Hybrid Measurement-Based WCET Analysis using Instrumentation Point Graphs

Adam Betts

This thesis is submitted in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy.

University of York

February 2010

Abstract

Precise operation of real-time systems depends on functionally correct computations that are delivered within imposed timing constraints. These temporal requirements are often modelled and verified assuming *a priori* knowledge of the *Worst-Case Execution Time* (WCET) of each task. Due to complexities resolving the *actual* WCET, estimates normally suffice. These estimates should be safe, so as not to compromise temporal correctness, and accurate, in order to maximise the often limited system resources. The aim of *WCET analysis* is to therefore compute a WCET estimate that is the actual WCET.

To date, the predominant research direction has been *static analysis*, which builds both program and processor models, and can therefore provide rigourous proofs regarding safety. However, the real-time sector is being infiltrated by more advanced processors that complicate processor modelling sufficiently so that simplfying assumptions are needed. Such assumptions lead to varying degrees of overestimation, depending on processor configuration. On the other hand, current *end-to-end testing* practices - most often employed in industry - do not target WCET estimation and could therefore underestimate unless the longest path is triggered. This is further complicated by advanced processors as the WCET can depend on a rare sequence of events at the architectural level, and not necessarily on the input causing the greatest number of operations.

In this thesis, we combine the relative strengths of testing and static analysis through a *Hybrid Measurement-Based* (HMB) framework based on a new program model, the *Instrumentation Point Graph* (IPG). We present an algorithm to construct the IPG from a reducible CFG* - an augmented Control Flow Graph (CFG) - such that arbitrary irreducible IPG loops are identified on the fly. Using these structural properties, we show how to map loop bounds obtained through static analysis onto the IPG and also how to extract observed loop bounds from *timing traces*.

However, since the IPG does not provide a means *per se* to compute WCET estimates, we remodel two common calculation techniques so that they pertain to arbitrary IPGs. For the purposes of tree-based calculations, we present an algorithm that decomposes the IPG into a new hierarchical form, the Itree; we also present the timing schema used to drive the calculation over the Itree. However, we show that the Itree representation must make a space/precision trade-off when modelling arbitrary irreducible IPGs, ultimately resulting in a margin of overestimation. As a consequence, we rework the Implicit Path Enumeration Technique (IPET) so that it applies to the IPG.

All these techniques have been implemented in a prototype tool which takes a disassembled program and a number of timing traces as input, illustrating the relative ease in which our HMB framework can be retargetted as neither a processor model nor user interaction is required. We use this prototype tool to evaluate a large-scale industrial application.

Contents

List of Figures

List of Tables

Acknowledgements

There are three people to whom I want to extend an indebted amount of gratitude.

First, my mother and father, who are the bestest parents in the whole wide world! Unfortunately, my mother (Margaret “Taffy” Betts) will never see the completion of this mammoth document, but without her love, her kind and caring nature, her sense of humour, her (bad) laundry service(!), I would never have fulfilled my lofty ambitions. I miss you every single second, mother. But of course, behind every good woman is a good man, and my father (Kenneth “Cowboy” Betts) has provided me with support in equal measure with his dodgy DIY, his occasional shoe polishing services, his slow and frustrating driving, and his general dopiness. I love you, honestly father!

Second, to my supervisor Dr. Guillem “Rapita” Bernat, who gave me the opportunity to come to York under his astute guidance. Along the undulating journey that is a PhD, Guillem has always been there in the background, ensuring that my ideas do not become too aloof or, plainly and simply, too stupid. Guillem has provided me with a plethora of opportunities in my professional career - sometimes through sheer blind faith it would appear - and that will never be forgotten. Moltes gràcies, Guillem!

Declaration

Some of the material presented in this thesis has previously been published in the following papers:

- A. Betts and G. Bernat, ”Issues using the Nexus Interface for Measurement-Based WCET Analysis”, Proceedings of the 5th international workshop on Worst-Case Execution Time Analysis, satellite workshop to the 17th international Euromicro conference on Real-Time Systems, July 2005.
- A. Betts and G. Bernat, ”Tree-Based WCET Analysis on Instrumentation Point Graphs”, Proceedings of the International Symposium on Object and component-oriented Real-time distributed Computing (ISORC’06), April 2006.
- A. Betts, G. Bernat, Raimund Kirner, Peter Puschner, and Ingomar Wenzel ”WCET Coverage for Pipelines”, Techincal report for the ARTIST2 Network of Excellence, August 2006.

Except where stated, all of the work contained within this thesis represents the original contribution of the author.

1 Introduction

In today's society, the dependability of computer-controlled systems manifests itself to a larger degree due to an increasing reliance on their correct functionality. These **embedded** systems are seldom visible to the naked eye since they are usually components of a larger system or machine. The modern world is littered with numerous pervasive examples, including: washing machines, printers, mobile phones, the Anti-lock Breaking System (ABS), and flight control systems for missiles and aircraft.

Embedded systems for which precise operation also depends on timing constraints are called **real-time systems**. A failure in some aspect of the temporal domain has a wide range of possible consequences, depending on the type of application. For instance, a jittery video streaming application is perhaps less of an inconvenience than the delayed release of an airbag after a high-speed impact. It is therefore critical — sometimes even *safety-critical* — that some analytical process has been undertaken to verify the temporal properties of a real-time system before its eventual dispatch into the real world.

The design of a real-time system revolves heavily around a model known as a *task schedule*, which allots computational resources to executing tasks, i.e. programs. Many different *scheduling algorithms* have been invented, all of which depend on a set of temporal properties relevant to each task. One such property is the **Worst-Case Execution Time** (WCET), intuitively described as the longest possible execution time. This is clearly essential in designing and verifying a *feasible* task schedule so that each task can be allocated a portion of CPU time. However, determining the WCET is not trivial because execution times vary as a consequence of underlying software and hardware properties. On the one hand, different input vectors cause deviations in the path followed through the software. On the other hand, the time taken for each instruction to complete depends largely on the hardware architecture. Due to these characteristics, **WCET estimates** are sought in which a typical requirement is to bound the actual WCET so that neither the task schedule nor the verification process are compromised. Yet, simply providing a *safe* upper bound is tempered by the desire for accuracy because embedded resources are restricted and need to be maximised accordingly. The epitome of **WCET analysis** is to therefore compute a WCET estimate that is the actual WCET.

1.1 Motivation

Mainstream industrial approaches for obtaining WCET estimates remain predominantly *ad hoc*. This is because the WCET is taken to be the longest observed **end-to-end** execution time during functional testing [117], using a particular kind of coverage criteria, such as Modified Condition/Decision Coverage (MC/DC) [23]. Sometimes, through sheer lack of confidence in the measured WCET, the observed time is factored in an attempt to bypass any optimism. However, this provides no guarantee of safety as the factoring scale is usually based on engineering wisdom, which might not sufficiently bound the actual WCET. On the other hand, if the actual WCET has been captured, the factoring is likely to lead to a very pessimistic WCET estimate.

Drawing motivation from this deficiency, **Static Analysis** (SA) emerged towards the end of the 1980s [94], which models the software and hardware instead of executing the program; a WCET estimate is then computed from these models. The most appealing aspect of SA is that it provides the framework for formal proofs — due to the properties of these models — that demonstrate the safety of the computed WCET estimate. Moreover, the embedded market has traditionally been dominated by simple and *predictable* processors (4-, 8-, and 16-bit), a trend which is due in part to power consumption and cost issues [100]. This considerably eases accurate and safe processor modelling since the effect of the CPU on the execution time of instructions is easily determined. However, processor modelling retains a number of undesirable features that are snowballing with the prevalence of significantly advanced CPU designs within the real-time sector.

First, there is an intrinsic SA requirement for *predictability* at each stage of the analysis, which is jeopardised in the presence of more advanced processors that include caches, dynamic branch predictors, and out-of-order execution units. Evidence suggests that the uptake of these processors within the embedded market is escalating [44], especially because core industries (e.g., the avionic and the automotive) request increased performance. This point is illustrated by the lane departure warning system implemented by Hella [24]; they specifically use Freescale Semiconductor's 32-bit MPC5200 microprocessor due to its computational power. Producing *precise* models of such processors quickly becomes intractable because of the degree of unpredictability introduced. Especially significant is the high level of interference between operations of disparate units; for example, an incorrect branch prediction pollutes the cache. The typical workaround is to decompose the analysis into specific speed-up features and merge the results together at some subsequent stage. However, some pessimism is inherent in such an analysis simply because the normal mode of processing encompasses concurrent operation of all features. Worse yet, this type of decomposition could yield unsafe WCET estimates because of *timing anomalies* [79, 123] whereby local worst-case behaviour, such as a cache miss, does not necessarily lead to the global WCET.

The second issue with SA ties in with the seemingly relentless increase in transistor density, and hence the advent of the *System on a Chip* (SoC). These systems can pack a mixture of one or more mircocontrollers, microprocessors, or Digital Signal Processor (DSP) cores — together with different memory storage mediums (e.g. ROM, RAM, Flash) — onto a single chip. A prime example is the TriCore^{TM} architecture produced by Infineon, which is a single core 32-bit microcontroller-DSP architecture that provides configurable memory in the size and type dimensions. In these cases, it is not sufficient to build a model of the CPU because of the tight coupling between peripheral units. For instance, the worst scenario could depend on the relative bus speed between the CPU and the on-chip memory. Following a similar decomposition strategy to that of processor modelling leads to yet more pessimism.

The third problem that accompanies hardware modelling is the reliance on the processor manufacturer to publish details of internal operation and implementation. In many cases, such sensitive information is withheld because of intellectual property and competition. This is also true of any hardware synthesis performed by the manufacturer, e.g. using VHDL. However, even when manuals are produced, they are often error strewn [37], and this challenges the validity of any model originating from this source.

The final bugbear of SA hardware modelling concerns the monumental effort required, which is highlighted by each of the above three points. Without question, undertaking this for high-end processors occupies significant resources, both in terms of time and money. For companies operating under strict time-to-market pressures, or for those with limited budgets, this could prove a decisive factor. This is more notable because state-of-the-art SA modelling techniques lag behind cutting-edge processor design [93]. Moreover, replacing or upgrading from one processor to another demands a fresh redesign of the model, incurring similar costs.

1.2 Contribution

This thesis contends that, in order to compute accurate WCET estimates — and not necessarily bounds thereof — a **Hybrid Measurement-Based** (HMB) framework should be employed rather than a pure end-to-end or SA approach. In particular, we believe that there are a myriad of **mission-critical** systems for which absolute upper bounds cause vast underutilisation of system resources, and as such would be ignored by the industrial sector due to the margin of pessimism. In reality, real-time systems are governed by self-checking and recovery mechanisms in case the actual WCET ever exceeds the computed estimate.

This thesis is based upon two points of contention. First, increasingly complex processors are emerging in the mission-critical sector of the embedded market, and building processor models requires predictable hardware. Computation of WCET estimates by SA techniques is therefore closely dependent on the accuracy of these models, whereas we argue that the most suitable model to analyse is the processor *itself*.

Second, existing end-to-end techniques rely on finding the actual worst-case test vector, thus the *longest path* could be missed. This is accentuated even further by higher-performance processors because the longest path also depends on the architectural state. Therefore, the accuracy of such estimates is intrinsically tied to the quality of test data and the percentage of coverage achieved, but even these remain insufficient as they solely target functional properties.

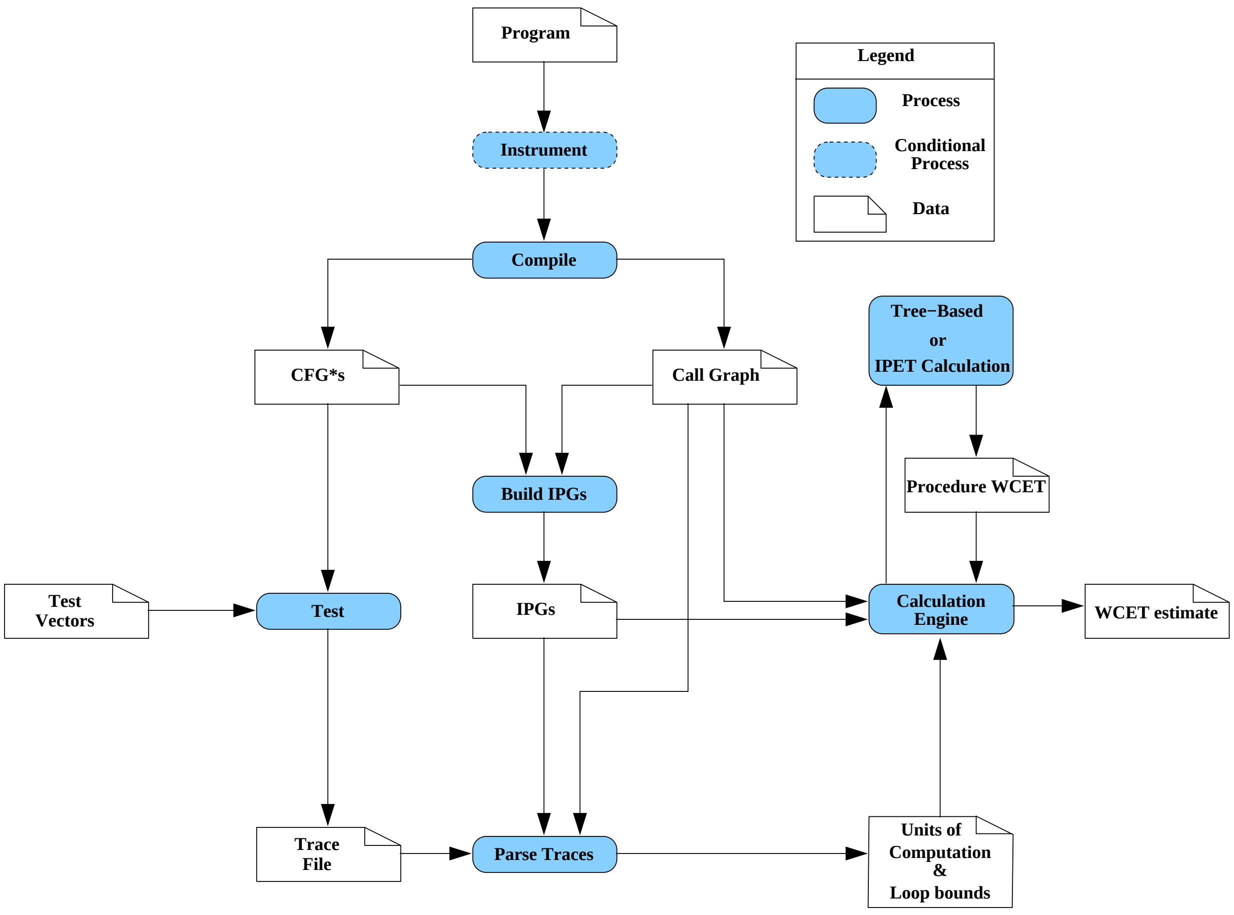


Figure 1.1. Overview of our WCET Toolchain

To prove the thesis, we develop a HMB framework to compute WCET estimates, a schematic overview of which is presented in Figure 1.1. In essence, we measure the execution times of program segments through **instrumentation points**[1] (ipoints). Upon execution during testing, these ipoints generate a number of **timing traces** of execution through the program. The static analysis part of our analysis then recombines

[1]Although we use the term ”instrumentation points”, we emphasise that these are are notional points in the program which do *not* always require software probes. There are often alternative means by which timing traces can be extracted — namely, through hardware debug interfaces — even if in practice the software solution proves to be the most desirable.

the measured execution times by means of a novel program model, the **Instrumentation Point Graph** (IPG). To realise this, the timing traces are first parsed in order to extract the basic unit of computation and, if required, loop bounds. The calculation engine then uses either a tree-based approach or the Implicit Path Enumeration Technique (IPET) — in both cases operating on the IPG — to compute a WCET estimate.

In developing this framework, the following key contributions are provided:

- **A new program model:** Chapter 3 motivates and presents the IPG, in which the atomic unit of computation is the transition among ipoints, as opposed to the more traditional basic block. We show how the IPG is constructed from a **CFG*** — which is basically an intermediate form similar to the Control Flow Graph (CFG) — through two algorithms. The first of these is very simple but its weakness is that it requires support from state-of-the-art techniques to identify loops in the IPG. We show that these often fail due to the arbitrariness of IPG *irreducibility* [3, 83]. Our solution is a more complex algorithm that instead uses the **Loop-Nesting Tree** (LNT) of the CFG* during construction, and hence identifies all such loops on the fly.

  This chapter also describes usage of the IPG in the context of **interprodecural analysis**. We describe a virtual inlining technique, **master ipoint inlining**, which avoids duplication of the callee's IPG at each call site in the caller and produces one IPG per procedure. We show how to use the set of IPGs in conjunction with the **call graph** to parse the trace file so as to extract WCET data on a per context basis. In particular, we show how to extract observed loop bounds using properties of IPG loops. The final contribution of this chapter describes how to use the call graph and the set of IPG loops in order to compute a final WCET estimate.

- **A new tree-based calculation engine:** Chapter 4 presents a new hierarchical form — the **Itree** — to facilitate tree-based calculations on the IPG. To this end, we present an algorithm to decompose an *arbitrary irreducible* IPG into an Itree, as well as the **timing schema** that drives the WCET computation.

  We show how to use the *dominance* relations to detect **Single-Entry, Single-Exit** (SESE), **Single-Entry, Multiple-Exit** (SEME), and **Multiple-Entry, Single-**

Exit (MESE) regions in acyclic IPGs. In particular, we show how these properties prevent redundant traversals of acyclic IPGs (whilst building a hierarchical representation) and how this results in a *forest of Itrees*, an **Iforest**.

We show that, when modelling cyclic IPGs in Itree form, irreducibility is generally the biggest contributing factor to overestimation in the calculation engine.

- **Remodelling of the IPET:** Chapter 5 describes how to remodel the IPET, a calculation technique that constructs an Integer Linear Program (ILP), so that it applies to arbitrary irreducible IPGs. We show that, in contrast to the Itree, the IPET does not cause any undue overestimation in the calculation on the IPG.
- **A prototype tool:** Chapter 6 describes the prototype tool developed to support HMB WCET calculations, which takes a disassembled program and a trace file as input. We use the prototype tool to evaluate the techniques presented with a large-scale industrial application that existing SA techniques cannot analyse due to the non-disclosure of fundamental system properties — in particular, both program source and processor configuration are withheld.

  We compare the WCET estimate that our HMB framework computes with end-to-end measurements. Our results indicate that our tree-based calculation engine can be competitive with the remodelled IPET, that analysis of execution contexts is essential to the quality of the WCET estimate when the IPG is the program model, and that computation of the WCET estimate is less sensitive to the amount of coverage achieved than end-to-end estimates.

In addition to these main chapters, a thorough account of related work is provided in Chapter 2, which also discusses our key assumptions regarding how timing traces are extracted and how testing is implemented. Chapter 7 draws general conclusions and indicates future directions of work. Appendix A reviews core terminology and notation used throughout the thesis, with particular regard to graphs, trees, and control flow analysis. It is recommended that the reader familiarises oneself with the material in the appendix before embarking on Chapters 3 through 5.

2 Background and Related Work

The central topic of this thesis is Hybrid Measurement-Based (HMB) WCET analysis. Section 2.1 begins by reviewing core material in real-time systems in order to provide the setting for WCET analysis research. This leads to Section 2.2, which is devoted to a survey of techniques used to generate WCET estimates. In particular, this section examines in detail the program and processor models used by Static Analysis (SA) and the testing techniques used by existing end-to-end measurements. An overview of the tool support available for WCET analysis from commercial and academic horizons is also presented. We conclude with a summary of the chapter in Section 2.3.

2.1 Real-Time Systems

Chapter 1 provided an intuitive distinction between embedded and real-time systems. Rather than rephrase previous definitions in the search for formalism, we present the following as provided by Laplante [64]:

> A **real-time system** is one whose logical correctness is based on both the correctness of the outputs and their timeliness.

The diversity implicated in this definition is mirrored by the extensive range of systems to which it applies: multimedia hand-held devices, heart pacemakers, and aircraft control systems, to name but a few. Clearly, the cost of failure of such systems differs enormously, but when loss of life is a possibility such systems are branded **safety-critical**. Other systems for which correct functionality is vital to the fulfilment of its goal are coined **mission-critical**, although the differences between them are not always abundantly transparent.

2.1.1 Scheduling

In reasoning about the temporal requirements of a real-time system, i.e. whether they can be satisfied or not, a model of the computations performed is constructed. Each sequential computation is referred to as a task, and the collective set of computations as a task set $\mathsf{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$. For each task, a task instance is a fresh invocation of that task, the regularity of which provides a categorisation as follows:

- Periodic if these arrive with a constant period,
- Aperiodic if these arrive irregularly,
- Sporadic if there is a minimum interarrival time between arrivals.

Each periodic task τ_i has several associated parameters that are assumed known:

- The release time R_i is the time at which the task becomes ready for execution.
- The period P_i is the regularity at which a new instance of τ_i is initiated.
- The deadline D_i is the time at which the computation should have completed.
- The worst-case execution time C_i is the time needed for the processor to perform uninterrupted execution of the task.

Given the task model T, a **schedule** is an assignment of tasks to the processor, so that each task is executed until completion [21]. Scheduling algorithms retain a number of characteristics that facilitate their categorisation. First, the actual selection policy of which task to execute can be decided on-line or off-line. On-line mechanisms choose the task during run-time, whereas off-line ones decide before the system is ever run. Second, how the scheduling decision is determined depends either on fixed priorities or on dynamic priorities of the task set. Fixed priority schemes assume that particular parameters remain fixed to determine a priority ordering of tasks statically. On the other hand, dynamic policies permit such parameters to change at run-time and utilise these dynamic values to determine the priority ordering. Third, the scheduler can operate a pre-emptive or a non-pre-emptive approach according to the priority of tasks. A pre-emptive algorithm permits a higher priority task to interrupt the execution of a lower priority task. In comparison, a non-pre-emptive algorithm forces the lower priority task to finish before invoking that of the higher priority.

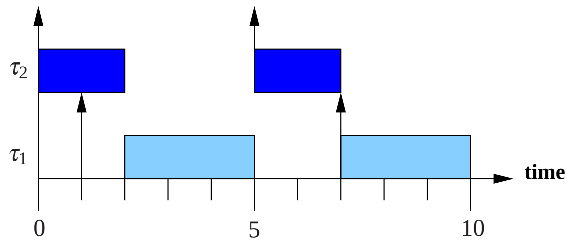
Three classical scheduling policies are:

- Rate monotonic [76]: A pre-emptive, fixed priority policy in which tasks with shorter periods between instances are given higher priority over those with longer periods.
- Deadline monotonic [67]: A pre-emptive, fixed priority policy in which tasks with shorter deadlines are given higher priority over those with longer deadlines.
- Earliest Deadline First (EDF) [76]: A pre-emptive, dynamic priority policy in which the task with the earliest deadline is assigned the highest priority. The difference between EDF and deadline monotonic is that the release times of tasks are modified during execution, and hence priorities evolve; therefore, EDF decides using these modified priorities.

Figure 2.1. Example Task Schedule using the Deadline Monotonic Algorithm.

Task	Release time	Period	WCET
τ_1	1	6	3
τ_2	0	5	2

(a) The task set and associated parameters.



(b) The task schedule in which arrows represent an invocation of each respective task. Task τ_2 has a higher priority than τ_1 since its deadline is shorter.

To illustrate a sample task schedule using the deadline monotonic algorithm, consider an example task set, the respective parameters of each task, and the resultant schedule in Figure 2.1. The task set consists of two periodic tasks τ_1 and τ_2 in which the deadline is assumed to be equal to the period; τ_2 has a shorter deadline than τ_1, thus it is deemed of a higher priority. The schedule in Figure 2.1(b) is over discrete time units whereby invocations of tasks is represented by arrows. Note that τ_1 does not pre-empt τ_2 at its release time due to the priority ordering. However, it does commence execution once τ_2 has finished.

When a task set can be scheduled in accordance with specified constraints, e.g. all periodic tasks meet their deadline, it is termed feasible. Moreover, the task set is schedulable if there exists at least one scheduling policy that is feasible. A **schedulability test** determines whether the task set is feasible for a particular scheduling algorithm.

Liu and Layland [76] provided a schedulability test for the rate monotonic algorithm:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} < n\left(2^{1/n} - 1\right) \tag{2.1}$$

In this inequality, the left-hand side represents the total processor utilisation and the right-hand side is the utilisation bound, which converges towards 0.69. This is a sufficient schedulability test but not a necessary one because a test set could fail the test yet still be schedulable according to the rate monotonic algorithm. Note that this is also a schedulability test for the deadline monotonic algorithm under the assumption that, for each task, $P_i = D_i$.

Liu and Layland [76] also provided a sufficient and necessary schedulability test for EDF:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{2.2}$$

2.2 Worst-Case Execution Time Analysis

In the previous section, the task model built in the design and verification of real-time systems was discussed. This model provides the framework for scheduling algorithms and their schedulability tests in which a fundamental assumption is that the WCET of each task is available and *fixed*. This is evident in the schedulability tests of the rate monotonic and EDF scheduling algorithms shown above in (2.1) and (2.2), respectively. However, in spite of the increased interest in scheduling theory during the 1970s and 1980s, research in WCET analysis surprisingly remained dormant until the seminal paper by Puschner and Koza [94] aroused interest. Being the great pioneers of WCET analysis, a reminder of their definition of a WCET estimate is in order:

> The Calculated Maximum Execution Time ($MAXT_C$)[1] of a program is the least upper bound for the Application Specific Maximum Execution Time ($MAXT_A$) that can be derived from the task's program code. The $MAXT_A$ of a program is the time it maximally takes to perform its functionality in the given application context, provided that all needed resources are available, the program is not interrupted and the performance of the hardware is known.

We crucially note the term "least upper bound", which indicates that merely providing an upper bound has never been the specific aim of WCET analysis. Besides presenting this definition, several key properties of the WCET problem were noted, namely:

- Providing a WCET estimate for an arbitrary program reduces to the Halting problem. Therefore, to ensure that the WCET problem is decidable, a minimal set of restrictions must be supplied. In particular, these are loop bounds and maximal depth of recursive procedures.
- The timing behaviour of all hardware components should be deterministic since execution times are hardware dependant.
- WCET estimates do not typically account for the interference produced by background activities, such as dynamic RAM refresh, nor that produced by preempting tasks.

Broadly speaking, the computation of WCET estimates is either achieved through SA or end-to-end testing techniques. Due to the conceptual requirement that a WCET estimate should be an upper bound on the actual WCET, SA solutions were the first to impact the literature. These techniques typically combine data derived from two disparate models: **program models** that we review in Section 2.2.1; and **processor models** that we review in Section 2.2.2. However, these models alone do not provide sufficient information to compute a WCET estimate because of the Halting problem, as discussed above. Calculable WCET estimates are therefore ensured by **flow analysis**, which computes path-related properties of the program, and are discussed in

[1]As an historical aside, although they provided the seminal paper on WCET analysis, the name "MAXT" was (unjustly) modified to "WCET" by native English-speaking authors.

Section 2.2.3.

As this thesis develops a HMB framework, we also survey test-oriented and coverage techniques to compute WCET estimates in Section 2.2.4. Finally, tool support for generating WCET estimates is an almost essential requirement, some of which have successfully evolved from academic prototypes into fully-fledged commercial toolsets. Tools having the greatest impact on the field are described in Section 2.2.5.

2.2.1 Program Models and WCET Calculations

In general terms, a program model represents the set of structurally feasible execution paths without considering the semantics of the code. From a WCET analysis perspective, the ultimate application of the program model is to derive the longest path by means of an appropriate calculation technique; how this is performed depends on the type of model. The **Control Flow Graph** (CFG) and the **Abstract Syntax Tree** (AST) are the *de facto* models since they are often a by-product of program compilation. In both the AST and the CFG, the atomic unit of computation is the **basic block**, which is a sequence of consecutive functional instructions (at the assembly/object code level) in which flow of control enters at the beginning and leaves at the end [3, 83]. Typically, the WCET of each basic block is deduced from either a processor model or from measurements.

Path-Based Calculations and the IPET

Both **path-based** approaches and the **Implicit Path Enumeration Technique** (IPET) to calculate WCET estimates require a graph-based program model. The standard model is the CFG, which is a flow graph of basic blocks in which edges represent the control flow relation between them. The graph-based model employed in our HMB framework is the IPG (see Chapter 3), thus these calculation techniques are also applicable to the IPG — Chapter 5 considers remodelling of the IPET towards the IPG.

The simplest way to implement a path-based approach is to enumerate each path through the graph and then select the longest amongst them — the clear benefit is that there is no pessimism in the calculation. However, because programs inevitably contain loops, this is not applicable in any practical setting as path enumeration causes exponential growth in the number of paths, e.g. a loop with bound n containing a unique `if-then-else` construct has 2^n paths. To limit the complexity, therefore, the enumeration of paths should be restricted to sections of straight-line code [105], such as within a loop body. In this way, calculations become localised (in a manner similar to that of a tree-based approach) and integrating global flow analysis data is compromised. However, a path-based approach working at the loop boundary level can virtually unroll paths and thus select a different path on each individual iteration [50].

To ensure a more precise analysis, the calculation must also be aware of infeasible paths. In [106], an algorithm was presented that iteratively computes a longest path until it finds a feasible execution path. This is basically done by rewriting the graph — inserting additional vertices and edges so that the infeasible path is not structurally feasible in the modified graph. (Their algorithm also accounts for pipeline effects.) Instead of rewriting the graph, infeasible path data can be included directly in the calculation [109]. In this approach, infeasibility data are derived for the acyclic region of each loop, which basically link edges (with branch vertices as sources) that cannot execute in sequence. Armed with this knowledge, the calculation engine then traverses the loop body in reverse topological order and propagates the longest path up to the loop header, discounting infeasible paths along the way. However, their approach is not yet able to handle cross-loop or interprocedural constraints.

The general weakness of path-based approaches is that, in order to incorporate global flow analysis data, the unit of analysis must be extended to complete execution paths. Any form of unrolling or graph rewriting provides a partial solution but can be quite costly. The alternative is to model feasible execution paths as a constraint model on the graph: this is how the IPET [73, 95] operates. The key observation to the IPET is that it generates bounds on the execution count of the atomic unit of computation, e.g. basic blocks, without explicitly enumerating all paths and can thus incorporate global flow analysis data. This is achieved by producing an **Integer Linear Program** (ILP) or a **Constraint Program** (CP). In both cases, an objective function needs to be maximised subject to a set of constraints expressing flow information. We defer a detailed examination of this calculation technique until Chapter 5.

One weakness of the basic constraint model of the IPET is that it simply provides a bound on the execution count of each variable. This means that it cannot sufficiently capture more detailed flow information, e.g. that a basic block is never executed on the first 20 iterations of a loop. The IPET has since been retargetted towards a scope graph [40], which is a CFG partitioned into regions, i.e. scopes. Every scope is essentially a hierarchical component of the program, such as a loop or a procedure, and carries a set of flow facts (derived from SA) that describe its dynamic properties. Flow facts can span across scopes. From the set of flow facts and the set of scopes, virtual scopes are created, which are basically duplicated scopes for which linear constraints can correctly bound the execution count of variables according to the flow facts. These duplicated scopes can be seen as “mini” constraint models that, when pieced together, give a complete description of flow through the CFG.

The long-standing issue with the IPET is that, in the worst case, solutions to ILPs (or CPs) have exponential time complexity [30]. This is particularly an issue when integrating global flow data into the constraint model because, otherwise, the problem can be reduced to the network flow problem [73] for which there are known polynomial-time solutions [30]. For this reason, a so-called clustered calculation [42] has been explored, which attempts to avoid the creation of a single global constraint model. This again uses the scope graph and the set of flow facts. Basically, flow facts determine the smallest unit of analysis (in the graph) to which a localised calculation can be confined, either using the IPET or an existing path-based approach. However, as the authors acknowledge, if flow data span the entire CFG, a global constraint model is the only option.

Tree-Based Calculations

A tree-based approach to calculating a WCET estimate was first proposed in the seminal paper on WCET analysis [94]. The original calculation engine operated at the source level of a program on its AST representation. This is created by parsing program source and identifying sequence, selective, and iterative constructs. Each internal vertex of an AST is one of these constructs and leaves are sequences of statements, i.e. basic blocks.

The calculation engine operates by traversing the AST bottom-up whilst conceptually collapsing each construct according to a particular timing rule, collectively referred to as the **timing schema** [88]. The original timing rules are shown in Table 2.1 in which: *S* is a basic block or an interior vertex; *C* is a conditional expression; *n* is an upper bound on the number of loop iterations.

Language construct	Timing rule
Sequence: $S_1, S_2 \ldots, S_n$	$\sum_{i=1}^{n} WCET(S_i)$
Alternative: `if` C `then` S_1 `else` S_2	$WCET(C) + max(WCET(S_1), WCET(S_2))$
Iteration: `while` C `do` S	$WCET(C) + (WCET(C) + WCET(S)) * n$

Table 2.1. Timing Schema for Tree-Based Calculations

However, the most poignant shortcoming of the original timing schema was its inability to account for the effect of hardware features, since it assumed fixed execution times of basic blocks. The timing schema has since been extended to account for the effects of pipelines and instruction caches on RISC architectures [75]. Colin and Puaut [28] proposed a data structure to represent the simulation results obtained from pipeline, cache, and branch prediction modelling in a modular way during the tree-based calculation.

There are also issues with the actual hierarchical representation. First, the timing schema impose localised calculations, and thus more complex flow analysis data, e.g. relating to infeasible paths or non-rectangular loops (where the number of iterations of an inner loop depends on the number of iterations of an outer loop), cannot be integrated into the calculation. The WCET estimate is thus less precise. This deficiency motivated the work in [25], which introduces a scope tree that has similar properties to those of the AST. The scope tree can handle more complicated flow analysis data by duplicating sub-trees.

Second, tree-based calculations are only suitable for well-structured programs whereby hierarchical relations hold. This essentially precludes any programs containing high-level statements that abruptly redirect flow of control to a different region, e.g. `goto`, `break`, and `continue`. Furthermore, there has to be a clear mapping between the source-level constructs in the AST and the compiled code at the intermediate level.

This is because processor models determine the WCETs of basic blocks, and these must then be transferred onto the AST for the calculation. However, as more complex hardware architectures prevail so too does the role of optimising compilers, which vastly complicate the mapping. The crux of the problem is that the AST should be created from the graph-based model, i.e. the CFG, in order to be compatible with the information extracted at the intermediate code level. (This is main reason that usage of the CFG is significantly more widespread as it can handle arbitrary program structure, including aggressively optimised code.)

In Chapter 4, we present an algorithm that does construct a hierarchical data structure, the **Itree**, from a graph-based program model (i.e. the IPG). As the IPG itself is derived from a program model similar to the CFG (as discussed in Chapter 3), this means that the Itree is able to support more control structures, e.g. `break` and `continue` statements, than those associated with well-structured programs. The Itree also models arbitrary irreducible loops in the IPG, although we show that these unstructured sections of code are the principal cause of inaccuracies in the resultant WCET estimate.

It might appear that tree-based approaches are inherently so weak that their study does not warrant further investigation. However, one clear advantage of a tree-based approach is that it allows computation of **probabilistic WCET estimates**; these can subsequently be used in schedulability analyses [20, 35] that assume execution times as probability distributions. This has motivated the introduction of a probabilistic timing schema [15, 16] which combines Execution Time Profile (ETP) instead of integer values. Each ETP represents the frequency of execution times for a particular code sequence (normally basic blocks), which are usually obtained from measurements. The algebra is then able to combine dependent ETPs arising from hardware effects that have not been captured in the measurement stage.

Interprocedural Analysis and Contexts

The calculation techniques described above typically operate on a per procedure basis, but there is clearly a need to drive calculations across procedure boundaries.

Interprocedural analysis is often aided by the **call graph** in which procedures are vertices and the procedure calls are modelled by its edges. To achieve maximum precision in the calculation, it is not sufficient to consider that the WCET of each call from a procedure f to a procedure g is the same at every call site. This neglects the **context** of the call and can be a major source of pessimism. For example, the loop bounds in a callee could be parametrised by the arguments supplied in the procedure call, thus simply assuming the maximum bound leads to overestimation across all calls to that callee.

Interprocedural analysis within the scope of the IPET has been researched by Theiling [113], which includes support for recursion. They consider a context to be a call stack state — a so-called call string. In order to manage complexity, the length of the call string can be constrained by a parameter, i.e. it determines how far back to go in the execution history.

In Chapter 3 we describe our interprocedural analysis technique using the IPG. The main difference between our work and that of Theiling is that we discover contexts from timing traces. Furthermore, our calculation engine is modularised, which means that either a tree-based approach, a path-based approach or the IPET can be chosen to calculate the WCET of each individual context.

A different approach to context-sensitive analysis is to represent the WCET of a piece of code as an algebraic expression, more widely referred to as **parametric analysis** [14, 25, 120]. This is especially useful for code that is heavily input-data dependent, such as an image processing application. In [14, 25] the formulae are constructed at the source level and a computational algebra system like Mathematica or Maple is used to simplify and evaluate these expressions. However, parametric analysis is sometimes performed at run time when the parameters become available in order to make dynamic scheduling decisions [120]. In such cases, it is unrealistic to assume such a computational algebra system is available. Rather, the formulae are much simpler so that the WCET can quickly be evaluated.

2.2.2 Processor Models

Although this thesis is not concerned with processor modelling, a picture of WCET analysis would not be complete without reviewing this area of research, especially because most work has been concentrated here. Moreover, the prime motivation for the development of a HMB framework is that there are a number of deficiencies in existing processor modelling techniques; thus, it is worthwhile reviewing the state of the art to support this claim.

A processor model synthesises the functional and temporal behaviour of an actual processor, which SA uses to compute the WCET of basic blocks without executing the software on the target hardware. The intricacy involved in providing a safe yet accurate model largely depends on the speed-up features present in the processor. **Pipelines** and **caches** have both been comprehensively studied. The former is widely used in contemporary embedded systems due to low implementation cost and power consumption. The latter has the greatest effect on the WCET [26, 61]. Some embedded systems wish to speed up memory accesses without the modelling complexity of caches: **scratchpads** provide an elegant workaround. Interest in **branch prediction** and **out-of-order execution** has only recently emerged, mainly because these features are reserved for processors striving for aggressive instruction throughput, which is a less common requirement in embedded systems. These more advanced features proliferate the presence of **timing anomalies**, which potentially invalidate traditional divide-and-conquer strategies. We survey the state of the art in each of these techniques.

Pipelines

Pipelining permits multiple instructions to be in flight simultaneously by exploiting the fact that an instruction must pass through multiple stages to complete execution. This idea contrasts with a non-pipelined architecture whereby the execution of an instruction only begins on completion of the previous instruction.

A pipeline thus consists of a number of stages — its depth — each instruction must pass through and allows several instructions to occupy independent stages. Each instruction normally progresses to the next stage on every clock cycle, although each instruction does not need to pass through each stage. The instruction latency is the number of cycles taken to pass through the pipeline. The idealised latency of an instruction is the pipeline depth, but this is hindered by hazards.

To obtain a safe and accurate WCET estimate in the presence of pipelines, the timing effect over basic block boundaries must be contemplated, since the effect within the basic block is quite easily determined using reservation tables [86]. In many cases, there is a relative speed-up in the execution time (of consecutive basic blocks), in comparison to their individual execution, due to the inherent overlapping of the pipeline. However, hazards between basic blocks can produce an increase in execution time, a so-called **positive timing effect**.

The first contribution in this area was to account for the overlap between adjacent basic blocks [86]. However, this is insufficient in the general case because the effects of pipelines can reach much farther [39]; for example, floating-point instructions occupy functional units much longer than do integer instructions. An alternative to capture such effects is trace-driven simulation [39, 37] in which a trace of a program for a fixed input is recorded and then simulated[2]; this is performed for sequences of basic blocks.

The effect of a pipeline on the WCET can also be analysed by **abstract interpretation** [31]. This is the approach taken in [101] in order to model the SuperSPARC I superscalar pipeline, whereby the abstract state of the pipeline is updated at each basic block until a fixed point solution is reached. However, state explosion must be managed by merging abstract states at selected (merge) vertices in the CFG; this makes the analysis more conservative, and ultimately translates into a loss of accuracy in the WCET estimation.

Caches and Scratchpads

For fast CPUs, a severe performance penalty would be incurred if both instructions and data were continually fetched from main memory, since the access time is relatively much slower and CPU execution would stall. A cache is an on-chip memory — thus

[2]Comparatively, execution-driven simulation dynamically interprets instructions for variable input.

providing significantly faster access times — whose contents subset the next lower level of memory, resulting in a memory hierarchy. Caches can contain instructions (instruction caches), data (data caches), or a mixture of both (unified caches). General-purpose faster processors typically contain two-levels of cache, normally configured with separate on-chip instruction and data caches at level one (L1), and a unified on-chip/off-chip cache at level two (L2) [56].

When the requested data resides in the cache then a cache hit ensues. Otherwise, it is a cache miss and a penalty is accrued whilst a fixed-size set of data, termed a cache line, is retrieved from the next lower level in the memory hierarchy. The location in which the incoming line is placed depends on the configuration as follows [53]:

- In a direct-mapped scheme, the block is placed in a specific location, usually as a function of its address.
- In a set-associative scheme, the cache is split between a number of sets. The block is then mapped onto a set, as a function of its address, and placed anywhere within that set.
- In a fully-associative scheme, the block can reside anywhere in the cache.

For set-associative and fully-associative configurations, a cache miss forces a block resident in cache to be displaced in order to accommodate the incoming block. There are typically three block replacement strategies:

1. The Least Recently Used (LRU) approach removes the block that has not been used for the longest time. To determine the exact LRU item to be evicted requires a number of status bits to track when each block was accessed, which becomes expensive when the number of blocks in each set is large. Instead, the pseudo-LRU approximates the LRU item, which is a very good approximation of LRU [81].
2. The First In, First Out (FIFO) strategy removes the block that has been resident for the longest duration.
3. The random approach arbitrarily chooses the block to be removed.

