the finite-cache effect (FCE). The infinite-cache performance is the CPI for the core processor under the assumption that no cache misses occur, and the FCE accounts for the effects in cycles of the memory hierarchy. On the program level the FCE can be expressed as:

$$FCE := (\text{miss penalty}) \cdot (\text{miss rate}). \tag{3.1}$$

Since the memory effects are one of the CPI components that are derived from run-time measurements we had to adapt our model to the information that the PowerPC performance monitor can gather (see Section 4.3.2). The performance monitor considers the instruction cache miss penalty as part of the instruction unit fetch stall cycles and not a cache miss forcing us to include data references only in FCE.

The instruction duration separation into finite- and infinite-cache performance is not enough to obtain a good grasp of the processor usage by the different instructions, and we therefore further divide the infinite-cache performance in busy and stall time:

$$CPI := \underbrace{busy + stall}_{\text{Infinite cache performance}} + FCE. \tag{3.2}$$

where the $stall$ component corresponds to the time the instruction is blocked in one the pipelines due to some hazard (e.g., a data dependence).

The PowerPC processor has a superscalar pipeline and may issue multiple instructions of different types simultaneously (one floating-point, one branch, one compare, one memory and three integer operations). For this reason we differentiate between the stall and busy component for each execution unit; let $parallelism$ be the average number of instructions that are simultaneously in the respective execution units, $stall_{unit}$ be the number of cycles the given instruction stalls in the execution unit, $stall_{pipeline}$ be the number of cycles the instruction stalls in the other pipeline stages (fetch, decode, dispatch, completion or write-back), and $exec_{unit}$ the number of cycles spent in an execution unit. The CPI for an instruction can then be expressed as:

$$CPI := \frac{exec_{unit} + stall_{unit}}{parallelism} + stall_{pipeline} + FCE. \tag{3.3}$$

For each instruction of a basic block we use Equation 3.3 to approximate the instruction's dynamic duration. The difficulty lies in obtaining a correct mapping between the static description of the processor and the dynamic information gathered with the runtime performance monitor to compute the components of Equation 3.3. The details are explained in the following paragraphs.

Finite cache effect

The PowerPC 604e performance monitor delivers precise information about the mean number of load and store misses and the duration of a load miss. Because of the similarity of the load and store operations [49], we can work with the same miss penalty and compute the FCE, in cycles, as follows:

$$FCE := (miss\ rate_{load} + miss\ rate_{store}) \cdot penalty_{load}. \tag{3.4}$$

Pipeline and unit stalls

Unfortunately, the performance monitor does not provide precise information about the stalls in the different execution units and the other pipeline stages (the performance monitor was not designed to be used for program analysis, and our predictor had to map the data gathered to something useful for the WCET estimation).

The first problem arises in the computation of the mean stall time in the different execution units as the performance monitor delivers the mean number of instruction dependences that occur in the reservation stations of the execution units only (a reservation station is a small two-entries queue in front of each execution unit). This number is not quite useful, because a dependence does not necessarily cause a stall, and we must find other ways to compute or approximate the impact of stalls.

The mean stall time for instructions executed by a single-cycle execution unit is easily computable as the difference between the time an instruction remains in the execution unit and its normal execution duration (i.e., 1). Let $idle_{unit}$ be the average measured idle time in a cycle (for an execution unit), and $load_{unit}$ be the average measured number of instructions handled in a cycle (at most 1):

$$stall_{unit} := \frac{1 - idle_{unit}}{load_{unit}} - 1. \tag{3.5}$$

Unfortunately the execution time of an instruction in a multiple-cycle execution unit is not constant, and Equation 3.5 cannot be used. Thanks to Little's theorem applied to M/M/1 queues we can compute the mean length of the reservation station queue N_q (at most 2) and then approximate the mean stall time $stall_{unit}$ as follows (let $\rho := 1 - idle_{unit}$ be the unit's utilization factor (arrival/service rate ratio), and p_{dep} the reported number (probability) of dependences):

$$N_q := \max\left(\frac{\rho^2}{1-\rho}, 2\right) \tag{3.6}$$

$$(1 - stall_{unit} \cdot load_{unit})^{N_q} = 1 - p_{dep} \tag{3.7}$$

$$stall_{unit} := \frac{1 - \sqrt[N_q]{1 - p_{dep}}}{load_{unit}}. \tag{3.8}$$

We assume that an absence of reported dependences corresponds to the absence of stalls in the execution unit (Equation 3.7).

A similar problem is present in the computation of the mean number of stall cycles in the dispatch unit: Seven different types of stall are reported for the whole four-instruction queue. We have no way to know how many instructions generate stalls, or how many different stalls are caused by the same instruction. These numbers, however, are important for the understanding of the dynamic instruction durations, because they include the effects of instruction cache misses and influence all the processed instructions. Several experiments with a set of simple well-understood test programs revealed that a reported stall can be considered as generated by a single instruction. The test programs were specially crafted by including different data dependencies to generate pipeline stalls and observe their effects on the performance counters. With p_{event} the reported number of stalls of a given type and $\prod_{event}(1 - p_{event})$ the probability no instruction stalls, we can compute the probability that an instruction stalls:

$$stall_{dispatch} := \frac{CPI \cdot \left(1 - \prod_{event} (1 - p_{event})\right)}{queue\_size} \tag{3.9}$$

The stalls in the dispatch unit ($stall_{dispatch}$) also consider stalls in the fetch stage and can be substituted for $stall_{pipeline}$ in Equation 3.3.

The need to heavily rely on statistics and queuing theory in the approximation of the pipeline behavior is determined by the granularity (or lack of it) of the hardware measurement tools we have at our disposal. Nevertheless, experimentation with small synthetic kernels to stress pipeline stalls shows the soundness of our mapping from the measured dependences to the actual stalls.

Execution duration

The normal execution duration (i.e., the execution time without stalls) of an instruction for single-cycle units is known from the processor specifications, but the multiple cycle units (FPU, MCIU, and LSU: Floating Point Unit, Multiple Cycle Integer Unit, and Load Store Unit) have internal pipelines that cause different throughput values depending on the instruction type. As a consequence there is no way to know the duration of a given instruction, even if the caches and pipeline stalls are not considered. The presence of preemption, and consequently the lack of knowledge about the pipeline status, forces us to approximate this value.

The architecture specifies, for each instruction, the time to execute it ($latency$) and the rate at which pipelined instructions are processed ($throughput$).

We define d_{unit} as the approximation of the maximal distance in the instruction flow between two instructions that can be pipelined by the same execution unit. We then consider an instruction pipelined, in the execution unit, if there was an instruction of the same type in the last d_{unit} instructions ($exec_{unit} := thoughput$), otherwise we consider the instruction duration to correspond to the integral computation time ($exec_{unit} := latency$); the value of d_{unit} is computed experimentally and varies—depending on the unit—from four to eight instructions.

Since the XO/2 instruction-duration analysis is done on a basic block basis, the approximation for the first instruction of a given type in a block is computed differently:

$$p_{instruction} := 1 - (1 - load_{unit})^{d_{unit}} \tag{3.10}$$

$$exec_{unit} := \begin{cases} latency & \text{if } p_{instruction} < threshold \\ throughput & \text{if } p_{instruction} \geq threshold \end{cases} \tag{3.11}$$

Where $p_{instruction}$ is the probability that a given instruction type was present in the last d_{unit} instructions and $threshold$ is an experimentally determined value representing the probability that a given instruction will be found in d_{unit} instructions.

To check how this approximation works in practice and to refine the d_{unit} and $threshold$ values, we compare for some test programs known mean instruction durations, obtained with the runtime performance monitor (see Section 4.3), with the predicted ones, confirming the soundness of the method.

Instruction parallelism

We compute the mean instruction parallelism ($parallelism$) solving Equation 3.3 and consider mean values over the whole task execution, due to the impossibility to get finer grained information. Furthermore, the task cannot be interrupted (or instrumented) to check the performance monitor; otherwise interference in the task's flow would destroy the validity of the gathered data. The FCE is taken into account only for the instructions handled by the load/store unit ($util_{unit}$ corresponds to the utilization factor of the given unit).

$$parallelism := \frac{exec + stall_{unit}}{CPI - stall_{dispatch} - util_{LSU} \cdot FCE} \tag{3.12}$$

To compute the mean instruction's execution duration $exec$ (needed by Equation 3.12), the mean stall value computed earlier can be used. We first compute the mean execution time of the instructions handled by a given execution unit ($exec_{unit}$)—in the same way as the stalls were approximated for the single cycle units.

$$exec_{unit} := \begin{cases} 1 & \text{single cycle units} \\ \frac{1 - idle_{unit}}{load_{unit}} - stall_{unit} & \text{multiple cycle units} \end{cases} \tag{3.13}$$

Finally, we compute the total mean instruction duration ($exec$) as the weighted mean of all the instruction classes:

$$exec := \sum_{unit} (util_{unit} \cdot exec_{unit}) \tag{3.14}$$

The duration of each instruction of a basic block can then be computed by applying Equation 3.3 with all the computed elements ($exec_{unit}$, $stall_{unit}$, $parallelism$, $stall_{pipeline}$ and FCE).

Heuristic and static information

The duration of an instruction is therefore computed using a mix of information derived from the CPU and system documentation, and data collected during some test runs of the application.

The latency and throughput of the different instructions along with the first level cache miss penalty are described in the CPU manual while the miss penalty of the second level of cache can be derived from the system description.

At run time we measure the first and second level cache hit probability ($miss\ rate$), the time the different execution units are idle ($idle_{unit}$), their load ($load_{unit}$) and their utilization factor ($util_{unit}$).

Since, contrarily to the WCET measurement by experiment, we do not sample the application running time but we only measure the application's behavior (average cache usage and pipeline stalls), we do not need many measurements to generate sound WCET estimations. It is thus not important to cover all the possible executions (a task that is impossible for many applications) but we must only gather information on the pipeline and cache usage of the given process.

Results

Testing the soundness of a WCET predictor is a tricky issue. To compare the computed value to a measured maximum execution time, we must force execution of the longest path at runtime. Such control of execution may be difficult to achieve, especially for big applications, where it may be impossible to find the WCET by hand. By consequence the WCET that we can experimentally measure is an estimation and could be smaller than the real value.

We tested our approximator with several simple tasks with a known and measurable WCET and with some bigger applications running on mechatronic systems based on the XO/2 system (see Table 3.1).

Task	Measured time	Predicted time	Overestimation
Matrix multiplications	279.577 ms	310.533 ms	+11%
Matrix multiplications FP	333.145 ms	351.753 ms	+6%
Array maximum	520.094 ms	555.015 ms	+7%
FP array maximum	854.930 ms	814.516 ms	-5%
Runge-Kutta	439.905 ms	495.608 ms	+13%
Polynomial evaluation	1 251.194 ms	1 187.591 ms	-5%
Distribution counting	2 388.816 ms	2 579.051 ms	+8%
LaserPointer::Watchdog	290 μs	287 μs	-1%
LaserPointer::Planner	297 μs	298 μs	0%
LaserPointer::Controller	730 μs	1 071 μs	+47%
Robojet::TrajectoryQ	8 μs	8 μs	0%
Robojet::TrajectoryN	339 μs	399 μs	0%
Robojet::TrajectoryX	164 μs	183 μs	+12%
Hexaglide::Learn	65 μs	65 μs	0%
Hexaglide::Dynamics	286 μs	451 μs	+58%

Table 3.1: WCET for some sample XO/2 applications.