Note that direct-mapped caches do not require a cache replacement strategy since there is a unique location for each block. In serving a cache miss, the CPU stalls until the entire block has been transferred. A wrap-around fill (critical word first [53]) deviates from such behaviour by transferring the requested word first and freeing the CPU to continue execution as soon as the data is available, whilst the transfer completes in parallel.

Scratchpads provide on-chip memory with predictable access latencies which are comparable to those of caches. The contents of a scratchpad are mapped into the address space of the processor and, unlike a cache, are normally allocated at compile time. Therefore, when the CPU requests data that falls within the range of the scratchpad, it fetches from the scratchpad; otherwise, it must fetch from the appropriate level in the memory hierarchy.

Caches significantly complicate WCET analysis because of the difficulty in determining which instructions and data reside in cache at a particular program point: their presence usually equates to a faster execution[3], otherwise a cache miss occurs and the associated penalty must be considered. Simply assuming uniform cache misses, or disabling the cache to force predictability, is likely to lead to gross overestimation and an underutilisation of processor resources. In general, predicting the caching behaviour of instructions is more straightforward than it is for data because the addresses of instructions are fixed and known at compile time. Comparatively, determining absolute data addresses at compile time is complicated by, for example, pointers. It is further aggravated by unique instructions whose data access different memory locations at run time, e.g. load/store instructions. Scratchpads, on the other hand, are often a desirable alternative because of their predictability.

One of the first techniques to emerge for WCET cache analysis was **Static Cache Simulation** [6, 85, 125, 126], which effectively simulates the effect of every path through the program on the state of the cache.

More specifically, static cache simulation is founded on **data-flow analysis** [3] and represents the **Abstract Cache State** (ACS) at each point in the program (that is, in its CFG). The ACS holds the program lines that *may* reside in cache if execution reaches that particular point, as opposed to the **Concrete Cache State** (CCS) that would otherwise represent the *exact* program lines resident in cache. The ACS at

[3]Due to timing anomalies, this might not always be the case.

each vertex is computed by taking the union of the output states of each immediate predecessor and simulating the effect of the instructions or data in that vertex on the state of the cache. The ACSs are propagated across the program through iteration until a fixpoint solution has been reached, which is a set of stable ACSs. Termination is guaranteed by the properties of the underlying data-flow framework.

Having computed ACSs, the next step is to determine if each instruction or data reference will either be a cache hit or a cache miss during program execution. It is not always possible, however, to categorise a reference as a hit or a miss, so instructions are instead categorised as follows:

- Always miss: the instruction is not guaranteed to be in cache.
- Always hit: the instruction is guaranteed to be in cache.
- First miss: the first access will miss but all subsequent accesses will hit.
- First hit: the first access will hit but all subsequent accesses will miss.
- Unknown: the caching behaviour of an instruction cannot be categorised. This is handled as always miss.

Static cache simulation has been extended to handle set-associative caches assuming perfect LRU [85, 125], multi-level caches [84], data caches [125, 126], and instruction caches employing a wrap-around fill mechanism [126]. However, it has the following limitations:

- At merge vertices in the CFG, cache states of all immediate predecessors must be unioned together to limit the complexity of the analysis (which would otherwise grow exponentially because all paths would effectively be simulated). If the cache states are very different then the model becomes pessimistic. This can be particularly problematic for loops (unless the first iteration is virtually unrolled) because no distinction is made between the first iteration of the loop and all other iterations.
- Replacement policies other than perfect LRU, e.g. pseudo LRU, cause unpredictability and force the analysis to be more conservative because the analysis should reflect the worst possible state of the cache at each vertex.

- Analysis overheads can be considerable, both in terms of time and space. The underlying iterative data-flow framework is known to have quadratic time complexity in the worst case because it must make several passes over the CFG until the cache states become stable from the initial cache state (of all invalid lines). On each pass, set unions must be performed at each vertex, which are typically quite slow especially when sets are not sparse as in the case of a cache model.

A similar way to categorise the caching behaviour of instructions and data is through abstract interpretation [43, 114], which combines ACS through a join function. In this work, three different types of analyses are performed:

1. Must analysis: determines which lines are always in cache at a particular vertex in the CFG. The join function is effectively a set intersection of the ACSs.
2. May analysis: determines which lines are never in cache by computing the set of lines that may be in cache. The join function is effectively a set union of the ACSs.
3. Persistence analysis: determines which lines are always in cache once loaded. The join function is effectively a set union of the ACSs.

After these analyses, memory accesses can be classified as always hit, always miss, persistent, or not classified in a similar manner to the categorisations proposed by static cache simulation. Likewise, it also suffers from the above mentioned problems.

Bounding the worst-case performance of data caches was originally studied in [60], which proposed two techniques. The first reduces the number of load/store instructions that are incorrectly classified as dynamic load/stores instructions; this is done by using data-flow analysis on the base register of such instructions. The second determines the maximum number of cache misses for array accesses in loops by using the pigeonhole principle. The approach, however, is limited to direct-mapped caches in which the entire loops fits in cache.

In a similar vein, data memory accesses can be classified as predictable or unpredictable [107]. Predictable accesses are those produced by scalar variables and predefined array accesses; otherwise, they are unpredictable. The impact of an unpredictable memory access on the current cache state, and its respective number of cache misses, are both bounded by observing that at most one block can be evicted from the cache due to that access.

However, effective modelling of data caches is impeded by the presence of pointers because addresses are not known at compile time. In these cases, it is possible to compute a conservative set of memory locations that each pointer references during program execution using pointer analysis techniques [83]. This leads to more overestimation, which is especially problematic given that C, a pointer-driven language, largely dominates the embedded sector.

Branch Prediction

In general, program execution does not proceed for long without evaluating a branch instruction that can alter the flow of control. Either the flow of control is always redirected (unconditional branch) or else it is decided at run time (conditional branch). Both types of branches can cause pipeline stalls because the CPU does not know the target of successive instructions. A Branch Target Buffer (BTB) stores the targets of unconditional and conditional branches. However, in deeply-pipelined processors where stalls seriously degrade instruction throughput, a branch prediction mechanism is inevitable, which tries to guess the outcome of conditional branches and hence keep the pipeline full.

A static branch prediction scheme is a compile-time directive that assigns a prediction to each conditional so that it is always predicted the same way during execution. However, to reach near optimal branch prediction accuracy, dynamic schemes are mandatory. In its simplest form, a single-level predictor indexes a Branch History Table (BHT) with the low-order bits of the branch address. Each entry of the BHT maps to either a one- or two-bit counter, which returns the prediction.

The first dynamic branch predictor studied was the Intel® Pentium® architecture [54]. The idea was to bound the number of mispredictions arising through the BTB [27] by defining branch instructions either as history-predicted if it is the BTB at prediction time, or default-predicted otherwise. These definitions allow branches to be classified as follows:

- Always default-predicted: the branch is always predicted not taken.
- First default-predicted: the branch is default-predicted on the first prediction, and history-predicted thereafter.
- First unknown: the branch is either default-predicted or history-predicted on the first prediction, and history-predicted thereafter.
- Always unknown: the branch prediction is never known.

In terms of BHTs, a global history register has been modelled in which each entry indexes a one-bit counter, although it can be extended to handle two-bit counters [82]. The number of mispredictions for each branch is incorporated into the IPET through additional linear constraints. Following this direction, the idea was extended in order to model the interaction between cache and instruction cache by making changes to the cache conflict graph [68].

Engblom has quantified the effects of dynamic branch predictors on WCET analysis by investigating the behaviour of the Intel® Pentium® III and 4 [54], the AMD Athlon$^{\text{TM}}$ [5], and Sun UltraSparc II and III [80] architectures [38]. These processors use the most sophisticated branch prediction mechanism: a two-level predictor in which a history register tracks the outcome of the most recent branches and indexes the BHT. Englom concludes that these predictors are not suitable for SA because, for example, there are cases where executing more iterations of a loop takes less time than executing fewer iterations. However, this counter-intuitive behaviour has since been explained with a theorem that bounds the number of mispredictions for nested loops [10].

One severe limitation of these WCET analysis techniques to analyse branch predictors is that they assume an absence of aliasing in the BHT. Aliasing describes the situation whereby branches compete for the same entry due to space restrictions in the BHT. Constructive aliasing is beneficial as branches correctly update the prediction for each branch mapping to that entry. However, destructive aliasing — where branches with different run-time behaviour result in more mispredictions — is much more common and is the largest limiting factor on prediction accuracy [127]. In the worst case, therefore, the presence of destructive aliasing would force static analysis into assuming a misprediction on each prediction, and hence a much more conservative analysis.

Out-of-Order Execution

To achieve the highest instruction throughput in a pipeline requires out-of-order execution, which permits instructions to be issued and executed in a different order to that ordained by the program. This prevents unnecessary stalls caused by in-order execution because an instruction can be dispatched to an idle functional unit once its operands are available. In theory, a processor employing out-of-order execution represents the most challenging aspect of CPU modelling since its operation depends on the support of other microarchitectural features, such as branch prediction, and hence introduces the greatest amount of unpredictability. The outcome is that there has been very little effort in this area.

A technique to bound the WCET of each basic block when operating with out-of-order resources has been proposed [69]. To this end, each basic block has an execution graph in which there is a vertex for each instruction and pipeline stage, and edges represent dependencies, e.g. data or resource. Every basic block was considered in isolation. However, their approach is only applicable under a number of simplifying architectural assumptions [36], e.g. a single-issue pipeline, a 4-entry instruction buffer and a 8-entry reorder buffer. Furthermore, although the authors claim to account for execution context of surrounding basic blocks in later work [70], they have not proven the absence of long-running timing effects in their model [98].

A possible way to handle out-of-order execution is to make hardware more predictable [98]. This means incorporating additional logic that decides if the instructions of a basic block can enter the pipeline. The decision depends on whether these instructions will be stalled by a structural or data hazards caused by some previous instruction. In this way, the execution of each basic block is independent from all others and long-running timing effects can never arise.

Timing Anomalies

Modern architectural trends complicate processor modelling due to the parallelism existing between units and the interference between them. For instance, a branch misprediction can result in instructions of the wrong path being fetched in cache. To handle the complexity, SA has adopted the traditional divide-and-conquer mindset: provide a WCET model of each unit and combine their effects in some subsequent analysis. However, this could yield *unsafe* WCET estimates because of the presence of timing anomalies [78], which Wenzel *et al.* have succinctly summarised as follows [123]:

> A timing anomaly is the unexpected deviation of real hardware behaviour contrasted with the modelled one, namely in the sense that predictions from models become wrong. This unexpected behaviour could lead to erroneous calculation results by WCET analysis when actually implemented. Thus, the concept of timing anomalies rather relates to the WCET analysis modelling process and does not denote malicious behaviour at runtime.

Timing anomalies were first reported by Lundqvist and Stenström [79]. In particular, they outlined three types:

1. Data cache hits, as opposed to misses, can lead to the WCET.
2. Cache miss penalties can be larger than expected due to changes in the instruction schedule.
3. The increase in cache miss penalties is not always bounded by a constant, but can be proportional to the length of the program.

They also claimed that timing anomalies were absent in processors that had exclusive in-order resources. However, an example was presented in [123] that disproved this claim, since timing anomalies can incur in processors with multiple in-order resources serving the same instructions. They instead proved that processors that do not allow resource allocations, i.e. assigning instructions to a given functional unit, are free of timing anomalies. Dynamic branch predictors are also subject to timing anomalous behaviour [11]. In particular, a lower number of branch mispredictions does not necessarily result in a lower WCET.

Improving the WCET

Traditional compiler optimisation has targeted average-case performance given that certain paths are more frequently executed. However, such techniques can be tweaked so that minimising the WCET is the primary goal. This has become an active area of research because of the belief that static modelling is already too complex, or that, with new generations of processors on the horizon, this will soon be the case [18].

Minimisation of the WCET can be achieved through one of the following:

- Controlling the contents of instruction caches [92]. The idea is to divide the program into regions at which the cache is to be reloaded. The contents at each reload point are determined by calculating the longest path and selecting instructions according to their execution frequencies. During program execution, the cache replacement strategy is disabled so that it is effectively locked.
- Statically predicting the outcome of branches [18, 19]. This generally works by initially assuming all branches are mispredicted and assigning predictions to these branches until a stable longest path is found.
- Determining the contents of data scratchpads, either for the entire lifetime of execution [108], or including dynamic updates [34]. For these approaches, the variables on the longest path are assigned to the scratchpad using linear programming techniques.
- Optimising the code [128] through one of three methods:
    1. Superblock creation: this is a duplicate sequence of basic blocks on the worst-case path. The basic blocks (of the superblock) are contiguous in memory so that penalties due to conditional branches are eliminated.
    2. Path duplication: the superblock path in a loop can be duplicated to eliminate more penalties associated with conditional branches.
    3. Loop unrolling: the entire body of a loop is duplicated as opposed to just the longest path.

Each of the these approaches must be aware of the stability of the longest path since it can possibly switch to a different path because of the optimisations. Normally, this is handled through a re-evaluation mechanism until the longest path becomes stable.

2.2.3 Flow Analysis

In order to compute WCET estimates statically, additional flow information is required to bound the iterative or recursive components of a program. Flow information pertains to any dynamic property of the program not captured by the static model, including knowledge of infeasible paths. The challenge is to glean sufficient flow information to enable safe and precise WCET computations. However, crucial to this process is the level at which these properties are derived. On the one hand, it is often more practical to assume the software developer undertakes this task at the source code level by including manual annotations in some specialised syntax. This is based on the hunch that the developer has detailed insight of software functionality. On the other hand, the final calculation stage operates at the object-code level, and therefore after program compilation. The consequence is that the entire process must be mindful of compiler transformations and optimisations to ensure exact correspondence between flow information at the various levels. This is more widely known as the mapping problem [62].

Manual Annotations

Manual annotations were first presented in the seminal paper on WCET analysis [94]. Additional constructs were incorporated into an extended version of C — the MARS-C language. This thread has been pursued further by developing the wcetC language, which is a superset of a subset of ANSI C [62]. The value of this approach is that annotations are mapped simultaneously — and transparently to the user — into object code during compilation, thus overcoming the mapping problem.

In a similar vein, Park [87] presented the Information Description Language (IDL), which supports quite sophisticated path-related information, especially interprocedural relations between high-level statements; for example, ”statement A in procedure `foo` is always executed whenever statement B in procedure `bar` is executed”. Annotations can also be added to SPARK Ada (a subset of the Ada language) programs through standard comment lines [22].

Automatic Analysis

Many researchers argue that annotations are error prone, thus potentially invalidating subsequent calculations. Indeed, Park [87] legislates for defective annotations (supplied through the IDL) by verifying them with assertional program logic. However, this is not a complete remedy due to the mapping problem, since compiler optimisations could still affect, for example, the number of loop iterations. Another problem highlighted by the IDL is that developers must sometimes learn new languages, clearly at additional cost to the project. Maintaining these annotations during successive project iterations is tedious, especially when new or different engineers are introduced. Instead, flow information can be derived automatically without user intervention, typically through SA.

Abstract execution is a way to simulate the program in the abstract domain [41, 46, 47]. This is a technique founded on abstract interpretation, which has abstract values for program variables, i.e. interval ranges for numeric variables, and abstract versions of the program operators. An example is an assignment operation, which calculates a new interval range instead of a unique value. The analysis of interval values at selected program points then enables loop bounds and infeasible paths to be deduced; for example, analysing the interval range of a loop counter variable at loop termination. However, potential state explosion must be managed by merging abstract states at merge points in the program. The effect of merging is loose loop bounds and the inability to uncover infeasible paths, although the analysis remains safe.

In fact, abstract execution is a form of symbolic execution, the latter which executes programs *without* input data. When a branch is encountered that contains a variable with an unknown value, the execution simulates both paths, except that path-merging must also be employed in the context of loops to prevent path explosion [78].

Data-flow analysis [83] is also a popular method to obtain loop bounds, especially since it can be integrated within a compiler framework. In [51, 52] three variations of loops are supported: those with multiple exits; those in which the exit condition is decided in relation to a non-constant loop-invariant variable; those in which the number of iterations of an inner loop depends on the control variable of an outer loop (non-rectangular loops).

One limitation of automatic static analyses is that they cannot always provide precise bounds for all loops due to the Halting problem [62]. In such cases, the user is expected to bridge the knowledge gap, but as we noted above, it is equally unreasonable to expect a user to obtain such information from a simple inspection of the code. Even if a user can supply a **relative loop bound**, i.e. relative to the next outer loop nesting level, it is typically much more difficult to provide an accurate bound on the actual number of executions due to non-rectangular loops.

In Chapter 3, we describe an alternative automatic flow analysis within the scope of trace parsing. The key difference between our technique and others described above is that we derive flow information from observations (in timing traces). This allows our HMB tool to operate without user interaction. In particular, we show how to use properties of the IPG to extract relative loop bounds, which are needed in both tree-based calculations and the IPET. In addition to determining relative loops bounds, our trace parser also counts the overall frequency of execution of loop bodies, and therefore, the analysis becomes more precise (provided testing is good enough).

Recently, an approach that is similar to our work has been explored [9]. They determine loop bounds through a combination of testing and machine learning. Their approach is able to deduce bounds relative to a particular loop-nesting level, including those common to non-rectangular loops. Our approach differs in a number of ways. First, we use properties of the IPG to obtain the loop bounds whereas they use pattern matching techniques. Second, they have not considered how to obtain traces of execution. We show that this actually affects the accuracy of the bound. In both cases, the accuracy of the derived bounds relies heavily on good test vectors, which we discuss further in the next section.

2.2.4 Measurement-Based Techniques

Interest in measurement-based approaches has been afforded increasing attention in recent years. The main motivation for these techniques is that processor modelling is either too complex, too time-consuming, or plainly and simply not possible.

Testing and Coverage Criteria

A myriad of testing techniques have been put forward with respect to functional behaviour. The main purpose of such techniques is to uncover errors, but because these have no bearing on WCET analysis, here we focus on the issues of testing that HMB frameworks must be mindful of.

Typically, testing takes places at the procedural level of a program. The idea of white box techniques is to utilise the structural properties of the CFG to construct **test vectors**. In order to measure the quality of testing, **coverage criteria** [129] are required. Some criteria are more stringent than others, thus the subsumes relationship offers a comparison between different coverage criteria: a criterion A subsumes a criterion B if, and only if, every test vector that satisfies A also satisfies B [23].

Statement coverage ensures that all statements in the program are exercised, and is clearly subsumed by basic block coverage. Full statement coverage can never be achieved in the presence of unreachable code. Branch coverage requires each edge of the CFG to be traversed, which also subsumes statement coverage. However, safety-critical systems require more stringent criteria, such as that provided by Modified Condition/Decision Coverage (MC/DC). In this metric, a condition is a boolean expression containing no boolean operators, whereas a decision is an outcome of a (composite) boolean valued statement in a high-level language. MC/DC is satisfied when every condition in a decision has been exercised, and each condition has been shown to independently affect the decision's outcome [23]. Path coverage is the most stringent but the least practical as programs generally contain loops and thus the number of paths grows exponentially.

End-to-end testing techniques and coverage criteria that target WCET estimation have not been extensively researched due to lack of confidence in the measured execution time. One approach is to use evolutionary algorithms to generate test vectors [121]. Initially, a random population of test vectors is generated, and the execution time is measured. Test vectors generating long execution times obtain high fitness values. New generations of test datum are bred through the combination and mutation mechanisms of evolutionary computation. This procedure continues until a particular stopping criterion has been met, usually when a certain number of generations have evolved. The accuracy of evolutionary testing was then compared with a SA method [122]. Results showed that the WCET estimates were in closer proximity to those obtained by SA, but that there was an underestimation in some cases.

This latter observation highlights the long-standing issue with end-to-end techniques in that they cannot guarantee a bound on the actual WCET. To do so would not only require full path coverage but also full state coverage at the architectural level; these are clearly intractable requirements. Research has thus digressed into HMB approaches in an attempt to combine the best features of both SA and end-to-end measurements. The key feature of such approaches is that measurements for program segments are collected via the testing phase and then recombined using the calculation techniques described in Section 2.2.1.

Instrumentation and Trace Generation

In order to collect measurements of program segments, **instrumentation points** (ipoints) are required. Once triggered during program execution, each ipoint emits a **trace identifier** and is timestamped accordingly, resulting in a timing trace of execution.

Definition 1. *A* ***(timing) trace*** *is a sequence of tuples (i,t) in which i is the trace identifier of an ipoint and t the time of observation.*

Therefore, testing the program with a set of test vectors produces multiple traces that are collated into a **trace file**. It is from this file that **trace parsing** extracts both the WCETs of ipoint transitions and observed loop bounds, both of which are explained further in Chapter 3.

How ipoints are inserted and how timing traces are generated and extracted depends on the mechanism available. Trace generation methods can generally be categorised as follows:

- Simulation: Cycle-accurate simulators, e.g. SimpleScalar [7], allow individual instructions to be traced (through the program counter) whereby the simulator provides the time stamp. However, because a simulator is a hardware model, this method encounters problems associated with SA as described above.

- Software: Ipoints are inserted into the source code via language extensions, and are thus compiled into the executable. When an ipoint is hit during execution, it is time stamped on target; traces are stored internally in a memory buffer to be downloaded on test completion. The advantage of this approach is that porting to new architectures is relatively straightforward. However, the additional ipoint routine incurs a timing penalty and increases overall code size, which is commonly referred to as the **probe effect**.
- Software/Hardware: This is similar to software only instrumentation, except that execution of an ipoint writes its trace identifier to an I/O port of the target. The port is monitored by a logic analyser, which both timestamps ipoints off target as they are produced and stores timing traces. Penalties associated with the probe effect are thus minimised. However, the target must have available and accessible pins to emit the data, which is not always practical with more advanced processors.
- Hardware: On-chip debug interfaces, such as Nexus [1] or the Embedded Trace Macrocell [33], allow programs to be traced without interference. In these cases, the trace data are either written to an on-chip trace buffer for subsequent download, or exported directly in real time through an external port. In order to limit the size of traces, only the program flow discontinuities are monitored, i.e. conditional and unconditional jumps. However, bandwidth remains the major technical obstacle because the port or debugger must keep pace with the rate at which trace data are produced; otherwise, **blackouts** arise in which parts of a timing trace are overwritten and essentially lost.

There is clearly a trade-off in using any of the above trace generation methods. On the one hand, source-level instrumentation provides greater flexibility, but this is inhibited by the probe effect. On the other hand, less intrusive instrumentation requires more technical support.

Clarification 1. *In this thesis, we embrace a more abstract view of how traces are generated by considering an ipoint simply as a program point at which a (timestamped) observation occurs. This could be intrusively by calls to a tracing library, or completely transparently through a hardware debug interface. Consequently, we do not* *quantify the impact of the probe effect on WCET estimates. We thus assume, without loss of generality, that there exists a suitable mechanism to generate and download traces.*

Given the freedom to select the locations of ipoints, various **instrumentation profiles** have been proposed, normally with functional coverage metrics or profiling in mind. The main goal of [2] is to select the fewest number of basic blocks (edges) to instrument so that basic block (branch) coverage can be measured by observing the number of ipoints hit. Tikir and Hollingsworth [116] detailed an approach that dynamically inserts and remove ipoints in order to reduce the run-time overhead of code coverage, as opposed to instrumenting statically. Instrumentation is only inserted into a procedure when executed for the first time during program execution; it is subsequently removed when it does not provide any additional coverage information. A widely-adopted instrumentation profile has been proposed by Ball and Larus [8, 65] in which a minimum number of ipoints are inserted such that the entire traversed path through a program can be reconstructed from a trace. We term such instrumentation profiles **path reconstructible**. In subsequent chapters, we will observe that this property impacts the accuracy of WCET estimates computed on the IPG program model.

Despite these novel techniques, the instrumentation profiles in modern HMB frameworks largely remain arbitrary. The archetypical case is an end-user inserting ipoints in an *ad hoc* fashion at the source level. However, arbitrary in this sense refers to instrumentation that is not program specific. Typically, there could be an upper bound on the permissible number of ipoints due to trace generation problems, e.g. blackouts, or because of the probe effect. Indeed, when a program contains a large number of procedures, it might only be possible to implement boundary instrumentation in which only the beginning and end of each procedure is instrumented.

Evidently, the instrumentation profile adopted impacts the amount of coverage sought in the testing phase of the HMB framework. During testing, the focus is to trigger the WCET of ipoint transitions, otherwise the final calculation could be compromised. In essence, smaller program segments, e.g. comprising basic blocks, between ipoint transitions require less stringent coverage because these are known to have a small number of different execution times [26]. On the other hand, program segments incorporating nested loops require greater coverage to ensure that all paths are exercised and that the WCET is indeed captured. This is further complicated by the influence of the hardware architecture on the amount of coverage sought. Relatively simple architectures with shallow, in-order pipelines result in smaller variations of execution times between ipoints. In comparison, state-of-the-art processors with multi-level caches, dynamic branch prediction, and out-of-order execution, exhibit a greater variability in execution time between ipoints due to, for example, cache misses and branch mispredictions.

Furthermore, as we noted in Section 2.2.3, we often wish to obtain more than the WCETs of transitions from timing traces. For example, determining loop bounds allows our HMB framework to operate automatically without user interaction, but this places a larger burden on the test phase as bounds need to be accurate.

Clarification 2. *How to stress any non-functional property of a program is the subject of **WCET coverage** [17]. Consequently, we do not make any assumptions regarding the way in which testing is conducted nor the amount of coverage achieved. This is considered beyond the scope of this thesis and we thus assume a uniform testing strategy provided by, for example, functional testing techniques.*

For this reason, when discussing the percentage of pessimism in the WCET estimate (computed through the IPG program model), we assume that testing is good enough, and therefore, the raw timing data provided are sufficiently representative of the worst case. Observe that existing SA techniques make a similar assumption as any error — either in the WCETs of basic blocks due to incorrect processor modelling or because the loop bound provided by a user is an underestimate — invalidates the entire analysis.

Hybrid Measurement-Based Approaches

The idea of a HMB framework is not a novel contribution of this thesis as it was first proposed elsewhere [90, 89]. Their work identifies paths in the CFG that must be exercised during testing. However, because of the path complexity problem, the CFG is manually split into measurement blocks, with the finest granularity being the basic block. Before executing each measurement block, cache contents and other hardware units are flushed so that history effects of other measurement blocks are isolated. The flushing mechanism clearly translates into pessimism in the WCET estimate as measurements blocks will not operate in isolation during execution.

The closest related work to our approach is the HMB framework proposed in [26]. In particular, they use timing traces generated by the SimpleScalar toolsuite [7] to extract the WCETs of basic blocks, from which a WCET estimate is computed using the standard timing schema of an AST. The crucial difference between their work and ours is that we allow for arbitrary instrumentation and employ the IPG program model in the calculation phase.

However, the main focus of their work was to quantify the effects of modern speed-up features on the WCET by using various processor configurations that the SimpleScalar framework allows. Their experiments demonstrated some interesting aspects relating the disabling/enabling of hardware features to WCET estimates. In particular, they showed that caches have the biggest impact in reducing the actual WCET, i.e. without instruction or data caches, the WCET is very large. Furthermore, the level of overestimation normally outweighs the loss of performance caused by disabling of the advanced speed-up features. This latter point reinforces the main motivation of HMB analysis in that we should not impose predictability at the hardware level to force easier analysis as it is detrimental to the entire system.

More recent work [63, 124] has approached the HMB problem from a slightly different angle by focusing more on test vector generation for WCET estimation. In particular, they partition the CFG into program segments using instrumentation. This is conceptually similar to the approach of [90, 89] described above, except that the partitioning criterion depends on the number of acyclic paths through the segment. They force execution of each path in the program segment using test vectors generated by a model checker. Paths through the program segment must be acyclic to avoid the high computational complexity that is associated with model checking.

2.2.5 WCET Tools

The dominance of SA is best reflected by the number of academic tools that have emerged from this field. Following is a selection of the most prominent:

- *Cinderella from the University of Princeton.* This was originally developed to support the IPET, including modelling of cache constraints.
- *Heptane from the Université de Rennes.* This has the capacity to analyse four different architectures: Pentium I, H8/300, StrongARM, and MIPS. There is a cache analysis unit supporting the LRU replacement policy in which the size, the block size, and the associativity of the cache can be configured. It supports source- and assembly-level analysis through the AST and the IPET, respectively.
- *Chronos from the University of Singapore.* The user can configure the processor model (pipeline, cache, branch prediction) with the help of the SimpleScalar toolset [7]. It produces WCET estimates through the IPET.
- *SWEET from the University of Mälardalen.* This is integrated with an ANSI C compiler so that flow analysis can operate at the intermediate code level. It supports timing analysis on both the ARM9 and NECV850E processors, and can produce WCET estimates through either a path-based approach or the IPET.

Industry has increasingly become aware of the importance of WCET analysis. This has inspired two spin-off companies from affiliated universities:

- AbsInt [45] was founded from research at the University of Saarland and first produced a WCET analyser, aiT, based on SA. Their processor modelling techniques are founded on AI and they currently support the following architectures: ARM7, HCS12/STAR12, PPC 555/565/755, C16x/ST10, TMS320C3x, TriCore 1796, and i386. They can analyse programs containing recursive procedures and the calculations are based on the IPET.
- Rapita Systems [77] was founded from research at the University of York and their WCET tool, RapiTime, is based on HMB analysis. In particular, they use a probabilistic tree-based approach to compute the WCET, which combines ETPs that are collected during testing.

2.3 Summary

This chapter has contextualized WCET analysis and described the state-of-the-art techniques to produce WCET estimates that derive from static and MB analyses. A deluge of research has emerged to support processor modelling, yet certain operational assumptions are still required to ensure safe analysis, many of which are not practically feasible. Prime examples are assuming perfect LRU cache replacement or the absence of destructive aliasing in a branch history table. Moreover, and perhaps more crucially, no techniques satisfactorily model the operational interaction between processor speed-up features *simultaneously*. Indeed, timing anomalies vastly complicate this task. A further observation is that the models only consider the activities of the CPU and disregard the impact of peripheral devices.

Complexities modelling the processor has led some researchers to promote usage of more predictable hardware. However, industry continues to choose off-the-shelf processors in line with its requirements, e.g. cost, and there is no evidence to suggest that processor manufacturers will alter future designs with predictability in mind. Indeed, processors are much more likely to increase in complexity as transistor size decreases and multi-core CPUs become prevalent.

In industry, end-to-end measurements remain the dominant means by which the WCET is estimated, but this truism is yet to fully impact research. This means that WCET estimates are normally computed using testing techniques and coverage metrics that target functionality. This is not sufficient, however, as more complex hardware infiltrates the real-time sector.

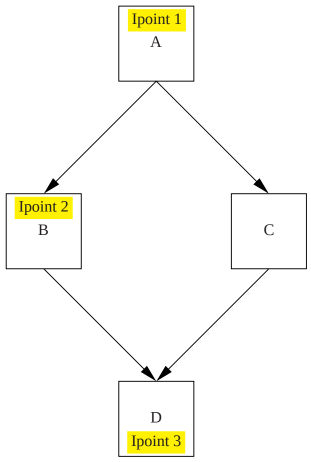
In response to this, HMB techniques are emerging, which includes some form of instrumentation. However, the biggest drawback so far is that they require very specific ipoint placement. This not only limits the type of instrumentation employed, i.e. software, but also prevents state-of-the-art instrumentation profiles from being considered. The remainder of this thesis is devoted to an analytical framework to compute WCET estimates given *arbitrary* instrumentation, under the assumption that there is a suitable test phase in place.

3 Instrumentation Point Graphs

Chapter 2 explored the program and processor models used in contemporary WCET techniques. In particular, we noted that the baseline program model is either the **Abstract Syntax Tree** (AST) or the **Control Flow Graph** (CFG). More important, however, is that the calculation engine operating on these data structures requires as input the WCET of basic blocks since these are the *atomic units*. For these purposes, static analysis constructs a processor model which attempts to bound the execution time of each basic block. Comparatively, current **Hybrid Measurement-Based** (HMB) approaches derive observed WCETs during testing after instrumenting the beginning of each basic block.

However, a severe shortcoming of these program models, from a HMB angle, is that they necessitate particular instrumentation profiles to avoid a pessimistic WCET estimate. This is best illustrated by considering a contrived example shown in Figure 3.1. The CFG in Figure 3.1(a) is a simple `if-then-else` construct with **instrumentation points** (ipoints) inserted in basic blocks A, B, and D; note that C is not instrumented and that the ipoint in D is misaligned with respect to those in A and B since it is not inserted at the beginning of the basic block. The table in Figure 3.1(b) exhibits the observed WCETs of these ipoint transitions. In order to combine these data in the calculation stage using the CFG (or indeed the AST), the WCETs of basic blocks must be derived from these measurements. Common practice assumes that this value is the maximum observed WCET amongst the ipoint transitions on which a basic block is executed; in this example, these WCETs are shown in Figure 3.1(c). However, the instrumentation employed causes overestimation of every basic block. The WCET of B is overestimated because transition 2 $\rightarrow$ 3 also executes D as a result of the misalignment of ipoints. Likewise, the WCETs of A, C, and D are overestimated because of both the missing ipoint in C and the misalignment of ipoints. A simple path-based approach on the CFG with these WCETs would deduce that $A \longrightarrow C \longrightarrow D$ is the worst path, resulting in a WCET estimate of $25+25+25=75$. Clearly, this a threefold overestimation since the actual WCET of the path $A \longrightarrow C \longrightarrow D$ is 25, which occurs when transition $1 \longrightarrow 3$ is followed.

Figure 3.1. Pessimism Intrinsic to the WCET Calculation Stage when using Sparse Instrumentation.



(a) A CFG with ipoints at the beginning of basic blocks A and B, and at the end of basic block D. However, basic block C is not instrumented.

Ipoint transition	Observed WCET
$1 \longrightarrow 2$	5
$2 \longrightarrow 3$	11
$1 \longrightarrow 3$	25

(b) Observed WCETs of ipoint transitions during testing.

Basic block	WCET
A	25
B	11
C	25
D	25

(c) Overestimated WCETs of basic blocks.

The general problem is that there is little leeway in the employed instrumentation profile for these program models: either each basic block is uniformly instrumented or an overestimation in their WCETs occurs (see Clarification 2 in Chapter 2). It is obvious that the overestimation becomes more problematic with coarser instrumentation profiles [2, 8, 65, 116], many of which are widely adopted in functional testing environments to limit the size of timing traces. This is undesirable as we want our HMB framework to integrate seamlessly with contemporary test harnesses so that it is applicable in an industrial setting.

In this chapter, we propose a novel program model — the **Instrumentation Point Graph** (IPG) — in which the atomic unit of computation is the transition among ipoints instead of basic blocks. This forcible modification essentially circumvents the pessimism associated with CFGs and ASTs when using *arbitrary* instrumentation.

The remainder of this chapter is organised as follows. Section 3.1 begins by introducing an intermediate form similar to the CFG — the CFG* — which is used to construct and analyse properties of the IPG. Following that, the section continues with a presentation of the IPG and its properties that are relevant in our HMB framework. Section 3.2 formulates the IPG construction problem as a data-flow problem [83], which leads to a simple *iterative algorithm* [29, 58, 59] that constructs the IPG independent of CFG* reducibility. However, merely constructing the IPG is generally not sufficient for it to be used as a program model (in WCET analysis) because of the problem of **irreducibility**. Section 3.3 demonstrates that irreducibility is especially prevalent in the IPG, that it often encompasses much larger subgraphs than canonical cases of CFG irreducibility, and consequently, that state-of-the-art techniques [49, 96, 97, 104] inevitably fail. We therefore detail a mechanism to identify all IPG loops — irrespective of irreducibility — using the **Loop-Nesting Tree** (LNT) of a reducible CFG*. This result forms the basis of a more complex algorithm, presented in Section 3.4, which constructs the IPG and identifies all IPG loops on the fly.

Following that, we consider usage of the IPG in the context of interprocedural analysis in Section 3.5. We describe a virtual inlining mechanism — **master ipoint inlining** — which provides visibility to procedure calls without virtually inlining the entire IPG of each callee. This results in one IPG per procedure and essentially black boxes procedure calls. We subsequently demonstrate how to parse timing traces using the set of IPGs and their properties. In particular, we show how to extract (observed) loop bounds and how to detect **contexts** dynamically so that the timing data retrieved applies to each context as opposed to each procedure, resulting in a more precise analysis. We then show how the call graph controls the calculation given the set of IPGs and the trace data. Finally, we conclude the chapter with a summary in Section 3.6.

3.1 The CFG* and the IPG

The program model in our HMB framework is the IPG, which basically arranges the transitions among ipoints into structural form. In order to construct the IPG, therefore, we require the locations of ipoints with respect to program structure at the intermediate code level. However, the standard graph-based structural model, the CFG, does not adequately model such information. Either ipoints are grouped together with other functional instructions in basic blocks (when software instrumentation is employed), or alternatively, they exist virtually on some part of the CFG (when a simulator or hardware debug interface is employed).

For this reason, our HMB framework replaces the CFG by a CFG* whose unique characteristic is that ipoints are decoupled into fully-fledged vertices; basic blocks thus only consist of functional instructions. These disjoint sets of vertices can be recognised by extending the set of *leaders* [3], which are first instructions inside a vertex. For the basic blocks of a CFG, each of the following is a leader:

- The first instruction of the procedure.
- Any instruction that is the target of a conditional or unconditional jump.
- Any instruction that immediately follows a conditional or unconditional jump.

In addition to these leaders, each of the following is also leader in the CFG*:

- Each ipoint instruction.
- Any instruction that succeeds an ipoint.

Analogously to a basic block, every vertex in the CFG* consists of its leader and all instructions up to, but not including, the next leader or the end of the program. Thus, CFG* generation partitions a program into a set of ipoints, denoted I, and a set of basic blocks, denoted B; these are linked together according to flow of control. Formally:

Definition 2. *Let* I *be the set of ipoints and* B *be the set of basic blocks in a procedure. A* ***CFG**** *is a flow graph* $C = \langle V_C = \mathsf{B} \cup \mathsf{I}, E_C, \mathsf{s}, \mathsf{t} \rangle$ *in which:*

- $\{\mathsf{s}, \mathsf{t}\} \subseteq \mathsf{I}$.
- $E_C = \{u \rightarrow v | u, v \in V_C \wedge \textit{there is possible flow of control from } u \textit{ to } v\}$.

For a CFG* C, a non-empty **ipoint-free path** p is a sequence $u \rightarrow b_1 \rightarrow b_2 \rightarrow \ldots \rightarrow b_n \rightarrow v$ such that $b_i \in \mathsf{B}$, $u, v \in \mathsf{I}$, and $n \geq 0$. For brevity of notation, we use $u \xrightarrow[\mathsf{B}]{+} v$ to denote an ipoint-free path of length one or more, and $u \xrightarrow[\mathsf{B}]{*} v$ to denote an ipoint-free path of length zero or more. An IPG is constructed from C by contracting the non-empty ipoint-free paths in C. More formally:

Definition 3. *Let $C = \langle V_C = \mathsf{B} \cup \mathsf{I}, E_C, \mathsf{s}, \mathsf{t} \rangle$ be a CFG*. The **IPG** of C is a flow graph $I = \langle \mathsf{I}, E_I, \mathsf{s}, \mathsf{t} \rangle$ in which:*

- $E_I = \{u \rightarrow v | u, v \in \mathsf{I} \wedge \exists\ u \xrightarrow[\mathsf{B}]{+} v \textit{ in } C\}$

Following are some important clarifications to make regarding the remainder of this chapter:

Clarification 3. *Each ipoint u conceptually belongs to both the CFG* and its IPG. When context does not disambiguate the graph to which u belongs, we shall use u_C to denote that $u \in V(C)$ and u_I to denote that $u \in V(I)$.*

Clarification 4. *We assume that, for each ipoint u_C, $|succ(u_C)| \leq 1$. In practice this is typically a valid assumption as neither software nor hardware ipoints decide the outcome of a conditional branch.*

Clarification 5. *We often need to partition a set of CFG* vertices into two disjoint subsets of basic blocks and ipoints. In particular, for a set S, we use the notation S_{I} to denote the set $\{u | u \in S \wedge u \in \mathsf{I}\}$ and S_{B} to denote the set $\{u | u \in S \wedge u \in \mathsf{B}\}$.*

3.1.1 Ghost Ipoints, Ghost Edges and Trace Edges

In Definition 2, the dummy vertices s, t of a CFG* are considered to be ipoints. The reason for this is that, without these being ipoints, the resultant IPG might not be weakly connected.

In general, ipoints inserted purely for analysis purposes — subsequent to program instrumentation, compilation, and testing — are termed **ghost ipoints** because they are never observed in a timing trace. Chapter 4 explores another usage of ghost ipoints whilst transforming the IPG into hierarchical form.

We say that an edge $u \to v \in E_I$ is a **ghost edge** if either u or v is a ghost ipoint, otherwise it is a **trace edge**. This distinction between edges is needed in trace parsing and the calculation engine, as described in Section 3.5.

3.1.2 Path Expressions

One essential difference between a CFG (or CFG*) and an IPG is that functional instructions reside on the edges of the IPG as opposed to its vertices. Code can appear in different execution contexts, depending on the instrumentation profile utilised, since more than one IPG edge can execute the same basic block. We term the section of code executed when a transition between ipoints occurs as its *path expression*, defined formally as follows:

Definition 4. *The* ***path expression*** *of an IPG edge $u \to v \in E_I$, denoted $P(u \to v)$, is the regular expression over E_C representing the set of all ipoint-free paths from u_C to v_C in C. We denote the set of basic blocks in $P(u \to v)$ as $\mathsf{B}(P(u \to v))$.*

When $|\sigma(P(u \to v))| = 1$, we say that $P(u \to v)$ is **reconstructible**, i.e. there is only one non-empty ipoint-free path from u_C to v_C. Furthermore, we say that an instrumentation profile is **path reconstructible** if, for all $u \to v \in E_I$, $P(u \to v)$ is reconstructible.

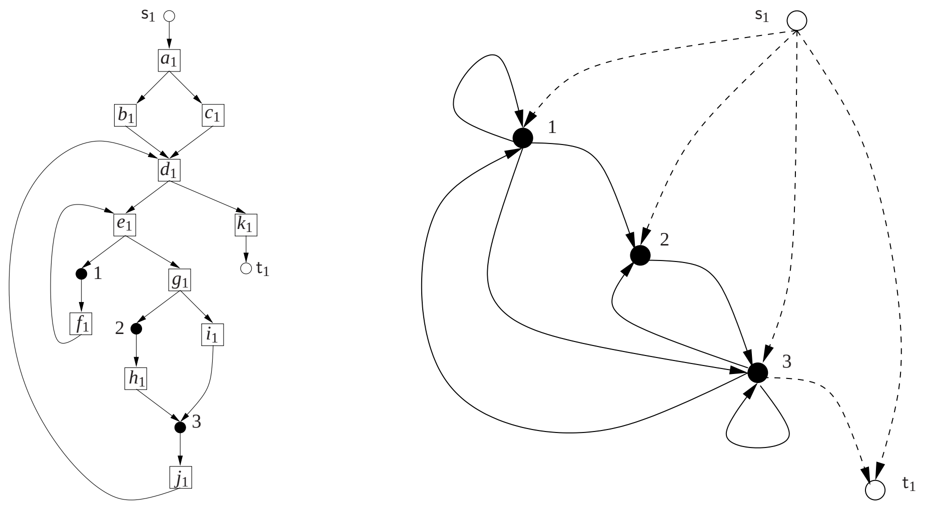
These properties are of interest for several reasons. First, a path reconstructible instrumentation profile allows the exact path through the CFG* to be regenerated from any sequence of ipoints[1]. For such profiles, every iteration of every loop must be observable in a trace, thus we can determine accurate observed loop bounds during trace parsing. Second, non-reconstructible path expressions highlight ipoint transitions for which testing should be stressed in the front end of a HMB framework, because such transitions traverse multiple CFG* paths. For example, if the path expression contains $\cup$ then the test harness could attempt to execute each acyclic path on that transition using model checking [124]. A third motivation is that the reconstructibility of path expressions determine whether extra flow information, e.g. relating infeasible paths,

[1]Hardware tracing mechanisms, such as Nexus [1], are implicitly path reconstructible because they monitor program flow discontinuities, i.e. jumps.

can be incorporated into the WCET calculation on the IPG. Normally, such information is obtained at the source level (through user annotations) and is mapped down to the basic block level [62]; in turn, this must be transferred onto the IPG.

An Example

Figure 3.2. Example of a CFG* and an IPG.



(a) The CFG.* *(b) Resultant IPG.*

IPG Edge	Path Expression
$s_1 \to 1$	$s_1 \to a_1 \cdot (a_1 \to b_1 \cdot b_1 \to d_1 \cup a_1 \to c_1 \cdot c_1 \to d_1) \cdot d_1 \to e_1 \cdot e_1 \to 1$
$s_1 \to 2$	$s_1 \to a_1 \cdot (a_1 \to b_1 \cdot b_1 \to d_1 \cup a_1 \to c_1 \cdot c_1 \to d_1) \cdot d_1 \to e_1 \cdot e_1 \to g_1 \cdot g_1 \to 2$
$s_1 \to 3$	$s_1 \to a_1 \cdot (a_1 \to b_1 \cdot b_1 \to d_1 \cup a_1 \to c_1 \cdot c_1 \to d_1) \cdot d_1 \to e_1 \cdot e_1 \to g_1 \cdot g_1 \to i_1 \cdot i_1 \to 3$
$s_1 \to t_1$	$s_1 \to a_1 \cdot (a_1 \to b_1 \cdot b_1 \to d_1 \cup a_1 \to c_1 \cdot c_1 \to d_1) \cdot d_1 \to k_1 \cdot k_1 \to t_1$
$1 \to 1$	$1 \to f_1 \cdot f_1 \to e_1 \cdot e_1 \to 1$
$1 \to 2$	$1 \to f_1 \cdot f_1 \to e_1 \cdot e_1 \to g_1 \cdot g_1 \to 2$
$1 \to 3$	$1 \to f_1 \cdot f_1 \to e_1 \cdot e_1 \to g_1 \cdot g_1 \to i_1 \cdot i_1 \to 2$
$2 \to 3$	$1 \to h_1 \cdot h_1 \to 2$
$3 \to 1$	$3 \to j_1 \cdot j_1 \to d_1 \cdot d_1 \to e_1 \cdot e_1 \to 1$
$3 \to 2$	$3 \to j_1 \cdot j_1 \to d_1 \cdot d_1 \to e_1 \cdot e_1 \to g_1 \cdot g_1 \to 2$
$3 \to 3$	$3 \to j_1 \cdot j_1 \to d_1 \cdot d_1 \to e_1 \cdot e_1 \to g_1 \cdot g_1 \to i_1 \cdot i_1 \to 3$
$3 \to t_1$	$3 \to j_1 \cdot j_1 \to d_1 \cdot d_1 \to k_1 \cdot k_1 \to t_1$

(c) Path expressions.

We illustrate the properties of the IPG through Figure 3.2, which depicts a CFG*, its IPG, and the path expressions of IPG edges. In Figure 3.2(a), circle vertices are ipoints, i.e. $\mathsf{I} = \{s_1, t_1, 1, 2, 3\}$, and square vertices are labelled basic blocks, i.e. $\mathsf{B} =$

$\{a_1, b_1, c_1, d_1, e_1, f_1, g_1, h_1, i_1, j_1, k_1\}$. We depict all ghost ipoints as unshaded circle vertices and the ghost edges of Figure 3.2(b) are depicted as dashed edges.

The path expression of each IPG edge is shown in Figure 3.2(c). Note that there is context-sensitive execution of all basic blocks except h_1, since h_1 only appears in the path expression of the edge $2 \rightarrow 3$. In addition, the path expressions of the edges $\mathsf{s}_1 \rightarrow 1$, $\mathsf{s}_1 \rightarrow 2$, $\mathsf{s}_1 \rightarrow 3$, $\mathsf{s}_1 \rightarrow \mathsf{t}_1$ are not reconstructible — thus the instrumentation profile is not path reconstructible — because there are multiple ipoint-free paths from a_1 to d_1 (indicated by the $\cup$ operator). However, all other path expressions are reconstructible.

3.2 IPG Construction

The previous section gave a formal description of the IPG and its properties. Here we describe how to construct the IPG from a CFG* using data-flow analysis.

Clarification 6. *This thesis does not consider how to construct the path expressions of IPG edges, principally because they have no practical bearing on the techniques developed. From the calculation perspective, path expressions are needed when path information relating basic blocks, e.g. infeasible paths, is to be transferred onto the IPG. However, such information is optional, i.e. it tightens the WCET estimate but is not essential. Note that this does not prevent us from mapping loop bounds obtained through static analysis onto the IPG because, as Section 3.3 describes, this only requires particular structural properties of the IPG.*

We begin by defining underlying properties of *Data-Flow Frameworks* (DFF), before elaborating upon the DFF that solves the IPG construction problem.

3.2.1 Data-Flow Analysis

In general terms, data-flow analysis gathers facts about how data are manipulated in programs. Such analyses are employed in compilers to assist in optimisation, the canonical example of which is the reaching definitions computation (see [3, 83]). Data-flow information is normally collected at each basic block by setting up and solving a system of data-flow equations, which model the effect of the basic block on the information across all executions.

Due to the similarities between many data-flow problems, they can often be treated in a unified way through a DFF. Central to DFFs is an algebraic structure called a semi-lattice:

Definition 5. *A **semi-lattice** is a set L with a binary meet operation $\sqcap$, and distinguished elements $\perp$ and $\top$ called bottom and top, respectively, such that:*

Commutativity *For all $x, y \in L$, $x \sqcap y = y \sqcap x$.*

Associativity *For all $x, y, z \in L$, $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$.*

Idempotency *For all $x \in L$, $x \sqcap x = x$.*

Bounded *For all $x \in L$, $x \sqcap \perp = x$ and $x \sqcap \top = \top$.*

Note that a semi-lattice can alternatively be defined as a partially-ordered set $\langle L, \preceq \rangle$. The connection between these two alternative definitions is that, for all $x, y \in L$, $x \preceq y$ if and only if $x \sqcap y = y$.

We now formally define a data-flow framework:

Definition 6. *A **data-flow framework** is a 4-tuple $\langle G, L, F, M \rangle$ such that:*

- *$G = \langle V_G, E_G, \mathsf{s}, \mathsf{t} \rangle$ is a flow graph.*
- *L is a semi-lattice.*
- *$F \subseteq \{f : L \to L\}$ is a set* transfer functions *such that:*
  - *F contains an identity function $I : L \to L$ such that $I(x) = x$ for all $x \in L$.*
  - *F is closed under composition. That is, $f \circ g \in F$, for all $f, g \in F$.*
  - *For all $f \in F$, $f : L \to L$ is* monotone. *That is, for all $x, y \in L$, $x \preceq y$ implies $f(x) \preceq f(y)$.*
  - *For all $f \in F$, $f : L \to L$* distributes *over $\sqcap$. That is, $f(x \sqcap u) = f(x) \sqcap f(y)$ for all $x, y \in L$.*
- *$M : E_G \to F$ is a map from flow graph edges to transfer functions.*

The semi-lattice essentially abstracts the effect of a vertex on the data-flow information, where the meet operation $\sqcap$ determines how the information is combined when it reaches a vertex. The monotonicity property of the set of functions ensures that the data-flow framework will halt as the information modelled in the lattice can only increase.

The function M can be extended to map every path in the flow graph G to a transfer function f. If $p = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n$ is a path in G, with $e_i = v_{i-1} \rightarrow v_i$, then the path transfer function $M(p)$ is defined to be $M(e_n) \circ M(e_{n_1}) \circ \ldots \circ M(e_1)$. The path transfer function of the empty path is the identity function I. The **Meet-Over-all-Paths** (MOP) solution [59] to the data-flow analysis problem is defined as follows:

$$\forall v \in V_G : MOP(v) = \sqcap_{p \in paths(\mathsf{s},v)} M(p) \tag{3.1}$$

Intuitively, the MOP solution produces the information at each vertex v that would result by applying the composition of each transfer function along all paths from s to v. However, the DFF must be distributive in order to compute the meet-over-all-paths solution. Otherwise, the DFF only computes the *maximum fixed point* solution, which is a safe approximation.

3.2.2 A Simple Data-Flow Framework to Build the IPG

We define a simple DFF to construct the IPG I from its CFG* C based on the observation that this problem is similar in nature to other classical data-flow problems, i.e. we want to know which ipoints can reach a particular program point.

Following are the elements of this DFF: the flow graph is the CFG*; the semi-lattice L_{I} is the powerset 2^{I} over the set of ipoints I; $F_{\mathsf{I}} \subseteq 2^{\mathsf{I}} \rightarrow 2^{\mathsf{I}}$ is the set of monotone, distributive transfer functions; and the meet operation is set union. Note that the bottom element of L_{I} is the empty set, and the top element of L_{I} is the set of all ipoints.

The data-flow problem that we want our DFF to solve encapsulates the intuitive idea that the ipoints reachable to each vertex $v \in V_C$ are the ipoint predecessors of v unioned with the ipoints that can reach the basic block predecessors of v. More formally, let us define the set:

$$\forall v \in V_C : \ ipoints(v) = \{u | u \in \mathsf{I} \wedge \exists u \xrightarrow[\mathsf{B}]{+} v \text{ in } C\}$$

which can be solved through the following data-flow equation:

$$\forall v \in V_C : ipoints(v) = \left(\bigcup_{p \in pred_{\mathsf{B}}(v)} ipoints(p) \right) \cup pred_{\mathsf{I}}(v) \tag{3.2}$$

3.2.3 The Iterative Algorithm

We can solve this DFF using the round-robin, *iterative* algorithm [29, 58, 59], which was designed specifically to handle common data-flow problems in a unified way. All that is required is to substitute an appropriate set of data-flow equations into its generic structure. The iterative algorithm to construct the IPG I from its sole parameter, the CFG* C, is shown in Figure 3.3.

```
   Input: $C$
   Output: $I$
1  foreach $v \in V_C$ do
2      $ipoints(v) := \emptyset$
3  $changed$ := true
4  while $changed$ do
5      $changed$ := false
6      foreach $v \in v_C$ in reverse post order do
7          $oldipoints(v) := ipoints(v)$
8          $ipoints(v) \cup= \left(\bigcup_{p \in pred_{\mathsf{B}}(v)} ipoints(p)\right) \cup pred_{\mathsf{I}}(v)$
9          if $oldipoints(v) \neq ipoints(v)$ then
10             $changed$ := true
```

Figure 3.3. Iterative Algorithm to Construct the IPG from a CFG*.

All iterative algorithms initially assign a conservative value to the information being computed; in this case, we assume no ipoints can reach a vertex $v \in V_C$ on a non-empty ipoint-free path (lines 1-2). The iterative part continues until a fixed-point solution has been found, which in this case is the meet-over-all-paths solution as the transfer functions in the semi-lattice are both monotone and distributive. Iteration is controlled by a boolean variable, *changed*, which is initially set to **true** (line 3). Each pass initially assumes that no changes will occur (lines 5). We then traverse each vertex v in reverse post order to update the values of $ipoints(v)$. Changes to the sets are discovered by recording the values computed in the previous iteration (line 7), updating the sets according to Equation (3.2) (line 8)[2], and then comparing the values (line 9). A change to any set forces another iteration (lines 9-10). On termination, therefore, the predecessors of each ipoint v in the IPG are stored in $ipoints(v)$, i.e. $pred(v_I) = ipoints(v)$.

One benefit of the iterative algorithm is that it is not restricted to reducible flow graphs, thus the IPG can always be built independent of CFG* reducibility. The disadvantage is that it requires quadratic time in the worst case. However, studies have shown that the iterative algorithm is very efficient in practice, requiring no more than $d(G)+3$ passes if we use the reverse post order of vertices [58]. Here, $d(G)$ is the *loop-connectedness* of G, which is the largest number of **Depth-First Search** (DFS) back edges found in any cycle-free path in G.

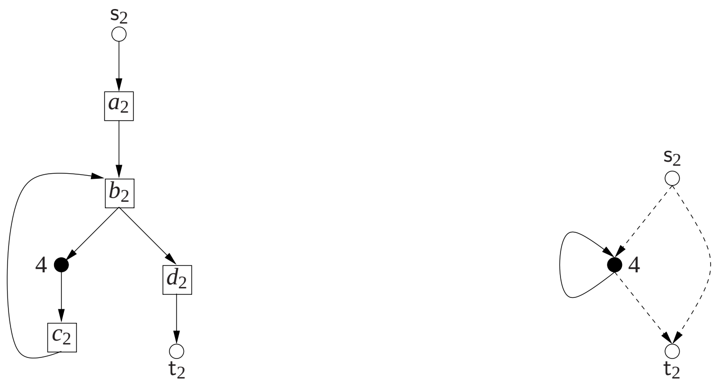
An Example

To illustrate the operation of the iterative algorithm, consider Figure 3.4, which depicts a CFG*, its IPG, and the iterative computations. Each row of the table shows how the values of $ipoints(v)$ change with successive iterations; coloured cells are sets that have changed from the previous iteration. The first column of the table lists the vertices in reverse post-order, which we have chosen arbitrarily, whereas all other columns display updates to the data-flow information. Before the first iteration, the set of ipoints reachable to each vertex is set to $\emptyset$. As vertex s_2 is the entry vertex, i.e. it has no predecessors, $ipoints(\mathsf{s}_2)$ remains empty on each iteration.

After the first iteration, only a subset of the correct edges in the IPG have been computed, i.e. only $\mathsf{s}_2 \rightarrow \mathsf{t}_2$ and $\mathsf{s}_2 \rightarrow 4$. After the second iteration, the remaining edges of the IPG, $4 \rightarrow 4$ and $4 \rightarrow \mathsf{t}_2$, are inserted. After the third iteration, none of the sets change, and thus the algorithm halts.

[2]Note that, for sets S and T, we use $S \cup_= T$ as a short form of $S := S \cup T$.

Figure 3.4. Example used to Demonstrate Construction of IPG using Algorithm in Figure 3.3.



(a) The CFG.* *(b) Resultant IPG.*

	Before #1	After #1	After #2	After #3
s_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
a_2	$\emptyset$	$\{s_2\}$	$\{s_2\}$	$\{s_2\}$
b_2	$\emptyset$	$\{s_2\}$	$\{s_2, 4\}$	$\{s_2, 4\}$
4	$\emptyset$	$\{s_2\}$	$\{s_2, 4\}$	$\{s_2, 4\}$
c_2	$\emptyset$	$\{4\}$	$\{4\}$	$\{4\}$
d_2	$\emptyset$	$\{s_2\}$	$\{s_2, 4\}$	$\{s_2, 4\}$
t_2	$\emptyset$	$\{s_2\}$	$\{s_2, 4\}$	$\{s_2, 4\}$

(c) Iterative computations.

3.3 Reducibility and Loop-Nesting Trees

The weakness of the algorithm described in the previous section is that it only builds the structure of the IPG without identifying its *structural properties*. As each calculation technique proposed in WCET analysis presumes knowledge of the loops in the program model — including their nesting relationship — a further analysis step is required to identify the IPG loops. Straightforward loop detection, however, is restricted to the class of *reducible* flow graphs.

Definition 7. *A flow graph G is* ***reducible*** *if its edges can be partitioned into two disjoint groups, called the* ***forward edges*** *and* ***loop-back edges****, respectively, with the following two properties [3]:*

1. *The* ***forward edges*** E_f *form an acyclic graph in which every vertex can be reached from the entry vertex.*

2. *The* ***loop-back edges*** E_b *consist only of edges* $u \rightarrow h$ *such that* $h \trianglerighteq u$.

Each loop-back edge $u \rightarrow h$ identifies a *reducible* loop in G, which is an induced subgraph, denoted L_h, of G whose vertices can reach h without passing through h [3]. The destination h of a loop-back edge is termed a **header** and satisfies the following two properties: h pre-dominates all vertices in L_h; a subset of $pred(h)$ do not belong to the loop, i.e. h is the unique entry vertex of the loop. The source of a loop-back edge is termed a **tail**. As h can be the destination of multiple loop-back edges $u_1 \rightarrow h, u_2 \rightarrow h, \ldots, u_n \rightarrow h$, the loops of each $u_i \rightarrow h$ are unioned together to create a single loop. When the loop contains a single vertex it is termed a **self-loop**, otherwise it is termed a **non-trivial** loop. A vertex u is termed a **loop exit** if u has a successor that is not in L_h. Let $h_1, h_2, \ldots, h_n$ be the headers of the loops in which a vertex u is contained; we say that h_i is the **immediate header** of u, denoted $h_i = header(u)$, if $h_i \triangleright u$ and there is no $h_j \neq h_i$ satisfying $h_i \triangleright h_j$. Loops are said to be **nested** in each other: for distinct loops L_x and L_y, either L_x is nested in L_y ($L_x \subseteq L_y$), L_y is nested in L_x ($L_y \subseteq L_x$), or L_x and L_y have no nesting relation ($L_x \nsubseteq L_y$ and $L_y \nsubseteq L_x$).

The containment relationship between loops in a reducible flow graph can be captured in a Loop-Nesting Tree (LNT):

Definition 8. *For a reducible flow graph* $G = \langle V_G, E_G \cup \{\mathsf{t} \rightarrow \mathsf{s}\}, \mathsf{s}, \mathsf{t}\rangle$, *its* ***Loop-Nesting Tree*** $T_L^G = \langle V_G, E_{T_L^G}, \mathsf{s}, H\rangle$ *is a tree with the following properties*[3]*:*

- $H \subset V_G$ *is the set of headers identified in* G.
- $E_{T_L^G} = \{(header(v), v) | v \in V_G - \{\mathsf{s}\}\}$.
- *For each* $h \in H$, *the vertices of the loop are the descendants of* h. *Therefore, every internal vertex is the header of a non-trivial loop.*

The algorithm to construct the LNT of a reducible flow graph was originally proposed by Tarjan [110]. It uses a DFS to identify DFS back edges: in a reducible flow graph loop-back edges and DFS back edges are equivalent because the pre-dominators of a vertex are always its ancestors in *any* DFS spanning tree [66].

[3]We explicitly add the edge $\mathsf{t} \rightarrow \mathsf{s}$ to the flow graph G to ensure that the flow graph becomes a maximal **Strongly Connected Component** (SCC). Therefore, all vertices in G are enclosed in a loop and the nesting relationship between loops can be captured in a tree as opposed to a forest.

However, when a flow graph G is irreducible[4], the pre-dominance relation is unable to identify all loop-back edges; that is, removing all edges $u \rightarrow h$ satisfying $h \trianglerighteq u$ does not ensure that G is acyclic, and consequently Tarjan's algorithm halts. For such irreducible flow graphs, there is no coherent view of what constitutes a loop [97] and, as a result, state-of-the-art techniques [49, 104] could produce different LNTs for the same program.

Havlak [49] proposed a refinement of Tarjan's original algorithm that continues to identify loops even when it finds a DFS back edge $u \rightarrow v$ satisfying $v \ntrianglerighteq u$, thus a loop is constructed for every DFS back edge, as opposed to every loop-back edge. (Ramalingam [96] has produced a modification of Havlak's algorithm so that it runs in almost linear time.) Note that the Havlak LNT chooses a unique vertex as the header of each irreducible loop. More importantly, because the algorithm piggybacks on a DFS, the set of loops computed depends on the order of the DFS, i.e. the same algorithm could produce a different set of loops due to a different DFS.

Another common technique used to identify irreducible loops was presented by Sreedhar [104] using the *DJ-Graph*. This is a data structure that basically unifies the flow graph and its pre-dominator tree together. Their algorithm first identifies reducible loops at each level in the underlying pre-dominator tree of the DJ-Graph. It then collapses any non-trivial SCC at the same level (not yet identified as a reducible loop) into an irreducible loop. Thus, in contrast to the Havlak LNT, a set of vertices is chosen as the header of irreducible loops, each of which is an entry to the SCC.

3.3.1 Identifying IPG Loops

The principal reason that loop identification is of concern in WCET analysis is that we must bound all loops in the program model. When a graph-based program model is in place, irreducibility implies that the computed WCET estimate is sensitive to the chosen LNT. To be conservative — and because the properties of the program model cannot be formally proven correct — the largest WCET estimate is selected. Accuracy could be compromised, however, as an alternative loop detection mechanism could be

[4]In practice, CFGs become irreducible as a result of `goto` statements and because of modifications to the program structure enforced by (optimising) compilers.

developed that produces a different WCET estimate.

Irreducibility is especially prevalent in the IPG and often encompasses much larger subgraphs than canonical examples of CFG irreducibility. To illustrate this point, reconsider the relatively simple IPG in Figure 3.2. In this example, the pre-dominance relation is able to detect self-loops $1 \rightarrow 1$ and $3 \rightarrow 3$ because $1 \trianglerighteq 1$ and $3 \trianglerighteq 3$. However, it cannot compute the loops in the SCC with vertex set $\{1,2,3\}$ because: $1 \not\trianglerighteq \{2,3\}$, $2 \not\trianglerighteq \{1,3\}$, and $3 \not\trianglerighteq \{1,2\}$. Therefore, unless we can provide a mechanism to correctly identify IPG loops, WCET calculations on the IPG could be inaccurate.

Clarification 7. *We term the cycle-inducing edges of the IPG as* ***iteration edges*** *as opposed to loop-back edges. As noted in Definition 7, loop-back edges have a very precise meaning in the literature, but as we shall observe, the pre-dominance relation between vertices of iteration edges rarely holds. Furthermore, this distinction clarifies the graph to which we refer when discussing cycle-inducing edges: "loop-back edges" only belong to the CFG* and "iteration edges" only belong to the IPG.*

The crux of the problem is that, because ipoint placement is not restricted to particular locations, the pre-dominator tree of the IPG can be very shallow. Valuable structural information is hence lost whilst constructing the IPG from the CFG*. This information can be retrieved, however. From Definition 3, it is obvious that every path in the IPG can be retraced through the CFG*. Thus, for every loop in the IPG, there must be a corresponding loop in the CFG* (although the converse might not be true). The following lemma captures this intuition by establishing the properties of the CFG* under which a cycle is induced in its IPG.

Lemma 1. *Let C be a CFG* and I its IPG. Then, I contains a cycle if, and only if, C is cyclic and there is at least one non-trivial SCC in C that contains an ipoint.*

Proof. $\Rightarrow$ If C is acyclic, then clearly, from Definition 3, contraction of non-empty ipoint-free paths in C cannot create cyclic paths in I. Therefore, C must contain cycles for I to be cyclic. Let $S_1, S_2, \ldots S_n$ denote the set of SCCs in C, and assume none of the non-trivial SCCs contain an ipoint (note that each trivial SCC is either a basic block or an ipoint). Consider the *component graph* C' of C. Since each non-trivial SCC does not contain an ipoint, the only transitions in I are constructed from the paths between trivial SCCs in C'. Because C' is a **Directed Acyclic Graph** (DAG) (see Lemma 22.13 in [30]), then the IPG I' created from C' must also be a DAG. Therefore, at least one SCC must contain an ipoint if I is cyclic.

$\Leftarrow$ Assume that there is only one SCC S_i in C containing a unique ipoint u. By the definition of a SCC, there must be a path $p : u_C \to b_1 \to b_2 \ldots \to b_n \to u_C$, $n > 0$, in C. From Definition 3, the IPG contracts p into the edge $u_I \to u_I$, which creates a cycle in I.

$\square$

This lemma decides upon the existence of cycles in the IPG I, but it does not indicate *which edges* are iteration edges in I. On the other hand, we may infer the following key observation from this result. Suppose initially that I was created from an acyclic CFG* C; then I is also acyclic. Now assume that the edges of C are updated to induce non-trivial SCCs $S_1, S_2, \ldots, S_n$ in C and that the edges of I are updated accordingly. Let E'_{scc} denote the set of edges added to I in this step. If no SCC contains an ipoint then I remains acyclic, i.e. $E'_{scc} = \emptyset$. However, if at least one S_i contains an ipoint then I would follow the same acyclic-to-cyclic transition as C. Specifically, because each edge $u \to v \in E'_{scc}$ causes a directed cycle in I, $u \to v$ is an iteration edge of I. This is precisely the information that the pre-dominance relation provides in standard reducibility, except in the opposite direction, i.e. it decides which edges should be removed for the flow graph to be acyclic.

This suggests that — provided C can be decomposed into a succession of acyclic regions — we can generalise this observation towards cyclic CFG*s. Such a decomposition mechanism is provided by the LNT T^C_L of C [97]. In particular, T^C_L enables the **loop DAG** $L'_h = \langle V_h, E_h \rangle$ of each loop L_h to be induced as follows. V_h consists of h and every vertex that is a child of h in T^C_L. Furthermore, if $h' \in V_h$ is a loop header such that $h' \neq h$ then h' is a termed an **abstract vertex** as it represents all vertices u of an inner loop $L_{h'}$. The edges E_h consist only of forward edges in L_h and are added as follows. For each abstract vertex h', any exit edge from $L_{h'}$ into L_h has h' as its source, and any entry edge into $L_{h'}$ from L_h has h' as its destination. Any edge between non-abstract vertices in L_h is added as normal.

The following theorem is the main result on how to identify iteration edges in I.

Theorem 1. *Let C be a reducible CFG*, I be its IPG, L_h be a non-trivial loop in C with a set of tails T, and $L'_h = \langle V_h, E_h \rangle$ be the loop DAG of L_h. Then, the edge $u \rightarrow v \in E_I$ is an iteration edge if there are paths $p : h \xrightarrow[\mathsf{B}]{*} v_C$ and $q : u_C \xrightarrow[\mathsf{B}]{*} t$, $t \in T$, in L_h and there is no path $r : u_C \xrightarrow[\mathsf{B}]{+} v_C$ in L'_h.*

Proof. Let $E_I^h \subset E_I$ be the set of edges added to I from L'_h. From lemma 1, every edge $u \rightarrow v \in E_I^h$ does not create a cycle in I. If path r exists then $u \rightarrow v \in E_I^h$, thus r cannot exist. Furthermore, there can be no path $r' : u_C \xrightarrow[\mathsf{B}]{+} v_C$ in the loop DAG of an outer loop $L_{h'}$ (i.e. h' is a proper ancestor of h in the LNT T_L^C of C) since u_C and v_C will both be represented by an abstract vertex.

If path p does not exist then $h \neq v_C$ and, because h cannot reach v_C on a non-empty ipoint free path, there must instead be non-empty ipoint-free paths from ipoints $w_C^1, w_C^2, \ldots, w_C^n$ to v_C, i.e. $w^i \rightarrow v \in E_I^h$. Similar reasoning can be used with respect to path q.

It follows that we can concatenate paths p and q together joined by the loop-back edge $t \rightarrow h$ to create the path $s : u_C \xrightarrow{*} t \rightarrow h \xrightarrow{*} v_C$. Clearly, s induces a cycle in C, and from lemma 1, $u \rightarrow v$ must create a cycle in I, thus proving the claim. $\square$

There is therefore a mapping from an *instrumented* CFG* loop L_h (i.e. h has ipoint descendants in T_L^C) to an IPG loop. We define a bijective function $\Omega : \mathscr{L} \rightarrow \mathscr{L}^I$ where $\mathscr{L}$ is the set of instrumented CFG* loops and $\mathscr{L}^I$ is the set of IPG loops.

Clarification 8. *In the remainder of the thesis, we use the notation L_h^I to refer to an IPG loop. Strictly speaking this is an abuse of notation as the header vertex h might not actually belong to the IPG loop (i.e. it could be a basic block). However, this notation succintly reflects the structural connection between CFG* and IPG loops.*

For an IPG loop L_h^I, we use the notation $IE(L_h^I)$ to denote its **iteration edge set**. Observe that an iteration edge $u \rightarrow v$ can belong to multiple iteration edge sets since $u \rightarrow v$ can “iterate through” more than a single CFG* loop, the conditions under which are established in the following corollary:

Corollary 1. *Let L_x and L_y be CFG* loops satisfying $L_x \subseteq L_y$ with respective tails t_x and t_y. Then, $IE(L_x^I) \subseteq IE(L_y^I)$ provided there are paths $p : y \xrightarrow[\mathsf{B}]{+} x$ and $q : t_x \xrightarrow[\mathsf{B}]{+} t_y$ in C and $y, t_y \notin \mathsf{I}$.*

Proof. Immediate from Theorem 1. □

Therefore, we may consider $u \rightarrow v$ to be a **multi-edge** in which the multiplicity of $u \rightarrow v$ is equal to the number of iteration edge sets to which it belongs; in essence, the iteration space of $u \rightarrow v$ is partitioned across the CFG* loops through which it iterates. Provided the instrumentation profile is path reconstructible, all iteration edge sets must be pairwise disjoint because, by definition, every iteration of every CFG* loop must be distinguishable in a trace. As we shall observe in Section 3.5.2, this property determines whether bounds gathered through trace parsing are accurate or not.

An Example

We illustrate the IPG loop detection mechanism by returning to the problematic IPG of Figure 3.2. In the CFG* of this figure, there are two loop-back edges, $f_1 \rightarrow e_1$ and $j_1 \rightarrow d_1$, identifying respective loops L_{e_1} and L_{d_1}. The LNT of the CFG* is shown in Figure 3.5. All headers of non-trivial loops are internal vertices in the LNT: $\mathtt{s}_1$ is the (root) internal vertex due to the dummy edge $\mathtt{t}_1 \rightarrow \mathtt{s}_1$.

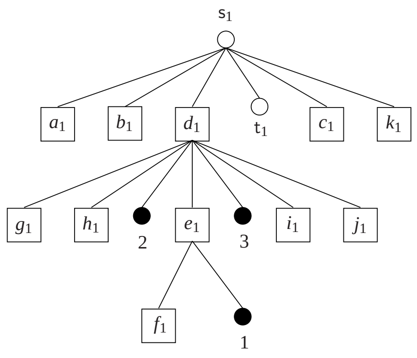


Figure 3.5. The LNT of the CFG* from Figure 3.2.

Using Theorem 1, we can identify IPG loops as follows:

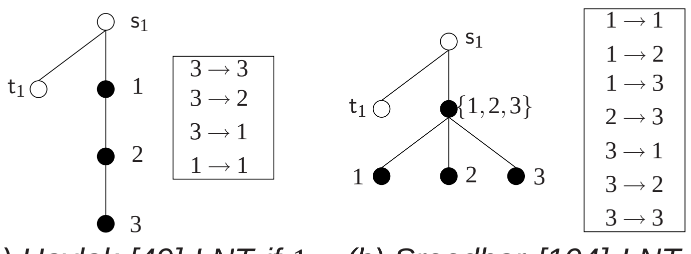
- In the inner loop L_{e_1}, there is a path $e_1 \xrightarrow[\mathsf{B}]{+} 1 \xrightarrow[\mathsf{B}]{+} f_1$, hence $L^I_{e_1} = \langle\{1\},\{1 \rightarrow 1\}\rangle$ is an IPG loop and $IE(L^I_{e_1}) = \{1 \rightarrow 1\}$.
- In the outer loop L_{d_1}, there are paths $p_1 : d_1 \xrightarrow[\mathsf{B}]{+} 1$, $p_2 : d_1 \xrightarrow[\mathsf{B}]{+} 2$, $p_3 : d_1 \xrightarrow[\mathsf{B}]{+} 3$ and $q : 3 \xrightarrow[\mathsf{B}]{+} f_1$. We can concatenate each p_i with q. Therefore, $L^I_{d_1} = \langle\{1,2,3\},\{1 \rightarrow 1, 1 \rightarrow 2, 1 \rightarrow 3, 2 \rightarrow 3, 3 \rightarrow 1, 3 \rightarrow 2, 3 \rightarrow 3\}\rangle$ is an IPG loop and $IE(L^I_{d_1}) = \{3 \rightarrow 1, 3 \rightarrow 2, 3 \rightarrow 3\}$.

Let us compare these sets of IPG loops with those identified by alternative techniques. The Havlak [49] LNT for this IPG is depicted in Figure 3.6(a). As we noted above, the Havlak LNT is sensitive to the order of the initial DFS, thus we have arbitrarily chosen the following pre-ordering of vertices: $\mathsf{s}_1, 1, 2, 3, \mathsf{t}_1$. The Havlak LNT identifies three IPG loops:

1. $\langle\{3\},\{3 \rightarrow 3\}\rangle$, where $3 \rightarrow 3$ is an iteration edge.
2. $\langle\{2,3\},\{3 \rightarrow 3, 2 \rightarrow 3, 3 \rightarrow 2\}\rangle$, where $3 \rightarrow 2$ is an iteration edge.
3. $\langle\{1,2,3\},\{1 \rightarrow 1, 1 \rightarrow 2, 1 \rightarrow 3, 2 \rightarrow 3, 3 \rightarrow 3, 3 \rightarrow 2, 3 \rightarrow 1\}\rangle$, where $3 \rightarrow 1$ and $1 \rightarrow 1$ are iteration edges.

Observe, first of all, that this Havlak LNT computes the exact set of iteration edges. However, it does not manage to construct the correct number of loops, and consequently, the correct nesting relationship.

Figure 3.6. Example LNTs Generated for IPG in Figure 3.2(b), and the Set of Iteration Edges each Identifies.



(a) Havlak [49] LNT if 1 *is visited first during DFS.*

(b) Sreedhar [104] LNT generated by DJ-Graph.

The DJ-Graph [104] LNT produced for this IPG is depicted in Figure 3.6(b). Each of the vertices in the set $\{1,2,3\}$ is chosen as a header of an irreducible loop since each is an entry into the SCC. The DJ-Graph LNT identifies three IPG loops:

1. $\langle\{1\},\{1 \to 1\}\rangle$, where $1 \to 1$ is an iteration edge.
2. $\langle\{3\},\{3 \to 3\}\rangle$, where $3 \to 3$ is an iteration edge.
3. $\langle\{1,2,3\},\{1 \to 1, 1 \to 2, 1 \to 3, 2 \to 3, 3 \to 3, 3 \to 2, 3 \to 1\}\rangle$, where $1 \to 2$, $1 \to 3$, $2 \to 3$, $3 \to 2$, and $3 \to 1$ are iteration edges.

This LNT correctly identifies one loop, i.e. $\langle\{1\},\{1 \to 1\}\rangle$, but it neither correctly identifies the set of iteration edges in the problematic SCC, nor does it compute the correct set of loops.

Thus, from this simple example, we may conclude the more general point that contemporary loop detection techniques do not adequately support IPG loop identification.

3.3.2 Bounding IPG Loops through Static Analysis

The mapping between CFG* loops and IPG loops provides the mechanism by which loop bounds supplied through static analysis techniques [47, 52] (or via an end-user) can be transferred onto the IPG. Equally, bounds obtained through trace parsing (by means of the properties of IPG loops) can be relayed back to the user at the source level, provided there is a clear mapping between the object and source code level.

Here we consider how to bound IPG loops assuming static analysis has provided a **relative bound** for each instrumented non-trivial CFG* loop L_h. We consider the relative bound to be the maximum number of times a vertex $v \in V(L_h)$ can execute (with respect to its outer nesting level) *minus* any execution of v on the loop exit path[5].

Let $b_{rel}(h)$ denote the relative bound of L_h. We want to provide a relative bound on each $u \to v \in E(L_h^I)$, denoted $b_{rel}(u \to v)$, according to $b_{rel}(h)$. There are two cases to consider:

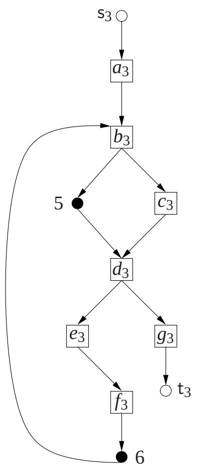
Forward edge: On every iteration of L_h, $u \to v$ can execute; therefore, $b_{rel}(u \to v) = b_{rel}(h)$.

[5]Considering bounds in this way avoids the discrepancies of general loop structures such as `for` loops or those containing `break` statements. In these cases, a subset of the vertices in the loop typically execute once more than others as they determine whether to exit the loop or not.

Iteration edge: The bound on $u \longrightarrow v$ depends on the location of u, v within L_h. Observe that, on the first execution of L_h, $u \longrightarrow v$ is *not* triggered since an acyclic path through L_h must be traversed before reaching u, i.e. $u \longrightarrow v$ essentially signifies a looping back into L_h. However, because we do not consider the exit path out of L_h as contributing to the bound on L_h, if v can reach an exit of L_h in the loop DAG L'_h, then $u \longrightarrow v$ can execute a further time on one of the exit paths out of L_h. Therefore:

$$b_{rel}(u \longrightarrow v) = \begin{cases} b_{rel}(h) & \text{if } v \text{ can reach exit of } L_h \text{ in the loop DAG } L'_h \\ b_{rel}(h) - 1 & \text{otherwise} \end{cases} \tag{3.3}$$

Figure 3.7. Demonstrating Different Relative Bounds on Iteration Edges depending on Locations of Ipoints.



(a) The CFG with a loop L_{b_3} whose sole exit is d_3.*

Entry	It #1	It #2	It #3	Exit
s_3	b_3	b_3	b_3	b_3
a_3	5	5	5	5
	d_3	d_3	d_3	d_3
	e_3	e_3	e_3	g_3
	f_3	f_3	f_3	t_3
	6	6	6	

(b) One trace of execution through the loop.

Entry	It #1	It #2	It #3	Exit
s_3	b_3	b_3	b_3	b_3
a_3	c_3	c_3	c_3	c_3
	d_3	d_3	d_3	d_3
	e_3	e_3	e_3	g_3
	f_3	f_3	f_3	t_3
	6	6	6	

(c) Another trace of execution through the loop.

Let us demonstrate Equation (3.3) through the example in Figure 3.7. For the CFG* in Figure 3.7(a), assume the relative bound of the loop L_{b_3} is 3 and note that $IE(L^I_{b_3}) = \{6 \rightarrow 5, 6 \rightarrow 6\}$. We have traced two different executions through this loop on successive iterations in Figures 3.7(b) and 3.7(c), respectively, together with the loop entry and loop exit path. In the trace of execution of Figure 3.7(b), the iteration edge $6 \rightarrow 5$ is executed three times, i.e. its bound is equal to $b_{rel}(b_3)$, whereas in Figure 3.7(c), the iteration edge $6 \rightarrow 6$ only executes twice, i.e. its bound is equal to $b_{rel}(b_3) - 1$. As explained above, the reason for this difference is that, on the path exiting the loop, it is possible to hit ipoint 5. Thus, *any* iteration edge with a destination of 5 is bounded exactly by $b_{rel}(b_3)$: one execution every time a loop-back edge is traversed plus one execution when the loop exits. On the other hand, we cannot hit ipoint 6 on the loop exit path and $6 \rightarrow 6$ can only be executed each time a loop-back edge is traversed, which is clearly $b_{rel}(h_i) - 1$.

3.3.3 Loop-Entry and Loop-Exit Edges in the IPG

Besides identifying the iteration edges in I, our HMB framework also requires detection of loop-entry and loop-exit edges *relative to the next outer nesting level*. For instance, the IPET constrains the execution count of iteration edges with respect to either of these sets of edges. In addition, our trace parsing mechanism uses the loop-exit edges to gather relative bounds. Here we show how to detect these edges using the LNT T^C_L of C.