For the majority of the tests with a known and measurable WCET (first part of the table) we achieve a prediction within 10% of the real value, and we avoid significant underestimations of the WCET. The second part of the table presents the WCET estimation and measurement for the real-time kernel of some real applications. *LaserPointer* is a laboratory machine that moves a laser pen on the tool-center point of a 2-joints (2 DOF) manipulator. *Hexaglide* is a parallel manipulator with 6 DOF used as a high speed milling machine [46] (the two real-time tasks we profile contain a good mix of conditional and computational code). *Robojet* is a hydraulically actuated manipulator used in the construction of tunnels [47] (we profile three periodic real-time tasks that compute the different trajectories and include complex trigonometric and linear algebra computations). The overestimation for some of the mechatronic kernels lies in the fact that the measured time is not the real WCET, which is unknown, but only the highest duration that we are able to measure. Trigonometric functions are a good example of the problem: The predictor computes the longest possible execution, while, thanks to optimized algorithms, different durations are possible at runtime. It is obviously difficult, or often even impossible, to compute an input set such that each call to these functions takes a maximum amount of time. We must therefore take into account that the experimentally measured maximum duration is probably smaller than the theoretical maximum execution time.

More details about the implementation and a more comprehensive analysis of the results can be found in [21].

In summary, for the real applications, our technique produces conservative approximations yet avoids noticeable underestimations of the WCET. Inaccuracies in the WCET estimation can only be caused by the approximation of the effects of caches and pipelines but not on the wrong understanding of the semantics of the program. These results demonstrate the soundness of the method for real-world products showing that a mixed approach with a precise semantic analysis and an approximated hardware analysis can safely compute good WCET estimations.

3.3 The road to partial trace simulation

The implementation of the WCET estimator for the XO/2 system described in Section 3.2.1 shows that a mixed approach (with a precise semantic analysis and an approximation of the instruction duration estimation) allows to analyze systems with a dynamic set of processes and preemptive multitasking delivering sound results for complex soft-real time applications.

To broaden the applications that we could use for the tool evaluation we refined both the semantic analyzer (as described in Chapter 2) and the hardware analyzer, reducing the amount of statistical information needed to compute the estimation (see Section 3.4). We also moved to a Linux platform on Intel processors and focused on the WCET estimation of fairly large soft-real-time Java applications running on commodity hardware (e.g., multimedia applications). This new hardware and software environment presents the same set of problems as the XO/2 system: the system is highly dynamic, and the set of running processes along with the preemption points are unknown. On the other hand the new environment offered us a larger amount of applications and did not require hard constraints on the computed WCET.

3.4 Partial trace emulation

As described in Chapter 2 our worst-case execution time approximator is composed of two tools: a machine independent semantic analyzer and a language independent instruction duration estimator.

The analyzer produces commented assembler files that, in addition to the native code, are enriched by additional semantic information: an annotated syntax tree for each method, the bounds of each basic block, the list of false paths, and the set of targets of each call. The hardware level estimator can then, from the input assembler files, easily reconstruct the structure of the program.

3.4.1 Locality

Our approach tries to compute a sound estimation of the instruction duration without precise and complete knowledge of the set of running processes on systems which support task preemption. For this reason we cannot predict when the analyzed process will be interrupted and therefore we cannot predict the state of the caches and pipelines when the process will be selected again for running.

As for the semantic analysis we have therefore decided to approach the problem locally without a global analysis and without simulating all the possible paths in an abstract domain.

Our idea is based on the principle of locality, limiting the effects of an instruction to a limited period of time: We assume that the effects of an instruction on the pipeline and caches will fade over time and that they will be no longer relevant after a certain number of executed instructions. In other words, the effects on the caches and pipelines of an instruction i in a given program trace are relevant only for the limited set of instructions following instruction i (the number n of relevant instruction is arbitrary and defined heuristically).

For every possible execution trace going through an instruction i we therefore define a *partial trace* $t(i, n)$ as the set of the last n instructions prior to i on that particular trace. Using the principle of locality, a partial trace also defines the set of instructions that could influence the duration of i on that given execution trace.

Although it would be more straightforward to use time (cycles or seconds) to define the length of a partial trace (instead of the number of instructions) that compose it, to do this we would need the duration of the various instructions that we still have to compute. The length of a partial trace does not directly influence the result of the WCET computation and only a rough estimate is necessary to build partial traces long enough to exploit the principle of locality. For this purpose we consider the number of executed instructions as directly proportional to the time it will take to execute them.

According to this definition, if we consider a particular path in the program and an instruction i (see Figure 3.1) only the instructions that are part of the partial trace $t(i, n)$ (in gray in Figure 3.1) can influence the duration of i. It also follows that everything that is before the instruction i_s in this particular path can be ignored for the computation of the duration of i.

We simulate the CPU behavior (pipeline and instruction cache) for the n instructions in the partial trace t. The duration of i given the partial trace t ($duration(i, t)$) can then be computed by looking at the time, in cycles, needed to execute the instruction at the end of the short

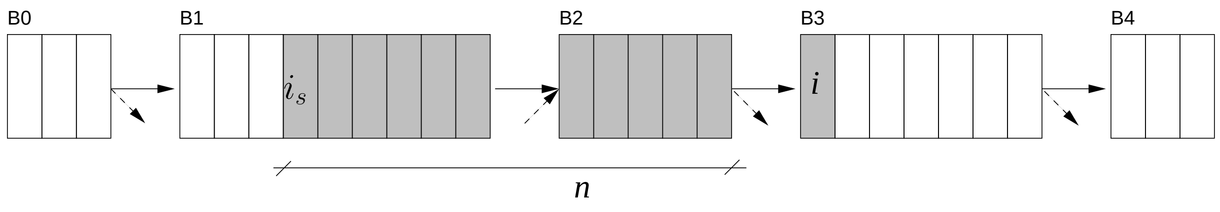


Figure 3.1: Partial trace and principle of locality.

simulation (see Section 3.4.2). The first $n-1$ instructions are used to warm up and initialize the simulator, which can then time the execution of the last instruction in the right context.

We define $T(i,n)$ to be the set of unique partial traces of length n that end in i (i.e., i is the last instruction of the partial trace) and we define the WCET of an instruction i as:

$$WCET(i) := \max_{t \in T(i,n)} (duration(i,t)) .$$

I.e., we compute the duration of i on every partial trace of length n which ends on i, and we conservatively consider the worst case only. The partial traces in the set $T(i,n)$ are limited in number by n and can be computed by a simple backward path enumeration pass (limited up to a certain path length).

To limit the number of traces that must be simulated, some optimizations are possible: Partial simulation results (e.g., the state of the pipeline) are cached and can be reused if the initial part of two traces is the same (this is particularly effective if we force traces to begin at basic block boundaries). Furthermore the same trace simulation can be extended to the following instructions as long as no jumps are encountered. We call the inclusion of all the instructions in the same block as the first instruction of the trace t and all the instructions in the same block as the last instruction of the trace t a *partial trace extension of* t.

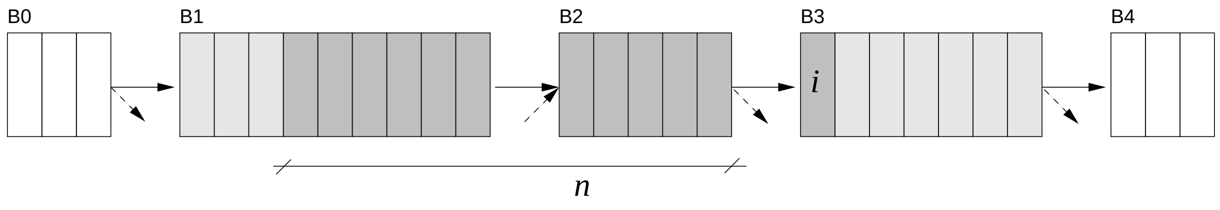


Figure 3.2: Trace extensions.

Figure 3.2 shows an example of the trace extensions: the trace $t(i,11)$ (dark gray instructions) is extended in the past to the beginning of block B1 and in the future to the last instruction of block B3 (light gray instructions).

This trace extension adds a small number of instructions to the trace but on one hand increases the probability that the initial part of the trace simulation could be used again (there might be more than one partial trace beginning in the same block). On the other hand, thanks to the additional blocks at the end, we can safely continue to simulate the code and record the duration of the whole block (B3). This is possible since no jumps or jump targets can disturb the flow of execution (which is guaranteed to be uninterrupted until the end of the block).

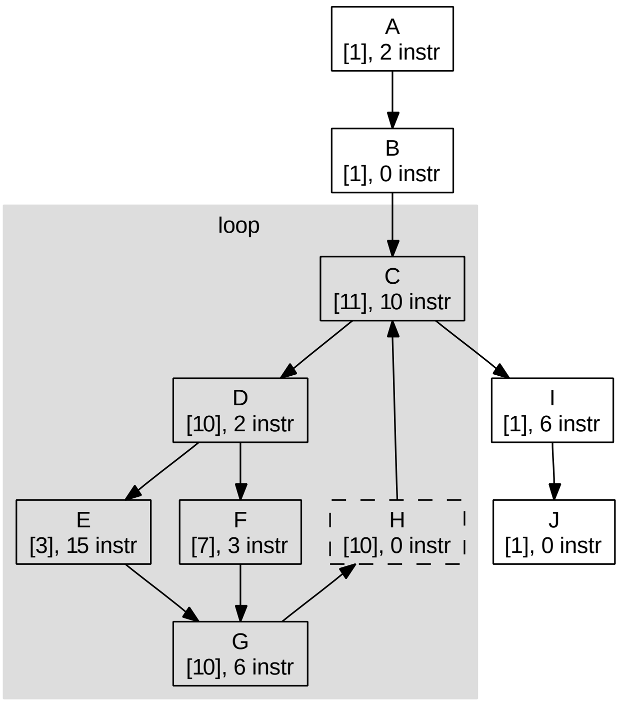


Figure 3.3: Partial traces: for each basic block we depict the name, the range of possible iterations in brackets and the number of instructions composing the block. The shaded area represents a loop.

Figure 3.3 shows an example control-flow graph (for each node the number of iterations is shown together with the number of instructions in the basic block). The set $T(H, 23)$ represents all the unique partial traces ending in the first instruction of block H with a length of 23 instructions. In this example $T(H, 23)$ is composed of HGFDCBA, HGED, and HGFDCHG. In addition we assign to each trace a ***weight*** w based on how many times it will be executed:

$$w(t \in T) := \frac{\min\limits_{b \in t} (\max (iter(b)))}{\max\limits_{b \in T} (\max (iter(b)))}.$$

The weight is computed using the minimum number of iterations of all the blocks that compose it and does not necessarily correspond to the actual number of times the trace will be executed. In our example: $w(\text{HGFDCBA}) = 1/11$, $w(\text{HGED}) = 3/11$, and $w(\text{HGFDCHG}) = 7/11$. Note that in our example the tool is not able to determine if the partial path HGED will be executed in the first cycle and is therefore part of HGFDCBA.

3.4.2 Partial trace simulation

To simulate the instruction cache and pipelines behavior of a partial trace we implemented a basic cycle precise simulator using a simplified Intel Pentium II processor [50] model (see Figure 3.4). For every clock tick we evaluate the state of the different CPU units keeping track of the instructions we are interested in. The following paragraphs describe which processor elements are considered in the model and how our simulator handles them.