Let us first examine the simpler case of how to identify these edges in C (assuming it is reducible), before extending the intuition to its IPG I. For a reducible loop L_h in C, the standard definition of a loop-entry edge $u \rightarrow v$ (respectively loop-exit edge $u' \rightarrow v'$) is one satisfying $u \notin V(L_h)$ (respectively $u' \in V(L_h)$) and $v \in V(L_h)$ (respectively $v' \notin V(L_h)$). Note that, because each L_h is reducible, h must always be the unique destination of $u \rightarrow v$. These edges can thus be identified using properties of T^C_L as follows. For the loop-entry edge $u \rightarrow v$, v is an internal vertex and u cannot be a descendant of h in T^C_L, otherwise it would be enclosed in the loop L_h; for example, in Figure 3.2(a), $b_1 \rightarrow d_1$ is a loop-entry edge for L_{d_1}, and in Figure 3.5, b_1 is not a descendant of d_1. For the loop-exit edge $u' \rightarrow v'$, either u' is h (exit from a `for` loop) or a child of h in T_L^C (exit from a `do-while` loop or `break` statement) and v' is not a descendant of h in T_L^C; for example, in Figure 3.2(a), $d_1 \to k_1$ is a loop-exit edge for L_{d_1}, and in Figure 3.5, k_1 is not a descendant of d_1.

Observe a key implicit property of these edges as defined: flow of control can only enter (or exit) the next inner (or outer) nesting level relative to the current loop. However, this might not be the case in I as an ipoint transition can enter (or exit) several nested loops at a time. (This property can also be present in a reducible CFG* whereby `goto` statements can redirect flow of control out of several nested loops at a time.) For example, in Figure 3.2(b), it is not obvious which IPG loop — either $L^I_{d_1}$ or $L^I_{e_1}$ — the loop-entry edge $\mathtt{s}_1 \to 1$ is relative to because 1 is in both loops. For this reason, we propose a stricter definition as follows:

Definition 9. *Let G be a flow graph, T_L^G be its LNT, $u \to v$ be an edge in G, h be a header in G, and $y = lca_{T_L^G}(u,v)$. We say that $u \to v$ is a* ***(relative) loop-entry*** *edge (respectively* ***(relative) loop-exit*** *edge) for the loop L_h provided all of the following conditions are met:*

1. *$u \to v$ is not a cycle-inducing edge;*
2. *$parent_{T_L^G}(u) \neq parent_{T_L^G}(v)$ if u, v are not internal vertices in T_L^G or $u \neq parent_{T_L^G}(v)$ if u is an internal vertex in T_L^G;*
3. *h is the first header on the path $p : y\ (\overset{*}{\to}]\ v$ (respectively $q : y\ (\overset{*}{\to}]\ u$) in T_L^G.*

This definition appears rather convoluted and requires some explanation. Condition 1 states that these edges must be forward edges (see Definition 7). Condition 2 states that u and v must reside in different loops. Thus, condition 3 determines the outermost loop that is entered (respectively exited) on traversing $u \to v$ because L_h is the only loop in the next nesting level from L_y (h is a child of y in T_L^G); this is precisely the information that we require to bound cycle-inducing edges relative to their outer nesting level.

Thus, the problem of identifying loop-entry and loop-exit edges in a flow graph G can be reduced to the problem of performing off-line least-common ancestor queries (with the query set E_G) on a static tree (T_L^G), for which there are known solutions [12, 48]. All queries can be answered in constant time, thus we can identify the loop-entry and loop-exit edges of G in $O(|E_G|)$ time.

Returning to the initial problem of identifying loop-entry and loop-exit edges in I, observe that the LNT on which we perform least-common ancestor queries is T_L^C (because we do not explicitly construct the LNT of I). How this is done can be demonstrated with the LNT in Figure 3.5 and the IPG in Figure 3.2(b). By condition 1 in Definition 9, $1 \rightarrow 1, 3 \rightarrow 1, 3 \rightarrow 2, 3 \rightarrow 3$ cannot be loop-entry or loop-exit edges since they are iteration edges. By condition 2 in Definition 9, $\mathsf{s}_1 \rightarrow \mathsf{t}_1, 2 \rightarrow 3$ cannot be loop-entry or loop-exit edges since they are in the same loop. This reduces the query set to $\{\mathsf{s}_1 \rightarrow 1, \mathsf{s}_1 \rightarrow 2, \mathsf{s}_1 \rightarrow 3, 1 \rightarrow 2, 1 \rightarrow 3, 3 \rightarrow \mathsf{t}_1\}$. Following are the answers to these queries:

- $lca_{T_L^C}(\mathsf{s}_1, 1) = lca_{T_L^C}(\mathsf{s}_1, 2) = lca_{T_L^C}(\mathsf{s}_1, 3) = \mathsf{s}_1$. Observe that d_1 is the first header on each path s_1 $(\xrightarrow{*}]$ 1, s_1 $(\xrightarrow{*}]$ 2, and s_1 $(\xrightarrow{*}]$ 3, thus $\{\mathsf{s}_1 \rightarrow 1,\ \mathsf{s}_1 \rightarrow 2,\ \mathsf{s}_1 \rightarrow 3\}$ are loop-entry edges for the IPG loop $L^I_{d_1}$.
- $lca_{T_L^C}(1, 2) = d_1$. In this case, e_1 is the first header on the path e_1 $(\xrightarrow{*}]$ 1, thus $1 \rightarrow 2$ is a loop-exit edge for the IPG loop $L^I_{e_1}$.
- $lca_{T_L^C}(3, \mathsf{t}_1) = \mathsf{s}_1$. In this case, d_1 is the first header on the path s_1 $(\xrightarrow{*}]$ 3, thus $3 \rightarrow \mathsf{t}_1$ is a loop-exit edge for the IPG loop $L^I_{\mathsf{s}_1}$.

3.4 A Modified Data-Flow Framework to Build the IPG

In the previous section we explained how to identify IPG loops using the LNT of the CFG*. Here we describe a more complex algorithm than that in Figure 3.3 to construct the IPG I from a CFG* C. To this end, we make some modest changes to the underlying DFF described in Section 3.2.2 and control how the iterative algorithm operates in order to detect iteration edges on the fly for each CFG* loop (using Theorem 1).

Changes to the DFF are needed to compute the iteration edges for each IPG loop L^I_h because, according to Theorem 1, this requires two pieces of information:

- The ipoints in the CFG* loop L_h that are reachable from the header h on a (possibly empty) ipoint-free path in the loop DAG L'_h, i.e. these are the destinations of iteration edges.
- The ipoints in L_h that can reach a tail of L_h on a (possibly empty) ipoint-free path in the loop DAG L'_h, i.e. these are the sources of iteration edges.

The previous DFF already computes the sources of iteration edges: for each basic block tail t, a subset of $ipoints(t)$ holds the ipoints that can reach t in L'_h. (It is a subset because ipoints from an outer loop could also reach t and these are not sources of iteration edges for L_h.) However, we cannot infer the destinations of iteration edges for *basic block* headers in the same way because the DFF only computes reachability information concerning ipoints. For this reason, we define another semi-lattice L_{H_B}, which is the powerset 2^{H_B} over the set of basic block headers H_B. We further define the set of monotone, distributive functions $F_{H_\mathsf{B}} \subseteq 2^{H_\mathsf{B}} \rightarrow 2^{H_\mathsf{B}}$.

The semi-lattice L_{H_B} is used to decide which basic block headers can reach other vertices in its loop DAG on ipoint-free paths. Let us define the following set:

$$\forall v \in V_C: \; headers(v) = \{h | h \in H_\mathsf{B} \land \exists\, h \xrightarrow[\mathsf{B}]{*} v \text{ in the loop DAG } L'_h\}$$

It would appear that, in order to compute $headers(v)$ for each $v \in V_C$, we can perform a union of the basic block headers that can reach the basic block predecessors of v. However, observe that, for every $h \in headers(v)$, h must be an ancestor of v in T_L^C but that not every predecessor p of v is at the same loop-nesting level. This implies that there could be some $h' \in headers(p)$ that is not an ancestor of v, and therefore, we should not insert h' into $headers(v)$. In essence, we do not want the reachability of a header to other vertices in its loop to spill into the next outer loop-nesting level. Therefore, for a predecessor p of v, let us define the following subset of $headers(p)$:

$$headers_v(p) = \{h | h \in headers(p) \land h \text{ is an ancestor of } v \text{ in } T_L^C\}.$$

From these observations, we arrive at the following data-flow equation to compute $headers(v)$:

$$\forall v \in V_C : \; headers(v) = \begin{cases} \left(\bigcup_{p \in pred(v)} headers_v(p)\right) \cup \{v\} & \text{if } v \in H_{\mathsf{B}} \\ \bigcup_{p \in pred(v)} headers_v(p) & \text{otherwise} \end{cases} \tag{3.4}$$

3.4.1 The Algorithm

Construction of I and identification of its iteration edges therefore requires a solution to Equation (3.2) and Equation (3.4). Observe that we cannot simply solve these equations and then determine the iteration edge sets associated with each IPG loop L_h^I off-line. This is because Theorem 1 states that an IPG edge $u \rightarrow v \in IE(L_h^I)$ if, and only if, there is no acyclic path from u to v in the loop DAG L'_h (of the CFG* loop L_h). Thus, the algorithm must also decide whether such a path exists before inserting $u \rightarrow v$ into $IE(L_h^I)$.

Consequently, this section presents a more complex algorithm than the simple iterative algorithm, although it operates in a similar fashion, but restricts computations to particular induced subgraphs of C. The pseudo-code is split across procedures in Figures 3.8 and 3.9, the conventions of which require some explanation. First, our intention is to focus on the insertion of edges into I. For this reason, we utilise "Update set S with set T" in several places to avoid unnecessary clutter. The actual meaning of this is to union S with all members of T, and if S changes due to this update, then *changed* is set to **true** to force another iteration. Second, we do not associate a *headers* set with each ipoint; rather, for each basic block header h, we associate a set $dest(h)$ that stores the ipoints that are reachable from h on a non-empty ipoint-free path. This saves us from having to scan each ipoint descendant v of h in T_L^C to check whether $h \in headers(v)$.

The main procedure called to initiate IPG construction is displayed in Figure 3.8, which takes C and T_L^C as parameters and commences with a series of initialisations (lines 11-18) as follows. We conservatively estimate the values of $ipoints(v)$ (for each $v \in V_C$) and $headers(v)$ (for each $v \in \mathsf{B}$) to be the empty set, i.e. the bottom element of their respective semi-lattices. Furthermore, we assume that, for each CFG* loop, there is no corresponding loop in the IPG and that there are no destinations of iteration edges for each basic block header.

The next step of the algorithm is to produce a reverse post-ordering of each non-trivial loop L_h in C (line 19). These reverse post-orderings are required in the iterative part of the algorithm, and can be obtained as follows. First initialise an empty list for each non-trivial header in C and initiate a DFS from the entry vertex s. Once all descendants of a vertex v in the DFS tree have been visited, append v onto the start of the list at $parent_{T_L^C}(v)$ provided $v \neq$ s. (Note that, if v is a header then v essentially represents an abstract vertex, i.e. a collapsed loop, in the reverse post-ordering of $parent_{T_L^C}(v)$.) Furthermore, if v is a header, append v onto the start of the list at v. (Note that this completes the reverse post-ordering for L_v because, as we assume that C is reducible, every vertex $u \neq v$ in L_v must be a proper descendant of v in the DFS spanning tree [66]. Therefore, by the nesting of descendants' intervals (see Corollary 22.8 in [30]), u must already be in the list at v.)

The algorithm then performs an inside-out decomposition of C, i.e. from inner loops outwards to outer loops, using T_L^C (lines 20-53). For each loop L_h in C, we first build the forward edges of I from inside the loop DAG L'_h. After this, we analyse the entire cyclic region in C induced by L_h and update I with any remaining transitions not present in L'_h: by Theorem 1, all edges added in this step must be iteration edges of the IPG loop $IE(L_h^I)$.

We do not explicitly induce the loop L_h nor its loop DAG L'_h from C. Rather, we use the reverse post-ordering previously computed in conjunction with an integer *it* that indicates whether we should analyse L'_h ($it = 1$) or L_h ($it = 2$). As the reverse post-orderings only contain abstract vertices for inner loops, we need to know which ipoints are reachable from an inner header h' on ipoint-free paths so that, on analysing edges entering $L_{h'}$ from L_h, we can build the ipoint transitions with destinations inside $L_{h'}$. Clearly, h' is the only such ipoint if h' is an ipoint. Otherwise, we place h in its set $headers(h)$ (lines 22-23) so that we can determine reachability of h to other vertices in L_h.

The algorithm then constructs the ipoint transitions inside L_h (lines 24-53). This section of pseudo-code has strong similarities to the iterative algorithm in Figure 3.3, the essential difference being that the iterative part of this algorithm is employed for

```
   Input: $C, T_L^C$
   Output: $I$
11 foreach $v \in V_C$ do
12     $ipoints(v) := \emptyset$
13     if $v$ is basic block then
14         $headers(v) := \emptyset$
15     if $v$ is internal vertex in $T_L^C$ then
16         $IE(L_v^I) := \emptyset$
17         if $v$ is basic block then
18             $dest(v) := \emptyset$
19 Reverse post-order each non-trivial $L_h$ in $C$
20 for $i := height(T_L^C) - 1$ downto 0 by-1 do
21     foreach internal vertex $h$ with $level(h) = i$ do
22         if $h$ is basic block then
23             $headers(h) := \{h\}$
24         $it := 0$
25         $changed :=$ true
26         while $changed$ do
27             $changed :=$ false
28             $it := it + 1$
29             foreach $v$ in reverse post-order of $L_h$ do
30                 foreach $p \in pred(v)$ do
31                     $analyse :=$ false
32                     if $v = h$ then
33                         if $p \rightarrow v$ is loop-back edge and $it > 1$ then
34                             $analyse :=$ true
35                     else if $v$ not an internal vertex in $T_L^C$ then
36                         $analyse :=$ true
37                     else if $p \rightarrow v$ is not a loop-back edge then
38                         $analyse :=$ true
39                     if $analyse$ then
40                         if $p$ is ipoint then
41                             Update($p, v$)
42                         else
43                             foreach $k \in ipoints(p)$ do
44                                 Update($k, v$)
45                             if $it = 1$ and $h \in headers(p)$ then
46                                 if $v$ is ipoint then
47                                     Update $dest(h)$ with $\{v\}$
48                                 else
49                                     if $v$ is header of non-trivial loop and $v \neq h$ then
50                                         Update $headers(u)$ with $\{h\}$ provided $u$ is exit of $L_v$
                                           and $v \in headers(u)$
51                                         Update $dest(h)$ with $dest(v)$
52                                     else
53                                         Update $headers(v)$ with $\{h\}$
```

Figure 3.8. Algorithm to Construct the IPG using the Loop-Nesting Tree of the CFG*.

each CFG* loop L_h and that, for each vertex v, we specifically analyse each predecessor p in turn (line 30). The reason for this is that, as noted above, we only want to analyse forward edges in C when $it = 1$, but because we do not explicitly induce loop DAGs, we must be able to ignore loop-back edges as and when required. Analysis of edges in C is therefore controlled through the boolean variable *analyse*.

For each $p \rightarrow v \in E_C$, we initially assume that we do not want to analyse $p \rightarrow v$ (line 31). If v is the header of the current loop under inspection (i.e. the first vertex in the reverse post-ordering of L_h), $p \rightarrow v$ is a loop-back edge and $it > 1$ (lines 32-34) then we analyse the edge because $it > 1$ indicates that we want to add the iteration edges in I associated with this loop. On the other hand, v could be the header of an inner loop. In this case, we do not analyse $p \rightarrow v$ if it is a loop-back edge because p does not appear in the reverse post-ordering of L_h, and consequently, none of the sets at p are affected whilst analysing L_h (lines 37-38). The only alternative is that v is a leaf in T_L^C such that $parent_{T_L^C}(v) = h$, and thus every $p \rightarrow v$ must be a forward edge (lines 35-36).

```
**Input**: $y, v$
**if** *$v$ is ipoint* **then**
    **if** $y \notin ipoints(v)$ **then**
        $changed :=$ **true**
        $ipoints(v) \cup_{=} \{y\}$
        **if** $it = 2$ **then**
            $IE(L_h^I) \cup_{=} y_I \rightarrow v_I$
**else**
    **if** *$v$ is header of non-trivial loop and $v \neq h$* **then**
        Update $ipoints(u)$ with $\{y\}$ provided $u$ is exit of $L_v$ and $v \in headers(u)$
        **foreach** $w \in dest(v)$ **do**
            **if** $y \notin ipoints(w)$ **then**
                $changed :=$ **true**
                $ipoints(w) \cup_{=} \{y\}$
                **if** $it = 2$ **then**
                    $IE(L_h^I) \cup_{=} y_I \rightarrow w_I$
            **else if** *$it = 2$ and $y_I \rightarrow w_I$ is iteration edge* **then**
                $IE(L_h^I) \cup_{=} y_I \rightarrow w_I$
    **else**
        Update $ipoints(v)$ with $\{y\}$
```

Figure 3.9. `Update`: Helper Procedure in IPG Construction.

The analysis of the edge $p \rightarrow v$ essentially updates the sets associated with v with the information at p, i.e. we partially solve Equation (3.2) and Equation (3.4). If p is an ipoint then this means that p can reach v on a non-empty ipoint-free path; otherwise, every ipoint $k \in ipoints(p)$ can reach v on a non-empty ipoint-free path. In both cases, the procedure `Update` is called, which takes two parameters, y and v, and performs one of three actions depending on the properties of v as follows:

v **is an ipoint:** As y is also an ipoint, we insert the edge $y_I \to v_I$ into I if it does not yet exist and mark it as an iteration edge for L_h^I if $it = 2$ (lines 55-59).

v **is a basic block header of a non-trivial loop:** As v essentially represents a collapsed loop, we need to update every exit u in L_v with the information that y can reach u *provided* $v \in headers(u)$ (line 62), since this signifies there is an ipoint-free path from v to u. (Observe that we are only interested in the ipoint-free paths that pass through L_v, thus it suffices to only update its exits with this information. In practice this speeds up the overall running time of the algorithm as we do not redundantly re-analyse all vertices in inner loops. In theory, however, the running time remains quadratic due to the iterative nature of the algorithm.) Furthermore, y can also reach every ipoint w in L_v such that $w \in dest(v)$. Hence, as for the first case, we add all edges $y_I \to w_I$ into I if they do not yet exist and mark them as iteration edges for L_h^I if $it = 2$ (lines 63-68). On the other hand, if $y_I \to w_I$ has already been added to I and marked as an iteration edge of some inner loop $L_{h'}^I$, then $y_I \to w_I$ is marked as an iteration edge for L_h^I as well provided $it = 2$ (lines 69-70). This is because, as noted above, an iteration edge can belong to multiple IPG loops.

v **is a basic block:** Obviously, y can reach v on a non-empty ipoint-free path, thus it is inserted into $ipoints(v)$ (line 72).

The analysis of the edge $p \to v$ is not complete when p is a basic block, $it = 1$ and $h \in headers(p)$ because we need to propagate the reachability information of h onto v. The actions taken here are similar to the actions taken when p is an ipoint, and hence we can summarise them according to the properties of v as follows (lines 45-53):

v **is an ipoint:** As h can reach p on an ipoint-free path, v is a destination of ipoint transitions entering the collapsed loop L_h (line 47).

v is a basic block header of a non-trivial loop: As for the case when p was an ipoint, we update every exit u in L_v satisfying $v \in headers(u)$ with the information that h can reach u on an ipoint-free path (line 50). Also, this indicates that all ipoints reachable from v in the collapsed loop L_v (i.e. those in $dest(v)$) are also reachable from h (line 51).

v is a basic block: h is inserted into $headers(v)$ (line 53).

An Example

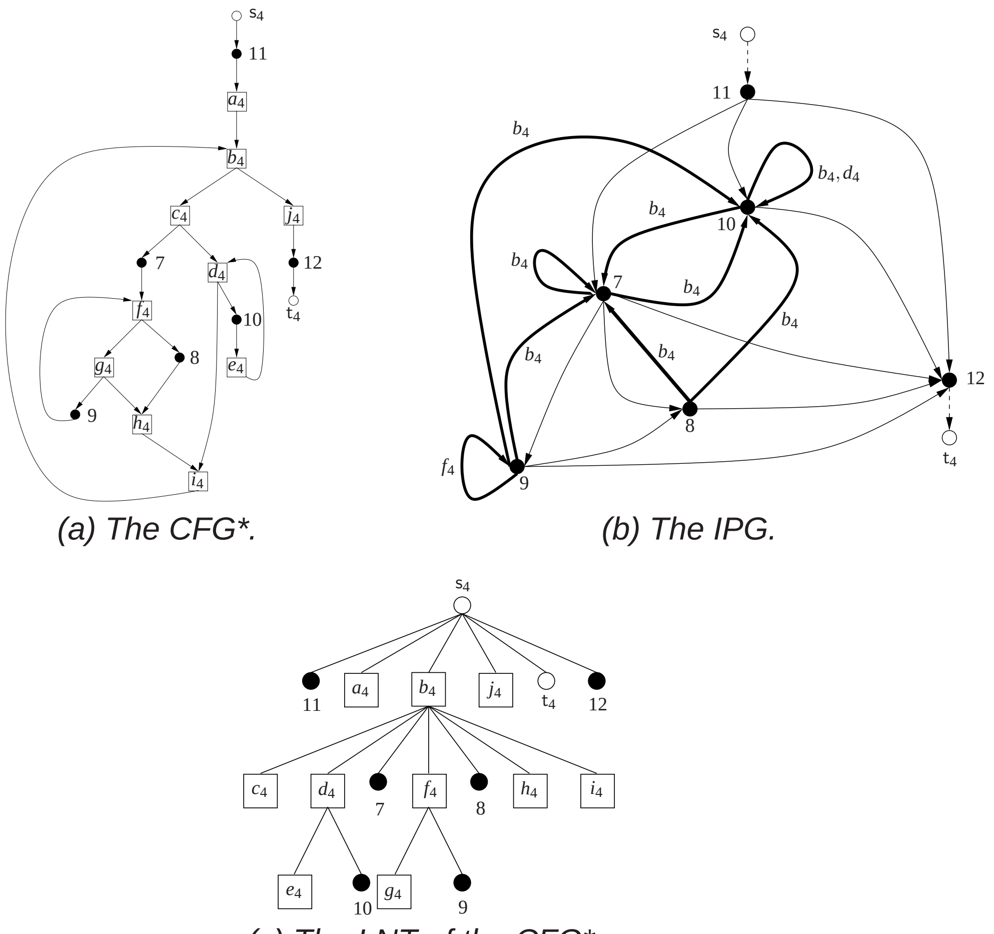
To illustrate the operation of the algorithm, consider Figure 3.10, which depicts a CFG*, its IPG, and its LNT. This example is deliberately more complicated than previous examples as we also wish to exhibit iteration edges that belong to multiple iteration edge sets. Observe that the CFG* of Figure 3.10(a) has four non-trivial loops: L_{d_4} and L_{f_4} are nested in L_{b_4} and L_{b_4} is nested in the dummy loop L_{s_4}. These data are represented in Figure 3.10(c).

In Figure 3.11, we have traced through the major stages of the algorithm for each loop in the CFG*. All tables in this figure display, for each vertex v, how the sets $ipoints(v), dest(v), headers(v)$ change on successive iterations during the iterative part of the algorithm. Shaded cells indicate sets that have changed from the previous iteration, and blank cells indicate that a particular set is not associated with a particular vertex. A further note to make is that the computations performed on the final iteration have been omitted because none of the sets change (this can easily be verified by the reader). The first column of each table lists an (arbitrarily chosen) reverse post-ordering of the vertices inside the loop, and all other columns display updates to the sets. We have ordered the tables with respect to the order in which the loops are processed by the algorithm.

Before processing each loop, the algorithm first places b_4, d_4, and f_4, into their respective sets $headers(b_4)$, $headers(d_4)$, and $headers(f_4)$ as all headers except s_4 are basic blocks.

For the inner loop L_{d_4}, we do not analyse the edge $e_4 \rightarrow d_4$ in the first iteration because it is a loop-back edge. Analysis of edges thus proceeds in the following

Figure 3.10. Example used to Demonstrate Construction of IPG using Algorithm in Figure 3.8.



(a) The CFG*.

(b) The IPG.

(c) The LNT of the CFG*.

order:

- $d_4 \rightarrow 10$: As $it = 1$ and $d_4 \in headers(d_4)$, $\{10\}$ is unioned into $dest(d_4)$.
- $10 \rightarrow e_4$: 10 can reach e_4 on a non-empty ipoint-free path and $\{10\}$ is unioned into $ipoints(e_4)$.

On the second iteration, edges are analysed in the following order:

- $e_4 \rightarrow d_4$: As $it > 1$, we now analyse the loop-back edge of L_{e_4}, and thus $\{10\}$ is unioned into $ipoints(d_4)$.
- $d_4 \rightarrow 10$: $10 \rightarrow 10$ is inserted into the IPG because $10 \in ipoints(d_4)$ and, because $it > 1$, $10 \rightarrow 10$ is an iteration edge for $L^I_{d_4}$.
- $10 \rightarrow e_4$: No changes.

On analysis of the other inner loop L_{f_4}, the steps taken are very similar and are summarised as follows: 9 is unioned into $dest(f_4)$; $9 \rightarrow 9$ is inserted into the IPG; $9 \rightarrow 9$ is flagged as an iteration edge for $L^I_{f_4}$.

Analysis of the outer loop L_{b_4} is markedly more involved. First observe that vertices d_4 and f_4 appearing in the reverse post-ordering are actually abstract vertices that represent the collapsed loops L_{d_4} and L_{f_4}. Whilst analysing the loop-entry edges of these loops, the sets of some of the vertices inside these loops change (specifically $10, g_4, 9$); we have therefore included them in the table, although it is important to stress that they do not appear in the actual reverse post-ordering of L_{b_4}.

Consider analysis of the following edges in the first iteration:

- $c_4 \rightarrow d_4$: As $it = 1$, $b_4 \in headers(c_4)$ and d_4 is the header of a non-trivial loop, two actions are undertaken. First, the set $dest(d_4)$ is unioned into $dest(b_4)$, and $dest(b_4) = \{10\}$. Second, b_4 is propagated to the only exit, namely d_4, of L_{d_4} such that $d_4 \in headers(u)$.
- $c_4 \rightarrow 7$: As $it = 1$ and $b_4 \in headers(c_4)$, 7 is reachable from b_4 on a non-empty ipoint-free path, thus $\{7\}$ is unioned into $dest(b_4)$. This is the last element inserted into $dest(b_4)$; hence, $dest(b_4) = \{7, 10\}$.
- $7 \rightarrow f_4$: As $9 \in dest(f_4)$, the edge $7 \rightarrow 9$ is inserted into the IPG. However, as $it = 1$, this edge is not identified as an iteration edge. Furthermore, note that both exits of L_{f_4}, namely f_4 and g_4, have f_4 in their respective *headers* sets, which causes $\{7\}$ to be unioned into their respective *ipoints* sets. It is worth re-emphasising that this step is necessary since the only path from 7 to 8 is through the collapsed inner loop.
- $f_4 \rightarrow 8$: As 7 and 9 are both in $ipoints(f_4)$, edges $7 \rightarrow 8$ and $9 \rightarrow 8$ are inserted into the IPG, neither of which are iteration edges.

Next consider the analysis of the following edges in the second iteration:

- $i_4 \rightarrow b_4$: The set $ipoints(i_4)$ is unioned into $ipoints(b_4)$.
- $c_4 \rightarrow d_4$: Because 10 is the only element of $dest(d_4)$, edges $8 \rightarrow 10, 9 \rightarrow 10, 7 \rightarrow 10$ are added to the IPG, all of which are iteration edges for L_{b_4}. However, also observe that, although $10 \rightarrow 10$ already exists in the IPG, it is also identified as an iteration edge for L_{b_4}.

Figure 3.11. Computations Performed by Algorithm in Figure 3.8 for Example in Figure 3.10.

	Before #1			**After #1**			**After #2**		
	$dest(v)$	$headers(v)$	$ipoints(v)$	$dest(v)$	$headers(v)$	$ipoints(v)$	$dest(v)$	$headers(v)$	$ipoints(v)$
d_4	$\emptyset$	$\{d_4\}$	$\emptyset$	$\{10\}$	$\{d_4\}$	$\emptyset$	$\{10\}$	$\{d_4\}$	$\{10\}$
10			$\emptyset$			$\emptyset$			$\{10\}$
e_4		$\emptyset$	$\emptyset$		$\emptyset$	$\{10\}$		$\emptyset$	$\{10\}$

(a) In CFG* loop L_{d_4}.

	Before #1			**After #1**			**After #2**		
	$dest(v)$	$headers(v)$	$ipoints(v)$	$dest(v)$	$headers(v)$	$ipoints(v)$	$dest(v)$	$headers(v)$	$ipoints(v)$
f_4	$\emptyset$	$\{f_4\}$	$\emptyset$	$\{9\}$	$\{f_4\}$	$\emptyset$	$\{9\}$	$\{f_4\}$	$\{9\}$
g_4		$\emptyset$	$\emptyset$		$\{f_4\}$	$\emptyset$		$\{f_4\}$	$\{9\}$
9			$\emptyset$			$\emptyset$			$\{9\}$

(b) In CFG* loop L_{f_4}.

	Before #1			**After #1**			**After #2**		
	$dest(v)$	$headers(v)$	$ipoints(v)$	$dest(v)$	$headers(v)$	$ipoints(v)$	$dest(v)$	$headers(v)$	$ipoints(v)$
b_4	$\emptyset$	$\{b_4\}$	$\emptyset$	$\{7,10\}$	$\{b_4\}$	$\emptyset$	$\{7,10\}$	$\{b_4\}$	$\{8,9,7,10\}$
c_4		$\emptyset$	$\emptyset$		$\{b_4\}$	$\emptyset$		$\{b_4\}$	$\{8,9,7,10\}$
d_4	$\{10\}$	$\{d_4\}$	$\{10\}$	$\{10\}$	$\{b_4,d_4\}$	$\{10\}$	$\{10\}$	$\{b_4,d_4\}$	$\{8,9,7,10\}$
10			$\{10\}$			$\{10\}$			$\{8,9,7,10\}$
7			$\emptyset$			$\emptyset$			$\{8,9,7,10\}$
f_4	$\{9\}$	$\{f_4\}$	$\{9\}$	$\{9\}$	$\{f_4\}$	$\{9,7\}$	$\{9\}$	$\{f_4\}$	$\{9,7\}$
g_4		$\{f_4\}$	$\{9\}$		$\{f_4\}$	$\{9,7\}$		$\{f_4\}$	$\{9,7\}$
9			$\{9\}$			$\{9,7\}$			$\{9,7\}$
8			$\emptyset$			$\{9,7\}$			$\{9,7\}$
h_4		$\emptyset$	$\emptyset$		$\emptyset$	$\{8,9,7\}$		$\emptyset$	$\{8,9,7\}$
i_4		$\emptyset$	$\emptyset$		$\{b_4\}$	$\{8,9,7,10\}$		$\{b_4\}$	$\{8,9,7,10\}$

(c) In CFG* loop L_{b_4}.

	Before #1			**After #1**		
	$dest(v)$	$headers(v)$	$ipoints(v)$	$dest(v)$	$headers(v)$	$ipoints(v)$
$\mathtt{s}_4$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
11			$\emptyset$			$\{\mathtt{s}_4\}$
b_4	$\{7,10\}$	$\{b_4\}$	$\{8,9,7,10\}$	$\{7,10\}$	$\{b_4\}$	$\{8,9,7,10,11\}$
7			$\{8,9,7,10\}$			$\{8,9,7,10,11\}$
10			$\{8,9,7,10\}$			$\{8,9,7,10,11\}$
j_4		$\emptyset$	$\emptyset$		$\emptyset$	$\{8,9,7,10,11\}$
12			$\emptyset$			$\{8,9,7,10,11\}$
$\mathtt{t}_4$			$\emptyset$			$\{12\}$

(d) In CFG* loop $L_{\mathtt{s}_4}$.

- $c_4 \rightarrow 7$: Edges $8 \rightarrow 7, 9 \rightarrow 7, 7 \rightarrow 7, 10 \rightarrow 7$ are inserted into the IPG and identified as iteration edges for $L^I_{b_4}$.

The final loop to analyse is that of L_{s_4}, which causes the remainder of the IPG edges, $\mathsf{s}_4 \rightarrow 11, 11 \rightarrow 7, 11 \rightarrow 10, 11 \rightarrow 12, 7 \rightarrow 12, 8 \rightarrow 12, 9 \rightarrow 12, 10 \rightarrow 12, 12 \rightarrow \mathsf{t}_4$, to be inserted.

The final IPG resulting from this construction is shown in Figure 3.10(b). Following are the iteration edge sets associated with each IPG loop:

- $IE(L^I_{d_4}) = \{10 \rightarrow 10\}$.
- $IE(L^I_{f_4}) = \{9 \rightarrow 9\}$.
- $IE(L^I_{b_4}) = \{10 \rightarrow 10, 9 \rightarrow 10, 8 \rightarrow 10, 7 \rightarrow 10, 10 \rightarrow 7, 9 \rightarrow 7, 8 \rightarrow 7, 7 \rightarrow 7\}$.

Figure 3.10(b) represents iteration edges as emboldened edges and labels them with the headers of the CFG* loops that they are associated with.

3.5 Interprocedural Analysis

Thus far we have described the analysis and construction of the IPG assuming a unique procedure. However, programs inevitably contain multiple procedures, and thus there has to be a mechanism to handle interprocedural relations. In particular, our HMB framework must parse timing traces (in order to extract the WCETs of ipoint transitions) and be able to drive the calculation engine across procedure boundaries. We begin the section with an explanation of our virtual inlining mechanism which provides visibility to procedure calls in the IPG of each procedure. Following that, we show how to use the set of IPGs to parse timing traces and to calculate a WCET estimate.

3.5.1 Master Ipoint Inlining

One straightforward way to handle procedure calls is to virtually inline the CFG* of the callee at each call site in the caller, resulting in a unique CFG*, and ultimately, a unique inlined IPG. This simplifies interprocedural analysis as the **Trace Parser** (TP) just walks the inlined IPG for each trace, and calculations across procedure boundaries are implicitly supported. The biggest weakness, however, is that the inlined IPG can grow exponentially in size because of the duplication process, thus limiting its usage to small-scale programs.

Instead of duplicating the IPG at each call site, we inline a subset of the ipoints — but none of the transitions — from the callee into the caller. These duplicated *master ipoints* provide visibility of the call to (and return from) each callee. Intuitively, master ipoints are those which are observed first and last in a trace on procedure invocation and return. Formally:

Definition 10. *For an IPG $I = \langle \mathsf{I}, E_I, \mathsf{s}, \mathsf{t} \rangle$:*

- $\overrightarrow{M_I} = \{u \in \mathsf{I} - \{\mathsf{t}\} | u \in succ(\mathsf{s})\}$ *is its set of* ***master entry ipoints****.*
- $\overleftarrow{M_I} = \{u \in \mathsf{I} - \{\mathsf{s}\} | u \in pred(\mathsf{t})\}$ *is its set of* ***master exit ipoints****.*

Therefore, inlining a master entry ipoint into the caller allows the TP to detect when a procedure call has arisen and hence it can switch to the IPG of the callee. On the other hand, inlining a master exit ipoint provides the return location in the IPG of the caller once the portion of the trace inside the callee has been processed.

To describe master ipoint inlining in greater detail, let us consider programs comprising multiple procedures $p_1, p_2, \ldots, p_n$ and assume that a call graph models the interprocedural relations, which is formally defined as follows:

Definition 11. *The* ***call graph*** *of a program with a set of procedures $\{p_1, p_2, \ldots, p_n\}$ is a digraph $\mathscr{P} = \langle V_{\mathscr{P}}, E_{\mathscr{P}}, S, r \rangle$ such that:*

- $V_{\mathscr{P}} = \{p_1, p_2, \ldots, p_n\}$.
- *$r \in V_{\mathscr{P}}$ is a distinguished vertex termed the* ***main*** *procedure for which $|pred(r)| = 0$. Every vertex v can be reached from r, thus preventing dead code. We also assume that r is instrumented so that each new trace begins with a master entry ipoint in r.*
- *S is a set of basic blocks termed* ***call sites****.*
- *$E_{\mathscr{P}} \subseteq V_{\mathscr{P}} \times V_{\mathscr{P}} \times S$ is the set of labelled edges that we term* ***contexts****. For an edge $(f, g, s) \in E_{\mathscr{P}}$, f is termed the* ***caller*** *and g is termed the* ***callee****. We also*

include a dummy context, denoted (r, r, λ), which is essentially the initial context when the program is invoked.

- *There are no cycles, thus preventing recursion. This is motivated by noting that recursion is seldom present in embedded software as the constrained resources of embedded systems do not provide large stack memory.*

In addition to the call graph, we also require the CFG* and the IPG of each procedure p_i. As we inline master ipoints from the IPG of a callee g into the CFG* of the caller f, the CFG* of f changes. Therefore, we use C_{p_i} to denote the CFG* of p_i *before* master ipoint inlining, C'_{p_i} to denote the CFG* *after* master ipoint inlining, and I_{p_i} to denote the IPG of p_i that is constructed from C'_{p_i}.

The mechanics of master ipoint inlining are presented in Figure 3.12, which takes the call graph $\mathscr{P}$ and the set of CFG*s $\{C_{p_1}, C_{p_2}, \ldots, C_{p_n}\}$ as parameters and operates as follows. Each context (f, g, s) (except the dummy context) is processed in reverse topological order so that procedure calls are considered in a bottom-up fashion (lines 73- 86). We only inline the master ipoints from g when I_g satisfies $|V(I_g)| > 2$ (line 74). This is because an IPG must always contain at least two *ghost* ipoints, namely s, t. Therefore, if $|V(I_g)| = 2$, then there are no ipoints in I_g that will be seen in a trace, and the call at s does not require inlining. On the other hand, when $|V(I_g)| > 2$, master entry and master exit ipoints from I_g are duplicated into C_f (line 75), which we denote as $\overrightarrow{M}'_g$ and $\overleftarrow{M}'_g$, respectively, to avoid confusion with the actual master ipoints of g.

To allow the TP to disambiguate between ipoints inlined from various procedures, we associate a **procedure identifier** to each ipoint u, denoted P_u, which is the procedure causing u to trigger in program execution. Here we set the procedure identifier of each inlined ipoint to that of the callee (line 76). Observe that this step subtly assumes that inlined ipoints do not become master ipoints in the caller; that is, each inlined ipoint must be triggered by g during program execution and not by another procedure g' that is called via g. This allows the TP to wind and unwind the call stack one procedure at a time.

Once these master ipoints have been duplicated and inserted into f, a simple relinking with the call site s takes place. Every predecessor p of s is redirected towards each

```
   Input: $\mathscr{P}, \{C_{p_1}, C_{p_2}, \ldots, C_{p_n}\}$
   Output: $\{C'_{p_1}, C'_{p_2}, \ldots, C'_{p_n}\}$
73 foreach $(f,g,s) \in E_{\mathscr{P}} - \{(r,r,\lambda)\}$ in reverse topological order do
74     if $|V(I_g)| > 2$ then
75         Add inlined master entries $\overset{\rightarrow\prime}{M}_g$ and master exits $\overset{\leftarrow\prime}{M}_g$ to $V(C_f)$
76         Set procedure identifier of each inlined ipoint to $g$
77         foreach $u \in \overset{\rightarrow\prime}{M}_g$ do
78             foreach $p \in pred(s)$ do
79                 Add $p \rightarrow u$ to $E(C_f)$
80                 Remove $p \rightarrow s$ from $E(C_f)$
81             Add $u \rightarrow s$ to $E(C_f)$
82         foreach $u' \in \overset{\leftarrow\prime}{M}_g$ do
83             foreach $p' \in succ(s)$ do
84                 Add $u' \rightarrow p'$ to $E(C_f)$
85                 Remove $s \rightarrow p'$ from $E(C_f)$
86             Add $s \rightarrow u'$ to $E(C_f)$
```

Figure 3.12. Algorithm to Effect Master Ipoint Inlining.

$u \in \overset{\rightarrow\prime}{M}_g$ (line 79), the edge $p \rightarrow s$ is removed (line 80) and then each u is connected to s (line 81). Thus, if C_f contained a path $w \xrightarrow[\mathsf{B}]{+} s$, where w is an ipoint, then C'_f will instead contain a path $w \xrightarrow[\mathsf{B}]{+} u$. Consequently, the TP can detect the call to g in I_f once the *trace edge* $w \rightarrow u \in E(I_f)$ has been traversed. In a similar fashion, every $u' \in \overset{\leftarrow\prime}{M}_g$ is connected to each successor p' of s (line 84), the edge $s \rightarrow p'$ is removed (line 85), and then s is redirected towards each u' (line 86). Thus, if C_f contains a path $s \xrightarrow[\mathsf{B}]{+} w'$, where w' is an ipoint, then C'_f will instead contain a path $u' \xrightarrow[\mathsf{B}]{+} w'$. This provides the location in I_f at which the TP should return once g has finished executing, since the next trace edge to traverse will be in I_f.

Note that every inlined master entry ipoint in I_f has the set of inlined master exit ipoints as its successors. Since there are a number of ipoint transitions between these inlined ipoints (in the call chain), these edges essentially black box the procedure call and are thus termed *context edges*. Formally:

Definition 12. *For a CFG* C and its IPG I, an edge $u \rightarrow v \in E_I$ is a **context edge** if u is an inlined master entry exit and v is an inlined master exit ipoint. Furthermore, $\mathsf{B}(P(u \rightarrow v))$ is a singleton set containing a call site in C.*

Context edges serve two purposes in our analysis. First, they allow the TP to reconstruct the exact context from the properties of the IPG, allowing trace data to be retrieved on a per context basis as required — how this is done is described in Section 3.5.2. Second, they allow the calculation engine to incorporate the (modularised) WCET calculation of the context into the caller — how this is done is described in Section 3.5.3.

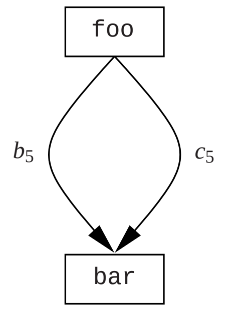
One potential, yet rare, problem with master ipoint inlining as described above is that it causes irreducibility in C'_f. This occurs when s is the header of a (reducible) loop L_s in C_f and I_g has multiple master entry ipoints, i.e. $\mid \overrightarrow{M_g} \mid > 1$. In this case, each inlined master entry ipoint will become an entry into the loop; that is, L_s will be transformed into an irreducible loop with headers $\overrightarrow{M}'_g$. Hence, I_f must be built through the simple iterative algorithm of Figure 3.3 (because C'_f is irreducible) and this complicates the identification of IPG loops.

We propose two solutions to this problem. The first is to ensure that every procedure contains a unique master entry ipoint. This is typically the most workable as a user (or automatic tool) can easily detect the problem. On the other hand, if no control over the instrumentation profile can be assumed, we can force reducibility in C'_f simply by inlining a unique ipoint that effectively represents all master entry ipoints in I_g. The weakness of this approach, however, is that we might lose some precision in the calculation on I_f as a portion of the code from the callee is effectively considered within the context of the caller due to master ipoint inlining. Therefore, by merging all master entry ipoints of I_g into a single vertex in I_f, every transition from an ipoint w to an inlined master entry ipoint forces the calculation engine to choose the transition (into g) with the largest value. That is, we cannot constrain the execution count of the transition with the larger value. Another disadvantage is that we must build the LNT of C_f (to detect headers that are call sites) and of C'_f (to build I_f), adding to the overall overhead, even though there are almost-linear time algorithms to construct a LNT, as explained in Section 3.3. In general, we attempt to avoid the problem entirely by choosing the first solution where applicable.

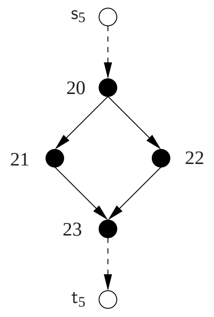
An Example

We illustrate master ipoint inlining in Figure 3.13 for a program containing two procedures (`foo` and `bar`) whose call relations are depicted in Figure 3.13(a). As `bar` is a leaf in the call graph, it does not require inlining, thus we have omitted its CFG*. The IPG of `bar` is, however, shown in Figure 3.13(b) because its master ipoints must be inlined into the CFG* of `foo`. Note that $\overrightarrow{M}_{\texttt{bar}} = \{20\}$ and that $\overleftarrow{M}_{\texttt{bar}} = \{23\}$.

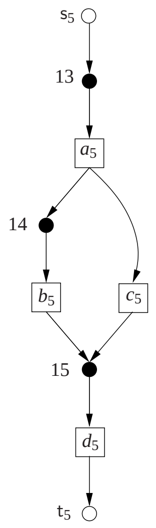
Figure 3.13. Example of Master Ipoint Inlining.



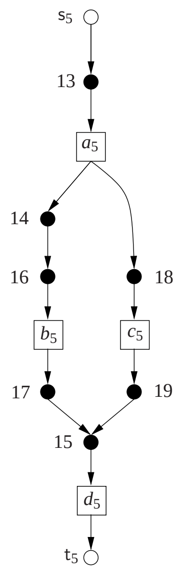
(a) The call graph.



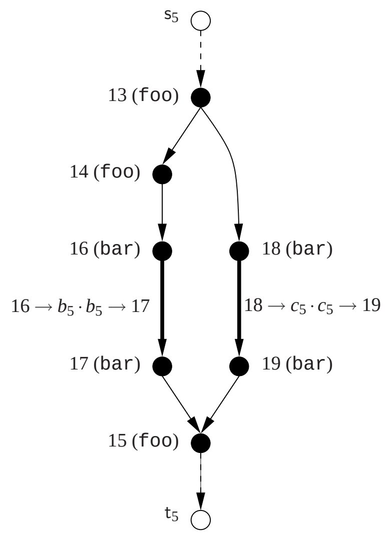
(b) IPG of `bar`.



(c) CFG of `foo` before master ipoint inlining.*



(d) CFG of `foo` after master ipoint inlining.*


s_5
13 (foo)
14 (foo)
16 (bar)
18 (bar)
$16 \rightarrow b_5 \cdot b_5 \rightarrow 17$
$18 \rightarrow c_5 \cdot c_5 \rightarrow 19$
17 (bar)
19 (bar)
15 (foo)
t_5

(e) IPG of `foo`.

The CFG*s of `foo` before and after master ipoint inlining are shown in Figures 3.13(c) and 3.13(d), respectively. For the context $(\texttt{foo}, \texttt{bar}, b_5)$, ipoints 16 and 17 are the inlined master ipoints from `bar`, i.e. $\overrightarrow{M}'_{\texttt{bar}} = \{16\}$ and $\overleftarrow{M}'_{\texttt{bar}} = \{17\}$. These additional ipoints are linked into $C'_{\texttt{foo}}$ through the edge insertions of $16 \rightarrow b_5$, $14 \rightarrow 16$, $b_5 \rightarrow 17$, and $17 \rightarrow 15$ together with the edge deletions of $14 \rightarrow b_5$ and $b_5 \rightarrow 15$. The context $(\texttt{foo}, \texttt{bar}, c_5)$ is analogously inlined, except that 18 and 19 are the inlined master ipoints.

The IPG of `foo`, which is constructed from the CFG* in Figure 3.13(d), is shown in Figure 3.13(e) in which trace ipoints are annotated with their procedure identifier. Following are properties to note: $13 \rightarrow 18$ and $14 \rightarrow 16$ are trace edges that detect a call to `bar` in trace parsing as the source and destinations have different procedure identifiers; $16 \rightarrow 17$ and $18 \rightarrow 19$ are context edges representing the call to `bar` at respective call sites b_5 and c_5 (indicated by the unique basic block in their path expressions); depending on the context invoked, the trace will return to either 17 or 19 once `bar` has finished executing, i.e. once ipoint 23 is seen in the trace.

3.5.2 Trace Parsing

After master ipoint inlining, we use the set of IPGs to extract **timing data** from the trace file on a per context basis. Timing data encompasses any non-functional property that is required, or is optional, in the calculation.

For our analysis, we must extract the WCETs of IPG edges as these are the IPG's atomic units of computation. The optional data concern flow analysis, such as loop bounds and infeasible paths. Here we focus on gathering two types of loop bounds: those relative to the next outer nesting level (**relative bounds**) and those bounding the execution in any single invocation of a context (**frequency bounds**). Both tree-based approaches and the IPET require relative bounds, whilst the IPET can also integrate frequency bounds to tighten the final estimate. For example, in the triangular loop nest of `Bubblesort`, the relative bound of the inner loop is n, whereas its frequency bound is $\frac{n(n+1)}{2}$, where n is the number of elements to sort.

Although Section 3.3.2 demonstrated how to map relative bounds obtained through static analysis onto the IPG, extracting such information from timing traces has its advantages. One limitation of contemporary static analysis techniques [47, 52], as noted in Chapter 2, is that they cannot always provide precise bounds for all loops due to the Halting problem. The user is then expected to fill in any gaps so that the calculation can actually run. Therefore, the prime motivation for extracting bounds from traces is that they allow our HMB framework to operate automatically without user interaction. Evidently, they can also serve to validate those provided by the user (as it is often error-prone), or alternatively, be fed into a SA tool as an initial guess on the bound, which is then verified.

The Automata of an IPG

Whilst parsing timing traces, the IPG becomes an automata as it is walked according to the sequence of tokens, i.e. trace identifiers (see Definition 1 in Chapter 2), in the trace file. As trace identifiers need not be unique, both Deterministic Finite Automatas (DFA) and Non-Deterministic Finite Automatas (NDFA) are possible.

In a DFA, given a particular ipoint in the IPG, the next token in the trace uniquely identifies the successor ipoint to advance to. It follows that, for any trace, we can resolve the *exact path* through the IPG. On the other hand, in a NDFA, the next token does not necessarily identify the successor ipoint. Path resolution is only satisfied if there are no two identical sequences of trace identifiers leading to some merge vertex in the IPG. This can lead to a less precise WCET estimate because we must conservatively assume that the timing data retrieved applies to each path identified by the trace.

For these reasons, it is desirable to assign trace identifiers to ipoints such that each trace resolves a unique path through the IPG. In practice, producing a DFA is easily attained by assigning a unique trace identifier to each ipoint *provided* there are no context disambiguation issues. For instance, if the CFG* of `foo` in Figure 3.13(c) did not contain the ipoint 14 then its IPG would instead contain the transitions $13 \rightarrow 16$ and $13 \rightarrow 18$. As 16 and 18 effectively represent the master entry ipoint 20 of `bar` (in Figure 3.13(b)), it is impossible to assign a trace identifier to 20 that resolves the ambiguity, and thus the IPG of `foo` becomes a NDFA.

Our TP assumes that each IPG is a DFA so that we can implement a simple one token lookahead scheme without the need to backtrack. We further assume that, for each IPG, none of its master exit ipoints is the source of an iteration edge; this basically means that, each time a master exit ipoint is encountered, a procedure is about to return, or if the IPG is the root procedure, a new trace is about to begin.

A complete description of trace parsing under these assumptions appears in Figure 3.14. Some of the conventions and notation require some explanation. First, it is useful to visualise the TP as writing some temporary variables whilst extracting the timing data, and at particular points, populating a database with the values in these variables. Consequently, the term "commit" appears in several places of the algorithm to indicate that the data associated with a particular IPG edge should be written to the database (if it exceeds the current value held) and that the temporary value held by the TP should be reset to zero. Second, we use t_u to denote the trace identifier of an ipoint u. Third, the TP extracts, for each *trace* edge $u \rightarrow v$ in a particular context (f,g,s), both its WCET, denoted $wcet^{(f,g,s)}(u \rightarrow v)$, and its frequency bound, denoted $b_{max}^{(f,g,s)}(u \rightarrow v)$. In addition, the TP uses the iteration edge sets associated with each IPG loop L_h^I to obtain the relative bound of L_h^I in a particular context (f,g,s), denoted $b_{rel}^{(f,g,s)}(h)$.

Trace parsing is initiated with the trace file *traces*, the set of IPGs $\{I_{p_1}, I_{p_2}, \ldots, I_{p_n}\}$, and the set of LNTs of the CFG* $\{T_L^{C_{p_1}}, T_L^{C_{p_2}}, \ldots, T_L^{C_{p_n}}\}$. The TP keeps track of the current context in the variable (f,g,s), beginning with the dummy context (r,r,λ) (line 87). Furthermore, the TP always walks the IPG of the callee g as the program being executed must be emitting ipoints in g; we always commence a new trace with the IPG of the main procedure. The occurrence of a new timing trace is tracked in the boolean variable *newtrace*.

The TP scans every tuple (i, j) in the trace file (lines 89-129), and undertakes one of two actions depending on the value of *newtrace* as follows:

- If $newtrace =$ **true** then we know that the next token is not the start of a new trace (line 91)[6]. The first ipoint u will thus be a master entry ipoint in the IPG of the main procedure satisfying $t_u = i$ (line 92). As the TP must determine

[6]In theory, the next token could be the start of a new trace provided the program contained a single ipoint u. In this case, there would be a single IPG I for the procedure which causes u to trigger. However, if u was not in a loop then I would only contain ghost edges, and there would be no need for trace parsing. Otherwise, there would be a unique trace edge $u \rightarrow u \in E(I)$, i.e. an iteration edge, and moreover, $u \in \overleftarrow{M}$; we disallow this occurrence from the assumptions above.

```
    Input: $traces, \{I_{p_1}, I_{p_2}, \ldots, I_{p_n}\}, \{T_L^{C_{p_1}}, T_L^{C_{p_2}}, \ldots, T_L^{C_{p_n}}\}$
87  $(f,g,s) := (r,r,\lambda)$
88  $newtrace :=$ true
89  foreach $(i,j) \in traces$ do
90      if $newtrace$ then
91          $newtrace :=$ false
92          $u := succ(s)$ with $t_u = i$
93          $j' := j$
94      else
95          $v := succ(u)$ with $t_v = i$
96          Commit $j - j'$ to $wcet^{(f,g,s)}(u \to v)$
97          Add one to $b_{max}^{(f,g,s)}(u \to v)$
98          if $u \to v$ is an iteration edge then
99              foreach IPG loop $L_h^I$ satisfying $u \to v \in IE(L_h^I)$ do
100                 Add one to $b_{rel}^{(f,g,s)}(h)$
101         if $u$ and $v$ in different loops and $u \to v$ is an iteration or a loop-exit edge then
102             if $u$ is internal vertex then
103                 $h := u$
104             else
105                 $h := parent_{T_L^{C_g}}(u)$
106             $y := lca_{T_L^{C_g}}(u,v)$
107             repeat
108                 Commit $b_{rel}^{(f,g,s)}(h)$
109                 $h := parent_{T_L^{C_g}}(h)$
110             until $h = y$
111         $j' := j$
112         if $v$ is inlined master entry ipoint then
113             Push(ContextStack, (f, g, s))
114             Push(ReturnStack, v)
115             Pick arbitrary $v' \in succ(v)$
116             $(f,g,s) := (g, P_v, \mathrm{B}(P(v \to v')))$
117             $u := succ(s)$ with $t_u = i$
118         else if $v$ is master exit ipoint then
119             foreach trace edge $u' \to v' \in E(I_f)$ do
120                 Commit $b_{max}^{(f,g,s)}(u' \to v')$
121             if $(f,g,s) = (r,r,\lambda)$ then
122                 $newtrace :=$ true
123             else
124                 $(f,g,s) :=$ Pop(ContextStack)
125                 $u :=$ Pop(ReturnStack)
126                 $v := succ(u)$ with $T_v = i$
127                 $u := v$
128         else
129             $u := v$
```

Figure 3.14. Algorithm to Parse Timing Traces to Extract WCET Data.

the WCET of each transition $u \rightarrow v$, we also record the time stamp of u in the variable j' (line 93).

- If $newtrace =$ **false** then we have to find the successor v of u satisfying $t_v = i$ (line 95). The TP thus commits the observed WCET $j - j'$ to the edge $u \rightarrow v$ in the current IPG (line 96) and increments the frequency bound of $u \rightarrow v$ in the current context (line 97).

  If $u \rightarrow v$ is an iteration edge then this indicates that an IPG loop has been iterated. Recall from Section 3.3.1 that $u \rightarrow v$ can belong to several iteration edge sets $IE(L^I_{h_1}), IE(L^I_{h_2}), \ldots, IE(L^I_{h_n})$, one for each CFG* header h_i whose loop it can iterate through. As we cannot distinguish the exact loop from the timing trace alone, we must conservatively increment the relative bound of *every* IPG loop L_{h_i} for which $u \rightarrow v \in IE(L^I_{h_i})$ (lines 99-100). In essence, the relative bound of the inner loop pollutes those of all outermost loops and generally leads to overestimation. However, it is important to stress that such overestimation can only occur when the instrumentation profile is not path reconstructible, otherwise every iteration of every CFG* is observable in a timing trace, i.e. $u \rightarrow v$ belongs to a single iteration edge set.

The next step is to determine which IPG loops have stopped iterating (lines 101-109). This is detected by the fact that u and v are in different loops (i.e. there is no ancestor-descendant between u and v in $T_L^{C_g}$) and that $u \rightarrow v$ is an iteration edge or a loop-exit edge. (Clearly, if $u \rightarrow v$ is just a loop-entry edge — and not a loop-exit edge as well — then the outer loops might not have stopped iterating so no relative bounds should be committed.)

Assuming these conditions are met, first observe that flow of control is currently contained within the IPG loop L^I_h, where $h = u$ if u is an internal vertex or $h = parent_{T_L^{C_g}}(u)$ otherwise (lines 102-105). Further note that, on traversing $u \rightarrow v$, flow of control gets redirected into the IPG loop L^I_y in which both u, v are contained — the header y of this loop is the least-common ancestor of u, v in $T_L^{C_g}$ (line 106). Thus, on making the transition from L_h to L_y, every IPG loop whose header is on the path $y\ (\xrightarrow{+}]\ h$ in $T_L^{C_g}$ has stopped iterating.

This completes the mapping of trace data onto the current transition, so the next task of the TP is to prepare for the next tuple in the trace. Thus the last time stamp observed becomes the current one (line 111) and the ipoint u is advanced to its next position (lines 112-129). There are three cases to handle:

Procedure Call: This is recognised by the fact that v is an inlined master entry ipoint (line 112). As we later wish to return to the current context and ipoint location once the procedure call has ended, the TP stores this information on respective stacks $ContextStack$ and $ReturnStack$ (lines 113-114).

The next step is to construct the elements of the new context from the properties of the IPG. Evidently, the current callee becomes the new caller and the identifier of the callee is stored in P_v, which was set during master ipoint inlining. In order to retrieve the call site, recall that every edge originating from v is a context edge whose path expression contains a unique basic block (i.e. the call site)[7]. As v might have multiple successors (there could be multiple master exit ipoints in the callee), we can choose an arbitrary successor of v as they all contain the same basic block. This completes the information required to switch to the new context (line 116).

As a call has occurred, the TP switches to the IPG of the callee and retrieves the appropriate master entry ipoint (line 117).

Procedure Return: This is recognised by the fact that v is a master exit ipoint, thus there are no more trace edges to be processed in the current context, and the frequency bounds of edges can be committed (lines 119-120).

If the current context is the dummy context then we have reached the end of program execution for the current test vector, and thus the next tuple in the trace file must be the beginning of a new trace (lines 121-122). Otherwise, the TP must unwind the stacks to return to the previous context (lines 124-125). Note that, because we push an inlined master entry ipoint onto $ReturnStack$ on a procedure call, we must now advance u to the inlined master exit ipoint in the caller IPG so that the TP is in the correct position (lines 126-127).

[7]Recall from Clarification 6 that we considered the computation of path expressions beyond the scope of the thesis. However, extracting the unique basic block of context edges is straightforward as the inlined master entry ipoint in the CFG* has the desired basic block as its unique successor.

Same Procedure: The final case is that v is triggered by the procedure of the current IPG, and therefore, the next transition will be a successor of v. Therefore, the TP advances u to the same location as v (line 129).

An Example

Let us illustrate the following two operations of the TP: how to switch between contexts; how to determine the relative bound of CFG* loops. The other tasks of the TP — namely, obtaining the WCETs and the frequency bounds of IPG edges — is trivial.

To demonstrate the switching between contexts, consider Figure 3.13. Note that the initial dummy context is $(\texttt{foo}, \texttt{foo}, \lambda)$, thus the TP starts with the IPG of `foo` in Figure 3.13(e). Let us assume that the timing trace has lead to the transition $14 \rightarrow 16$. As 16 is an inlined master entry ipoint from `bar`, a procedure call is detected. (Evidently, the caller is `foo` and the callee is `bar`.) The only successor of 16 is the inlined master exit ipoint 17, and because $\mathsf{B}(P(16 \rightarrow 17)) = \{b_5\}$, b_5 is the call site of the new context. Hence, $(\texttt{foo}, \texttt{foo}, \lambda)$ and 16 are pushed onto *ContextStack* and *ReturnStack*, respectively, and the TP switches to the new context of $(\texttt{foo}, \texttt{bar}, b_5)$. Then, we fetch the IPG of `bar` and move to the master entry ipoint with the current trace identifier, which is ipoint 20 in Figure 3.13(b).

When the TP reaches ipoint 23 in Figure 3.13(b), a procedure return is identified because 23 is a master exit ipoint. Thus, we return to calling context $(\texttt{foo}, \texttt{foo}, \lambda)$, which is popped from *ContextStack*. On return, we pop the inlined master entry ipoint 16 (in Figure 3.13(e)) from *ReturnStack* and advance to the inlined master exit ipoint successor of 16, which is ipoint 17.

To demonstrate the extraction of relative bounds for IPG loops, consider Figure 3.10 (since the IPGs in Figure 3.13 do not contain any loops). Observe that, in the IPG of Figure 3.10(b), 11 is the sole master entry ipoint and 12 is the sole master exit ipoint. Also recall that $10 \rightarrow 10$ is in both the iteration edge sets of $L^I_{b_4}$ and $L^I_{d_4}$.

Consider the following timing trace:

$$11 \rightarrow 7 \rightarrow 9 \xrightarrow{f_4} 9 \xrightarrow{f_4} 9 \xrightarrow{f_4} 9 \xrightarrow{b_4} 7 \xrightarrow{b_4} 10 \xrightarrow{b_4,d_4} 10 \xrightarrow{b_4,d_4} 10 \xrightarrow{b_4,d_4} 10 \rightarrow 12$$

We have annotated each iteration edge transition $u \rightarrow v$ with the IPG loop header for which $u \to v$ is an iteration edge. How this timing trace is processed can be summarised by the following steps:

- On encountering the three $9 \to 9$ transitions, the TP increments the relative bound for $L^I_{f_4}$, and $b_{rel}(f_4)$ is currently 3.
- On the next transition $9 \to 7$, two actions are taken. First, the relative bound of $L^I_{b_4}$ is set to 1. Second, because 9 and 10 are in different loops and $9 \to 10$ is an iteration edge, we know a transition has been made from an inner loop to an outer loop, thus we have to commit some relative bounds. Observe that $lca_{T^C_L}(9,10) = b_4$ and that f_4 is the only header on the path $b_4\ (\xrightarrow{+}]\ f_4$. Therefore, $b_{rel}(f_4) = 3$ is committed to the database.
- On encountering the three $10 \to 10$ transitions, we cannot determine through which IPG loop execution has iterated, and the TP increments the bound for both $L^I_{b_4}$ and $L^I_{d_4}$. Thus, $b_{rel}(b_4)$ is currently 4 and $b_{rel}(d_4)$ is currently 3. Observe that this pollutes the bound of $L^I_{b_4}$ if $10 \to 10$ only iterated through $L^I_{d_4}$; however, as $L^I_{b_4}$ could have iterated, we are forced to be conservative and assume that the bound on both loops should be incremented.
- The final transition $10 \to 12$ is a loop-exit edge. Observe that $lca_{T^C_L}(10,12) = \mathsf{s}_4$. We have to commit the relative bound for the IPG loops $L^I_{b_4}$ and $L^I_{d_4}$ as the headers of these loops appear on the path $\mathsf{s}_4\ (\xrightarrow{+}]\ d_4$. Therefore, $b_{rel}(b_4) = 4$ and $b_{rel}(d_4) = 3$ are committed to the database.

3.5.3 Calculation Engine

Once all timing traces have been parsed, our HMB framework is in a position to generate a WCET estimate. We combine the timing data retrieved from trace parsing using the set of IPGs and the call graph.

Since our trace parser associates timing data with a particular context, the user (or our tool) can decide whether to *unify* or *expand* contexts off-line. Context unification, e.g. consider each call to g from f in its worst case, brings about a more conservative WCET estimate, which is typically desired if confidence in testing is lacking. For example, covering all trace edges of each IPG in every context places a greater burden on the test framework, and consequently, we might wish to unify the contexts where coverage is thin. On the other hand, full context expansion provides the most accurate WCET estimate. Observe that, if the TP only obtains timing data on a per procedure basis, any context expansion in the calculation engine would require a (costly) re-parsing of timing traces.

Our calculation engine operates in the following manner. We order the contexts in the call graph in reverse topological order, which is possible because we assume there is no recursion. For each context (f,g,s), we first elicit whether it should be considered expanded or unified: if expanded, we use the timing data applicable to (f,g,s) retrieved by the trace parser; if unified, we take the maximum value observed for each call from f to g. The WCET of (f,g,s) is then calculated using the IPG I_g. Every ghost edge in I_g is assigned the value of 0 as it is not observed in a trace, by definition. If g calls other procedures then we map the calculated WCETs of contexts (g,g',s') onto the appropriate context edges in I_g. Our HMB framework then uses a tree-based approach or the IPET to calculate the WCET of I_g, the specifics of such are detailed in Chapters 4 and 5, but for now we assume either one is in place. We also defer a discussion of how to incorporate loop bounds into the calculation until then. However, it suffices to say that frequency bounds are optional (they tighten the estimate), whereas relative bounds are compulsory. Finally, the WCET estimate for the program has been generated once we reach the dummy context (r,r,λ).

An Example

To illustrate the operation of the calculation engine, consider the example of Figure 3.13 and, for the IPG in Figure 3.13(b), assume the TP has extracted the following timing data in the given contexts:

$(\texttt{foo},\texttt{bar},b_5)$	
IPG Edge	WCET
$20 \rightarrow 21$	100
$21 \rightarrow 23$	10
$20 \rightarrow 22$	30
$22 \rightarrow 23$	65

$(\texttt{foo},\texttt{bar},c_5)$	
IPG Edge	WCET
$20 \rightarrow 21$	75
$21 \rightarrow 23$	30
$20 \rightarrow 22$	20
$22 \rightarrow 23$	25

If these contexts are expanded in the calculation then the WCET of $(\texttt{foo},\texttt{bar},b_5) =$

110 and the WCET of $(\texttt{foo}, \texttt{bar}, c_5) = 95$. Thus, when calculating the WCET of the context $(\texttt{foo}, \texttt{foo}, \lambda)$, 110 is mapped onto the edge $16 \rightarrow 17$ and 95 is mapped onto the edge $18 \rightarrow 19$ (c.f. Figure 3.13(e)).

Now assume that we want to consider every call to `bar` to be unified. Following is the timing data associated with its IPG edges:

IPG Edge	WCET
$20 \rightarrow 21$	100
$21 \rightarrow 23$	30
$20 \rightarrow 22$	30
$22 \rightarrow 23$	65

In this case, the WCET of `bar` is 130, which is clearly more conservative than either of the expanded contexts $(\texttt{foo}, \texttt{bar}, b_5)$ or $(\texttt{foo}, \texttt{bar}, c_5)$. The pessimism also propagates into the caller as 130 must be mapped onto both edges $16 \rightarrow 17$ and $18 \rightarrow 19$ because we only have one WCET for `bar`.

3.5.4 Discussion

There are a few issues with our interprocedural analysis that warrant further discussion.

The first is that the TP cannot yet handle timing trace *blackouts*, which arise when the port or debugger cannot keep pace with the rate at which ipoints are emitted, and data are essentially lost. A second issue with the TP is that all IPGs must be DFAs. This is particularly undesirable as the number of unique trace identifiers available is often restricted by practical considerations. For instance, if the data are written to an I/O port (using a logic analyser) then this number is determined by how many pins are available. Although an easy workaround is to initiate multiple writes, such a solution increases the overhead of ipoints (the *probe effect*) and forces the tracing mechanism to become non-atomic, which could create problems in multi-threaded applications.

Our mechanisms to handle contexts could be generalised further by considering loops and recursion. For example, the first procedure call in a loop often has a larger WCET than the other iterations due to cache misses. However, also observe that considering more precise forms of contexts places a greater burden on the test framework as we must have a sufficient amount of confidence in the timing data extracted. Clearly, this does not only mean stressing the WCETs between ipoint transitions but also forcing loops to iterate as much as possible so that the bounds on iteration edges are accurate. As noted in Chapter 2, this is now being addressed by the concept of WCET coverage [17].

3.6 Summary

A HMB WCET analysis framework must be retargettable to different instrumentation profiles without causing any additional pessimism in the calculation stage. In this respect, existing program models, such as the CFG and the AST, prove unsuitable as basic blocks are the atomic units of computation. As a HMB framework does not model the processor, the only way to extract the WCETs of basic blocks is to parse timing traces generated during testing. However, when sparse instrumentation profiles are employed, basic blocks rarely execute in isolation on any single ipoint transition, thus their WCETs can become grossly inflated. This dominoes into the calculation stage since any analysis is tied to the accuracy of its input parameters: overestimation ensues.

The core contribution of this chapter was a novel program model — the IPG — that forcibly changes the unit of computation to the transitions among ipoints. The timing data retrieved from trace parsing can therefore be mapped directly onto an IPG, avoiding any overhead associated with basic blocks as a consequence. This chapter also made the following additional contributions:

- We showed how to construct and analyse structural properties of the IPG using the CFG*, an intermediate form similar to the CFG. In particular, we demonstrated how to use the structural connection between a *reducible* CFG* and an IPG in order to identify *arbitrary* irreducible loops in the IPG.
- We demonstrated how to use the structural relation between CFG* and IPG loops to transfer loop bounds obtained through static analysis [47, 51] onto the IPG. Alternatively, we also presented a way to extract loop bounds from tim-

ing traces using properties of the IPG. Although the accuracy of such bounds is mainly tied to the amount of testing undertaken (bounds can be underestimated), we also showed that how the program is instrumented can be equally as influential (bounds can be overestimated).

- We showed how to use the IPG in the context of interprocedural analysis. In particular, we described how to virtually inline a subset of the ipoints from each callee into the caller (but none of the transitions), resulting in one IPG per procedure. This inlining mechanism gives the trace parser visibility to procedure calls and returns and facilitates retrieval of timing data on a per context basis as opposed to a per procedure basis. This provides the user with the flexibility to determine the precision of the analysis as the calculation engine can unify or expand contexts.

At this stage we are currently not in a position to evaluate the IPG as a program model within WCET analysis. This is because the IPG is a mere static representation and does not provide a means *per se* to compute WCET estimates. The remainder of this thesis thus addresses this issue by exploring tree-based calculations on the IPG (in Chapter 4) and remodelling of the IPET towards the IPG (in Chapter 5). Once these calculation techniques are in place, we evaluate our entire framework, including the effect of context expansion and unification, in Chapter 6.

4 Tree-Based Calculations on the IPG

In the previous chapter, we introduced the **Instrumentation Point Graph** (IPG) as a novel program model for WCET analysis in a **Hybrid Measurement-Based** (HMB) framework, which is constructed from a **CFG*** - an augmented **Control Flow Graph** (CFG). Instead of modelling the transitions between basic blocks, the IPG structures the transitions among **instrumentation points** (ipoints) and therefore forcibly changes the unit of computation.

This chapter considers how to perform *tree-based* calculations on the IPG. One advantage of tree-based approaches is that they incur low computational complexity. Moreover, in addition to being able to combine pure integral values, tree-based methods can combine *execution time profiles* derived from measurements to produce *probabilistic* WCET estimates [15, 16], in contrast to path-based approaches and the **Implicit Path Enumeration Technique** (IPET).

The first contribution of this chapter is how to transform the IPG into a novel hierarchical form, the **Itree**. This new representation is needed because the **Abstract Syntax Tree** (AST) is constructed from program source and hence it does not suitably model the transitions among ipoints at the intermediate code level. The Itree models standard control structures found in high-level languages (selection, sequence, and iteration), thus it is conceptually similar to the AST. The second contribution of this chapter is the **timing schema** [88, 94] associated with the Itree, which are formulae that compute a WCET estimate from the Itree structure.

The remainder of the chapter is structured in the following manner. Section 4.1 first recalls some properties of the IPG that were presented in Chapter 3, before Section 4.2 presents the formal properties of the Itree. Next, Section 4.3 considers how to transform an acyclic IPG into an Itree. For these purposes, we introduce the notion of **acyclic reducibility**, which basically decides if acyclic regions in the IPG can be decomposed into a hierarchy of **Single-Entry, Single-Exit** (SESE), **Multiple-Entry, Single-Exit** (MESE), and **Single-Entry, Multiple - Exit** (SEME) regions. We show how to identify these regions using the pre-dominance, post-dominance, pre-dominance frontier, and post-dominance frontier relations. In essence, branch and merge vertices are classified as being either reducible or irreducible, analogously to how headers of loops are categorised in standard (cyclic) reducibility [3, 83]. We show how to use these properties to prevent redundant traversals of acyclic IPGs (whilst building a hierarchical representation) and how this results in a *forest of Itrees*, an **Iforest**.

Section 4.4 then gives a complete description of the algorithm that creates an Iforest from the IPG. Section 4.5 gives the timing schema, which then enables us to evaluate our tree-based calculation engine against the AST in Section 4.6. We compare the core contributions of the chapter with previous work in Section 4.7, before finally concluding the chapter in Section 4.8.

4.1 Preliminaries

An IPG $I = \langle \mathsf{I}, E_I, \mathsf{s}, \mathsf{t} \rangle$ has a set of cycle-inducing edges called **iteration edges**. In Chapter 3, we showed that it is generally very difficult to identify iteration edges using state-of-the-art loop detection techniques due to the arbitrariness of IPG **irreducibility** (see Definition 7 in Chapter 3). We instead assumed the CFG* $C = \langle V_C = \mathsf{B} \cup \mathsf{I}, E_C, \mathsf{s}, \mathsf{t} \rangle$ from which I is constructed to be reducible and used the **Loop-Nesting Tree** (LNT) (see Definition 8 in Chapter 3) T_L^C of C to determine which edge insertions into I cause cycles.

A structural connection therefore exists between a CFG* loop, denoted L_h, and an IPG loop, denoted L_h^I (denoted in this way to reflect the structural connection to L_h). We defined a function $\Omega : \mathscr{L} \to \mathscr{L}^I$, where $\mathscr{L}$ is the set of *instrumented* CFG* loops and $\mathscr{L}^I$ is the set of IPG loops. Every IPG loop L_h^I has an **iteration edge set**, denoted $IE(L_h^I)$. In the following, it will be useful to partition $IE(L_h^I)$ into two disjoint subsets: the set of **non-self-loop** iteration edges $\overline{SL(L_h^I)} = \{u \to v | u \to v \in IE(L_h^I) \wedge u \neq v\}$ and the set of **self-loop** iteration edges $SL(L_h^I) = \{u \rightarrow v | u \rightarrow v \in IE(L_h^I) \wedge u = v\}$.

Detection of **loop-entry** and **loop-exit** edges for each IPG loop L_h^I is achieved by performing least-common ancestor queries on T_L^C with the set of *forward* edges in E_I.

4.2 Itree Representation

For an IPG $I = \langle \mathsf{I}, E_I, \mathsf{s}, \mathsf{t} \rangle$, each *leaf* of the Itree is an edge $u \rightarrow v \in E_I$ because this is the unit of computation of the IPG. Following are the properties of *internal vertices*:

- A **loop** vertex, denoted `LOOP`, is a rooted binary tree that models a subset of the IPG transitions in a CFG* loop L_h. The properties of its children satisfy either of the following:

  – The right tree models an iteration edge $u \rightarrow v \in SL(L_h^I)$ and the left tree is empty.

  – The right tree models the set of iteration edges $\overline{SL(L_h^I)}$ and the left tree models the ipoint transitions created from the induced **Directed Acyclic Graph** (DAG) L_h'.

  As a loop vertex L only has two children, we shall denote its left tree with the notation $succ_{left}(L)$ and its right tree with the notation $succ_{right}(L)$.

- An **alternative** vertex, denoted `ALT`, is a rooted n-ary tree that either models:

  1. The paths $p_1, p_2, \ldots, p_n$ from a branch vertex b to $ipost(b)$.
  2. The paths $p_1, p_2, \ldots, p_n$ from a branch vertex b to a merge vertex $m \neq ipost(b)$ such that m post-dominates all edges on all p_i.
  3. The selection between the execution of the iteration edges in $\overline{SL(L_h^I)}$ (provided $|\overline{SL(L_h^I)}| > 1$) that are identified for a CFG* loop L_h.
  4. The selection between `LOOP` vertices $L_1, L_2, \ldots, L_n$, in which each L_i models an iteration edge $u \rightarrow v \in SL(L_h^I)$, and the (unique) `LOOP` vertex modelling the set $\overline{SL(L_h^I)}$ (provided $\overline{SL(L_h^I)} \neq \emptyset$ and $SL(L_h^I) \neq \emptyset$). The iteration edges $SL(L_h^I)$ and $\overline{SL(L_h^I)}$ are those identified for a CFG* loop L_h.

5. The selection between the loop-entry edges $u \rightarrow v_1$, $u \rightarrow v_2$, …, $u \rightarrow v_i$ into an IPG loop or the selection between the loop-exit edges $w \rightarrow y_1$, $w \rightarrow y_2$, …, $w \rightarrow y_i$ out of an IPG loop.

- A **sequence** vertex, denoted `SEQ`, is an ordered rooted n-ary tree that either models:

  1. The set of paths $p_1 : b \rightarrow s_1 \overset{+}{\rightarrow} m$, $p_2 : b \rightarrow s_2 \overset{+}{\rightarrow} m$, …, $p_n : b \rightarrow s_n \overset{+}{\rightarrow} m$ such that b is branch vertex, $s_i \in succ(b)$ and m is the first merge vertex on all p_i satisfying $m \triangleleft u \rightarrow v$, where $u \rightarrow v$ is an edge on some p_i.
  2. The set of paths $u \overset{+}{\rightarrow} v$ such that u is *not* a branch vertex and v satisfies $v \triangleleft u$.

4.3 Acyclic Reducibility

A hierarchical representation of a flow graph represents every execution path by decomposing the flow graph into a number of regions which are notionally contained in others. When the flow graph is cyclic, *reducibility* assesses whether the flow graph can be decomposed into a hierarchy of cyclic regions and, if so, which vertices “control” entry into a particular cyclic region – these vertices are called *reducible* (loop) headers. Which vertices are controlled by such headers, and which headers control other headers, can be represented in a LNT.

However, reducibility does not assess which vertices in the flow graph “control” acyclic regions (mainly because compiler optimisations are performed almost exclusively within loops, i.e. only cyclic properties are of interest). This is undesirable since, with such information, we could collapse the region from where control is assumed to the vertex at which control is relinquished into an **abstract** vertex, and proceed in this manner until a unique abstract vertex remained (note the deliberate similarity to how loops are collapsed in constructing the LNT). Furthermore, the reduction process naturally organises such regions hierarchically.

Let us demonstrate this notion of control through a running example, which is depicted in Figure 4.1. Observe in this IPG that $\mathtt{s}_1$ controls access to all edges in the region until $\mathtt{t}_1$ and that 4 controls access to the set of edges $\{4 \rightarrow \mathtt{t}_1, 4 \rightarrow 5, 5 \rightarrow \mathtt{t}_1\}$.

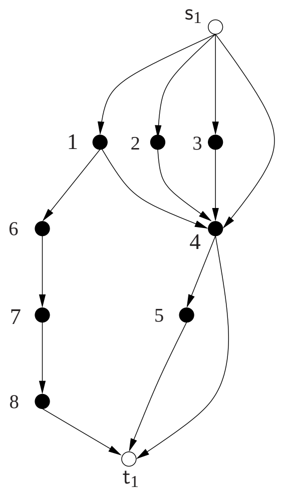


Figure 4.1. Example IPG To Demonstrate Acyclic Irreducibility.

Also note that both vertices are branch vertices and that the point at which control is relinquished (or merged) is their respective immediate post-dominator. (Clearly, if a vertex is not a branch vertex then it must always control access to its unique successor edge.)

However, the important vertex to note is 1 which, although controlling access to the set of edges $\{1 \rightarrow 6, 6 \rightarrow 7, 7 \rightarrow 8, 8 \rightarrow \mathtt{t}_1, 1 \rightarrow 4\}$, does *not* control access to the set of edges from 4 onwards. In this case, because $ipost(1) = \mathtt{t}_1$, control is relinquished in a region *before* flow of control reaches its immediate post-dominator, specifically once the edge $1 \rightarrow 4$ has been traversed.

These observations are important (to the problem of building an Itree from an acyclic IPG) because, as described in the previous section, the Itree must model the set of alternative paths $p_1, p_2, \ldots, p_n$ from a branch vertex b to the first vertex common to all p_i, which, by definition, is $ipost(b)$. (This must be modelled so that the timing schema is able to select the longest path amongst all p_i.) In particular, because b does not control all edges on all p_i up to $ipost(b)$, the transitions on these uncontrolled regions will have to be duplicated. Besides causing redundant graph traversals, this will inflate the Itree space requirements quadratically.

For instance, consider Figure 4.2, which depicts the Itree resulting from the IPG in Figure 4.1. Note, in particular, how the two emboldened alternative subtrees are duplicates, which represent the paths from 4 to $ipost(4)$.

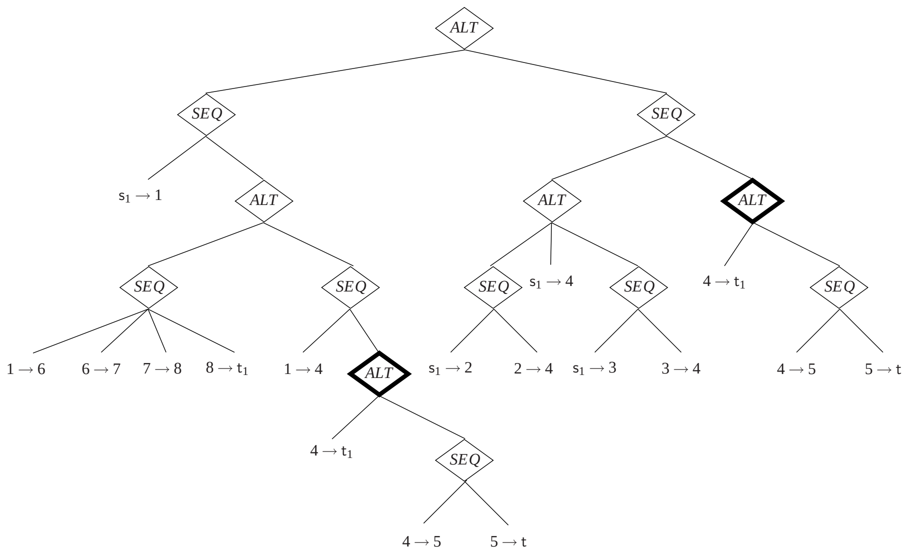


Figure 4.2. The Itree of the IPG from Figure 4.1.