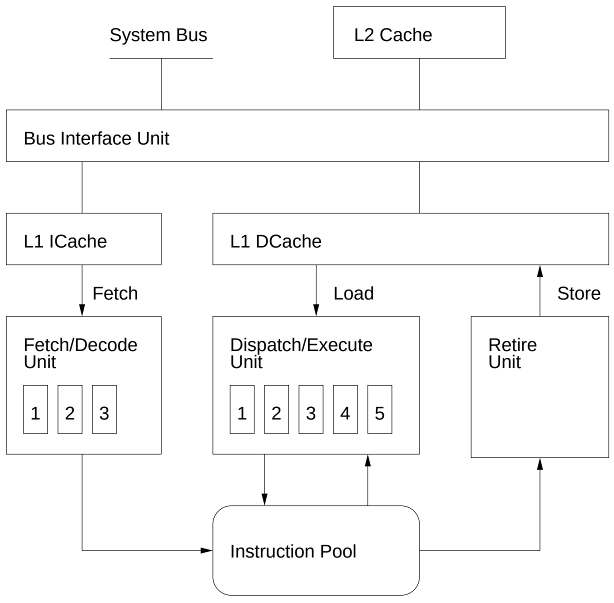


Figure 3.4: Simplified CPU model.

Pipeline

The Pentium Pro pipeline can be broadly divided in three main stages: a fetch and decode phase, a dispatch and execute phase and finally a retire phase (see Figure 3.4).

Fetch and decode. The fetch and decode unit fetches one 32-byte cache line at atime from the instruction cache and decodes it to the internal micro-ops. Every IA-32 instruction is decoded to one or more pre-programmed micro-ops which are then passed to the instruction pool. The unit can decode up to six mirco-ops per clock tick.

Instruction pool. The instruction pool holds from 20 to 30 micro-ops, which are then dispatched to the correct execute unit in any order. The Pentium processors are able to dispatch micro-ops out-of-order using Tomasulo's algorithm[99] to minimize the stalls caused by data dependences.

The instructions are then dispatched to several execution units divided among five ports (pipelines). Several execution units can execute micro-ops on each pipeline, but only one micro-op per cycle can be fed to a port.

To be able to correctly simulate the dispatching mechanism we therefore need to know the dependences (i.e., the def-use chains) among the micro-ops on the analyzed path. Since paths are simulated in isolation the simulator can simply keep track of the dependences during the traversal.

Execution. The five execution pipelines processes the instructions according to their type. Since the execution of some operations can depend on the values of the operands we consider the worst-case scenario.

Retire unit. The retire unit has the task to reorder the micro-ops and can process up to three instructions per clock tick.

Memory operations

Partial trace simulation does not allow us to simulate the data cache behavior since at the beginning of the instruction sequence (i.e., the partial trace) the memory and data cache states are unknown. For this reason we do not currently model data dependences in memory and we assume that a write to a store buffer will not influence the instruction duration. Assuming enough memory to handle the process and assuming that no swap space is present (all the processes fit in memory), our model, similarly to the XO/2 predictor (see Section 3.2.1) uses a statistical approximation of the memory accesses, computing the cost, in cycles, of a read (c_{read}) in the following way:

$$c_{read} := c_{L1_{hit}} + (1 - p_{L1_{hit}}) \cdot (c_{L2} + (1 - p_{L2_{hit}}) \cdot c_M)$$

Where $p_{Ll_{hit}}$ is the measured average probability to have a hit on the cache level l, $c_{Ll_{hit}}$ is the cost of a cache hit on level l and c_M is the cost of a memory read. The values for $c_{L1_{hit}}$,$c_{L2_{hit}}$ and c_M are determined statically by the Intel specifications [50] while the global hit probabilities for the whole program are determined empirically by measurements (see Section 4.3) performed on several test runs of the analyzed application.

Possible heuristics Even if partial trace evaluation does not allow a precise data cache simulation some heuristic assumptions can help to reduce the cache overestimation. A points-to analysis could help to identify the references pointing to a single object. The references could the be assumed to generate a cache hit if found inside loops.

Although more complex heuristics and analyses could improve the analyzer knowledge of the accessed memory, our simple approximation is sufficient to estimate the memory effects on computation intensive applications (see Chapter 5)..

Branch prediction

To estimate the effects of branch prediction we use the bounds on the maximum number of basic block iterations computed with the static semantic analysis (see Section 2.5.2). We consider the probability of a branch from a block b to a block s as:

$$p_{branch(b,s)} := \frac{max(iter_{edge(b,s)})}{max(iter_b)}$$

where $iter_{edge(b,s)}$ is the number of times we will jump from b to s, and $iter(b)$ is the number of times that block b will be executed. Comparing the expected CPU behavior described in the documentation with our predicted branch direction we can estimate how often a branch misprediction will occur.

Discussion

Our partial trace simulator is based on a simplified model of the Intel CPU and its accuracy could be improved by a complete cycle precise Pentium Pro simulator, but to the best of our knowledge no such tool is available in a form that allows its integration in our WCET estimator.

Although several aspects as the instruction reordering and the cache model are handled in a very simplified way our simple model delivers results that confirm the soundness of the approach (see Chapter 5).

3.4.3 Longest path

Once we have the worst-case execution time of each instruction we can easily compute the duration of a basic block b as the sum of the durations of all the instructions i it contains:

$$WCET(b) := \sum_{i \in b} WCET(i)$$

Since, due to pipelining, an instruction execution could be spread over the execution of two basic blocks, we define the duration of a basic block as the time between the beginning of the execution of its first instruction to the last clock tick before the beginning of the execution of the first instruction of the next block.

If traces are extended to finish on block boundaries (see Section 3.4.1) the length of a block is simply the time resulting from the simulation of the block at the end of the trace (see Figure 3.2).

To be able to compute the longest path from a method source to the method exit, we must transform the control-flow graph representation removing structural cycles: For every loop we remove the back edges from our representation and replace them with an edge to all the possible loop's exit points. We then update the weight of each basic block, multiplying it with the block's maximum number of iterations (the initial weight of a block is defined as its worst-case execution time). We obtain an acyclic representation of the control-flow graph that retains information about block iterations in the form of nodes weights. Note that cycles are removed from the representation only, and that we do not perform any loop unrolling on the actual code. Figure 3.5 shows an example: The back-edge ($4 \to 2$) of the loop (`Loop(2,3,4)`) is removed and substituted with a set of edges to the loop's exit points ($4 \to 5$ and $4 \to 6$). The weights w of the blocks are updated according to the maximum possible number of iterations of each block.

To find out the longest path from the entry to the exit block of the graph which is not a false path there are essentially two different approaches: The first one transforms the graph in such a way that false paths are removed from the graph. An algorithm that allows to do this efficiently for very large graphs has been presented by Blaauw, Panda and Das in [8]. The algorithm was conceived for the field of digital circuit optimization, where the encountered graphs are far more complex and bigger than the ones that must be dealt with in computer programs.

The second and simpler approach is to compute the k longest structural paths in the graph—where k is a user-defined parameter—and then choose the longest of these paths that is not a false path. An efficient algorithm for finding the k longest paths in a directed acyclic graph efficiently has been proposed by Yen, Du and Ghanta in [106] which is able to perform the path

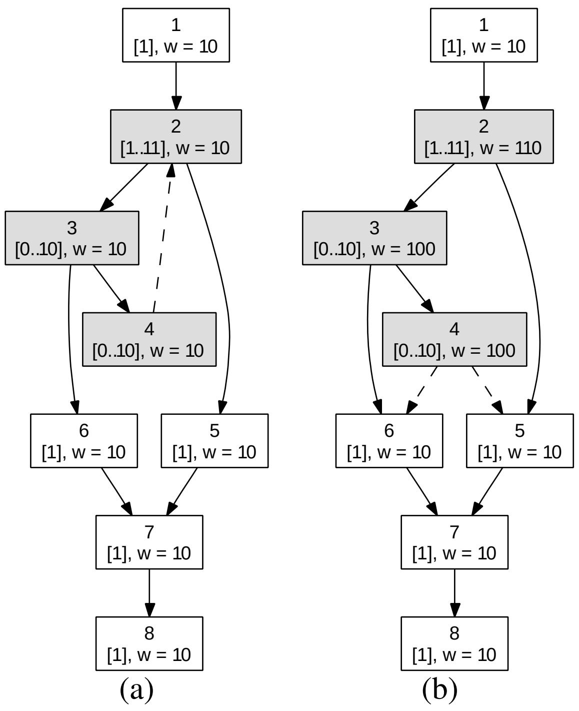


Figure 3.5: Virtual loop unrolling: for each block we depict the number, the number of possible iterations in brackets and the weight w.

enumeration in logarithmic time when k is small. The problem of this algorithm is that for our particular case, where false paths also have to be considered, there is no sensible way to define the parameter k. Depending on the particular program instance being analyzed, it could turn out that all k longest paths are in fact false paths. In such a pathological case, the only assertion that can be made is that the longest non-false path is shorter than the k-th longest path, but the actual longest path is not known. Despite this problem, the algorithm is efficient and relatively easy to implement. Since program graphs are relatively small—the number of nodes in the graph is rarely greater than a couple of hundred nodes—it is sufficient to write a modified version of the algorithm of the second approach, in such a way that no user-defined parameter k is needed (the algorithm adaptively increases the factor k depending on the presence of false paths).

Distance to the exit node In a first phase the algorithm computes the longest distance d from each basic block to the control-flow graphs exit node. The nodes are processed in reverse topological order computing the maximum distance (in time) to the exit node:

$$d(B) := WCET(B) + \max_{p \in pred(B)} (WCET(p))$$

For each node we then sort the successors according to their distance to the exit (in decreasing order). In absence of false paths the WCET corresponds to the distance from the entry to the exit node ($d(entry)$).

Path enumeration If this longest path ($P_{max} := \langle entry, b_1, b_2, \ldots, b_n, exit \rangle$) contains a false partial linear path (see Section 2.3.5) we try to find an alternative path by choosing a different successor for one the blocks b_i (see Figure 3.6). Since the number of possible paths is exponential in the number of conditional branch nodes of the graph, the algorithm only considers paths that are longer than a threshold value T.

```
$P := P_{max}$
mark all the blocks in $P$ as visited
**if** $P$ does not contain a false partial linear path **then**
   return $P$
**end if**
$T := |P|$ $\{$initialize the threshold value$\}$
$\{$backward trace$\}$
look backward for a $b_i$ which has a non visited successor
**while** $b_i$ exist **do**
   $b'_{i+1}$ is the non visited successor of $b_i$ with the highest distance from the exit node
   **if** duration of the path up to $b'_{i+1} > T$ **then**
      $\{$forward trace$\}$
      $P$ is completed by choosing the successors with the highest distance from the exit node
      $P' := \langle entry, b_1, b_2, \ldots, b_{i-1}, b_i, b'_{i+1}, b'_{i+2}, \ldots, b'_n, exit\rangle$
      mark $b'_{i+1}$ as visited
      **if** $P'$ does not contain a false linear path **then**
         $\{$update the current longest path and the threshold$\}$
         $P := P'$
         $T := d(entry)$ along $P$
      **end if**
   **end if**
   $\{$backward trace$\}$
   look backward for a $b_i$ which has a non visited successor
**end while**
**if** $P$ does not contain a false linear path **then**
   return $P$
**else**
   $\{$all the paths are infeasible$\}$
   return $\emptyset$
**end if**
```

Figure 3.6: Finding the longest non-false path.