Vertices that control acyclic regions can be identified with the **dominance relations**, since they determine which vertices and edges must precede or proceed the execution of other vertices and edges across *all* paths. The specifics of how these can be detected is established in Theorem 2 below, the proof of which requires the following simple lemma:

Lemma 2. *For a flow graph G, a vertex u pre-dominates all edges with source v if, and only if, u pre-dominates v.*

Proof. Immediate from the definition of the pre-dominance relation. □

Theorem 2. *Let $G = \langle V_G, E_G, \mathsf{s}, \mathsf{t}\rangle$ be an acyclic flow graph, b be a branch vertex in G, and G_b be the induced subgraph of G such that all vertices in G_b are reachable from b and satisfy $ipost(b) \trianglelefteq u$. Then, b pre-dominates all edges in G that belong to $E(G_b)$ if, and only if, $b = ipre(ipost(b))$ or $|DF_{pre}(b)| = 1$[1].*

Proof. $\Rightarrow$ From Lemma 2, it suffices to show that the source of every edge in G_b is pre-dominated by b.

There are two cases to consider:

[1]Note that the dominance relations can be reversed to make them apply to merge vertices instead.

$b = ipre(ipost(b))$**:** Suppose to the contrary that $u \to v \in E(G_b)$ is an edge such that $b \not\triangleright u \to v$. This implies that there is a path $\mathsf{s} \xrightarrow{+} u \to v \xrightarrow{*} ipost(b)$ that avoids b; evidently, $b \not\triangleright ipost(b)$ and hence $b \neq ipre(ipost(b))$.

$|DF_{pre}(b)| = 1$**:** We will show that $ipost(b)$ is the unique element in $DF_{pre}(b)$ and, because $ipost(b)$ is *not* the source of an edge in G_b, the implication holds.

Let b' the unique element in $DF_{pre}(b)$. We show that $ipost(b) \neq b'$ implies $|DF_{pre}(b)| > 1$. Since $ipost(b) \neq b'$, there must be a path $b \xrightarrow{+} u \xrightarrow{+} \mathsf{t}$ that avoids b' (because b' does not post-dominate b). There are two cases to consider:

- $b \not\triangleright u$. Either $b' \notin DF_{pre}(b)$, and the proposition is contradicted; or $u \in DF_{pre}(b)$, and therefore $|DF_{pre}(b)| > 1$.
- $b \triangleright u$. Let z denote the vertex at which the paths $u \xrightarrow{+} \mathsf{t}$ and $b' \xrightarrow{+} \mathsf{t}$ converge. Either $z \in DF_{pre}(b)$, or there is another path $u \xrightarrow{+} z' \xrightarrow{+} z$ such that $b \not\triangleright z'$, and therefore $z' \in DF_{pre}(b)$. In either case, $|DF_{pre}(b)| > 1$.

$\Leftarrow$ There are two cases to consider:

$b \triangleright ipost(b)$: We want to show the stronger condition that $b = ipre(ipost(b))$. It suffices to show that there is no vertex $b' \in V(G_b)$ such that $b \triangleright b' \triangleright ipost(b)$. Observe that b' must be a branch vertex (the immediate pre-dominator of a merge vertex is always a branch vertex) and that $ipost(b) \triangleleft b'$, otherwise b could avoid $ipost(b)$ via b'. There are two cases to consider:

- Every path from b to $ipost(b)$ includes b'. But then $b' = ipost(b)$.
- There is a path from b to $ipost(b)$ that avoids b'. Therefore, because b' is also reachable from b, b' cannot pre-dominate $ipost(b)$.

$b \not\triangleright ipost(b)$: As b pre-dominates all edges in G that belong to G_b, by Lemma 2, it also pre-dominates all vertices in G that are sources of edges in G_b. Therefore, the only vertex it cannot pre-dominate is the destination of an edge. The only such destination must be $ipost(b)$, i.e. $DF_{pre}(b) = \{ipost(b)\}$, and the claim holds.

□

This theorem is important because it enables ordered vertex pairs (b,m) in a flow graph G to be identified, where each b is a branch vertex and each m is a merge vertex. Each pair can be categorised as one of the following:

- If $ipost(b) = m$ and $ipre(m) = b$ then (b,m) is a SESE region because flow of control always enters this region at b and always leaves at m.
- If $ipost(b) = m$ and $|DF_{pre}(b)| = 1$ then (b,m) is a SEME region because flow of control always enters this region at b and always exits through one of the edges $u \longrightarrow m$ where $b \triangleright u \longrightarrow m$.
- If $ipre(m) = b$ and $|DF_{post}(m)| = 1$ then (b,m) is a MESE region because flow of control always enters this region through one of the edges $b \longrightarrow u$ where $m \triangleleft b \longrightarrow u$ and always exits through m.

When all branch vertices in G control entry into a SESE or SEME region and all merge vertices in G control exit out of a SESE or MESE region, we say that G is *acyclic reducible* because these regions can be organised hierarchically, similar to how reducible loops are organised in cyclic flow graphs. Formally:

Definition 13. *An acyclic flow graph $G = \langle V_G, E_G, \mathtt{s}, \mathtt{t}\rangle$ is **acyclic reducible** provided:*

- *All its branch vertices b satisfy $ipre(ipost(b)) = b$ or $|DF_{pre}(b)| = 1$, and*
- *All its merge vertices m satisfy $ipost(ipre(m)) = m$ or $|DF_{post}(m)| = 1$.*

In an acyclic reducible IPG, therefore, we can build the Itrees of its constituent SESE, SEME, and MESE regions without redundant graph traversal and without unnecessary duplication of subtrees.

On the other hand, if G is acyclic irreducible, this indicates that there must be branch vertices b satisfying $|DF_{pre}(b)| > 1$ and merge vertices m satisfying $|DF_{post}(m)| > 1$: such vertices are termed *acyclic-irreducible vertices*. The reason for this terminology is that, for an acyclic-irreducible branch vertex b, some parts in the region from b to $ipost(b)$ can be reached through paths that do not pass through b, and thus this region cannot be collapsed until its outermost enclosing region is collapsed. (Similar reasoning can be applied to a merge vertex except that the region begins at $ipre(m)$ and ends at m.)

In an acyclic irreducible IPG, therefore, there are irreducible merge vertices at which flow of control merges from distinct branch vertices that results in subtree duplication and hence a re-traversal of a subgraph in the IPG.

For instance, let us return to our running example in Figure 4.1, which contains branch vertices $\{\mathtt{s}_1, 1, 4\}$ and merge vertices $\{\mathtt{t}_1, 4\}$. Note that:

- $ipost(\mathtt{s}_1) = ipre(\mathtt{t}_1)$, thus $(\mathtt{s}_1, \mathtt{t}_1)$ is a SESE region.
- $ipre(ipost(4)) \neq 4$ but $DF_{pre}(4) = \{\mathtt{t}_1\}$, thus $(4, \mathtt{t}_1)$ is a SEME region.
- $ipre(ipost(1)) \neq 1$ and $DF_{pre}(1) = \{4, \mathtt{t}_1\}$, thus 1 is an acyclic-irreducible branch.
- $ipost(ipre(4)) \neq 4$ and $DF_{post}(4) = \{1, \mathtt{s}_1\}$, thus 4 is an acyclic-irreducible merge.

From these observations, we may infer that the IPG is acyclic irreducible. Also note that, because 4 is an acyclic-irreducible merge, $DF_{post}(4) = \{1, \mathtt{s}_1\}$ implies that there will be *two* traversals of the subgraph from 4 to $\mathtt{t}_1$: one whilst building the subtree modelling the paths from the branch vertex $\mathtt{s}_1$ (which merge at $\mathtt{t}_1 = ipost(\mathtt{s}_1)$); the other whilst building the subtree modelling the paths from the branch vertex 1 (which also merge at $\mathtt{t}_1 = ipost(1)$). This explains why the emboldened alternative subtrees in Figure 4.2 are duplicates.

Duplication can be avoided, however, by instead building a forest of Itrees (an Iforest) and having conceptual pointers from one subtree to the root of a particular Itree. As long as the calculation engine is aware of the order in which trees in the Iforest need to be processed, production of an Iforest is not an issue and hence does not affect the WCET calculation.

For example, Figure 4.3 depicts the hierarchical representation that we wish to move towards for the IPG in Figure 4.1. This is similar to the Itree of Figure 4.2, the key difference being that there is now only one emboldened alternative vertex which is a tree root. Furthermore, there are now two pointers to this subtree, which are indicated by the leaves $\overrightarrow{T_4}$. The calculation engine must therefore generate the WCET estimate of the Itree on the left (before the Itree on the right) and then propagate its value to the leaves that are pointer references.

In order to build this representation, however, it is not sufficient to merely detect

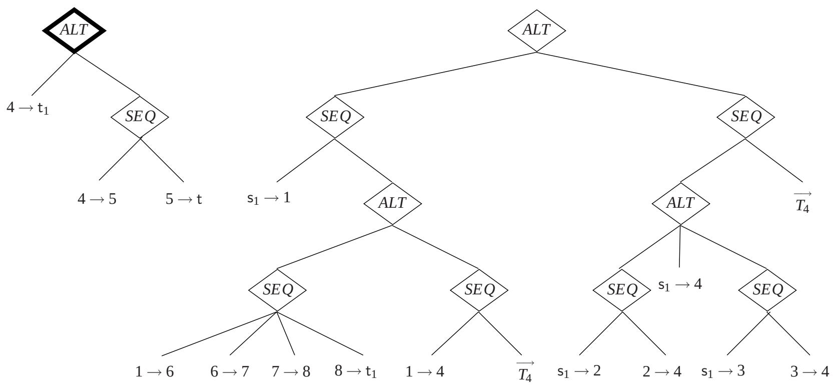


Figure 4.3. The Iforest of the IPG from Figure 4.1.

the merge vertices that are acyclic irreducible — we also need to know the vertex at which to terminate Itree construction. Following are the key observations to identify these additional vertices:

- Because the subtree constructed from an acyclic-irreducible merge vertex m will be duplicated for each branch vertex $b \in DF_{post}(m)$, and because we must model the paths from b to $ipost(b)$, duplication must halt at another merge vertex m'. In particular, m' will either satisfy $ipost(b) = m'$ or $ipost(b) \triangleleft m'$, i.e. it has to be enclosed in the region $(b, ipost(b))$.
- m cannot pre-dominate m' because, if m did pre-dominate m', this would imply that m also pre-dominates $ipre(m')$. Therefore, m would control the SESE region $(ipre(m'), m')$ and, consequently, m' cannot be a merge vertex at which distinct branch vertices in $DF_{post}(m)$ merge.

This suggests the following sequence of steps in order to build the Iforest of an acyclic IPG I:

- For each acyclic-irreducible merge vertex m, the nearest merge vertex m' satisfying both $m \not\trianglerighteq m'$ and $m' \triangleleft m$ is identified. We call m', denoted $m' = imerge(m)$, the *stopping vertex* of the acyclic-irreducible merge vertex m.
- The vertices of I are traversed in reverse topological order. For each acyclic-irreducible merge vertex m, we build an Itree from m to its stopping vertex. For each branch vertex b, we construct the `ALT` vertex modelling the paths from b to $ipost(b)$. However, during this construction, we do not construct the transitions

from any acyclic-irreducible merge vertex m; rather, we add a pointer to the subtree constructed from m and "jump" to its stopping vertex. This mechanism continues until $imerge(m) = ipost(b)$, thus avoiding traversal of the IPG edges in these regions.

In our running example, observe that $\mathtt{t}_1 = imerge(4)$. Traversing this IPG in reverse topological order, 4 is the first (and only) acyclic-irreducible merge vertex encountered; hence the algorithm outlined above will construct the Itree of the region $(4, \mathtt{t}_1)$. The first branch vertex encountered (except for 4) is 1. When building its `ALT` vertex, we know that flow of control will arrive at vertex 4 because it is on a subset of the paths from 1 to $\mathtt{t}_1$. When analysing this vertex, the algorithm will instead add a pointer to the previously computed Itree (for the region $(4, \mathtt{t}_1)$) and will then jump to $\mathtt{t}_1$ because $\mathtt{t}_1 = imerge(4)$. The construction of the `ALT` vertex (modelling the subset of paths on which vertex 4 is encountered) will then halt because $\mathtt{t}_1 = ipost(1)$.

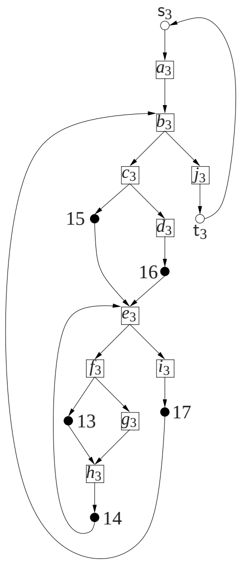
4.4 The Algorithm

This section presents the algorithm that constructs an Iforest from an IPG $I = \langle \mathsf{I}, E_I, \mathsf{s}, \mathsf{t} \rangle$. It is assumed that the CFG* $C = \langle V_C = \mathsf{B} \cup \mathsf{I}, E_C \cup \{\mathsf{t} \rightarrow \mathsf{s}\}, \mathsf{s}, \mathsf{t} \rangle$ from which I was built is reducible. This restriction is needed because the algorithm requires the LNT T_L^C of C, and as noted in Chapter 3, there is no consistent view of the properties of a LNT in the presence of irreducibility. Also note that the presence of the edge $\mathsf{t} \rightarrow \mathsf{s}$ simplifies our discussion since all vertices are then enclosed in a loop, and because I is constructed from C, this property also holds in I; therefore, *every* edge in I will be rooted under a `LOOP` vertex.

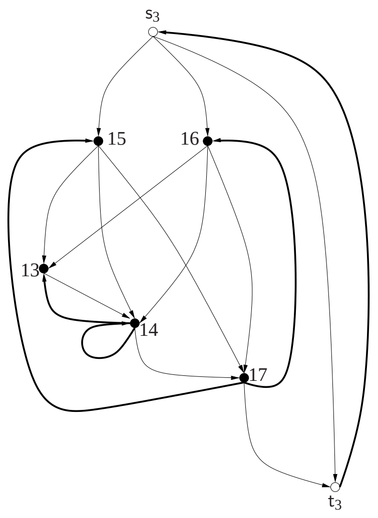
It is important to emphasise that the algorithm can handle IPGs that are generated from CFG*s with multiple-exit loops (commonly associated with `break` statements) and loops with multiple tails (commonly associated with `continue` statements).

Due to the complex nature of the algorithm, we will illustrate its operation with a running example depicted in Figure 4.4, which shows the three data structures required. Note that the CFG* has three non-trivial loops (L_{b_3}, L_{e_3}, and L_{s_3}) whose nesting relationship is shown in the LNT. With respect to the IPG, the following properties

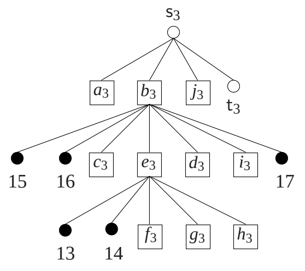
Figure 4.4. Example to Demonstrate Itree Construction Algorithm.



(a) The CFG.*



(b) Resultant IPG.



(c) The LNT of the CFG.*

are of note:

- For the IPG loop $L^I_{e_3}$, $IE(L^I_{e_3}) = \{14 \to 14, 14 \to 13\}$, its loop-entry edges are $\{15 \to 13, 15 \to 14, 16 \to 13, 16 \to 14\}$ and its loop-exit edges are $\{14 \to 17\}$.
- For the IPG loop $L^I_{b_3}$, $IE(L^I_{b_3}) = \{17 \to 15, 17 \to 16\}$, its loop-entry edges are $\{\mathsf{s}_3 \to 15, \mathsf{s}_3 \to 16\}$ and its loop-exit edges are $\{17 \to \mathsf{t}_3\}$.
- For the IPG loop $L^I_{\mathsf{s}_3}$, $IE_{\mathsf{s}_3} = \{\mathsf{t}_3 \to \mathsf{s}_3\}$ but it has neither loop-entry edges nor loop-exit edges as it encloses the entire IPG.

The pseudo-code of the Iforest algorithm is split across Figures 4.5, 4.8, 4.9, 4.10, 4.12, and 4.14. Before we embark on a description of each of these, some remarks about the conventions employed need clarifying. First, we try and avoid unnecessary parameter passing between procedures by assuming the parameters passed to the main procedure (`Build-Iforest`) are globally visible. This allows us to concentrate on the parameters to procedures that change between calls. Second, for sets S and T, we use $S \cup_= T$ as a short form of $S := S \cup T$.

4.4.1 `Build-Iforest`

This procedure takes I, C, and T_L^C as parameters to initiate construction and is the only procedure that analyses cyclic properties of I; that is, all other procedures operate exclusively on (induced) acyclic subgraphs of I.

The basic operation of the algorithm is akin to that of the IPG construction algorithm (see Chapter 3) in the way that it performs an inside-out decomposition of I, i.e. from inner loops outwards to outer loops, using T_L^C (lines 1-37)[2]. During this decomposition, the Itree representation of each IPG loop L_h^I is constructed as follows.

First of all, recall from Section 4.1 that we partition the iteration edge set of L_h^I into non-self-loop edges and self-loop edges. The reason for this is that we may consider L_h^I as several sub-loops (one for each iteration edge) that effectively contribute to the execution time of one larger loop, namely the CFG* loop L_h (from which L_h^I is constructed). Therefore, the Itree representation should piece together these sub-loops in such a way that, during the calculation, the sub-loop with the greatest WCET estimate is selected. That is, we want an `ALT` vertex whose children are the Itree representations of these sub-loops.

Clearly, by definition, any self-loop iteration edge cannot execute any of the forward edges in L_h^I. Thus the algorithm first constructs individual `LOOP` vertices for all $u \rightarrow u \in SL(L_h^I)$ (lines 3-12). The left child of each `LOOP` vertex is empty, as required, and the right child models the unique transition $u \rightarrow u$. If there are multiple self-loop iteration edges then all of these `LOOP` vertices are rooted under an `ALT` vertex which effectively models the entire self-loop region of L_h^I (line 5).

Every non-self-loop iteration edge $u \rightarrow v \in \overline{SL(L_h^I)}$ is modelled in the right tree of a *single* `LOOP` vertex whose left tree models all forward edges in L_h^I (lines 13-21). If $|\overline{SL(L_h^I)}| > 1$ then the right tree is an `ALT` vertex so that the calculation engine can

[2]The algorithm must use the LNT of C to perform the decomposition because we do not specifically create the LNT of I.

```
Input: $I, C, T_L^C$
for $i := height(T_L^C) - 1$ downto $0$ by $-1$ do
    foreach $internal\ vertex\ h\ with\ level(h) = i\ and\ IE(L_h^I) \neq \emptyset$ do
        if $SL(L_h^I) \neq \emptyset$ then
            if $|SL(L_h^I)| > 1$ then
                $l_s :=$ ALT
                foreach $u \rightarrow v \in SL(L_h^I)$ do
                    $l_c :=$ LOOP
                    $succ_{right}(l_c) := u \rightarrow v$
                    $succ(l_s) \cup_= \{l_c\}$
            else
                $l_s :=$ LOOP
                $succ_{right}(l_s) \cup_= \{u \rightarrow v \in SL(L_h^I)\}$
        if $\overline{SL(L_h^I)} \neq \emptyset$ then
            $l_m :=$ LOOP
            if $|\overline{SL(L_h^I)}| > 1$ then
                $a :=$ ALT
                foreach $u \rightarrow v \in \overline{SL(L_h^I)}$ do
                    $succ(a) \cup_= \{u \rightarrow v\}$
                $succ_{right}(l_m) := a$
            else
                $succ_{right}(l_m) := u \rightarrow v \in \overline{SL(L_h^I)}$
            Induce loop DAG $L_h^{I\prime}$ for $L_h^I$
            body := BuildAcyclic($s'$,$t'$)
            $succ_{left}(l_m) := body$
        if $SL(L_h^I) \neq \emptyset\ and\ \overline{SL(L_h^I)} \neq \emptyset$ then
            if $|SL(L_h^I)| = 1$ then
                $r :=$ ALT
                $succ(r) \cup_= \{l_s, l_m\}$
            else
                $r := l_s$
                $succ(r) \cup_= \{l_m\}$
            $loopRoots(h) := r$
        else if $SL(L_h^I) \neq \emptyset$ then
            $loopRoots(h) := l_s$
        else if $\overline{SL(L_h^I)} \neq \emptyset$ then
            $loopRoots(h) := l_m$
        Append $loopRoots(h)$ onto $calculationOrder$
```

Figure 4.5. `Build-Iforest`: Main procedure to Build Iforest from IPG choose the longest iteration back into the acyclic region (we shall observe in Section 4.6 that this is one cause of overestimation). Otherwise, the right tree models the unique iteration edge in $\overline{SL(L_h^I)}$.

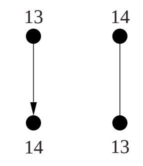
In order to construct the left tree of this `LOOP` vertex, we must first induce the loop DAG $L_h^{I\prime} = \langle V_h, E_h, \mathsf{s}', \mathsf{t}' \rangle$ of L_h^I as follows (line 22)[3]. V_h consists of the set of ipoints $\{u | parent_{T_L^C}(u) = h \vee u = h\}$ and the set of vertices $\{v | parent_{T_L^C}(v) = h \wedge v \text{ is internal vertex in } T_L^C\}$. Each internal vertex is termed an **abstract vertex** because it represents all vertices u of an inner loop $L_{h'}^I$. The edges E_h consist only of *forward* edges (i.e. there are no iteration edges) and are added as follows. Let $u \rightarrow v \in E_I$ be an edge such that $u = h$ or either u or v is a child of h in T_L^C. Then, if u is the source (respectively destination) of a loop-exit (respectively loop-entry) edge from an inner loop $L_{h'}^I$, the abstract vertex representing $L_{h'}^I$ becomes the source (respectively destination) of the edge in E_h; otherwise, u itself is the source (respectively destination).

The vertices s', t' represent the unique entry and exit vertices, respectively, of $L_h^{I\prime}$. These are needed because the dominator tree construction algorithms [4, 66] always assume their existence. If $L_h^{I\prime}$ has *multiple* vertices u for which $|pred(u)| = 0$ or $|succ(u)| = 0$, we insert additional **ghost ipoints** (see Section 3.1) to V_h and link the vertices without predecessors or without successors to these ghost ipoints.

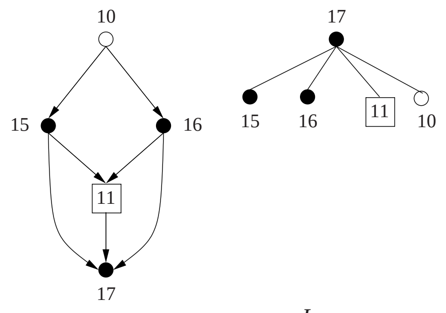
The loop DAGs generated for our running example are depicted in Figure 4.6 together with their respective post-dominator trees (on the right of each sub-figure). In Figure 4.6(a), only one IPG edge $13 \rightarrow 14$ has been added because the other edges, namely $14 \rightarrow 14$ and $14 \rightarrow 13$, are iteration edges. In Figure 4.6(b), vertex 11 is an abstract vertex representing the IPG transitions in the IPG loop $L_{e_3}^I$. Observe that IPG edges $15 \rightarrow 13$, $15 \rightarrow 14$, $16 \rightarrow 13$, and $16 \rightarrow 14$ have 11 as their destination since they are all loop-entry edges into the collapsed loop. Also note that we have added the ghost ipoint 10 because both $15, 16$ are entries into this loop and therefore have no predecessors in the loop DAG. Figure 4.6(c) depicts the last loop DAG, where vertex 12 represents the collapsed IPG loop $L_{b_3}^I$.

[3]Recall that the IPG construction algorithm used reverse post-orderings of vertices in loops to avoid explicit inducement of each loop DAG. In contrast, `Build-Iforest` requires each loop DAG to be induced in order to construct its respective pre-dominator and post-dominator trees so that, for example, acyclic-irreducible merge vertices can be detected.

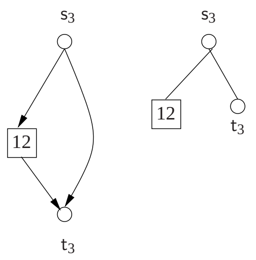
Figure 4.6. Induced Loop DAGs and their Post-Dominator Trees for IPG in Figure 4.4



(a) For loop $L^I_{e_3}$.



(b) For loop $L^I_{b_3}$.



(c) For loop $L^I_{s_3}$.

Once the loop DAG has been generated, the Itree modelling this acyclic region is constructed via a call to `BuildAcyclic` with parameters s', t' (line 23); the returned Itree is subsequently rooted under the left tree of the `LOOP` vertex (line 24).

The next step of `Build-Iforest` pieces together the respective Itrees constructed to model the non-self-loop iteration edge region (a `LOOP` vertex) and the self-loop iteration edge region (either an `ALT` vertex or a `LOOP` vertex). There are three cases to handle:

1. $SL(L^I_h) \neq \emptyset$ and $\overline{SL(L^I_h)} \neq \emptyset$: When $|SL(L^I_h)| > 1$, the Itree modelling the self-loop region of L^I_h is already an `ALT` vertex, thus the `LOOP` vertex modelling $\overline{SL(L^I_h)}$ is added as a child (lines 30-31). Otherwise, we insert an additional `ALT` vertex and add the respective `LOOP` vertices as children (lines 27-28).

2. $SL(L^I_h) \neq \emptyset$: The Itree modelling $SL(L^I_h)$ becomes the Itree representation for the IPG loop L^I_h (line 34).

3. $\overline{SL(L^I_h)} \neq \emptyset$: The Itree modelling $\overline{SL(L^I_h)}$ becomes the Itree representation for the IPG loop L^I_h (line 36).

The root of the Itree modelling the IPG loop L^I_h is stored in a temporary variable *loopRoots* for subsequent retrieval (because it will become a child of a subtree when building the Itree representation of its outer loop). We also append the root onto a list, *calculationOrder*, in case the abstract vertex representing L^I_h in its outer nesting-level is an irreducible-merge vertex and will therefore only have pointers to it (line 37).

Let us consider the (partial) state of the Itrees modelling the three cyclic regions of the IPG in Figure 4.4 before any of their acyclic regions have been constructed. This is depicted in Figure 4.7 in which Itrees are arranged from left to right according to the order that they are built. For the IPG loop $L^I_{e_3}$, there are two `LOOP` vertices rooted under an `ALT` vertex since $SL(L^I_{e_3}) = \{14 \rightarrow 14\}$ and $\overline{SL(L^I_{e_3})} = \{14 \rightarrow 13\}$. For the IPG loop $L^I_{b_3}$, there is a unique `LOOP` vertex whose right tree is an `ALT` vertex since $\overline{SL(L^I_{b_3})} = \{17 \rightarrow 15, 17 \rightarrow 16\}$. On the other hand, for the IPG loop $L^I_{\mathtt{s}_3}$, there is only one (dummy) iteration edge in the right tree because $\overline{SL(L^I_{\mathtt{s}_3})} = \{\mathtt{t} \rightarrow \mathtt{s}\}$. Also note that the parametrised call to `BuildAcyclic` is shown for each respective `LOOP` vertex.

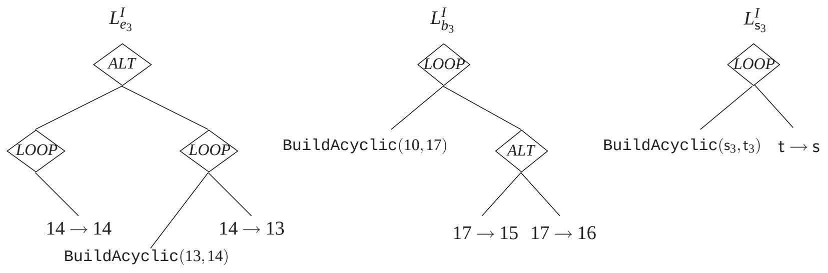


Figure 4.7. Itrees modelling Cyclic Regions in IPG of Figure 4.4

4.4.2 `Build-Acyclic`

This procedure builds the acyclic region induced for each IPG loop DAG $L^{I'}_h$; its pseudo-code is presented in Figure 4.8. It uses the algorithm described in the previous section to avoid redundant traversal of the IPG. Therefore, it processes vertices in $L^{I'}_h$ in reverse topological order (lines 39-43), and on encountering an acyclic-irreducible merge vertex y, the subtree modelling the region from y to it stopping vertex $imerge(y)$ is built (line 41). The subtree returned is subsequently stored in an array *acyclicRoots* (line 42), which is indexed by acyclic-irreducible merge vertices, so that we can later analyse the properties of the subtree constructed. Furthermore, the subtree is appended onto *calculationOrder* (line 43) to allow the calculation engine to generate the WCET estimate of this subtree before others that point to it. The final step of this procedure is to build the Itree modelling the region from s' to t' (line 44).

```
   Input: $u, v$
38 Sort vertices in $L_h^{I\,\prime}$ topologically
39 foreach $y$ in reverse topological order of $L_h^{I\,\prime}$ do
40     if $y$ is irreducible merge vertex then
41         $r :=$ Which-Sub-Tree($y$, $imerge(y)$)
42         $acyclicRoots(y) := r$
43         Append $r$ onto $calculationOrder$
44 return Which-Sub-Tree($u$, $v$)
```

Figure 4.8. `Build-Acyclic`: Helper Procedure in Iforest Construction.

For the loop DAG $L_{e_3}^{I\,\prime}$ (c.f. Figure 4.6(a)), because there are no acyclic-irreducible merge vertices, only the call `Which-Sub-Tree`$(13,14)$ is made. On the other hand, for the loop DAG $L_{b_3}^{I\,\prime}$ (c.f. Figure 4.6(b)), the call `Which-Sub-Tree`$(11,17)$ precedes that of `Which-Sub-Tree`$(10,17)$ since 11 is an acyclic-irreducible merge vertex. Again, for the loop DAG $L_{\mathrm{s}_3}^{I\,\prime}$ (c.f. Figure 4.6(c)) there are no acyclic-irreducible merge vertices, thus only the call `Which-Sub-Tree`$(\mathrm{s}_3,\mathrm{t}_3)$ is made.

4.4.3 `Which-Sub-Tree`

This is a helper procedure which determines the type of subtree required in modelling the paths from a source vertex u to a target vertex v; its pseudo-code is presented in Figure 4.9. Specifically, this procedure is called from `BuildAcyclic` when u is an acyclic-irreducible merge vertex and $v = imerge(u)$ or when $u = \mathrm{s}'$ and $v = \mathrm{t}'$. Note, therefore, that v always post-dominates u.

When u only has a unique successor s, we need to first determine if $s = v = ipost(u)$. If this is satisfied, the Itree created by `Unique-Edge-Action` is returned (line 47), otherwise we have to build a `SEQ` tree from u to v (line 49).

On the other hand, u must be a branch vertex. If $v = ipost(u)$ then this means that all alternative paths from u will merge at v, i.e. we only need an `ALT` vertex (line 52).

```
   **Input**: $u, v$
45 **if** $|succ(u)| = 1$ **then**
46     **if** $ipost(u) = v$ **then**
47         **return** Unique-Edge-Action($u, v$)
48     **else**
49         **return** Build-SEQ-Root($u, v$)
50 **else**
51     **if** $ipost(u) = v$ **then**
52         **return** Build-ALT-Root($u$)
53     **else**
54         **return** Build-SEQ-Root($u, v$)
```

Figure 4.9. `Which-Sub-Tree`: Helper Procedure in Iforest Construction.

Otherwise, in addition to modelling the region from u to $ipost(u)$, we have to model the region from $ipost(u)$ to v, i.e. we need a `SEQ` vertex (line 54).

In our running example, the following calls were made from `BuildAcyclic`:

- `Which-Sub-Tree`$(13, 14)$: this returns `Unique-Edge-Action`$(13, 14)$.
- `Which-Sub-Tree`$(11, 17)$: this returns `Unique-Edge-Action`$(11, 17)$.
- `Which-Sub-Tree`$(10, 17)$: this returns `Build-ALT-Root`$(10, 17)$.
- `Which-Sub-Tree`$(\mathrm{s}_3, \mathrm{t}_3)$: this returns `Build-ALT-Root`$(\mathrm{s}_3, \mathrm{t}_3)$.

4.4.4 `Unique-Edge-Action`

This procedure builds the Itree for the edge $u \rightarrow v$ in the loop DAG $L_h^{I\,\prime}$; its pseudo-code is presented in Figure 4.10. Because $L_h^{I\,\prime}$ might contain abstract vertices, this procedure also reconstructs the actual IPG transitions (i.e. the loop-entry and loop-exit edges of E_I) that the dummy edges between abstract vertices represent.

There are four cases to handle:

1. If u and v are both abstract vertices then we have to model all the loop-exit edges of the IPG loop L_u^I that are also loop-entry edges of the IPG loop L_v^I. When there is only one such edge, we simply return that edge (line 58), otherwise we construct an `ALT` vertex modelling the selection between them (lines 60-63).
2. Similarly to the previous case, if u is an abstract vertex then we have to model all the loop-exit edges of the IPG loop L_u^I that have the destination v. Again, when

```
    Input: $u, v$
55  if $u$ and $v$ are abstract vertices then
56      $X = \{y \rightarrow z | y \in L^I_u \wedge z \in L^I_v\}$
57      if $|X| = 1$ then
58          return $y \rightarrow z \in X$
59      else
60          $a :=$ ALT
61          foreach $y \rightarrow z \in X$ do
62              $succ(a) \cup_= \{y \rightarrow z\}$
63          return $a$
64  else if $u$ is abstract vertex then
65      $X = \{y \rightarrow v | y \in L^I_u\}$
66      if $|X| = 1$ then
67          return $y \rightarrow v \in X$
68      else
69          $a :=$ ALT
70          foreach $y \rightarrow v \in X$ do
71              $succ(a) \cup_= \{y \rightarrow v\}$
72          return $a$
73  else if $v$ is abstract vertex then
74      $X = \{u \rightarrow y | y \in L^I_v\}$
75      if $|X| = 1$ then
76          return $u \rightarrow y \in X$
77      else
78          $a :=$ ALT
79          foreach $u \rightarrow y \in X$ do
80              $succ(a) \cup_= \{u \rightarrow y\}$
81          return $a$
82  else
83      return $u \rightarrow v$
```

Figure 4.10. `UniqueEdgeAction`: Helper Procedure in Iforest Construction.

there is only one such edge, we simply return that edge (line 67), otherwise we construct an `ALT` vertex modelling the selection between them (lines 69-72). For example, Figure 4.11 depicts the subtree created for the unique transition $s_3 \rightarrow 12$ (c.f. Figure 4.6(c)).

3. Analogous to the previous case, except that v is an abstract vertex and we instead have to model all the loop-exit edges of the IPG loop L^I_v that have the source u. Again, when there is only one such edge, we simply return that edge (line 76), otherwise we construct an `ALT` vertex modelling the selection between them (lines 78-81).

4. The edge $u \rightarrow v$ actually belongs to E_I and so it is simply returned (line 83).

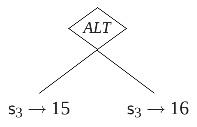


Figure 4.11. Itree modelling Loop-Entry Edges into IPG Loop $L^I_{b_3}$

Observe that the loop-exit subtree of an IPG loop can handle IPG transitions that were constructed from CFG* loops containing `break` statements. The reason is that the algorithm is not concerned with the source of the loop exit out of the CFG* loop; rather, it only needs to know that a transition $u \to v$ can occur from an ipoint u inside a CFG* loop L_h to an ipoint v inside an outer CFG* loop $L_{h'}$.

4.4.5 `Build-ALT-Root`

This procedure builds an `ALT` root modelling the paths in I from a given branch vertex u to $ipost(u)$; its pseudo-code is presented in Figure 4.12. It depends on the construction of an auxiliary data structure that we call the *Compressed Post-Dominator Tree* (CPDT) (line 70). This basically models the nearest merge node m that is the immediate post-dominator of a subset of $succ(u)$ and enables the `ALT` subtree to be pieced together during a bottom-up traversal.

The CPDT, denoted T^{com}_{post}, is iteratively computed by performing Least-Common Ancestor (LCA) queries on $T^{L^{I'}_h}_{post}$ with a query set Q. The queries in Q are all unordered pairs taken from a *dynamic* vertex set X; initially $X = succ(u)$. Each lca query returns a vertex w. If, for a vertex $x \in X$, w is the vertex with the largest height in $T^{L^{I'}_h}_{post}$ amongst all lca queries involving x, then w is set as the parent of x in T^{com}_{post}. Furthermore, if w was not initially a vertex in T^{com}_{post}, it is inserted into a temporary vertex set X'; once all queries in Q have been answered, X is assigned the elements of X' and the query process continues until there is a unique vertex in X'. The unique vertex remaining in X' must be $ipost(u)$ (otherwise u would have a different immediate post-dominator).

For example, let us consider how the CPDT is constructed for the branch vertex 10 in Figure 4.6(b). The initial vertex set is $X = \{15, 16\}$, and therefore, there is only one lca query to be answered, namely $lca(15, 16)$ in $T^{L^{I'}_h}_{post}$. The answer returned is

$17 = ipost(10)$. Thus the query process stops and the parent of both 15 and 16 in the CPDT is 17.

```
    Input: $u$
84  $a :=$ ALT
85  $T_{post}^{com} :=$ Build-Compressed-Tree($T_{post}^{L_h^{I'}}, succ(u)$)
86  for $i := height(T_{post}^{com})$ downto $1$ by $-1$ do
87      foreach $v$ with $level(v) = i$ do
88          $s :=$ Build-SEQ-Root($v, parent(v)$)
89          if $v$ is leaf then
90              $succ(s) \cup_= \{$Unique-Edge-Action($u, v$)$\}$
91          else
92              $a' := mergeRoots(v)$
93              if $v \in succ(u)$ then
94                  $succ(a') \cup_= \{$Unique-Edge-Action($u, v$)$\}$
95              $s \cup_= \{a'\}$
96              if $parent(v) = ipost(u)$ then
97                  $succ(a) \cup_= \{s\}$
98              else
99                  $a' :=$ ALT
100                 $succ(a') \cup_= \{s\}$
101                 $mergeRoots(parent(v)) := a'$
102 if $ipost(u) \in succ(u)$ then
103     $succ(a) \cup_= \{$Unique-Edge-Action($u, ipost(u)$)$\}$
104 return $a$
```

Figure 4.12. `Build-ALT-Root`: Helper Procedure in Iforest Construction.

The `ALT` vertex a is subsequently built by visiting the nodes of the CPDT level by level in a bottom-up fashion (lines 86-101). For each vertex v at level i, we first construct the subtree to model the paths from v to $parent(v)$ (line 88). Note that we explicitly call `Build-SEQ-Root` to carry out this construction (and not `Which-Sub-Tree`) because we know it has to be a `SEQ` vertex s; that is, the subtree must represent the transition $u \to v$ *and* the paths from v to $parent(v)$.

The properties of v within T_{post}^{com} are then examined. When v is a leaf, we append the subtree modelling the unique transition $u \to v$ to the `SEQ` vertex (line 90). Otherwise, v is an internal vertex, indicating that there are *alternative* paths from the children of v that merge at v. In this case, the algorithm will have constructed another `ALT` vertex a' (as discussed below) modelling these paths. (Observe that $v \neq ipost(u)$ because the level by level of traversal of T_{post}^{com} halts at level 1 and $ipost(u)$ is the root.) Therefore, a' is added to the `SEQ` root (line 95).

This completes the construction of the SEQ vertex s, so now the algorithm determines which ALT vertex s should be added as a child. If $parent(v) = ipost(u)$ then s should be added to the main ALT vertex a because we have reached the end of the traversal of the CPDT (line 97). Otherwise, as noted above, we need another ALT vertex a' to model the flow of control that merges before reaching $ipost(u)$; s is added as a child of a' in this case (line 100) and a' is stored in array *mergeRoots* indexed by $parent(v)$ so that we can retrieve it at the next level in the CPDT (line 101).

The final task of this procedure is to check whether there is a transition $u \rightarrow ipost(u)$ (line 102), in which case we also have to add its Itree model to the ALT vertex (line 103).

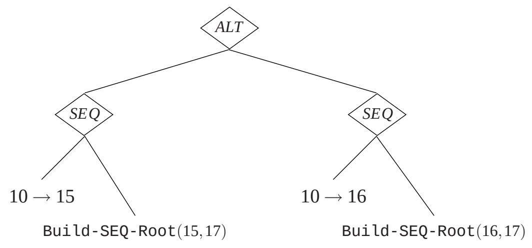


Figure 4.13. Itree modelling Paths from Branch Vertex 10 in IPG Loop $L^I_{b_3}$

For example, let us consider how the ALT vertex is constructed for the branch vertex 10 in Figure 4.6(b). There are only two edges to analyse in its CPDT: $17 \rightarrow 15$ and $17 \rightarrow 16$. Let us consider the edge $17 \rightarrow 15$ because the operations performed for $17 \rightarrow 16$ will be analogous. The paths $15 \xrightarrow{+} 17$ are constructed with the call Build-SEQ-Root$(15, 17)$, which will return a SEQ vertex having a single ALT child (because 15 is a branch vertex itself). The transition $10 \rightarrow 15$ is then added to the SEQ vertex by a call to Unique-Edge-Action: this will return the single edge $10 \rightarrow 15$ since neither of these vertices are abstract ones. The (partial) state of the Itree modelling this is shown in Figure 4.13.