Method calls. Method calls are handled by traversing the call graph in depth-first order and inserting the maximum duration of all the possible callees at the corresponding call site.

3.4.4 Complexity

The limited length of the user-chosen partial traces length limits the number of paths, and consequently the number of instructions, that we must consider for simulation. For a program with N instructions, B basic blocks and a partial trace length of n, our analysis has to simulate a maximum of

$$O\left(2^{\left(\frac{\frac{B\cdot n}{N}-1}{2}\right)}\cdot B\cdot n\right)=O\left(2^{\left(\frac{B\cdot n}{N}\right)}\cdot B\cdot n\right)=O(B)$$

steps. Where $\frac{B}{N}\cdot n$ is the number of blocks in a trace and $2^{(B/N\cdot n-1)/2}$ is the number of partial traces to be simulated. Since n (the length of a partial trace) and $\frac{N}{B}$ (the average number of instructions per basic block) are constant, the total running time is in the asymptotic case linear.

The small number of paths is a consequence that considering partial traces, instead of real program traces, the time needed to simulate the code does not depend on the number of loop iterations, which normally contributes to the exponential path explosion problem, but depends only on the number of conditional branches in the code (a static property).

3.5 Summary

In this chapter we presented one possible backend for our semantic WCET analyzer. The dynamism of our environment suggested us to approximate the computation of the instruction duration on the given hardware platform using simulations of short program trace snippets.

We have shown on two different platforms (XO2 on PowerPC [21] and Linux on Pentium Pro [22]) that a mixed approach that includes heuristic information or that approximates the state of the various CPU components can safely estimate the WCET for soft-real time processes.

The approximations performed by the hardware-level analyzer can only have a limited impact on the total WCET estimation since the bounds on cyclic structures are safely computed and an eventual over- oder underestimation is only dependent on the caches and pipeline stalls approximations.

4 Implementation

This chapter presents issues related to the chosen software and hardware platform: In particular we describe the challenges derived by the choice of a modern object-oriented language like Java (see Section 4.1) and the implementation of our analyzer as part of a general purpose native-to-bytecode compiler (see Section 4.2). Finally we describe the software and hardware infrastructure we use to measure the processor and system performance and behavior (see Section 4.3).

4.1 The Java language

The importance of the Java language in the real-time is quickly growing [45]: despite its advanced memory management and high dynamism (with class loading and unloading at runtime), the Java language has a lot of characteristics that make it a good choice for embedded and real-time programming. The language, as many other modern languages, offers, for example, an environment without pointer arithmetic (thanks to enforced strong typing) and with automatic array index checks. These two features alone help the embedded programmer to write safer code and to enhance predictability: the absence of pointer arithmetic strongly reduces aliasing effects improving the results of automatic analyzers.

In the last years three proposals for real-time programming with the Java language originated from different industry groups: the first was drafted by Kelvin Nielsen from NewMonics and the National Institute for Standards and Technology (NIST) [76, 1], the second by Greg Bollella from IBM and the J-Consortium (a group of companies lead by HP and including NewMonics) [38] and the last one by a Java expert group chartered under the Java community process lead by Sun, working on real-time Java extensions (this group also included several other companies, some of them involved in the J-Consortium as well).

Despite some differences in the proposals—some industries wanted a heavily modified language with some ad hoc constructs for hard-real-time programs, while Sun insisted in keeping the Java spirit and did not want any change in the language itself—the three groups merged in 1998 and their joint efforts resulted in a set of requirements which led to the *Real-Time Specification for Java* (RTSJ) [11, 12].

This section gives a short overview of the real-time Java specifications explaining the position of our approach in respect to the current deployment of Java in the embedded world.

4.1.1 Scheduling and event handling

The RTSJ does not specify a specific scheduling strategy (although every implementation is required to have at least one implementation of a real-time scheduler) but emphasizes the need of *timely execution of threads*. Each implementation should also provide a lock mechanism implementation that prevents priority inversion allowing a real-time safe synchronization. The Java exceptions mechanism is also extended to allow applications to programmatically change the point of control of another thread.

The requirement for real-time scheduling emphasizes the need of predictability of the real-time Java process. This aspect played a central role in the J-Consortium specification drafts (they contained strict restrictions on the code to make it analyzable) but lost importance in the final specifications.

4.1.2 Memory management

The RTSJ provides several extensions to the memory management to overcome the unpredictable latencies of garbage collectors. The RTSJ introduces the concept of memory areas, differentiating several allocation (and deallocation) strategies. In addition to the classical garbage collected objects it is possible to allocate immortal objects (never removed) and to allocate scoped objects (their lifetime is defined by a syntactic scope).

Immortal objects can be safely used by a real-time application since no external garbage collection mechanism will ever interrupt the execution, but on the other hand a careful offline analysis is needed to ensure that the applications will never allocate more memory than available (see Section 5.10).

4.1.3 Our approach

With this thesis we want to show that it is possible to safely approximate the maximum time a soft-real-time Java application needs, for large real-time applications (including soft-real-time multimedia applications). For this reason instead of focusing on a reduced set of the Java programming language, or focusing on a specific RTSJ implementation, we decided to work on unmodified single-threaded Java code.

We therefore target unrestricted Java applications emphasizing the generality of the analysis as we want to handle in addition of real-time Java programs written with timeliness in mind, normal multimedia applications.

4.1.4 Multi-threading

The presence of more than one thread of the same application on a preemptive system has several implications for a WCET analyzer.

First of all the semantic analyzer (see Chapter 2) must take into account that another thread could change object fields at any time: As we consider only local variables for our loop bounding analysis (see Section 2.4.2) this does not directly affect our algorithm (but an extension of it that considers object fields as possible loop control variables would have to perform escape and alias analyses to ensure the consistency of the data).

Resource locking, on the other hand, makes the computation of one thread's WCET, in the general case, impossible. The time a thread would have to wait for a shared resource does not only depend on conditions known at compile time, and therefore the presence of locking makes an automatic analysis very difficult. The main difference between the lock analysis and the loop bounds problem, which also suffers from the undecidability of the halting problem, is that while loops frequently depend on local variables, locks involve shared resources often related to the underlying system. For this reasons our analyzer does not support locking constructs but our loop bounding analysis, which does not consider object fields for loop termination, is robust with respect to multi-threading.

The presence of multiple threads, on the other hand, does not affect the hardware-level instruction duration analyzer (see Chapter 3), which already considers the presence of preemption.

4.2 The compiler

Our WCET estimator is embedded as a module in a general purpose research Java ahead-of-time bytecode to native compiler. This tight integration allows us to take advantage of the many analysis and optimizations that are not normally employed in a standalone WCET analyzer but that can greatly improve the results.

An example are the intermediate representation in static single assignment form [25] on which we perform a classical set of optimizations (e.g., constant folding, copy propagation, common subexpression elimination) and the advanced call graph reduction algorithms (see Section 2.8.1). These whole-program analyses, although not directly related to the timing analysis, help on hand to gather preciser information on the loop induction's variables and on the other hand to reduce the set of possible targets for each method call. As an example, we were able to reduce the number of unbounded loops from 12% for the Linpack benchmark to 57% for the JavaLayer application (see Section 5.2.1) by propagating and folding constants only. The variable type based analysis, was able to reduce the average number of targets per method calls by around 5%.

The WCET analysis is performed as a last step just before the code generation phase to ensure the consistency of our analysis with the produced code (i.e., the control-flow and call-graphs will not be changed).

4.2.1 Memory management

Our bytecode-to-native compiler relies on parts of the *GNU Compiler for the Java Programming Language* [33] (version 2.96) for the runtime system: namely memory allocation, garbage collection, and thread management. The compiler also relies on the Classpath libraries [34] for the implementation of the core Java classes.

We are therefore not able to analyze JVM functionalities as memory management or the native implementation of part of some core classes. Characteristics of native methods (memory and WCET) must be consequently manually specified with code annotations (see Section 2.7).

4.3 Hardware-level measurements

Precise measurements of the processor usage by the analyzed processes are crucial: on one hand to determine the exact duration of a test run and on the other hand to have detailed information on the behavior of the different hardware components to tune the hardware level estimator.

4.3.1 On-chip performance monitoring units

Modern processors as the Motorola PowerPC, the DEC Alpha, the Intel Pentium (from the Pro version on) and the MIPS R10000 have an on-chip performance monitoring unit that is able to perform cycle precise measurements about the CPU behavior. Examples of the measured values are: clock cycles, stalls at the various pipeline stages, cache misses and their penalties, and branch predictor performance. The performance monitoring hardware can be usually switched on and off at any time allowing very fine grained measurements.

Performance monitors were not specifically designed to help in program analysis but to better understand processor performance and, due to pipelining, the performance monitor events are usually not precisely reported at the right time. Events could be signaled some cycles after the event has taken place [27]: at the time we stop the measurements some instructions could be halfway in the pipeline and they could be visible to certain events only. As an example we can consider the Intel Pentium Pro Processor, which has a buffer of 20 instructions and two levels of branch prediction: when the measuring process is started or stopped, the events related to these instructions cannot be precisely accounted since it is not known if they will be really executed or not (instructions in the pipeline at a given point in time could be later discarded due to a branch misprediction). For a given event, we can therefore just keep summary data over a large number of instruction executions.

This means that in the general case the gathered data is sufficient to evaluate how the processor performs but is too coarse to precisely understand the correlation between a particular event and the instruction generating it.

In addition many events are not disjoint, and the performance monitors are not usually able to report their intersection. Consider, e.g., stalls in the dispatch unit of a PowerPC 603e: seven different stall types are reported, but there is no way to know how many apply to the same instruction (up to four instructions of six different types could be in the dispatch queue when the event is reported).

4.3.2 XO/2 measurements

The XO/2 system (see Section 3.2.1) runs on PowerPC processors and specifically, our tool is using a PowerPC 604e on VME boards with several megabytes of RAM (depending on the application).

The PowerPC 604e performance monitor [59, 86] offers four 32 bit counters that can be used to monitor 40 unique events. For each counter it is possible to simply count the occurrences of given event or to decrease the counter each time the event occurs and trigger an interrupt when the counter reaches zero. The performance monitoring is controlled with a bit in the process status register, which is process specific and is saved across context switches (i.e., the monitor is automatically switched on and off when the process is scheduled and suspended). This feature helps us to gather data only on the task of interest and avoids interference from other processes.

Although the input-triggering solution allows a safe usage of the counters since there is no risk of overflow (2^{32} events can quickly happen on a 200MHz machine and even faster on a 2GHz machine), the addition of unnecessary interrupts was not acceptable for the XO/2 real-time system.

Instead, we check the values of the counter at each scheduler period storing the measured values in a per-process structure (the XO/2 scheduler is called often enough to avoid counter overflows). In this way one counter must be relinquished to measure the clock cycles between each preemption point leaving the other three counters free for the actual measurements. On the other hand the bookkeeping overhead is minimal (i.e., we do not require additional interrupts).

Since the XO/2 WCET analyzer needs information on more than three different events (see Section 3.2.1) we have either to make several measurement runs, each one with a different set of events, or to alternate the measurements during the same run and consider the average behavior only. The second approach divides the execution in segments and, for each segment, we measure three of the needed events. The tool iterates over the list of needed events ensuring that each one is measured the same amount of time. For each event we compute the average value for one segment and we then approximate the behavior on the program level of a certain event, with the average value in the measured segments.

Both approaches deliver, with a negligible difference, the same results (real-time kernels have a homogeneous usage of the CPU) and we therefore require only one run reducing the complexity and the duration of the test phase.

4.3.3 Linux measurements on Intel Pentium processors

From the Pentium Pro family on, Intel Pentium processors include an on-chip performance monitoring unit [7, 88] that allows the user to count specific events with two 40 bit special purpose registers.

As for the PowerPC, we use this facility to measure (over several passes) the exact duration in cycles of the program, the number of cache misses and their penalties, and the branch predictor performance of the analyzed processes.

The Intel performance monitor does not have the possibility, as the PowerPC one, to mark one process for monitoring but only provides mechanisms to start and stop the counters regardless of which process is running. For this reason, we modified the Linux scheduler to keep track of the running processes and only measure the processes we are interested in.

We therefore performed the following changes to the Linux kernel:

- We added a module with a new set of system calls to manipulate the counters and the registers to configure the performance monitor (these instructions can only be executed in privilege mode and hence in kernel space).

- We inserted additional fields in the process descriptor storing if the process is currently monitored and the temporary values of the counters between context switches.

- We created a new special device to perform the user and kernel space communication via the ioctl mechanism.

- We included a check at every context switch to see if the current process must be monitored, and we included a small event count bookkeeping routine when a monitored process is suspended.

We provide both the possibility to measure the program behavior with a separate tool or to instrument the code to gather fine-grained data on a per-method basis.

In contrast to the XO/2 approach where the different events where measured in alternation during one single test run, the Linux measurement tool automatically performs several runs avoiding to reconfigure the counters during the test. Although this solution lengthens the time needed to analyze an application it allows us to minimize the required changes to the kernel, easing maintenance and portability. In addition we are also able to better profile applications with an heterogeneous usage of the CPU since measurements of different events are not alternated during one run.

4.3.4 Discussion

Both the Pentium and the PowerPC monitoring systems offer a plethora of events than can be monitored, but these units were not specifically designed to be used to perform precise measurements and to clearly describe the performance behavior (see Section 4.3.1).

This means that in both cases (XO/2 on PowerPC and Linux on Intel) we lacked the possibility to gather data as the stalls in the execution units for the PowerPC, that could have been very useful. We therefore had to rely on approximations to compute the processor behavior as explained in Sections 3.2.1 and 3.4.2.

Although on-chip performance monitoring units are generally badly documented and they lack a standardized system support they have a general utility in addition to their classic applications as chip design debugging and high-end application tuning. Both the XO/2 and the Linux hardware-level analyzer show that the measured processor behavior can not only help on the analyzer evaluation (see Section 5.1) but also on the WCET estimation (see Chapter 3).

5 Evaluation

Testing the soundness of a WCET predictor is a tricky issue since, for complex examples, the real maximum execution time is difficult or even impossible to measure. To compare the estimated values with the measured time we must force the execution of the longest path, which is not normally known. Another option to experimentally determine the WCET would be to measure the applications with all the possible direct and indirect inputs. This approach is clearly impossible for the majority of the nontrivial cases.

Since the simulation of all the possible executions is not feasible we validate our tool by comparing the computer WCET estimation with the measured longest running time for small known synthetic applications where the real longest path is known or computable by hand (see Section 5.4). If these tests yield good results and the validity of the hardware-level analyzer is confirmed, the WCET of larger applications can be estimated but no direct comparison with the real WCET is possible (see Section 5.5).

Section 5.1 presents the software and hardware environment we used to evaluate our WCET estimations. Section 5.3 describes the effects of the partial abstract interpretation pass presented in Section 2.3. Sections 5.4, 5.5 and 5.6 show the estimations of various benchmarks along with the improvements due to the introduction of semantical analysis. Section 5.7 categorizes the loops according to their possible number of iterations. Section 5.8 shows the number of manual annotations needed by the loop bounder while Section 5.9 describes the effectiveness of the automatic inlining (see Section 2.8.2). Section 5.10 describes the results of a maximum memory allocation analysis, and the last section shows the resources needed by our analyzer and its impact on the compilation time.

5.1 Testing environment

One of the most difficult issues in the implementation of a WCET analyzer lies in the evaluation of the results. The output of the semantic analyzer can, for small programs, be analyzed manually by carefully comparing the source code with the computed bounds on the number of iterations for each basic block. This technique allows to quickly identify problems and to easily establish a direct relationship between the results and the corresponding semantic structures.

But on the other hand, the output of the hardware-level analyzer (i.e., the WCET itself) captures a global property and it is difficult to relate an eventual over- or under- estimation to the corresponding part of the analyzed program.

To tune and debug the hardware-level estimator during the development, and to later validate it, we included the possibility to gather the same information on the processor behavior that we can measure using the on-chip performance monitor (see Section 4.3). In other words, both the hardware-level monitor and the hardware-level estimator deliver the same statistics (e.g., cycles per instruction, branch prediction performance and memory related penalties).

5.1.1 Platform

All the measurements in this chapter are performed on a 1GHz Intel Pentium III machine with 512MB of RAM running our patched version of the Linux 2.4.22 kernel (the patches as described in Section 4.3.3 allow the reading and writing of the on-chip performance counters and keep track of the measurements between context switches). We use a multiprocessor Pentium IV 1.4GHz machine with 8GB of RAM for the compilation and the hardware-level analysis.

5.2 Benchmarks

This section describes the benchmarks that we use the evaluate the WCET estimation performed by our tool. We divide them in two groups: simple benchmarks where the WCET is known and measurable and some bigger application benchmarks where a manual analysis in infeasible.

5.2.1 Small synthetic kernels

- BubbleSort on random arrays [93].
- Division performs integer divisions.
- ExpInt computes several exponential integrals [93].
- Jacobi performs Jacobi successive over-relaxations (SORs).
- JanneComplex is the Java version of `janne_complex` from the Uppsala WCET benchmark package in [93].
- MatMult multiplies several integer matrices.
- MatrixInversion inverts several integer matrices.
- Sieve is an implementation of Eratostene's sieve.

5.2.2 Application benchmarks

- _201_compress is part of the SPEC JVM Client 98 benchmark suite [97]. It is an implementation of the Lempel-Ziv (LZW) compression algorithm [104]. It basically finds common substrings and replaces them, with a variable size code. To make it bounded in time, we removed the file input and output mechanisms (we compress 4KB of data in memory).
- JavaLayer [51] is a pure Java library that decodes, converts and plays MP3 files (in our benchmark we decode some sample MP3s to raw audio data).

- Linpack is a benchmark [29] containing various codes from the Linpack library collection which analyzes and solves linear equations and linear least-squares problems.

- SciMark 2.0 [80] is a composite Java benchmark measuring the performance of numerical kernels occurring in scientific and engineering applications (FFT, SOR, sparse matrix-multiply, Monte Carlo integration, and dense LU matrix factorization).

- Whetstone is a benchmark [24] intentionally written to measure computer performance and designed to simulate floating point numerical applications.

5.3 Partial abstract interpretation

Partial abstract interpretation (see Section 2.3) is effective in two ways: it helps to discover false paths and, in some cases, reduces the possible values a variable can assume at a certain program location.

Table 5.1 shows the number of false paths that our estimator was able to discover for a number of benchmarks from the SPEC JVM98 suite and for the JavaLayer decoder (see Section 5.5). Most of these programs are not bound in time and a complete WCET analysis is not possible, but these benchmarks show that although we consider linear paths only, the partial abstract interpretation pass is able to discover several infeasible paths on real applications.

Table 5.1: Infeasible paths.

Benchmark	Infeasible paths
_201_compress	2
_202_jess	3
_205_raytrace	7
_209_db	2
_213_javac	240
_222_mpegaudio	19
_228_jack	22
JavaLayer	238
Linpack	2
SciMark	0
Whetstone	0

In some time-bound benchmarks we discovered some infeasible paths that are indeed the longest paths in the program. These false paths when considered as valid during the analysis cause some WCET overestimation (in the case of JavaLayer by 1%).

In addition to discover infeasible paths the context sensitive computation of the range of possible values of local variables showed to be useful to help the loop bounder to determine the maximum number of loop iterations. Table 5.2 shows the number of times the range of values of a variable helped to compute the iteration bound on an otherwise unbounded loop (see Figure 2.25 for an example). This number includes the occurrences when the lower (or upper) limit or the initial value of the loop's control variable are determined by a range of possible values. The second column lists the total number of analyzed loops.

Table 5.2: Partial abstract interpretation results used for loop bounding.

Program	Loop bounding improvements due to ranges of possible values	Loops (total)
_201_compress	1	17
JavaLayer	1	117
Linpack	8	24
SciMark	7	43
Whetstone	0	14

These results show the usefulness of the introductory partial abstract interpretation pass, which in addition to allow the user to insert annotations in a clean and safe way (see Section 2.7) also allows to increase the number of automatically bounded loops (see Section 5.6) and helps to reduce WCET overestimation due to infeasible paths.

5.4 Small synthetic kernels

For many large applications the WCET is not known and is, due to the complexity of the code, not manually computable. For these examples, it is therefore impossible to enforce the execution of the longest path and validate our estimation: we cannot, in the JavaLayer example, craft a special MP3 input which will generate the longest running time.

The best way to validate such a tool is therefore the comparison of the results for small known synthetic applications where the real longest path is known or computable by hand (BubbleSort, Division, ExpInt, Jacobi, JanneComplex, MatMult, MatrixInversion, and Sieve).

Table 5.3 shows the results for the small benchmarks: the first two columns identify the benchmark showing the number of loops, while columns three and four show the measured and estimated WCET finally the last column shows the WCET overestimation.

All the measurements were executed on a standard Linux system with a small load minimizing the interference effects of other programs.

These small synthetic benchmarks show the soundness of the analysis: we never underestimate the WCET and all the results are close to the measured longest running time.

5.5 Application benchmarks

After the validation of both the semantic and the hardware-level analyzer on small synthetic benchmarks, this section presents the results obtained with bigger applications. The challenge in the evaluation of bigger applications is that the actual WCET of the applications is not known and that it is not possible to force the execution of the WCET at run-time with a specially crafted input.

Table 5.3: WCET estimations of small kernels.

Program	Loops	Measured [cycles]	Estimated [cycles]	Overestimation
BubbleSort	4	$9.16 \cdot 10^{9}$	$1.53 \cdot 10^{10}$	67%
Division	2	$1.40 \cdot 10^{9}$	$1.55 \cdot 10^{9}$	10%
ExpInt	3	$1.28 \cdot 10^{8}$	$2.38 \cdot 10^{8}$	86%
Jacobi	5	$0.88 \cdot 10^{10}$	$1.08 \cdot 10^{10}$	22%
JanneComplex	4	$1.39 \cdot 10^{8}$	$2.48 \cdot 10^{8}$	78%
MatMult	6	$2.67 \cdot 10^{9}$	$2.73 \cdot 10^{9}$	2%
MatrixInversion	11	$1.42 \cdot 10^{9}$	$1.55 \cdot 10^{9}$	10%
Sieve	4	$1.29 \cdot 10^{10}$	$1.40 \cdot 10^{10}$	9%

In principle, these real-life programs could be tested using a “black box” approach with the execution time as result. In this case testing consists of maximizing the result function, i.e., the dynamic path length, varying the input data and submission time [83, 73]. Unfortunately, this approach is not feasible with real applications where the amount of input data is huge.

We therefore compare our estimation with the highest running time we observe over several runs of the analyzed applications (_201_compress, JavaLayer, Linpack, SciMark and Whetstone).