4.4.6 Build-SEQ-Root

This procedure builds a SEQ root from a given source vertex u to a target vertex v; its pseudo-code is presented in Figure 4.14. Note that that v is guaranteed to post-dominate u because the places from which it is called — namely in Which-Sub-Tree (at lines 49 and 54) and `Build-ALT-Root` (at line 88) — always satisfy this property. The basic intuition of this procedure, therefore, is to append the subtrees on the path $u \xrightarrow{+} ipost(u) = v_1 \xrightarrow{+} ipost(v_1) = v_2 \xrightarrow{+} \ldots \xrightarrow{+} ipost(v_i) = v$ onto the `SEQ` root in that order.

```
    Input: $u, v$
105 $s :=$ SEQ
106 $x := u$
107 repeat
108     if $x$ is abstract vertex then
109         if $x$ is irreducible merge then
110             $succ(s) \cup_{=} \{\overrightarrow{loopRoots}(x)\}$
111         else
112             $succ(s) \cup_{=} \{loopRoots(x)\}$
113     if $x$ is irreducible merge then
114         if $acyclicRoots(x)$ is IPG edge then
115             $succ(s) \cup_{=} \{acyclicRoots(x)\}$
116         else
117             $succ(s) \cup_{=} \{\overrightarrow{acyclicRoots}(x)\}$
118         $y := imerge(x)$
119     else
120         $y := ipost(x)$
121         if $|succ(x)| > 1$ then
122             $succ(s) \cup_{=} \{$Build-ALT-Root($x$)$\}$
123         else
124             $succ(s) \cup_{=} \{$Unique-Edge-Action($x, y$)$\}$
125     $x := y$
126 until $y = v$
127 return $s$
```

Figure 4.14. `Build-SEQ-Root`: Helper Procedure in Iforest Construction.

This is done iteratively by successively updating two temporary vertices x and y that always satisfy $y \trianglelefteq v$. Initially, x is set to u (line 106) and iteration halts once $y = v$ (line 126).

On each iteration, we determine which subtree should be appended to the `SEQ` root by analysing the properties of x as follows:

- When x is an abstract vertex, this indicates that the path from u to v executes the IPG loop L^I_x. If x is an acyclic-irreducible merge vertex then we insert a pointer to the `LOOP` vertex modelling L^I_x (line 110). Otherwise, L^I_x controls a region in the current loop DAG (i.e. it is either a SESE or MESE region) and thus its `LOOP` vertex is retrieved from *loopRoots* and rooted under the `SEQ` vertex (line 112).

- When x is an acyclic-irreducible merge vertex, then we first inspect the properties of the subtree that was constructed in `BuildAcyclic` and stored in $acyclicRoots(x)$. If it is an IPG edge $u \rightarrow v$ then we add this to the `SEQ` vertex (line 115). The reason is that adding a pointer to a leaf unnecessarily increases the size of the Iforest as it merely adds a layer of indirection to the smallest possible Itree entity. Otherwise, we add a pointer to the subtree as required (line 117). We then advance y so that it jumps over the already modelled subgraph (line 118).
- When x is *not* an acyclic-irreducible merge vertex, y is updated to its immediate post-dominator (line 120). If x is a branch vertex then we append the `ALT` vertex returned by `Build-ALT-Root` onto the `SEQ` root (line 122); otherwise there is a unique transition $x \rightarrow y$ and the subtree returned by `Unique-Edge-Action` is instead appended (line 124).

At the end of the iteration, x is advanced to the position of y (line 125).

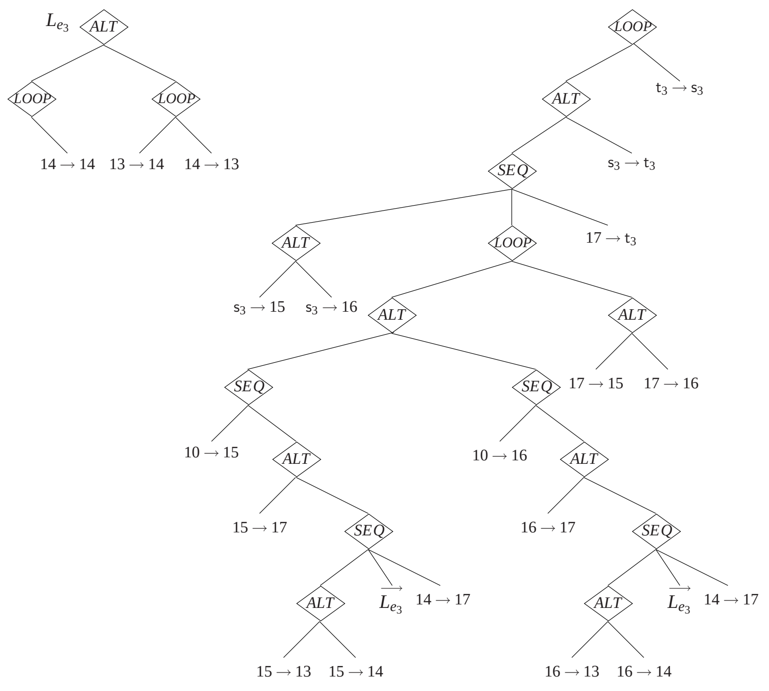


Figure 4.15. The Iforest of the IPG in Figure 4.4

The final Iforest generated for the IPG in Figure 4.4 is shown in Figure 4.15.

4.5 Iforest Calculations: Timing Schema and Calculation Order

An Iforest is just another program representation and therefore, like the IPG, it does not implicitly provide a mechanism from which a WCET calculation can be computed. In this section, we first discuss how to control the order of the computations on individual Itrees in the Iforest, before introducing the timing schema that drive the calculation over Itrees.

Controlling the calculation order of Itrees is necessary because of the creation of a forest of Itrees as opposed to a unique Itree. For this reason, when the timing schema traverses a particular Itree, the WCETs of any pointer references that it has must already have been computed. The algorithm described in the previous section constructed this ordering on the fly by appending the roots of Itrees onto the list *calculationOrder* as they are built. Therefore, all the calculation engine need do is process the Itrees in *calculationOrder* and then transfer the WCET estimate determined at the root to all pointer references.

Informally, the timing schema are a set of formulae that decide how to compute a WCET at each internal vertex in the hierarchical representation (either the AST or the Itree) from the WCETs of their children and the type of vertex. In this way, the calculation engine traverses the tree in a bottom-up fashion, and upon reaching the root, the WCET estimate is produced.

The original timing schema [88, 94] proposed for the AST, however, are not applicable to our tree-based calculation engine because the two representations have different properties; for example, our subtrees representing conditional constructs (i.e. the alternative vertex) are not the same as `if-then-else` constructs in the AST which always evaluate the conditional expression regardless of the path taken. We instead present the following set of timing schema to be used on each Itree in the Iforest, which are conceptually similar to the original formulae[4]:

$$\sigma(\texttt{SEQ}) = \sum_{s \in succ(\texttt{SEQ})} \sigma(s) \tag{4.1}$$

$$\sigma(\texttt{ALT}) = \max_{s_i \in succ(\texttt{ALT})} (\sigma(s_1), \sigma(s_2), \ldots, \sigma(s_n)) \tag{4.2}$$

$$\sigma(\texttt{LOOP}) = \sigma(succ_{left}(\texttt{LOOP})) * k + \sigma(succ_{right}(\texttt{LOOP})) * k \tag{4.3}$$

In these equations, σ denotes the WCET of a particular Itree construct. Equation (4.1) states that the WCET of a sequence vertex is the sum of the execution times of its children. Equation (4.2) states that the WCET of an alternative vertex is the child that has the greatest execution time. Equation (4.3) states that the WCET of a loop vertex is the sum of the contribution of the acyclic region modelled by the left tree and the iteration edge selected from the right tree, both of which are factored by the loop bound obtained for L_h^I (either through static analysis techniques or through trace parsing).

4.6 Evaluation

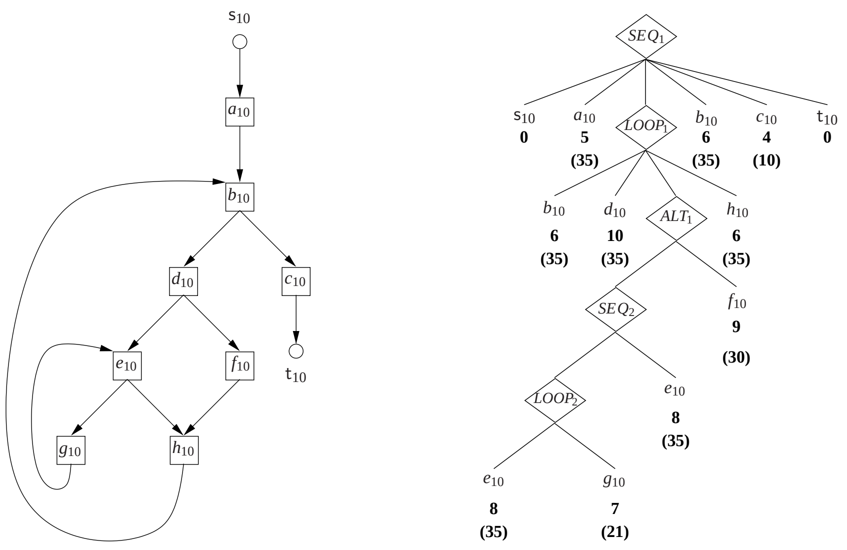
In this section, we evaluate the Itree by considering a synthetic example program instrumented according to two different instrumentation profiles. We also evaluate this program in Chapter 5 in order to contrast the precision of our tree-based calculation engine against that of the remodelled IPET.

Figure 4.16 depicts the program under consideration in both CFG and AST formats. Here, we will compare the Itree specifically with the AST since the latter is the only hierarchical representation we are aware of that drives tree-based calculations (in static analysis). Moreover, this allows us to pinpoint sources of pessimism in the Itree.

Let us assume that the WCETs of basic blocks have been extracted: either through

[4]Recall that the basic units of computation of an IPG are its edges. If an edge is a **trace edge** then its WCET is extracted during trace parsing as described in Section 3.5.2. If an edge is a **context edge** then its WCET is computed by the calculation engine as described in Section 3.5.3. Otherwise, if an edge is a **ghost edge** then it is assigned a WCET of 0.

Figure 4.16. Example Program.



(a) The CFG. *(b) The AST.*

building a processor model or from measurements. This information is directly below each basic block in the AST of Figure 4.16(b) whereby $\mathtt{s}_{10}$ and $\mathtt{t}_{10}$ have WCETs of 0 because they are ghost ipoints (the meaning of the WCETs in brackets will become clear shortly but suffice to say they are used for an additional comparison). Let us further assume that the loop bound of $L_{e_{10}}$ is 5 and that the loop bound of $L_{b_{10}}$ is 10.

A simple tree-based calculation on the AST, according to the timing schema originally proposed in [88], would proceed bottom-up in the following manner:

1. At $LOOP_2$: the WCET is $(8+7)*5 = 75$.
2. At SEQ_2: the WCET is $75+8 = 83$.
3. At ALT_1: the WCET is $max(83,9) = 83$.
4. At $LOOP_1$: the WCET is $(6+10+83+6)*10 = 1050$.
5. At SEQ_1: the WCET is $0+5+1050+6+4+0 = 1065$.

Observe that 1065 is an accurate WCET estimate with the timing information provided.

4.6.1 Instrumentation Profile One

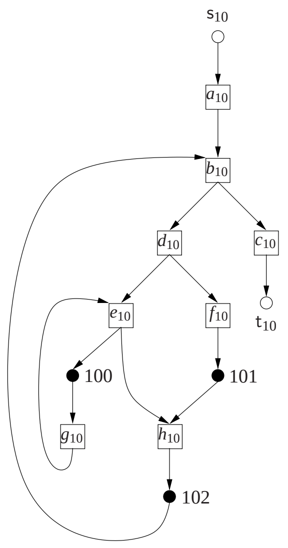
Figure 4.17 presents the first instrumentation profile from which we will evaluate the Itree. The CFG* in Figure 4.17(a) has been generated from the CFG in Figure 4.16(a), thus the loops have the same relative loop bounds. The Itree in Figure 4.17(c) represents edges in the IPG by the labels associated with them in Figure 4.17(b) — these edge labels will also be useful in Chapter 5 when building the Integer Linear Program from the IPG.

Observe in particular that the instrumentation profile is path reconstructible, which means that there is a unique path between ipoints in the CFG* (see Section 3.1). Let us therefore consider the WCET of each IPG edge $u \rightarrow v$ to be the sum of the WCET of each basic block in the set $\mathsf{B}(P(u \rightarrow v))$. For example, because $\mathsf{B}(P(\mathsf{s}_{10} \rightarrow 100)) = \{a_{10}, b_{10}, d_{10}, e_{10}\}$, the WCET of $e_{100} = 5+6+10+8 = 29$. (Recall from Clarification 1 that this thesis does not consider the probe effect, thus we assume ipoints incur zero overhead.) The WCET of every IPG transition derived in this way is shown underneath the corresponding edge label in Figure 4.17(c). Furthermore, let us assume that the loop bound supplied for each CFG* loop is transferred onto the corresponding IPG loop. (Recall from Section 3.3.2 that, for an iteration edge $u \rightarrow v$, the position of v with respect to the loop exits of L_h determines how the bound for L_h is transferred onto $u \rightarrow v$. In particular, see Equation (3.3).)

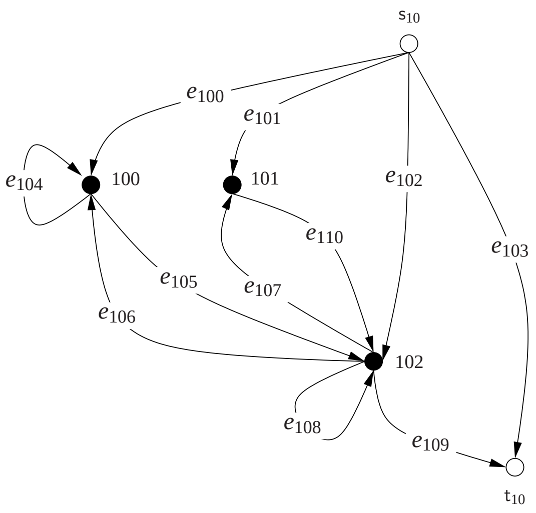
The timing schema presented in the previous section proceeds in a bottom-up manner on the Itree as follows:

1. At $LOOP_5$: the WCET is $15 * 4 = 60$.
2. At SEQ_4: the WCET is $60 + 21 = 81$.
3. At ALT_5: the WCET is $max(6, 81) = 81$.
4. At ALT_6: the WCET is $max(24, 25) = 25$.
5. At $LOOP_4$: the WCET is $(81 * 10) + (25 * 9) = 1035$.
6. At $LOOP_3$: the WCET is $30 * 9 = 270$.
7. At ALT_4: the WCET is $max(270, 1035) = 1035$.
8. At ALT_3: the WCET is $max(29, 30, 35) = 35$.

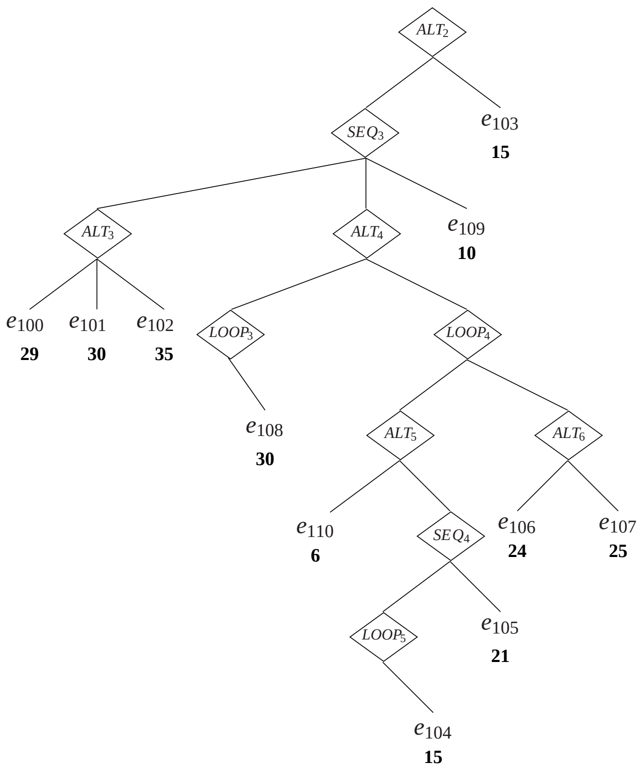
Figure 4.17. First Instrumentation Profile on Program from Figure 4.16 to Evaluate Itree.



(a) The CFG.*



(b) Resultant IPG.



(c) The Itree.

9. At SEQ_3: the WCET is $35 + 1035 + 10 = 1080$.
10. At ALT_2: the WCET is $max(1080, 15) = 1080$.

Because the WCET estimate on the AST was 1065, we may conclude that the WCET estimate derived from the Itree has been overestimated by 15 cycles. The underlying cause of the overestimation is the irreducibility in the outer IPG loop $L^I_{b_{10}}$. For this reason, the Itree models the entry transitions into that region, the acyclic transitions in that region, and the iteration edges of that region in distinct subtrees without relation to each other. This forces the calculation engine into producing a structurally infeasible path because it conservatively picks the subtrees with the longest execution times. In this case, the actual longest path through the outer IPG loop includes execution of the inner loop, which the calculation engine correctly chooses in the acyclic region modelled by ALT_5; however, this implies that the iteration edge $102 \rightarrow 100$ must also be selected, whereas the calculation engine chooses the iteration edge $102 \rightarrow 101$ at ALT_6, accounting for 9 cycles of overestimation. The other 6 cycles of overestimation come from ALT_3 because the chosen entry transition $\mathsf{s}_{10} \rightarrow 102$ should have been $\mathsf{s}_{10} \rightarrow 100$ given that all iterations of that loop pass through vertex 100.

However, the original motivation for introducing the IPG was not when the units of computation are basic blocks, but when timing traces are the only sources of such information. Let us therefore re-calculate a WCET estimate using the AST by instead extracting the WCETs of its basic blocks from the ipoint transitions (these values are shown in brackets in Figure 4.16(b)); this will serve to validate our original claim at the beginning of Chapter 3 that WCET estimates derived from static analysis program models become inflated due to sparse instrumentation.

The steps taken in the calculation on the AST are as follows:

1. At $LOOP_2$: the WCET is $(35 + 21) * 5 = 280$.
2. At SEQ_2: the WCET is $280 + 35 = 315$.
3. At ALT_1: the WCET is $max(315, 30) = 315$.
4. At $LOOP_1$: the WCET is $(35 + 35 + 315 + 35) * 10 = 4200$.
5. At SEQ_1: the WCET is $0 + 35 + 4200 + 35 + 10 + 0 = 4280$.

In comparison to the WCET estimate computed through the Itree, this WCET estimate is an overestimation of $\approx 300\%$. In this case, therefore, it is the IPG's unit of computation that prevents conservative analysis.

4.6.2 Instrumentation Profile Two

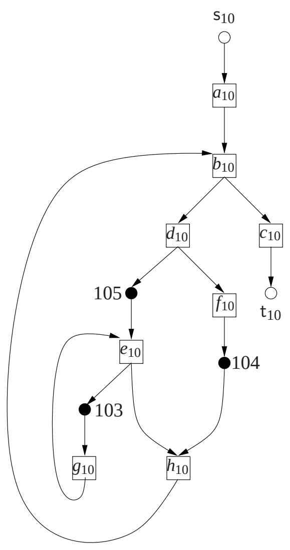
Figure 4.18 presents the second instrumentation profile from which we will evaluate the IPG. The CFG* in Figure 4.18(a) is similar to that in Figure 4.17(a): ipoint 103 is in the same location as ipoint 100 and ipoint 104 is in the same location as ipoint 101. However, the essential difference is that ipoint 102 has effectively been moved to the new location occupied by ipoint 105. We will observe that this minor modification to the instrumentation profile changes the Itree calculation considerably.

As for the previous instrumentation profile, let us again consider the WCET of each IPG edge $u \rightarrow v$ to be the sum of the WCET of each basic block in the set $\mathsf{B}(P(u \rightarrow v))$, as shown underneath the corresponding edge label in Figure 4.18(c), and that the loop bound supplied for each CFG* loop is transferred onto its corresponding IPG loop. Note that both Itrees contain the same timing information, thus we are able compare the WCET estimates that they each compute.

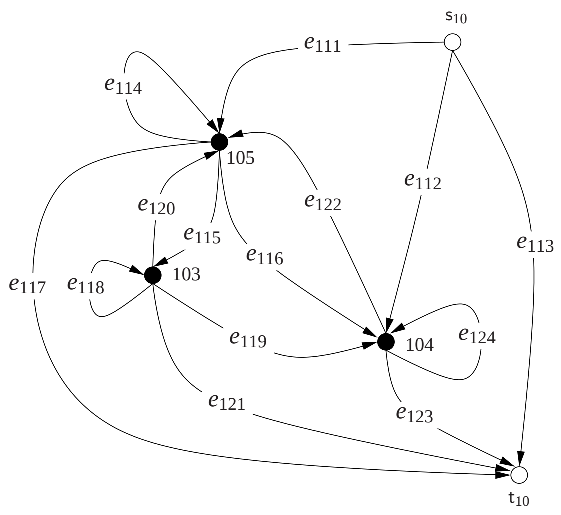
Following are the steps taken in this calculation:

1. At $LOOP_6$: the WCET is $15 * 4 = 60$.
2. At SEQ_6: the WCET is $60 + 8 = 68$.
3. At ALT_7: the WCET is $max(39, 46, 37, 22) = 46$.
4. At $LOOP_9$: the WCET is $(68 * 10) + (9 * 46) = 1094$.
5. At $LOOP_8$: the WCET is $9 * 31 = 279$.
6. At $LOOP_7$: the WCET is $9 * 30 = 270$.
7. At ALT_8: the WCET is $max(279, 1094, 270) = 1094$.
8. At ALT_{10}: the WCET is $max(21, 30) = 30$.
9. At ALT_9: the WCET is $max(24, 31, 16) = 31$.
10. At SEQ_5: the WCET is $30 + 1094 + 31 = 1155$.
11. At ALT_4: the WCET is $max(1155, 15) = 1155$.

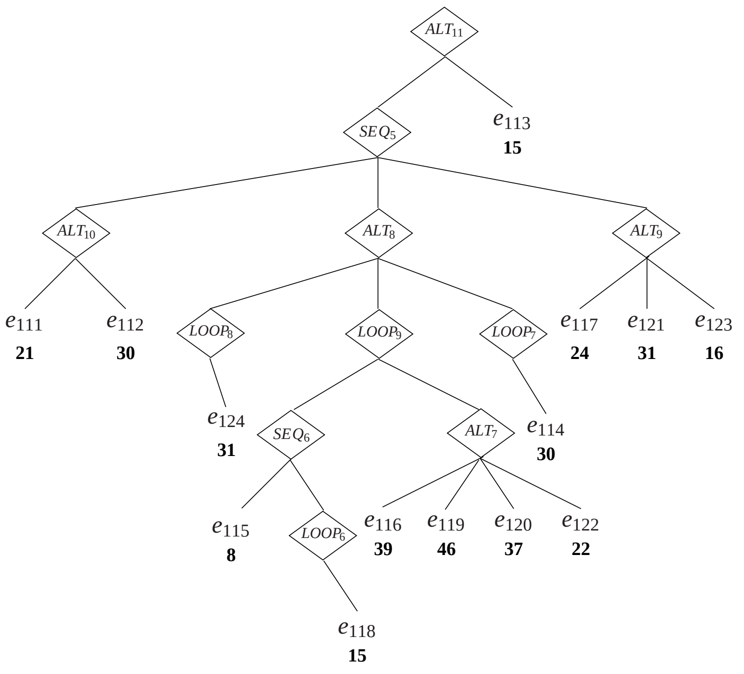
Figure 4.18. Second Instrumentation Profile on Program from Figure 4.16 to Evaluate Itree.



(a) The CFG.*



(b) Resultant IPG.



(c) The Itree.

In comparison to the Itree calculation performed in the first instrumentation profile, this is an $\approx 7\%$ overestimation. Again, the underlying cause of the overestimation is the irreducibility of the outer IPG loop, specifically because the $LOOP_9$ chooses iteration edge e_{119} to execute 9 times in sequence. Clearly, from the structural properties of the IPG, this is not a feasible execution path.

However, also observe that this WCET estimate is still an improvement on the WCET estimate (of 4280) that was computed from the AST in which the WCETs of basic blocks were extracted from timing traces.

4.7 Discussion and Related Work

A similar technique to break up a flow graph into a number of control regions has been introduced [55] in which SESE regions are identified and the hierarchical containment between them is captured in a program structure tree. Although at first this might appear analogous to the SESE, SEME, and MESE regions introduced above, our consideration of control regions differs in two fundamental aspects. First, the notion of acyclic reducibility clearly precludes cyclic flow graphs, but their SESE regions do not; however, the intention of our work was to specifically assess the hierarchical relationship between acyclic regions, and not to redefine (cyclic) reducibility. Second, the conditions that they require to identify a SESE region use the pre-dominance and post-dominance relations. In particular, a SESE region (a,b) is defined to be one satisfying both $a \trianglerighteq b$ and $b \trianglelefteq a$ (together with another property that is of no relevance here). Our conditions are more stringent in that we use the *immediate* pre-dominance and post-dominance relations in conjunction with the pre-dominance and post-dominance frontier relations.

The acyclic reducibility property can be considered equivalent to whether a flow graph is Two-Terminal Series-Parallel (TTSP) [119]. A TTSP graph is recursively defined as follows:

- A graph consisting of an entry and an exit vertex joined by a single edge is TTSP.
- *Series-Composition*: If G_1 and G_2 are TTSP graphs with terminals (s_1,t_1) and

(s_2, t_2), respectively, then the graph G with terminals (s_1, t_2) obtained by identifying vertices t_1 and s_2 is TTSP.

- *Parallel-Composition*: If G_1 and G_2 are TTSP graphs with terminals (s_1, t_1) and (s_2, t_2), respectively, then the graph G with terminals (s_1, t_1) obtained by identifying vertices s_1 with s_2 and t_1 with t_2 is TTSP.

In essence, a series-composed graph would be modelled by a `SEQ` vertex, whereas a parallel-composed graph would be modelled by an `ALT` vertex. Note in particular that these rules of construction would only result in acyclic-reducible graphs since two disjoint graphs are only joined at their respective entry and exit points, i.e. the single entry and single exit points are maintained.

TTSP graphs fall into a broader category of graph, those which have bounded *tree width* [115]. This is essentially an estimate of how close a graph is to a tree. The acyclic reducibility property can be similarly viewed, although we principally use it to identify vertices in the graph at which reduction into a tree would cause redundant graph traversal. To our knowledge, nobody has proposed how to use the properties of TTSP graphs or tree width with the same task in mind.

4.8 Summary

A popular calculation technique used in WCET analysis is based on a hierarchical program representation. In this chapter, we considered a tree-based approach to computing a WCET estimate from the IPG program model and made the following key contributions:

- We presented properties of a novel hierarchical form called the Itree, which models traditional high-level constructs such as sequence, selection, and iteration, except with respect to the IPG.
- We introduced the acyclic reducibility property, which is applicable to the class of (generalised) acyclic *flow graphs*, and essentially categorises branch and merge vertices either as acyclic-reducible or acyclic-irreducible. We showed that detection of acyclic-irreducible merge vertices can prevent redundant traversals of the IPG when building its Itree representation. Avoiding such traversals

instead produces a forest of Itrees (called the Iforest) in which Itrees essentially point to other Itrees.

- We presented an algorithm that decomposes the IPG into an Iforest. The algorithm is not restricted to a particular class of IPGs; that is, it handles arbitrary instrumentation even if that produces arbitrary irreducible regions in the IPG. The only restriction is that the CFG* (from which the IPG is constructed) is reducible, but the algorithm supports multiple exits out of loops and multiple loop-back edges commonly associated with `break` and `continue` statements, respectively.
- We introduced the timing schema that drives the calculation over individual Itrees in the Iforest. In addition, we showed how the calculation engine controls the order of computations over the Iforest so that the WCET estimate of a referenced Itree is available when required.
- Finally, we evaluated our tree-based calculation engine by considering an example program instrumented with two different (sparse) instrumentation profiles, from which we concluded the following:
  - When the atomic units of computation are derived from trace parsing, the Itree generates more accurate WCET estimates than the traditional AST-based calculation. This forms part of the validation of our initial conjecture at the beginning of Chapter 3, that the IPG program model should be chosen ahead of existing static analysis program models when instrumentation is sparse. This will be confirmed in Chapter 6 when evaluating a real application.
  - Our tree-based calculation is sensitive to the locations of ipoints because of the problem of irreducibility. This forces the Itree into making a trade-off between the space overhead incurred and the precision of the analysis.

5 IPET Calculations on IPGs

The previous chapter presented a tree-based calculation engine operating on the **Instrumentation Point Graph** (IPG), which first decomposes the IPG into a novel hierarchical form, the **Itree**. However, tree-based calculations suffer from the inability to incorporate extra path information, obtained normally through flow analysis techniques [46, 47, 99], into the calculation, resulting in a less precise analysis.

A more pronounced problem is the overestimation caused by arbitrary irreducible IPGs in hierarchical form. To avoid the complexity, it is possible to place **instrumentation points** (ipoints) at particular locations so that the resultant IPG is reducible. However, there are clear disadvantages of forcing instrumentation to particular locations. First, we must assume control of the instrumentation profile, which contravenes our goal of targeting systems that are instrumented as-is. This is especially true when a hardware debug interface, such as Nexus [1] or the Embedded Trace Macrocell (ETM) [33] is the chosen tracing mechanism. The second problem is that we might inadvertently increase the number of ipoints, potentially incurring a timing penalty, depending on how timing traces are extracted. Even more, it is sometimes not possible to increase the number of ipoints due to so-called *ipoint budgets*, particularly when analysing large systems. Lastly, if a user is selecting the location of ipoints, it is much more convenient to do this at the source level without the concern of how they affect structural properties of the underlying analysis engine.

The unequivocal outcome is that the calculation technique operating on the IPG must handle arbitrary irreducibility without causing undue pessimism. In this respect, the **Implicit Path Enumeration Technique** (IPET) has already been proven to be suitable [73, 95] because it does not explicitly model *global* flow graph structure. Rather, the IPET expresses the computation of the WCET on a given flow graph as an

Integer Linear Program (ILP), which has a known solution method. Therefore, the IPET essentially builds a constraint model relating *local* flow graph structure, i.e. at each vertex, to path-related information regarding loop bounds and infeasible paths. The key aspect of the IPET, therefore, is that irreducibility is not an issue as long as there is support from loop identification algorithms. In Chapter 3, we presented a mechanism to identify irreducible IPG loops, thus implying that the IPET can be reformulated so that it pertains to arbitrary IPGs — this is the main contribution of the chapter.

We begin, in Section 5.1, by recalling some properties of the IPG that were presented in Chapter 3. Section 5.2 then remodels the basic ILP created by the IPET towards the IPG and discusses how ILP solvers compute an *upper bound* on the WCET from this basic ILP. In Section 5.3, we discuss the **disconnected circulation problem** [95], which relates to any flow graph modelled by the ILP. For **Control Flow Graphs** (CFG), this can be solved through the addition of **relative capacity constraints** into the basic ILP: this extended model has been proven to return an exact WCET estimate [95]. Thus, we show how to model relative capacity constraints for the IPG. Next, Section 5.4 evaluates the WCET estimates returned by the IPET against those produced by our tree-based calculation engine, which was described in Chapter 4. We show that, in contrast to the tree-based approach, the IPET always returns an accurate WCET estimate as it determines a feasible execution path. Section 5.5 discusses the current limitations of our remodelled IPET, before we finally summarise the chapter in Section 5.6.

5.1 Preliminaries

An IPG $I = \langle \mathsf{I}, E_I, \mathsf{s}, \mathsf{t} \rangle$ has a set of cycle-inducing edges called **iteration edges**. In Chapter 3, we showed that it is generally very difficult to identify iteration edges using state-of-the-art loop detection techniques due to the arbitrariness of IPG **irreducibility** (see Definition 7 in Chapter 3). We instead assumed the CFG* $C = \langle V_C = \mathsf{B} \cup \mathsf{I}, E_C, \mathsf{s}, \mathsf{t} \rangle$ from which I is constructed to be reducible and used the **Loop-Nesting Tree** (LNT) (see Definition 8 in Chapter 3) T_L^C of C to determine which edge insertions into I cause cycles.

A structural connection therefore exists between a CFG* loop, denoted L_h, and an IPG loop, denoted L_h^I (denoted in this way to reflect the structural connection to L_h). We defined a function $\Omega : \mathscr{L} \rightarrow \mathscr{L}^I$, where $\mathscr{L}$ is the set of *instrumented* CFG* loops and $\mathscr{L}^I$ is the set of IPG loops. Every IPG loop L_h^I has an **iteration edge set**, denoted $IE(L_h^I)$.

An iteration edge $u \rightarrow v$ can belong to multiple iteration edge sets. In essence, $u \rightarrow v$ is a **multi-edge** such that the multiplicity of $u \rightarrow v$ is equal to the number of iteration edge sets to which it belongs. For this reason, this chapter considers an IPG $I = \langle \mathsf{I}, E_I \cup \{\mathsf{t} \rightarrow \mathsf{s}\}, \mathsf{s}, \mathsf{t} \rangle$ to be a **multi-digraph**. (The additional edge $\mathsf{t} \rightarrow \mathsf{s}$ is included to guarantee that the IPG is a maximal **Strongly Connected Component** (SCC), a condition necessary to represent execution paths through circulations.)

Detection of **loop-entry** and **loop-exit** edges for each IPG loop L_h^I is achieved by performing least-common ancestor queries on T_L^C with the set of *forward* edges in E_I.

There are two mechanisms by which edge frequencies in an IPG can be bounded. The first is by counting the maximum number of times each executes during **trace parsing**, resulting in a **frequency bound**. However, this potentially creates underestimation because it relies more heavily on good quality testing to exercise the maximum number of executions of each IPG edge.

The alternative is to obtain **relative bounds**, which constrain execution of IPG loops relative to the next outer nesting level. These can also be extracted by the trace parser using structural properties of I. However, we also allow for relative bounds supplied by static analysis techniques [47, 52] (or via interaction with a user). As such bounds are typically with respect to a CFG* loop L_h, these can be subsequently transferred onto the IPG loop L_h^I.

5.2 Basic ILP of the IPET

Both Puschner and Schedl [95] and Li and Malik [71, 72, 74, 73] have proposed almost analogous ways to reduce the WCET computation stage to an ILP. Each approach has its merits. On the one hand, Puschner-Schedl presented a rigorous theoretical model based on the **circulation problem** [57] — which itself is a generalisation of the **network flow problem** [30, 57] — before transforming this into an ILP. They were first to elaborate upon the disconnected circulation problem, which is especially problematic in the IPG due to irreducibility, as discussed in Section 5.3. For this reason, and to retain consistency, we attempt to use terminology and notation as presented by Puschner-Schedl, only deviating as necessary. On the other hand, Li-Malik presented the ILP in a more descriptive manner, and thus we often digress to their explanations. Furthermore, they discussed how to solve sets of **disjunctive constraints**, which are normally needed when extra path information is included in the calculation.

The basic ILP produced by the IPET consists of the following components:

- An objective function.
- Program structural constraints.
- Capacity constraints.
- Non-negativity constraints.

5.2.1 The Objective Function

In general terms, a solution to an ILP minimizes or maximizes an objective function composed of n decision variables, subject to a number of constraints that must be satisfied simultaneously. For the WCET problem, the objective function models the execution counts of IPG edges (i.e. these are the decision variables), which should be maximized to deliver the WCET estimate.

Following is the objective function:

$$Z = \sum_{u \to v \in E_I} wcet(u \to v) \cdot f(u \to v) \tag{5.1}$$

where Z is the returned WCET estimate, $wcet(u \to v)$ the WCET of an IPG edge (derived from trace parsing), and $f(u \to v)$ a non-negative execution count of an IPG edge that is set by the ILP solver.

Note that, for an iteration edge $u \to v$ with multiplicity n, the objective function contains n decision variables for $u \rightarrow v$, all of which have the same WCET.

5.2.2 Program Structural Constraints

These constraints represent the basic properties of program structure, intuitively stating that flow into a vertex equals flow out. These can be derived directly from the IPG and are stated formally as follows:

$$\forall\, v \in \mathsf{I} : \sum_{p \in pred(v)} f(p \rightarrow v) = \sum_{s \in succ(v)} f(v \rightarrow s) \tag{5.2}$$

Note here the subtle difference between the network flow and the circulation problems. The flow conservation property of the network problem is applicable to all vertices, *except* the dummy vertices s and t. However, the flow conservation property of the circulation problem is applicable to all vertices because of the existence of the edge $\mathsf{t} \rightarrow \mathsf{s}$.

5.2.3 Capacity Constraints

As Li-Malik affirmed, the maximisation of (5.1) is ∞ because each $f(u \rightarrow v)$ can be assigned the value ∞, i.e. each loop can iterate forever. Capacity constraints are therefore needed which bound both the minimum and maximum execution count of each edge. In the circulation problem, these capacity constraints are functions $b : E_I \rightarrow \mathbb{R}$ and $c : E_I \rightarrow \mathbb{R}$ such that:

$$\forall\, u \rightarrow v \in E_I : b(u \rightarrow v) \leq f(u \rightarrow v) \leq c(u \rightarrow v) \tag{5.3}$$

For an IPG, the capacity constraints of each $u \rightarrow v \in E_I$ are defined as follows:

$$b(u \rightarrow v) = \begin{cases} 1 & \text{if } u \rightarrow v = \mathtt{t} \rightarrow \mathtt{s}, \\ 0 & \text{otherwise.} \end{cases} \tag{5.4}$$

$$c(u \rightarrow v) = \begin{cases} 1 & \text{if } u \rightarrow v = \mathtt{t} \rightarrow \mathtt{s}, \\ b_{max}(u \rightarrow v) & \text{otherwise.} \end{cases} \tag{5.5}$$

where $b_{max}(u \rightarrow v)$ represents the maximum number of executions of an IPG edge in a single execution. The upper capacity constraint on the dummy edge $\mathtt{t} \rightarrow \mathtt{s}$ is 1 to indicate that the path through the procedure is executed once.

For maximum precision, b_{max} needs to be provided for each edge as this constrains the valid execution paths further. However, the minimal information that we must provide is a bound on each iteration edge $u \rightarrow v$ because these are the cycle-inducing edges of I — the solver can then determine b_{max} of other edges from the structural constraints.

An iteration edge $u \rightarrow v$ is bounded as follows. Let L_h^I be the *innermost* IPG loop such that $u \rightarrow v \in IE(L_h^I)$ and assume $b_{rel}(h)$ is its relative bound, either obtained through trace parsing or through static analysis; then, $b_{max}(u \rightarrow v) = b_{rel}(h)$. However, $b_{rel}(h)$ is only relative to the next outer nesting level, thus $b_{max}(u \rightarrow v)$ must also be factored by the relative bounds on all outer loops, i.e. $b_{max}(u \rightarrow v)$ is an upper capacity constraint representing its worst-case number of executions. If the multiplicity of $u \rightarrow v$ is greater than one then all other decision variables for $u \rightarrow v$ are set to 0. This is because the upper capacity constraint on $u \rightarrow v$ represents the maximum number of executions of $u \rightarrow v$ across all IPG loops in which it is contained.

Further note that incorporating frequency bounds into the ILP offers better precision as it constrains the execution count of an IPG edge to that only observed in testing. As already noted above, the accuracy of frequency bounds depends on suitable testing and coverage, although Chapter 6 will observe a marked improvement in the WCET estimate when such bounds are incorporated.

5.2.4 Non-Negativity Constraints

The non-negative constraints state that the execution count of edges must never be negative. That is, $f(u \rightarrow v) \geq 0$, for all $u \rightarrow v \in E_I$.

5.2.5 Solution to the IPET

The basic ILP is therefore formed of Equations (5.1), (5.2), and (5.15). In solving this model via standard **Linear Program** (LP) solvers, the WCET is returned, together with a setting of the execution counts for each IPG edge in the worst case. In this way, all paths are implicitly considered since the solver attempts different assignments to the execution counts in determining the worst case [73]. In general, we cannot determine the *exact* longest path from the execution counts because the order of execution is missing.

5.2.6 An Example

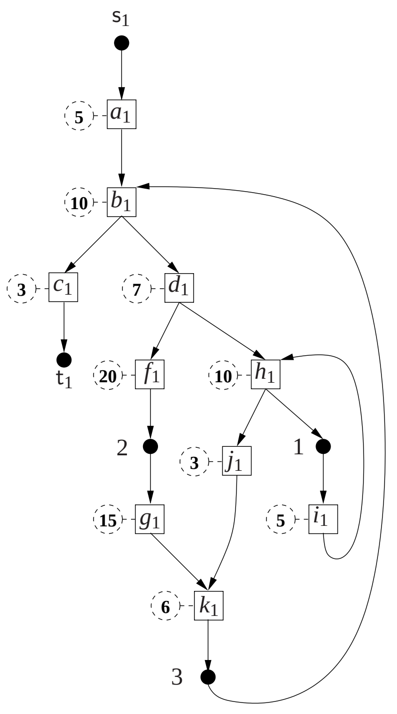
We illustrate the computation of a WCET estimate from a basic ILP using Figure 5.1. In particular, we compare the WCET estimate with that computed for its CFG* because this allows us to describe the disconnected circulation problem in greater detail.

Figure 5.1(a) depicts a CFG* in which every basic block has been annotated with its WCET. Also note that there are two loops L_{b_1} and L_{e_1}, whose relative bounds we assume are 10 and 5, respectively.