Table 5.4 shows for each benchmark the number of classes, methods, and loops. *Observed* corresponds to the maximum observed running time (in cycles) while *Estimated* is our WCET estimation. The WCET estimation is almost always (with the exception of the SciMark benchmark) relatively close to the highest observed running time confirming the soundness of the method.

The reasons for the WCET overestimation of the SciMark benchmark is twofold: on one hand the benchmark uses several mathematical functions (e.g., trigonometric functions) that normally take less time than in the worst case and on the other hand the benchmark contains triangular loops (i.e., the number of iterations of the inner loop is dependent on the iteration count of the outer loop). Our semantic analyzer treats triangular loops conservatively by assuming the same maximal number of iterations for the inner loop at each outer loop iteration (see Section 2.4.5).

Table 5.4: WCET estimations.

Program	Classes	Methods	Loops	Observed [cycles]	Estimated [cycles]	Overestimation
_201_compress	13	43	17	$7.20 \cdot 10^{9}$	$1.05 \cdot 10^{10}$	46%
JavaLayer	63	202	117	$6.09 \cdot 10^{9}$	$1.18 \cdot 10^{10}$	94%
Linpack	1	17	24	$1.40 \cdot 10^{10}$	$2.72 \cdot 10^{10}$	94%
SciMark	9	43	43	$1.91 \cdot 10^{10}$	$1.22 \cdot 10^{11}$	538%
Whetstone	1	7	14	$1.86 \cdot 10^{9}$	$2.11 \cdot 10^{9}$	13%

5.5.1 Tuning and evaluation of the partial traces length

The length of the partial traces we simulate to estimate the duration of an instruction i or a block (see Section 3.4) represents the number of instructions executed before i that could influence its duration (e.g., by modifying the caches or by blocking the pipeline).

Experiments show that after a threshold increasing the partial traces length does not change the WCET estimation any more since the additional instruction that are simulated are not able to influence the duration of the analyzed instruction.

Figure 5.1 shows the WCET overestimation of the _201_compress benchmark: if the length of the partial trace is sufficiently long (100 instruction) the result of the hardware-level WCET estimator is stable. The same behavior can be observed by the other benchmarks.

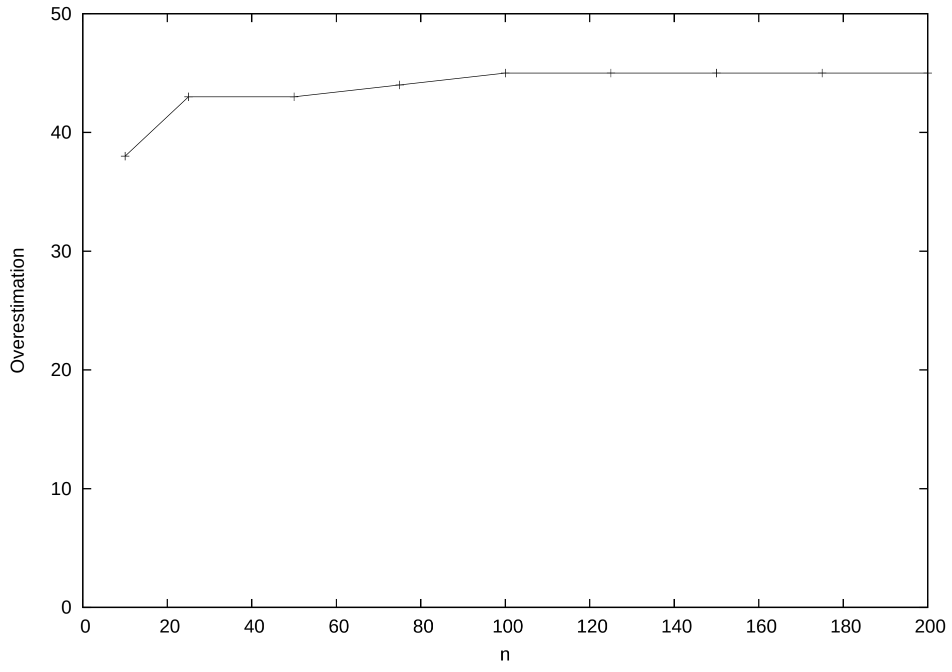


Figure 5.1: Influence of the partial trace length (n) on the overestimation of the _201_compress benchmark.

5.6 Effects of semantic analysis

Structural analysis allows to propagate the bounds on the maximum number of iterations of a loop header to all the loop's basic blocks, taking into account different path frequencies (see Section 2.5). Table 5.5 shows the results for some benchmarks where this propagation is able to reduce the WCET overestimation. The first column represents the maximum execution time that we were able to observe (in cycles), while the second and the third are the estimated WCET using the conservative assumption that the header bounds are safe for the whole loop (see Section 2.4), and our enhancement (i.e., the ability to consider different path frequencies in a loop body).

Three tests (_201_compress, JavaLayer, and Scimark) present some improvement over the

Table 5.5: Effects of semantic analysis.

Program	Maximum observed [cycles]	Estimated [cycles]		Enhanced [cycles]	
_201_compress	$7.20{\cdot}10^9$	$4.26{\cdot}10^{12}$	(59 066%)	$1.05{\cdot}10^{10}$	(45%)
JavaLayer	$6.09{\cdot}10^9$	$1.35{\cdot}10^{10}$	(121%)	$1.18{\cdot}10^{10}$	(94%)
Linpack	$1.40{\cdot}10^{10}$	$2.72{\cdot}10^{10}$	(94%)	$2.72{\cdot}10^{10}$	(94%)
SciMark	$1.91{\cdot}10^{10}$	$1.82{\cdot}10^{11}$	(852%)	$1.22{\cdot}10^{11}$	(538%)
Whetstone	$1.86{\cdot}10^9$	$2.11{\cdot}10^9$	(13%)	$2.11{\cdot}10^9$	(13%)

base algorithm that bounds the number of iterations of basic blocks using the loop headers only (see Section 2.5). _201_compress shows a huge improvement since our algorithm is able to detect that a significant part of the compressor main loop is executed only every 10'000 iterations. The small synthetic kernels show no improvement since their simple structure does not present any different path frequencies inside loop bodies.

5.7 Representation of ranges of iterations

The introduction of the sets of disjoint ranges as a common data structure for both the partial abstract interpretation and the loop bounding pass allows us to easily handle loops with noncontinuous ranges of possible iterations (i.e., ranges with holes). Table 5.6 shows the number of loops that have a constant number of iterations (i.e., the minimum and maximum number of iterations overlap). The number of loops that have a noncontinuous set of iterations is shown in the last column.

Table 5.6: Loop header iterations representation.

Program	Loops (total)	Range representation suffices	Noncontinuous representation needed
BubbleSort	4	2	2
Division	2	2	0
ExpInt	3	3	0
Jacobi	5	5	0
JanneComplex	4	2	0
MatMult	6	6	0
MatrixInversion	8	7	1
Sieve	4	3	0
_201_compress	17	3	1
JavaLayer	117	67	5
Linpack	24	8	1
SciMark	43	9	0
Whetstone	7	7	0

The presence of loops with a noncontinuous set of possible iterations shows that our algorithm is able to handle complex cyclic structures keeping track of all the possible execution scenarios (and not only the best and worst case).

5.8 Annotations

The number of manual annotations that were needed to bound the number of iterations for each basic block are shown in the third column of Table 5.7. Small tests (i.e., small scientific kernels) do not normally require annotations while some bigger applications as JavaLayer need some manual intervention.

Table 5.7: Manual annotations.

Program	Loops	Annotations
BubbleSort	4	0
Division	2	0
ExpInt	3	0
Jacobi	5	0
JanneComplex	4	2
MatMult	6	0
MatrixInversion	11	0
Sieve	4	1
_201_compress	17	10
JavaLayer	117	46
Linpack	24	19
SciMark	43	33
Whetstone	14	0

The main reason is that these applications where not designed considering timing concerns. In the case of JavaLayer, 15 annotations where needed to specify that the number of sub-bands is a constant and known number. Many other loops, in the JavaLayer benchmark, have a constant number of iterations which was expressed in an object field (see Table 2.2), hindering the automatic analysis.

5.9 Automatic inlining

Automatic inlining is performed when one or more loops depend on method parameters and the code of the method is inlined to analyze it in the caller context (see Section 2.8.2). The advantage of this operation is that, in many cases, the analyzer is able to find the correct loop iteration bounds in the caller context (see Table 5.8).

5.10 Memory allocation

Once all the basic blocks in the analyzed program have a bound on the maximum number of iterations, in addition to the computation of the WCET, it is also possible to estimate the maximum allocated memory (assuming no garbage collection).

Table 5.8: Method inlining.

Program	Annotations	Inlined methods
_201_compress	10	5
JavaLayer	46	1
Linpack	19	7
SciMark	33	6
Whetstone	0	0

This is useful for real-time applications using immortal objects (see Section 4.1.2) where, before scheduling, the system must be able to ensure that it will have enough free memory for the process.

The maximum amount of allocated memory can be computed by a complete traversal of the call and control-flow graphs without iterating over loops and recursive procedures since their maximum number of iterations is already computed by the semantic WCET analyzer (see Chapter 2).

During the traversal the analyzer computes the maximum sum of all the objects allocated on the heap and the maximum size of the stack. Our tool supports a prototypical implementation of a wort-case memory allocation analyzer which does not consider false paths and uses a simplified model of the memory management but nevertheless Table 5.4 shows some results for a small set of benchmarks (the prototype is not able to handle all the benchmarks).

Table 5.9: Memory allocation estimation.

Benchmark	Measured [KB]	Estimated [KB]	Overestimation [KB]
Drystone	62 567	62 634	0.1%
Linpack	16 057	16 057	0%
Whetstone	0.822	0.822	14%

Although the analysis of the maximum amount of allocated memory and the analysis of the worst-case execution time are similar on the semantic level (they both rely on the maximum number of basic block executions) they differ on the backend: runtime environment for memory and processor and system for the instruction duration estimation.

The computation of the space requirements for a given object (with a known type) is solely depending on the runtime environment and, in contrast to the instruction duration, is context independent

Our prototype makes some basic assumption about memory management from runtime system but a more in-depth analysis of GCJ's memory allocation scheme (see Section 4.2.1) would be required for the analysis of more complex examples.

5.11 Resources

Table 5.10 shows the time needed to compile the given benchmarks: *Size LOC* show the size of the benchmark in lines of code, *Compile* represents the total time needed to compile the application, *Semantic* represents the part of the total compilation time spent in the semantic analyzer module and *HW* shows the time needed by the hardware-level analyzer to estimate the WCET.

Table 5.10: WCET Analyzer resources.

Program	Size (LOC)	Compile [s]	Semantic [s]	HW [s]
_201_compress	574	12	3	6
JavaLayer	5 816	302	10	661
Linpack	318	17	8	12
SciMark	756	11	1	6
Whetstone	128	9	1	5

Both the overhead of the semantic analyzer on the whole compilation time and the time needed by the hardware-level analyzer (which includes scanning and parsing of the assembler files) remain small and manageable even for large tests.

5.12 Discussion

Our WCET estimator is able to deliver sound results for a number of small and medium-sized Java applications thanks to a precise semantic analysis and a quick context-sensitive hardware-level analysis to estimate the duration of the program's instructions.

The partial abstract interpretation pass is able to discover several false paths but more significantly is able to help in the computation of the bounds on the number of loop iterations. The integration of precise semantic structural information in the bounding process of the number of possible iterations for each basic block allows to consider different execution frequencies for distinct paths inside loop bodies.

The hardware-level analysis based on the principle of locality is able to estimate the duration of each instruction determining the effects of caches and pipelines in the right computational context.

6 Conclusions

The computation of the worst-case execution time of an application is crucial for real-time systems (where the WCET is needed for schedulability analysis) and a useful tool for the analysis of many soft-real-time and multimedia systems.

Although the problem of finding the WCET of a program is, in the general case, undecidable, its importance motivated the growth of several pragmatic approaches to compute tight estimations of the WCET for a limited set of analyzed applications. A considerable amount of related research exits in both the field of semantic analysis and low-level hardware analysis (see Chapter 1) offering solutions to several practical application domains.

These techniques differ on several aspects: Some approaches compute a precise and conservative WCET for simple and deterministic systems, while other allow some approximations (e.g., in the instruction duration estimation) to be able to handle larger inputs.

This thesis presents a set of analyses that do not rely on path enumeration to estimate the WCET for fairly large Java applications that do not include explicit timing specifications.

6.1 Summary and contributions

This thesis addresses both the challenges in the semantic and hardware-level analyses and makes the following contributions:

Semantic analysis. The dissertation presents a set of analyses to derive safe bounds on the maximum number of iteration for each semantic construct in the analyzed program.

- We define a fast partial abstract interpretation pass able to significantly reduce the possible values of local variables at given program points. This pass able to discover some false paths. The abstract interpretation is limited to code segments that do not include cycles, as a result the complexity of is linear.

  The produced sets of possible values for each local variable are used to increase the precision of the loop bounder and to allow the user to easily manually specify bounds on the values of loop induction variables (see Section 2.7).

- We developed a loop iterations bounding technique that takes into account different path frequencies inside a loop body, without the need to perform path enumeration.

  The analysis, which operates on the loops in isolation and is tightly integrated in the high-level compiler structures, cooperate with classical compiler analyses both by using their results and by providing useful information on the loops behavior (one simple example is the detection of dead or infinite loops).

- We analyze unmodified Java programs without posing restrictions to the semantics of the code (programs using recursion are allowed but have to manually annotated with the maximum recursion depth) and we reduce the size of the call graph by resolving polymorphism at call sites by a sophisticated whole-program variable type based analysis (see Section 2.8).

Instruction duration analysis. In this thesis we describe a technique to approximate the duration of the various assembler instructions in their execution context. The approach can be adapted to different architectures with a configurable precision: the level of approximation which is dependent on the length of the simulated partial traces (see Section 3.4) can be specified by the user. The technique can be used with any cycle-precise CPU simulator allowing a high portability.

The simulation of short program traces before and after a given instruction has shown to be a fast and effective solution to analyze the duration of the instructions in their possible execution contexts. The whole analysis in performed in asymptotically linear time over the number of instructions that are present in the program source (approaches that are based on simulation or abstract interpretation depend on the other hand from the number of executed instructions).

Language and platform. Although our implementation is Java (frontend) and Intel (backend) specific, the solution can be applied to a plethora of languages and platforms.

- The loop bounding algorithm and the partial interpretation pass are language independent and work directly on the SSA form of the compiler intermediate representation. The precision of the analysis is however strongly influenced by the chosen language: the presence of pointer arithmetic (as in C or C++) could, because of aliasing problems, strongly reduce the amount of successfully bounded loops.
- The partial trace evaluation technique is architecture independent. The quality of the results depends on the faithfulness of the simulator model to the actual hardware implementation, and on the predictability of the CPU. The effects of preemption on a nonpipelined processor without caches are inherently smaller than the same effects on a modern CPU as the PowerPC or the Pentium.

6.2 Future work and open issues

There are three major directions for future research based on our WCET estimator: On the semantic analysis level to tighten the number of iterations on the cyclic structures, on the hardware-level analysis to better model the hardware architecture and finally on the analysis of other restricted resources (e.g., memory allocation).

Semantic analysis. Although our semantic analyzer is able to bound the iterations of a large number of loops several improvements are possible. We do not consider dependences among loops and we are not able to consider loop bounds depending on the value of an outer loop's induction variable (e.g., triangular loops). This limitation is one of the major causes behind the WCET overestimation of many scientific applications where these loops are common (the WCET is overestimated since we conservatively consider triangular inner loops as rectangular).

A parametrization of the data structures (e.g., the intervals holding the number of iterations) could allow the analyzer to keep track of the execution of a loop with dynamic bounds depending on an outer loop. Parametrization can also be broadly applied to a whole set of problems as loops depending on the methods parameters, or on a program's input [101].

Automatic recursion depth bounding is another issue that needs attention and further research. The manual specification of the bounds can be a difficult and tedious task: it requires a precise knowledge of the theoretical limits of the analyzed algorithms and it's implementation (recursion could involve a large set of methods). Users could benefit from an automatic analysis by reducing the number of manual annotations and therefore by reducing the possibility of misleading user interventions.

Hardware model. Our implementation makes use of a simplified processor model for the cycle-precise CPU simulator, and a more accurate simulation would certainly increase the precision of the results. The main source of imprecision however is not coming from the incorrect modeling of the processor but from the impossibility to track data references and therefore to properly estimate the influence of data caches on memory operations.

The whole predictor would benefit from the substitution of the probabilistic cache model used by our predictor (see Section 3.4.2) by a context sensitive data reference analysis able to compute or approximate the state of the data caches.

Maximum allocated memory. The most usual and obvious use of the information on the semantics of the program computed by the WCET estimator is the estimation of the longest running time of the application. Memory is, as time, another limited resource and future research to automatically estimate the maximum amount of allocated memory of processes would be beneficial to the stability of the real-time systems (see Section 5.10).

6.3 Concluding remarks

Many pragmatic solutions to the worst-case execution time estimation have been presented addressing different aspects: from precise estimation of small hard-real-time application running on predictable platform to the estimation for big applications using probabilistic methods.

This thesis describes a complete set of analyses addressing both the semantic aspects as well as the analysis on a given hardware platform, with the goal to deliver an approximation of the WCET of fairly large soft-real time applications on a modern architecture sufficient for practical purposes.

Our work shows that it is possible to approximate the WCET of soft-real-time applications written with object-oriented languages (Java in our case) on modern platforms without performing path enumeration. The complete analysis time is dependent solely on the number of instructions or blocks present in the code and not as in the case of simulation on the number of executed instructions.

Bibliography

[1] Requirements for real-time extensions for the Java^{TM} platform, April 1999. NIST Special Pubblication, U.S. Department of Commerce, Lisa Carnahan, NIST and Markus Ruark, Commotion Technology, Inc. Editors.

[2] P. Altenbernd. On the false path problem in hard real-time programs. In *Proc. 8th Euromicro Workshop on Real-Time Systems*, pages 102–107, L'Aquila, Italy, June 1996.

[3] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. 11th Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, San Jose, CA, October 1996. ACM.

[4] G. Bernat, A. Burns, and A. Wellings. Portable worst-case execution time analysis using Java byte-code. In *Proc. 12th Euromicro Workshop on Real-Time Systems*, Stockholm, Sweden, June 2000.

[5] G. Bernat, A. Colin, and S. Petters. WCET analysis of probabilistic hard real-time systems. In *Proc. 23rd IEEE Intl. Real-Time Systems Symp.*, pages 279–289, Austin, TX, December 2002. IEEE.

[6] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[7] D. Bhandarkar and J. Ding. Performance characterization of the Pentium Pro processor. In *Proc. 3rd Intl. Symp. on High-Performance Computer Architecture*, pages 288–297, San Antonio, TX, February 1997.

[8] D. Blaauw, R. Panda, and A. Das. Removing user specified false paths from timing graphs. In *Proc. 37th Conf. on Design Automation*, pages 270–273, Las Vegas, NV, June 2000. IEEE.

[9] J. Blieberger. Real-time properties of indirect recursive procedures. *Information and Computation*, 171(2):156–182, 2001.

[10] J. Blieberger and R. Lieger. Real-time recursive procedures. In *Proceedings of the 7th Euromicro Workshop on Real-Time Systems*, pages 229–235, Odense, Denmark, June 1995.

[11] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, and R. Belliardi. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[12] G. Bollella and J. Gosling. The real-time specification for Java. *Computer*, 33(6):47–54, June 2000.

[13] H. Börjesson. Incorporating worst-case execution time in a commercial C-compiler. Technical Report 69, Department of Computer Systems, Uppsala University, Uppsala, Sweden, January 1996.

[14] R. Brega. A real-time operating system designed for predictability and run-time safety. In *Proc. 4th Intl. Conf. Motion and Vibration Control (MOVIC)*, pages 379–384, Zurich, Switzerland, August 1998.

[15] R. Brega. *A Combination of System Software Techniques Aimed at Raising the Run-Time Safety of Complex Mechatronic Applications*. PhD thesis, Swiss Federal Institute of Technology (ETH) Zurich, 2002. Nr. 14513.

[16] J. Busquets-Mataix and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proc. 8th Euromicro Workshop on Real-Time Systems*, pages 271–276, L'Aquila, Italy, June 1996.

[17] G. Buttazzo. *Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications*, volume 416 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, MA, September 1997.

[18] M. Campoy, A. Ivars, and J. Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *Proc. IEEE/IEE Real-Time Embedded Systems Workshop*, London, UK, December 2001.

[19] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland Univ. of Technology, School of Computing Science, Brisbane, Australia, July 1994.

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

[21] M. Corti, R. Brega, and T. Gross. Approximation of worst-case execution time for preemptive multitasking systems. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, Vancouver, Canada, June 2000.

[22] M. Corti and T. Gross. Instruction duration estimation by partial trace evaluation. In *Proc. WIP Session 10th Real-Time and Embedded Technology and Applications Symp.*, Toronto, Canada, May 2004. IEEE.

[23] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

[24] H. Curnow and B. Wichmann. A synthetic benchmark. *Computer Journal*, 19(1):43–49, February 1976.

[25] R. Cytron et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[26] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. 9th European Conf. on Object-Oriented Programming*, pages 77–101, Årtus, Denmark, August 1995.

[27] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proc. 30th Intl. Symp. on Microarchitecture*, pages 292–302, Los Alamitos, CA, December 1997. IEEE/ACM.

[28] Z. Deng and J. W. S. Liu. Scheduling real-time applications in open environment. In *Proc. 18th Real-Time Systems Symp.*, pages 308–319, San Francisco, CA, December 1997. IEEE.

[29] J. Dongarra. Performance of various computers using standard linear equations software in a Fortran environment. In Walter J. Karplus, editor, *Proc. 3rd Conf. on Multiprocessors and Array Processors*, pages 15–32, San Diego, CA, January 1987. Society for Computer Simulation.

[30] P. Emma. Understanding some simple processor-performance limits. *IBM Journ. Research and Development*, 43(3):215–231, 1997.

[31] C. Ferdinand et al. Reliable and precise WCET determination for a real-life processor. In *Proc. 1st Intl. Workshop on Embedded Software*, pages 469–485, Tahoe City, CA, October 2001. ACM.

[32] M. F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proc. 8th Conf. on Programming Language Design and Implementation*, pages 103–115, La Jolla, CA, June 1995. ACM.

[33] Free Software Foundation. GCJ: The GNU compiler for the Java programming language. http://gcc.gnu.org/java/.

[34] Free Software Foundation. GNU Classpath. http://www.gnu.org/software/classpath/.

[35] M. R. Garey and D. S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal of Computing*, 4(4):397–411, 1975.

[36] N. Gehani and K. Ramamritham. Real-time concurrent C (C++): A language for programming dynamic real-time systems. *Real-Time Systems*, 3(4):377–405, December 1991.

[37] R. Govindan and D. P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proc. 13th Symp. on Operating Systems Principles*, pages 68–80, Pacific Grove, CA, October 1991. ACM.

[38] Real-Time Java Working Group. Real-time core extensions for the Java platform. http://www.j-consortium.org/, September 2000.

[39] J. Gustafsson. Calculation of execution times in realtimetalk-an object-oriented language for real-time. In *Proc. 1st Workshop on Object-Oriented Real-Time Dependable Systems*, pages 153–162, Dana Point, CA, October 1994. IEEE.

[40] J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Department of Computer Systems, Information Technology, Uppsala Univ., 2000.

[41] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. on Computers*, 48(1):53–70, January 1999.

[42] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proc. 4th Real-Time Technology and Applications Symp.*, pages 12–21, Denver, CO, June 1998.

[43] C. Healy, R. van Engelen, and D. Whalley. A general approach for tight timing predictions on non-rectangular loops. In *WIP Proc. 5th Real-Time Technology and Applications Symp.*, pages 11–14, Vancouver, Canada, June 1999. IEEE.

[44] C. Healy and D. Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proc. 5th Real-Time Technology and Applications Symp.*, pages 79–88, Vancouver, Canada, June 1999.

[45] M. T. Higuera-Toledano and V. Issarny. Java embedded real-time systems: An overview of existing solutions. In *Proc. 3rd IEEE Intl. Symp. on Object-Oriented Real-Time Distributed Computing*, pages 392–399, Newport, CA, March 2000.

[46] M. Honegger, R. Brega, and G. Schweitzer. Application of a nonlinear adaptive controller to a 6 dof parallel manipulator. In *Proc. Intl. Conf. Robotics and Automation*, pages 1930–1935, San Francisco CA, April 2000. IEEE.

[47] M. Honegger, G. Schweitzer, O. Tschumi, and F. Amberg. Vision supported operation of a concrete spraying robot for tunneling work. In *Proc. 4th Conference on Mechatronics and Machine Vision in Practice*, pages 230–234, Toowoomba, Australia, September 1997.

[48] M. Humphrey and J. A. Stankovic. Predictable threads for dynamic, hard real-time environments. *IEEE Trans. on Parallel and Distributed Systems*, 10(3):281–296, March 1999.

[49] IBM Microelectronic Division and Motorola Inc. *PowerPC 604/604e RISC Microprocessor User's Manual*, 1998.

[50] Intel Corporation. *Intel Architecture Optimization, Reference Manual*, 1999.

[51] JavaZOOM. Javalayer. http://www.javazoom.net/javalayer/javalayer.html.

[52] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proc. 17th Real-Time Systems Symp.*, pages 89–99, Phoenix, AZ, December 1992. IEEE.

[53] S.-K. Kim, S. Min, and R. Ha. Efficient worst-case timing analysis of data caching. In *Proc. 2nd Real-Time Technology and Applications Symp.*, pages 230–240, Boston, MA, June 1996. IEEE.

[54] D. Kirk. SMART (Strategic Memory Allocation for Real-Time) cache design. In *Proc. 10th Real-Time Systems Symp.*, pages 229–239, Santa Monica, CA, December 1989. IEEE.

[55] E. Kligerman and A. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Trans. on Software Eng.*, 12(9):941–949, September 1986.

[56] C.-G. Lee et al. Bounding cache-related preemption delay for real-time systems. *IEEE Trans. on Software Eng.*, 27(9):805–826, September 2001.

[57] C.-G. Lee, J. Hahn, Y.-M. 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 Real-Time Systems Symp.*, pages 264–274, Washington, D.C., December 1996. IEEE.

[58] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, December 1982.

[59] F. Levine and C. Roth. A programmer's view of performance monitoring in the PowerPC microprocessor. *IBM Journ. Research and Development*, 41(3):345–356, May 1997.

[60] Y.-T. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond directed mapped instructions caches. In *Proc. 17th Real-Time Systems Symp.*, pages 254–263, Washington, D.C., December 1996. IEEE.

[61] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proc. 3rd Real-Time Technology and Applications Symp.*, Montreal, Canada, June 1997. IEEE.

[62] S.-S. Lim, Y. Bae, G. Jang, B.-D. Rhee, S. Min, C. Park, H. Shin, K. Park, S.-M. Moon, and C.-S. Kim. An accurate worst-case timing analysis for RISC processors. *IEEE Trans. on Software Eng.*, 21(7):593–604, July 1995.

[63] S.-S. Lim, J. Han, J. Kim, and S. Min. A worst-case timing analysis technique for multiple-issue machines. In *Proc. 19th Real-Time Systems Symp.*, pages 334–345, Madrid, Spain, December 1998. IEEE.

[64] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[65] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers Univ. of Technology, Göteborg, Sweden, June 2000.

[66] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2/3):183–207, November 1999.

[67] T. Lundqvist and P Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proc. 20th Real-Time Systems Symp.*, pages 12–21, Phoenix, AZ, December 1999. IEEE.

[68] J. Miecznikowski and L. Hendren. Decompiling Java using staged encapsulation. In *Proc. Working Conf. on Reverse Engineering*, pages 368–374, Stuttgart, Germany, October 2001. IEEE.

[69] T. Mitra and A. Roychoudhury. A framework to model branch prediction for worst case execution time analysis. In *Proc. 2nd Intl. Workshop on Worst-Case Execution Time Analysis*, Vienna, Austria, June 2002.

[70] A. Mok. Evaluating tight execution time bounds of programs by annotations. In *Proc. 6th Workshop on Real-Time Operating Systems and Software*, pages 74–80. IEEE, May 1989.

[71] H. Mössenböck and N. Wirth. The programming language Oberon-2. *Structured Programming*, 12(4):179–195, 1991.

[72] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[73] F. Müller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *Proc. 19th Real Time Technology and Applications Symp.*, pages 179–188, Madrid, Spain, June 1998. IEEE.

[74] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proc. 1st IEEE/ACM/IFIP Intl. Conference on Hardware/Software Codesign and System Synthesis*, pages 201–206, Newport Beach, CA, October 2003.

[75] D. Niehaus. Program representation and translation for predictable real-time systems. In *Proc. 12th Real-Time Systems Symp.*, pages 53–63, San Antonio, TX, December 1991. IEEE.

[76] K. Nilsen. Adding real-time capabilities to Java. *Comm. ACM*, 41(6), June 1998.

[77] S. Oikawa and H. Tokuda. User-level real-time threads. In *Proc. 11th Workshop on Real-Time Operating Systems and Software*, pages 7–11, Seattle, WA, May 1994.

[78] C. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 1(5):31–62, 1993.

[79] G. Pospischil, P. Puschner, A. Vrchoticky, and R. Zainlinger. Developing real-time tasks with predictable timing. *IEEE Software*, 9(5):35–44, September 1992.

[80] B. Pozo, R. Miller. Scimark2. http://math.nist.gov/scimark2.

[81] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proc. 23rd IEEE Intl. Real-Time Systems Symp.*, pages 114–123, Austin, TX, December 2002. IEEE.

[82] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journ. Real-Time Systems*, 1(2):160–176, September 1989.

[83] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *Proc. 19th Real-Time Systems Symp.*, pages 134–143, Madrid, Spain, December 1998. IEEE.

[84] P. Puschner and A. Schedl. Computing maximum task execution times — A graph-based approach. *Real-Time Systems*, 13(1):67–91, July 1997.

[85] P. Puschner and A. Vrchoticky. Problems in static worst-case execution time analysis. In *9. ITG/GI-Fachtagung Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, Kurzbeiträge und Toolbeschreibungen*, pages 18–25, Freiberg, Germany, September 1997.

[86] C. Roth, F. Levine, and E. Welbon. Performance monitoring on the PowerPC 604 microprocessor. In *Intl. Conf. on Computer Design: VLSI in Computers and Processors*, pages 202–218, Austin, TX, October 1995.

[87] Li Y.-T. S. and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 88–98, La Jolla, CA, June 1995.

[88] K. Safford. A framework for using the Pentium's performance monitoring hardware. Master's thesis, University of Illinois, Urbana, IL, 1997.

[89] Y. Seo, J. Park, and S. Hong. Supporting preemptive user-level threads for embedded real-time systems. Technical report, Seoul National University, Seoul, Korea, August 1998. SNU-EE-TR-98-1.

[90] O. Serlin. Scheduling of time critical processes. In *Proc. Spring Joint Computer Conf.*, pages 925–932, Atlantic City, NJ, May 1972.

[91] M. Sharir. Structural analysis: a new approach to flow analysis in optimizing compilers. *Computer Languages*, 5(3-4):141–153, 1980.

[92] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Trans. on Software Eng.*, 15(7):875–889, January 1989.

[93] F. Stappert. WCET benchmarks. http://www.c-lab.de/home/en/download.html.

[94] J. Staschulat and Ernst R. Multiple process execution in cache related preemption delay analysis. In *Proc. 4th ACM Intl. Conf. on Embedded Software*, pages 278–286, Pisa, Italy, September 2004.

[95] A. D. Stoyenko, V. C. Hamacher, and R. C. Holt. Analyzing hard-real-time programs for guaranteed schedulability. *IEEE Trans. on Software Eng.*, 17(8):737–750, August 1991.

[96] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proc. 15th Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 264–280, Minneapolis, MN, October 2000. ACM.

[97] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1996.

[98] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proc. 19th Real-Time Systems Symp.*, pages 144–153, Madrid, Spain, December 1998. IEEE.

[99] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11(1):25–33, January 1967.

[100] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proc. 8th Intl. Workshop on Hardware/Software Co-Design*, pages 67–71, San Diego, CA, May 2000.

[101] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. *SIGPLAN Not.*, 36(8):88–93, 2001.

[102] C. von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. PhD thesis, Swiss Federal Institute of Technology (ETH) Zurich, Zurich, Switzerland, May 2004. Nr. 15524.

[103] A. Vrchoticky. Compilation support for fine-grained execution time analysis. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, Orlando, FL, 1994.

[104] T. Welch. A technique for high performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[105] N. Wirth. From Modula to Oberon. *Software Practice and Experience*, 18(7):671–690, July 1988.

[106] S.H. Yen, D.H. Du, and S. Ghanta. Efficient algorithms for extracting the K most critical paths in timing analysis. In ACM Press, editor, *Proc. 26th Conf. on Design Automation*, pages 649–654, Las Vegas, NV, June 1989. IEEE/ACM.

[107] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst-case execution times. *Real-Time Systems*, 5(4):319–343, October 1993.

List of Figures

List of Tables

Curriculum Vitae

Matteo Corti

September 12, 1974	Born in Lugano, Switzerland
1980 – 1985	Primary school, Breganzona
1985 – 1989	Secondary school, Breganzona
1989 – 1993	Liceo scientifico (High School), Lugano
1993	Maturità tipo C
1993 – 1997	Studies in Computer Science, ETH Zurich
1998	Internship at FIDES/EDS, Zurich
1998	Diploma in Computer Science, ETH Zurich
1998 – 2005	Research and Teaching Assistant Laboratory for Software Technology, ETH Zurich