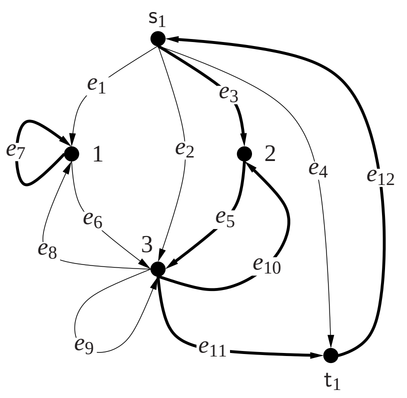
Let us perform the calculation on the CFG* using a simple path-based approach[1]. The longest path through L_{b_1} is either $p : b_1 \rightarrow d_1 \rightarrow f_1 \rightarrow 2 \rightarrow g_1 \rightarrow k_1 \rightarrow 3$, or $p' : b_1 \rightarrow d_1 \rightarrow h_1 \xrightarrow{5} 1 \xrightarrow{5} i_1 \xrightarrow{5} h_1 \rightarrow j_1 \rightarrow k_1 \rightarrow 3$. Furthermore, the longest *acyclic* path from $\mathtt{s}_1$ to $\mathtt{t}_1$ is $q : \mathtt{s}_1 \rightarrow a_1 \rightarrow b_1 \rightarrow c_1 \rightarrow \mathtt{t}_1$. Following are the WCETs of these paths:

[1]Producing the ILP of the CFG* causes unnecessary clutter and the reader can easily verify that both a path-based calculation and an ILP deliver the same WCET estimate.

Figure 5.1. Example to Demonstrate an ILP for an IPG.



(a) The CFG.*



(b) The IPG.

IPG edge $u \longrightarrow v$	$\mathsf{B}(P(u \longrightarrow v))$	$wcet(u \longrightarrow v)$
e_1	$\{a_1, b_1, d_1, h_1\}$	32
e_2	$\{a_1, b_1, d_1, h_1, j_1, k_1\}$	41
e_3	$\{a_1, b_1, d_1, f_1\}$	42
e_4	$\{a_1, b_1, c_1\}$	18
e_5	$\{g_1, k_1\}$	21
e_6	$\{i_1, h_1, j_1, k_1\}$	24
e_7	$\{i_1, h_1\}$	15
e_8	$\{b_1, d_1, h_1\}$	27
e_9	$\{b_1, d_1, h_1, j_1, k_1\}$	36
e_{10}	$\{b_1, d_1, f_1\}$	37
e_{11}	$\{b_1, c_1\}$	13
e_{12}	λ	0

(c) Data associated with IPG edges.

Path	WCET
p	58
p'	111
q	18

Evidently, the worst-case path through the CFG* is either $10p+q$ or $10p'+q$. Simple arithmetic shows that $10p'+q$ is the chosen path with an *accurate* WCET of 1128.

The IPG resulting from this CFG* is depicted in Figure 5.1(b), including the dummy edge $\mathsf{t}_1 \rightarrow \mathsf{s}_1$. Observe that there is a unique sequence of basic blocks on each ipoint transition, i.e. the instrumentation profile is path reconstructible. Let us therefore consider the WCET of each IPG edge $u \rightarrow v$ to be the sum of the WCET of each basic block in the set $\mathsf{B}(P(u \rightarrow v))$. This information is displayed in Figure 5.1(c) in which the edge $\mathsf{t}_1 \rightarrow \mathsf{s}_1$ has a WCET of 0 as its path expression is empty.

Let us provide upper capacity constraints on the iteration edges of the IPG by transferring the relative bounds of the CFG* loops onto the corresponding IPG loop. For the inner IPG loop, $IE(L^I_{h_1}) = \{e_7\}$. According to Equation (3.3) (in Chapter 3), the relative bound of e_7 is 4 because L_{h_1} has a relative bound of 5. Therefore, $c(e_7) <= 4 * 10 = 40$ as $L^I_{h_1}$ is nested in $L^I_{b_1}$, the latter which has a relative bound of 10. For the outer IPG loop, $IE(L^I_{b_1}) = \{e_8, e_9, e_{10}\}$. As the relative bound of L_{b_1} is 10, these edges have a relative bound of 9, which is also their upper capacity constraint.

These observations lead to the following ILP:

$$32e_1+41e_2+42e_3+18e_4+21e_5+24e_6+15e_7+27e_8+36e_9+37e_{10}+13e_{11}+0e_{12} \tag{5.6}$$

$$e_1+e_2+e_3+e_4=e_{12} \tag{5.7}$$

$$e_5=e_3+e_{10} \tag{5.8}$$

$$e_6=e_4+e_{11}+e_7 \tag{5.9}$$

$$e_8+e_{10}+e_{11}=e_2+e_5+e_6 \tag{5.10}$$

$$e_{12}=e_4+e_{11} \tag{5.11}$$

$$e_{12}=1 \tag{5.12}$$

$$e_7<=40 \tag{5.13}$$

$$e_8+e_9+e_{11}<=9 \tag{5.14}$$

The objective function is Equation (5.6). Structural constraints are Equations (5.7) through (5.11), which can be generated directly from the IPG. Upper capacity constraints are Equations (5.12) through Equation (5.14) in which Equation (5.12) states that the procedure is executed once.

Solving this ILP returns the following non-zero execution counts of edges:

Edge $u \to v$	$f(u \to v)$
e_3	1
e_5	10
e_7	40
e_{10}	9
e_{11}	1
e_{12}	1

Therefore, the WCET estimate is $(1*42)+(10*21)+(40*15)+(9*37)+(1*13)+(1*0)=1198$. As both the CFG* and the ILP contain the same timing information, we may conclude that this is an overestimation. The execution path through the IPG that these execution counts induce is exhibited by the thick lines in Figure 5.1(b). Clearly, this is not a feasible execution path as e_7 is completely disconnected. This is the sole cause of overestimation, the reasons for which are explored further in the next section.

5.3 Inaccuracies in the Basic ILP: Disconnected Circulations

Despite the fact that the basic ILP produced by the IPET always returns an upper bound on the WCET — thus conforming with the safety requirement of WCET analysis — the previous example demonstrated that it can be subject to unnecessary pessimism. This is because initial reduction to a circulation problem does not precisely characterise the set of execution paths through the IPG.

In particular, the basic ILP models a number of "self-contained" circulations, i.e. loops. As the ILP solver can satisfy all upper bounds on capacity constraints simultaneously, an inner circulation f will become disconnected from the execution path selected through its outermost circulation f' *unless* the longest path through f' always includes f. For example, in Figure 5.1(b), the iteration edge e_7 is disconnected because the longest path through its outermost circulation does not include execution of 1, i.e. $wcet(e_5) + wcet(e_{10}) = 21 + 37 > wcet(e_6) + wcet(e_8) = 24 + 27$.

Puschner-Schedl termed this the *disconnected circulation problem* (which was not discussed by Li-Malik) and gave the following formal description. Let $I = \langle \mathsf{I}, E_I \cup \{\mathsf{t} \rightarrow \mathsf{s}\}, \mathsf{s}, \mathsf{t} \rangle$ be an IPG and let f be a circulation in I. The IPG $I^f = \langle \mathsf{I}, E_I^f, \mathsf{s}, \mathsf{t} \rangle$, where $E_I^f = \{u \rightarrow v \in E_I | f(u \rightarrow v) > 0\}$, is called the circulation subgraph. Then, the execution counts returned by the ILP solver do not form a structurally feasible path — and hence lead to overestimation — whenever I^f is not a SCC.

As Puschner-Schedl proved, the disconnected circulation problem can be solved by *relative capacity constraints*, which replace the capacity constraints and essentially constrain the execution count of subgraphs relative to their outer loop-nesting level. To this end, it is necessary to identify the set of **implicating edges** $E_{imp} \subset E_I$ for each IPG loop L_h^I in I. An edge $u' \rightarrow v' \in E_{imp}$ implies the execution of L_h^I provided $u' \rightarrow v' \notin E(L_h^I)$ and every execution path containing $u' \rightarrow v'$ contains at least one edge $u \rightarrow v \in E(L_h^I)$.

The implicating edges for L_h^I are either its loop-entry edges or its loop-exit edges because either of these sets “imply” that the loop in question has executed. However, it is possible that an IPG loop has empty sets of loop-entry and loop-exit edges according to Definition 9 (in Chapter 3). For instance, this arises when an instrumentation profile is not path reconstructible. In this case, the implicating edges are the iteration edges from the next outer-nesting level of L_h^I for which a subset of $V(L_h^I)$ are destinations.

Given the implicating edges for each IPG loop, the relative capacity constraints can be formally expressed as follows:

$$\sum_{u \to v \in IE(L_h^I)} f(u \to v) \circ \sum_{u' \to v' \in E_{imp}} k_i \cdot f(u' \to v') \tag{5.15}$$

where $\circ \in \{<, \leq, =\}$ and at least one k_i is greater than zero.

To demonstrate relative capacity constraints, let us return to the example of Figure 5.1 and observe that:

- The set of implicating edges for $IE(L_{h_1}^I)$ is $\{e_6\}$.
- The set of implicating edges for $IE(L_{b_1}^I)$ is $\{e_1, e_2, e_3\}$.

Since 10 and 5 are the relative loop bounds for L_{b_1} and L_{e_1}, respectively, following are the relative capacity constraints on these iteration edge sets:

$$e_7 <= 4e_6 \tag{5.16}$$

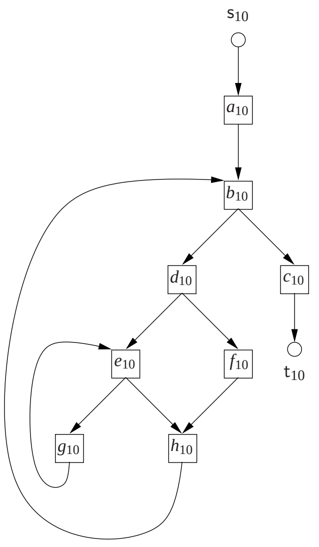
$$e_8 + e_9 + e_{11} <= 9e_1 + 9e_2 + 9e_3 \tag{5.17}$$

Substituting these constraints for Equations (5.13) and (5.14) in the ILP results in an accurate WCET estimate of 1128.

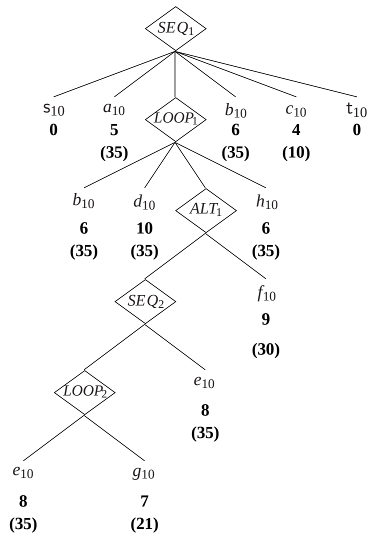
5.4 Evaluation

In this section, we evaluate the remodelled IPET by re-considering the synthetic example program introduced in Section 4.6, thus allowing a comparison between the precision of the two different calculation engines that operate on the IPG. Recall that we instrumented the program according to two particular instrumentation profiles so that we could evaluate the sensitivity of the analysis to the locations of ipoints.

Figure 5.2. Example Program (Same as Figure 4.16).



(a) The CFG.



(b) The AST.

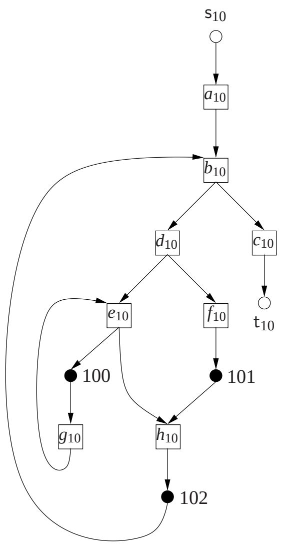
Figure 5.2 repeats the program under consideration, in both CFG and AST formats, for ease of reference. Recall that we assumed the WCETs of basic blocks (displayed below each basic block in Figure 5.2(b)) had been given and that the loop bounds were 5 and 10 for $L_{e_{10}}$ and $L_{b_{10}}$, respectively. An accurate WCET estimate of 1065 was calculated after performing the calculation on the AST using the timing schema proposed in [88].

5.4.1 Instrumentation Profile One

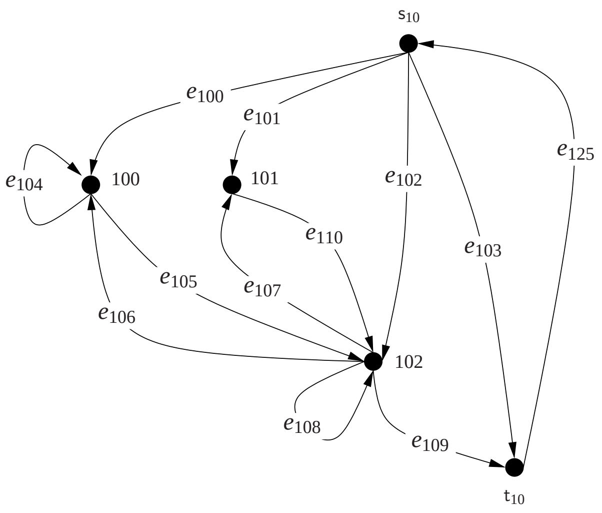
The CFG* and the IPG constructed from the assumed instrumentation profile are again depicted in Figure 5.3. Recall that, because the CFG* has been generated from the CFG, the loop bounds from the CFG effectively transfer onto the CFG*.

Further recall that, because the instrumentation profile is path reconstructible, we considered the WCET of each IPG edge $u \rightarrow v$ to be the sum of the WCET of each basic block in the set $\mathsf{B}(P(u \rightarrow v))$. For example, because $\mathsf{B}(P(\mathsf{s}_{10} \rightarrow 100)) = \{a_{10},$

Figure 5.3. Instrumented Program from Figure 5.2 to Evaluate the IPET (Same as Figure 4.17 without Itree).



(a) The CFG*.



(b) Resultant IPG.

$$29e_{100} + 30e_{101} + 35e_{102} + 15e_{103} + 15e_{104} + 21e_{105} + \\ 24e_{106} + 25e_{107} + 30e_{108} + 10e_{109} + 6e_{110} + 0e_{125} \tag{5.18}$$

$$e_{100} + e_{101} + e_{102} + e_{103} = e_{125} \tag{5.19}$$
$$e_{105} = e_{100} + e_{106} \tag{5.20}$$
$$e_{110} = e_{101} + e_{107} \tag{5.21}$$
$$e_{106} + e_{107} + e_{109} = e_{102} + e_{105} + e_{100} \tag{5.22}$$
$$e_{125} = e_{103} + e_{109} \tag{5.23}$$
$$e_{125} = 1 \tag{5.24}$$
$$e_{106} + e_{107} + e_{108} <= 9e_{100} + 9e_{101} + 9e_{102} \tag{5.25}$$
$$e_{104} <= 4e_{105} \tag{5.26}$$

(c) The ILP created from the IPG.

$b_{10}, d_{10}, e_{10}\}$, the WCET of $e_{100} = 5+6+10+8 = 29$. Furthermore, we also assumed that the loop bound supplied for each CFG* loop is transferred onto its corresponding IPG loop.

This timing information leads to the ILP shown in Figure 5.3(c) in which: Equation (5.18) is the objective function; Equations (5.19) through (5.23) are structural constraints; Equations (5.25) and Equation (5.26) are relative capacity constraints; and Equation (5.24) is the capacity constraint bounding execution through the procedure.

Solving this model by means of an ILP solver returns the following non-zero execution counts of edges:

Edge $u \rightarrow v$	$f(u \rightarrow v)$
e_{125}	1
e_{100}	1
e_{105}	10
e_{104}	40
e_{106}	9
e_{109}	1

Therefore, the WCET estimate is $(1*0)+(1*29)+(10*21)+(40*15)+(9*24)+(1*10) = 1065$. As this was the value computed from the AST, we may conclude that it is accurate. More important is that, in comparison to the Itree, which generated a WCET estimate of 1080, there is no pessimism in the ILP. This is because the Itree cannot adequately model the irreducible IPG loop $L^I_{b_{10}}$ consisting of iteration edges $\{e_{106}, e_{107}, e_{108}\}$, in contrast to the ILP which simply models the execution counts.

5.4.2 Instrumentation Profile Two

The CFG* and the IPG constructed from the second instrumentation profile are again depicted in Figure 5.4. Recall that the CFG* in Figure 5.4(a) is similar to that in Figure 5.3(a) except that ipoint 102 has effectively been moved to the new location occupied by ipoint 105. Whilst evaluating the Itree, we observed that this slight modification to the instrumentation profile resulted in a WCET estimate of 1155, adding more pessimism to the analysis.

The corresponding ILP is shown in Figure 5.4(c) in which: Equation (5.27) is the objective function; Equations (5.28) through (5.32) are structural constraints; Equations (5.34) and Equation (5.35) are relative capacity constraints; and Equation (5.33) is the capacity constraint bounding execution through the procedure.

Solving this model by means of an ILP solver returns the following non-zero execution counts of edges:

Edge $u \rightarrow v$	$f(u \rightarrow v)$
e_{126}	1
e_{111}	1
e_{115}	10
e_{118}	40
e_{120}	9
e_{121}	1

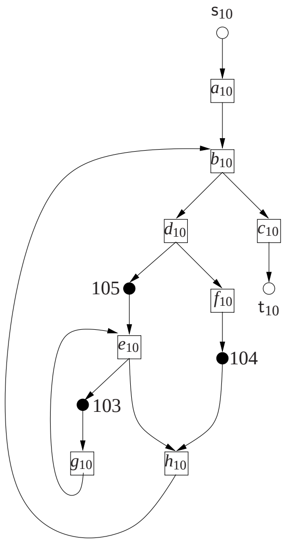
Therefore, the WCET estimate is $(1*0)+(1*29)+(10*15)+(40*25)+(9*10)+(1*6)=1065$. Once more we observe that there is no pessimism in the ILP. In particular, the IPET is not sensitive to the locations of ipoints.

5.5 Discussion

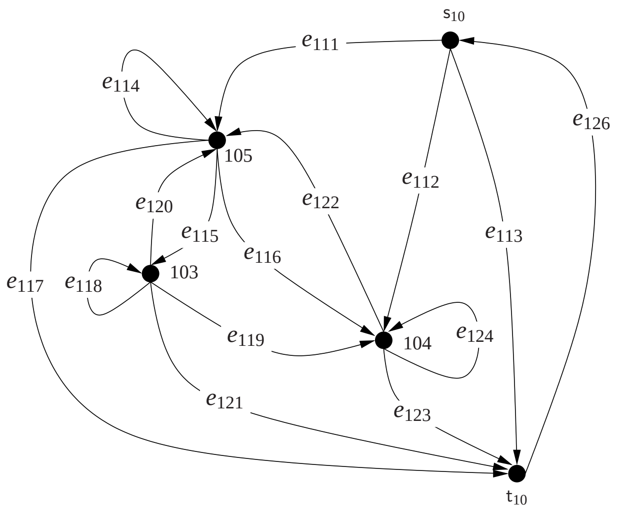
Puschner-Schedl also detailed another inaccuracy in the basic ILP in that, although the execution path induced by the returned execution counts is connected, it is infeasible. Both Puschner-Schedl and Li-Malik have described how to include more sophisticated path data, e.g. mutually inclusive and exclusive paths, to constrain the feasible execution paths further.

However, we have not considered how to include such additional constraints in our remodelled IPET. There are a couple of reasons for this. First, we have no mechanism by which such data are collected. On the one hand, our trace parser (described in Section 3.5.2) only collects frequency bounds on the IPG edges, but without correlation to

Figure 5.4. Second Instrumentation Profile on Program from Figure 5.2 to Evaluate the IPET (Same as Figure 4.18 without Itree).



(a) The CFG.*



(b) Resultant IPG.

$$21e_{111}+30e_{112}+15e_{113}+30e_{114}+8e_{115}+39e_{116}+24e_{117}+15e_{118}+ \\ 46e_{119}+37e_{120}+31e_{121}+22e_{122}+16e_{123}+31e_{124}+0e_{126} \tag{5.27}$$

$$e_{111}+e_{112}+e_{113}=e_{126} \tag{5.28}$$

$$e_{115}+e_{116}+e_{117}=e_{111}+e_{122}+e_{120} \tag{5.29}$$

$$e_{119}+e_{120}+e_{121}=e_{115} \tag{5.30}$$

$$e_{122}+e_{123}=e_{112}+e_{116}+e_{119} \tag{5.31}$$

$$e_{126}=e_{117}+e_{121}+e_{123}+e_{113} \tag{5.32}$$

$$e_{126}=1 \tag{5.33}$$

$$e_{118}<=4e_{115} \tag{5.34}$$

$$e_{114}+e_{116}+e_{119}+e_{120}+e_{122}+e_{124}<=9e_{111}+9e_{112} \tag{5.35}$$

(c) The ILP created from the IPG.

other IPG edges. On the other hand, assuming static analysis could provide such data (with respect to basic blocks), we cannot yet transfer these onto the IPG because our HMB framework does not compute path expressions of IPG edges (see Clarification 6 in Chapter 3).

Another issue with path constraints is that they typically cross procedure boundaries. Recall, however, that our calculation engine (described in Section 3.5.3) modularises the calculation of each context due to master ipoint inlining. Although this is not an issue for our tree-based calculation engine (because it cannot handle such constraints) incorporation into the IPET would require a different inlining mechanism. For these reasons, path-related constraints are considered beyond the scope of this thesis.

5.6 Summary

Programs that are arbitrarily instrumented often create irreducibility in the IPG, even if the underlying graph-based model of the program (the CFG*) is reducible. This is particularly problematic for the tree-based calculation engine proposed in Chapter 4 since the hierarchical representation must trade space overheads against loss of precision. One way to avoid such a trade-off is to assume control of the instrumentation profile and guarantee well-structured IPGs. However, this restricts the type of instrumentation employed (normally software so that an automatic tool can place ipoints carefully) and generally prohibits the use of state-of-the-art instrumentation profiles [2, 8, 65, 116]. Both of these limitations ultimately complicate re-targeting of our HMB framework to new systems.

In order to avoid pessimistic WCET estimates arising from path modelling, this chapter demonstrated how to remodel the IPET so that it applies to the IPG. In this context, we made the following additional contributions:

- We showed how to determine relative capacity constraints from the IPG using its structural properties, which are needed to accurately characterise the set of feasible execution paths through the IPG.
- We compared the IPET model with our tree-based calculation engine using the

same instrumentation profiles as in Section 4.6 and showed that the IPET always returned a precise WCET estimate, in contrast to those computed through an Itree. The main reason is that the IPET does not model program flow explicitly and can therefore handle arbitrary irreducible IPGs without undue pessimism. Although this is not a completely novel observation (similar weaknesses with ASTs fuelled the migration to graph-based models and the IPET), what we may conclude is that, *for arbitrarily instrumented programs*, the IPET should be the calculation engine of choice when the IPG is the program model.

6 Prototype Tool and Evaluation

In Chapters 4 and 5 we evaluated the **Instrumentation Point Graph** (IPG) program model against existing static analysis program models using both a tree-based approach and the **Implicit Path Enumeration Technique** (IPET). Although we were careful to select code that is representative of real-world applications, a thorough evaluation of the techniques proposed demands actual code executed with actual test vectors. This issue is now addressed by evaluating a large-scale industrial application.

This chapter commences in Section 6.1 with an overview of our prototype tool. Following that, Section 6.2 motivates our evaluation of an industrial case study — as opposed to the emerging WCET benchmarks [91] — and gives a detailed description of its properties. Section 6.3 then presents four different experiments that we undertook and evaluates the results computed through the prototype tool. Finally, Section 6.4 summarises our main findings.

6.1 Prototype Implementation

In order to automate the analytical process described in Chapters 3 through 5, we have implemented these techniques in a prototype tool. The operation of the tool, its main features, assumptions, and limitations can be summarised as follows:

- The first input to the tool is structural knowledge of the program at the intermediate code level. In particular, we require the following information:
  - The basic blocks and the transitions amongst them.
  - Which basic blocks are call sites, and moreover, the target (i.e. procedure) of the call. This means that we cannot analyse programs containing

function pointers. Also recall that we assume there are no cycles in the **call graph** (see Definition 11 in Section 3.5), thus no tool support is yet available for programs containing recursion.

– Which basic blocks contain **instrumentation points** (ipoints) and the trace identifier associated with each ipoint (to enable trace parsing). This subtly implies that we assume no control over the assignment of trace identifiers to ipoints; consequently, we must also assume that the assignment mechanism does not result in an IPG becoming a Non-Deterministic Finite Automata (NDFA), otherwise our trace parsing mechanism halts. Furthermore, we do not yet support other tracing mechanisms provided by logic analysers or, for example, Nexus [1].

The reason that the tool requires information about the program in this form is that, for the industrial application under analysis, all object code operations had to be stripped out before it could be released off-site due to the sensitive nature of the application. Therefore, we have no knowledge of the Instruction Set Architecture (ISA) and instead had to rely on a third-party tool[1] to provide the basic properties of program structure.

It is clearly a trivial programming task to extend the input of the tool to handle object code of other ISAs. As we do not require a processor model, therefore, porting to new architectures is relatively straightforward.

From this information, our tool constructs the **CFG*** (a data structure similar to the **Control Flow Graph**) of each procedure together with the call graph of the program. Master ipoint inlining is then carried out before building the IPG of each procedure. As noted in Chapter 3, we assume that each CFG* is reducible *after* master ipoint inlining in order to detect loops in the IPG. Currently, we have implemented Havlak's loop detection algorithm [49, 96], which identifies both reducible and irreducible loops. We envisage that, in the future, we can extend the IPG construction algorithm to handle both reducible and irreducible CFG*s.

- The second input is a trace file, which then triggers trace parsing. The trace

[1]RapiTime produced by Rapita Systems Ltd. [77]

parser populates an internal database structure that stores both the WCETs of IPGs transitions and the loop bounds of CFG* loops on a per context basis. In addition, we also obtain the **Measured Execution Time** (MET), which is simply the maximum observed time from the start of a timing trace to the end; this value is used as a comparison against the WCET estimate that our tool computes.

- Both full **context** expansion and full context unification calculations are currently supported (see Section 3.5.3). To calculate a WCET estimate for each context, the calculation engine can choose whether to use the tree-based calculation engine (described in Chapter 4) or the remodelled IPET (described in Chapter 5). To produce a solution from the Integer Linear Program (ILP) created by the IPET, our tool connects to the `lp_solve` library [13], which is freely available under the GNU public license and is generally very fast.

  We do not currently accept any path information from the user. This implies that all loop bounds used in the calculation are obtained directly from trace parsing; any loop not triggered during testing is therefore assumed to have a bound of zero.

- Interaction between the user and the analysis engine is handled by a Graphical User Interface (GUI). The GUI has been implemented using the plug-in architecture and Rich Client Platform (RCP) provided by Eclipse [103].

  Interaction with the GUI allows the user to view a particular intermediate data structures (e.g. the pre- and post-dominator trees), which are displayed by connecting to the uDraw(Graph) API [118]. Furthermore, the GUI displays a summary of the WCET report, which gives individual WCET estimates of individual procedures, their measured end-to-end execution times, and very basic coverage derived from trace parsing, such as trace edge coverage.

6.2 Properties of the Industrial Case Study

For our evaluation, we have chosen to analyse an industrial application rather than the *WCET benchmarks* [91] that have recently emerged. There are a couple of reasons for this choice that are worth discussion.

First, we initially did run our analysis on a subset of the WCET benchmarks using random testing and, in some cases, running the benchmark with its worst-case test vector, e.g. using a reverse sorted array for the `bubblesort` application. We found that our WCET estimates always bounded the MET. This is an expected result as we combine smaller portions of the measurements in computing a WCET estimate — thus our HMB framework can never produce a smaller value than the MET. However, we are particularly interested in the sensitivity of the analysis to the coverage of procedures/contexts and how expanding or unifying contexts affects the computed WCET estimate. Most, if not all, of the WCET benchmarks are relatively small, i.e. they only have a few procedures with a small number of procedure calls. We believe this is one limitation of the current set of WCET benchmarks, mainly because the analysis of contexts is pivotal to the accuracy of the WCET estimate with increasingly larger applications.

Second, it is very difficult to compare the WCET estimates computed by our tool with those computed by others, even with access to the same set of benchmarks. This is because, we would additionally need the same target hardware and the same compiler suite configured with the same options. Although the SimpleScalar toolset [7] provides, in theory, such a suitable framework, we are not aware of any direct comparison between existing WCET analysis tools using the same SimpleScalar configuration. Rather, the SimpleScalar architecture is typically configured so as to isolate the effect of a particular hardware feature, e.g. the instruction cache, to show the relative improvement.

Finally, to reiterate the point made at the onset of this chapter, we want to evaluate our HMB framework on an application for which we can neither control the properties of the program nor disable particular hardware effects.

Unfortunately, for non-disclosure reasons, we are not able to describe the functionality of the industrial application. Nor can we give a breakdown of the system properties as both the source code and details of the hardware architecture were withheld. (It is worth stressing that no static analysis tool could analyse this application because it is impossible to build a processor model.) However, what we are able to describe is properties of the application that are derived from the CFG*s, the call graph, and the trace file.

6.2.1 Structural Properties

Table 6.1 gives a summary of the main properties of the program, including its instrumentation.

In total there are 73 loops; thus, 42 are self-loops, many of which we believe are array initialisations as they most often appear at the beginning of a procedure. There is a maximum loop-nesting level of 2 for non-trivial loops. In general, there are only a few loops with respect to the overall size of the application, which is quite a common property in embedded software.

Property	Total	Instrumented
Procedures	223	105
Contexts	432	228
Non-trivial Loops	31	21
Basic Blocks	7010	958

Table 6.1. Structural Properties of Industrial Case Study.

Regarding the instrumentation profile, we had no control of where or how many ipoints were inserted. However, the third-party tool, RapiTime [77], used to instrument the application generally ensures that each procedure has a single master entry ipoint and a single master exit ipoint, i.e. 210 of the 958 ipoints are master ipoints. From our understanding of the application, most of the procedures that were not instrumented are error-handling routines, which do not contribute to the execution time during normal operation and are not instrumented as a result.

Observe that the instrumentation profile is very sparse. The total number of basic blocks across all instrumented procedures is 6488, thus $\approx$ 15% of this number are instrumented. We use this property to validate our claim that the IPG should be chosen ahead of the CFG or the Abstract Syntax Tree (AST) in a HMB framework. Although the instrumentation profile is not path reconstructible, every iteration of every CFG* loop could be observed in a trace, thus bounds extracted from trace parsing are not subject to pessimism (see Section 3.5.2). It is also worth noting that all of the IPGs are well structured, i.e. the instrumentation does not create IPG irreducibility as each header of a non-trivial loop (in the CFG) contains an ipoint.

6.2.2 Testing and Coverage Properties

Table 6.2 gives a summary of the main properties of testing and coverage as extracted from trace parsing.

The trace file contains $97,528$ traces, i.e. this is the number of test vectors, and the longest MET observed in these traces is $127,373$.

Clarification 9. *We use the MET to evaluate the precision of the WCET estimate computed through our tool. This has become best practice in the field of WCET analysis as the actual WCET is non-computable in the general case.*

Property	Value
Test Vectors	97, 528
MET	127, 373
Procedures Covered	87
Contexts Covered	166
Non-trivial Loops Covered	11

Table 6.2. Testing and Coverage Properties of Industrial Case Study.

Observe from Table 6.1 that it is desirable for the test framework to cover all 118 procedures, all 228 contexts, and all IPG trace edges across all contexts. This would increase our confidence in the WCET estimate (and indeed in the MET) because no sections of code that potentially contribute to the WCET would remain uncovered. From Table 6.2, we see that $\approx$ 83% of procedures and $\approx$ 73% of contexts were covered. Furthermore, after trace parsing, we counted the number of trace edges triggered in each context and then calculated how many trace edges were covered in each procedure. On average, across all 87 covered procedures, $\approx$ 85% of trace edges were covered, and across all 166 covered contexts, $\approx$ 73% of trace edges were covered. Finally, we note that only about half of the non-trivial loops were covered.

Normally we would expect a better attainment of coverage, particularly because functional coverage metrics attempt to cover every instruction at least once. From a timing analysis perspective, we would especially prefer better coverage of the non-trivial loops given that most of the (average-case and worst-case) execution time of programs is spent in loops. However, we recall from Clarification 2 (in Chapter 2) that these issues relate to WCET coverage and are considered beyond the scope of the thesis. For this reason, we assume that testing is good enough in analysing this application.

6.3 Experimental Set-Up and Results

We use the industrial application described in the previous section to perform four experiments which aim to validate particular hypotheses in the thesis. Each of these is now described, together with the results computed through our prototype tool.

6.3.1 Experiment 1: IPG versus CFG*

In Chapter 3, we motivated the introduction of the IPG program model by claiming that, when instrumentation is sparse, calculation techniques on existing static analysis program models are subject to unnecessary pessimism (in a HMB framework). We can validate this claim with the industrial application since it has been sparsely instrumented.

To do this, we used the IPET on both the IPG and the CFG* (since the CFG* is conceptually similar to the CFG) and then compared their WCET estimates. We initially considered all contexts unified, but we shall consider expanded contexts shortly. The IPET in particular was picked as we do not have a tree representation of the CFG*, and moreover, it generally offers greater precision.

We constructed the objective function for the CFG* as follows:

- We retrieved the WCET of each basic block *b* by taking the maximum observed WCET amongst the ipoint transitions on which *b* is executed unless *b* is a call site[2]. In this case, as we construct one CFG* per procedure, the WCET of *b* was assigned the WCET of the procedure that it calls.

- Each ipoint was given a value of 0 as we do not quantify the impact of the probe effect in this thesis — see Clarification 1 (in Chapter 2).

The bounds on CFG* loops were those obtained from trace parsing from the corresponding iteration edge set. Because every iteration of every CFG* loop is observable in a trace, these bounds are accurate provided testing has triggered the worst-case number of iterations.

Table 6.3 gives the computed WCET estimates and the associated (rounded) pessimism relative to the MET (which is $127,373$).

Program Model	WCET Estimate	+%
IPG	646, 392	408
CFG*	17, 471, 631	13, 617

Table 6.3. Results of Experiment One.

There are a couple of interesting observations from these results. First, the WCET estimate computed from the CFG* is several orders of magnitude higher than that computed from the IPG. One potential problem with this — besides giving overinflated values — is that the longest path identified can be biased towards procedures that are sparsely instrumented rather than the procedures that actually contribute to the WCET. Engineers often want to reduce the WCET by optimising code on the worst-case path, thus identifying code that does not contribute to worst-case behaviour will result in fruitless optimisation.

Second, the WCET estimate computed from the IPG is (approximately) a four-fold overestimation. This contrasts with most results published in the literature, which normally report little margin of overestimation. This is even more surprising given that the program is structurally simple, only containing a few loops.

[2]The astute reader will recall that, in Clarification 6, we considered the computation of path expressions beyond the scope of the thesis. The natural question to ask, therefore, is how we obtained the WCETs of basic blocks from ipoint transitions. In fact, our tool implements a more sophisticated version of the data-flow framework that we presented in Chapter 3 to construct the IPG. Simple extensions to the data-flow equations allow the basic blocks executed on each ipoint transition to be computed during construction.

We investigated the reason for the pessimism by examining the individual WCET estimates of procedures compared with their METs. (As each procedure has a single master entry ipoint and a single master exit ipoint, these METs could be extracted by the trace parser in the same way that the MET of the program is extracted.) As the number of procedures is large, we have selected a few procedures with different properties and displayed their comparisons in Table 6.4.

Procedure	Properties	MET	WCET estimate	+%
1	SLC	242	242	0
2	SLC	544	544	0
3	SLC	1, 388	1, 673	21
4	SLC	19, 092	22, 158	16
5	One loop	25, 581	57, 700	126
6	SLC & many procedure calls	44, 742	419, 862	838
7	SLC & many procedure calls	9, 668	38, 344	296
8	SLC & many procedure calls	24, 549	58, 008	136
9	SLC & many procedure calls	96, 600	565, 309	485

Table 6.4. Comparison of Measured Execution Times (MET) and WCET estimates of Selected Procedures. SLC signifies Straight-Line Code.

Procedures 1 through 4 are leaves in the call graph and generally have extremely accurate WCET estimates — the margin of overestimation is comparable to that reported in the literature using the WCET benchmarks.

Procedure 5 is also a leaf in the call graph but the overestimation is more considerable due to the loop. Observe that this is precisely the advantage that a HMB framework offers vis-a-vis an end-to-end testing strategy. That is, the latter relies on a test vector to trigger the loop for its maximum number of iterations with worst-case timing behaviour, whereas the HMB framework is able to piece this information together from the smaller units of computation and the loop bounds provided. For this reason, we may also infer that inserting ipoints in loops provides increased confidence in the WCET estimate as there is less reliance on the test framework to trigger each loop for its worst number of iterations.

Procedures 6 through 9 are relatively close to the root of the call graph and do not contain loops. Comparing their margins of overestimation with the other straight-line code procedures (1 through 4), we observe a marked increase. The key difference is that the paths through these procedures contain many procedure calls. Therefore, any overestimation in a callee is propagated into the caller, and as a consequence, the overestimation widens as the calculation engine approaches the root. This explains the pessimism in the IPG calculation as the program under analysis contains many procedure calls.

6.3.2 Experiment 2: Context Expansion versus Context Unification

The previous experiment suggested that how contexts are considered in the calculation is pivotal to the accuracy of the WCET estimate. To quantify the margin of improvement (if any), we also ran the WCET calculation engine on the IPG considering all contexts expanded and all contexts unified; these are on opposite ends of the sliding scale and therefore offer the best measure of comparison.

Table 6.5 gives the computed WCET estimates and the associated (rounded) pessimism relative to the MET (which is $127,373$).

Contexts	WCET Estimate	+%
Unified	646, 392	408
Expanded	610, 814	378

Table 6.5. Results of Experiment Two.

As we can see, the unified calculation is tighter by a margin of $\approx$ 30%, principally because the unit of computation is more accurate and this propagates up the call graph. Clearly, more significant improvements can be expected with more powerful context-handling techniques. For example, on architectures with a cache, the first call to a procedure often induces a greater WCET than on all subsequent calls due to cache misses; contexts could thus be extended to distinguish between these cases.

6.3.3 Experiment 3: Itree versus the IPET

Chapters 4 and 5 presented a tree-based and an ILP-based calculation engine, respectively, which operate on the IPG. We showed, by means of a synthetic, yet realistic, example that the IPET is generally less sensitive to the instrumentation profile. Here we wish to compare the actual differences using real code with real numbers.

Table 6.6 gives the computed WCET estimates and the associated (rounded) pessimism relative to the MET (which is $127,373$). We performed two types of IPET calculations. The first one (referred to as "IPET") was the simple ILP model with structural constraints and relative loop bounds. The second one (referred to as "IPET-Freqs") integrated the maximum frequency of execution of trace edges (that were extracted during trace parsing) as execution count constraints. We did this because tree-based calculations cannot include extra path information and we wanted to evaluate its affect on the precision of the WCET estimate.

Calculation	WCET Estimate	+%
Itree	646, 392	408
IPET	646, 392	408
IPET-Freqs	566, 527	345

Table 6.6. Results of Experiment Three.

An interesting result is that both the Itree and the IPET calculations produced the same WCET estimate. This is due to the fact that the instrumentation profile created reducible IPGs. This reaffirms our earlier finding from Chapter 4 that the Itree can be competitive with other calculation methods provided instrumentation is well placed in loops.

On the other hand, the IPET-Freqs calculation offers much more precision than that of the Itree, which is to be expected as the bounds on the execution counts essentially constrain the feasible execution paths. Also observe that the tightness gained with IPET-freqs outweighs than when considering expanded contexts (c.f. Table 6.5). Although, in general, using frequency bounds creates the possibility of underestimation, what we can infer is that the IPG can gain significantly from path information supplied by a user or from conventional static analysis tools.

6.3.4 Experiment 4: Sensitivity to Coverage

Our final experiment explored how sensitive our HMB framework is to coverage. In particular, we wanted to know how many procedures had to be covered (and the number of test vectors required) to produce a WCET estimate that bounded the MET.

To do this, we stopped the trace parser once a particular timing trace covered a set of (as yet uncovered) procedures and then performed the calculation with the timing data retrieved until that point. We unified all contexts and used the basic IPET (without frequency bounds), although the previous experiment suggests that we can expect the same WCET estimates using the Itree.

Procedures covered	Test vectors	MET	WCET estimate	+/-%
11	1	34, 643	48, 797	-62
11	3, 405	39, 928	77, 647	-39
38	3, 406	114, 403	323, 150	+154
39	3, 407	114, 403	377, 899	+197
48	3, 909	114, 403	377, 899	+197
54	3, 953	114, 403	394, 155	+209
56	4, 492	114, 403	402, 206	+216
59	4, 495	114, 403	402, 206	+216
60	5, 136	114, 403	536, 147	+321
62	5, 603	114, 403	541, 754	+325
68	6, 138	114, 403	542, 563	+325
71	6, 139	114, 403	542, 604	+326
73	6, 140	114, 403	542, 604	+326
75	8, 147	114, 403	549, 538	+331
76	13, 117	117, 762	580, 594	+356
78	15, 228	117, 762	581, 899	+357
79	30, 123	124, 163	598, 488	+370
80	72, 816	124, 163	610, 178	+379
82	72, 819	124, 163	610, 178	+379
87	76, 556	127, 373	613, 707	+382

Table 6.7. Results of Experiment Four.

Table 6.7 presents: the number of procedures covered in the increments observed; the number of test vectors that achieved the given coverage; the MET after that number of test vectors; the computed WCET estimate after that number of test vectors; and the associated (rounded) optimism/pessimism in the WCET estimate relative to the longest MET (which is $127,373$). The final row in the table has 87 covered procedures as this was the maximum covered by the test vectors (see Section 6.2). Also note that there is no increment in the number of procedures covered between the first and second rows; we provided this additional row because this is the last number of test vectors for which our tool underestimates the longest MET.

Figure 6.1 is a graphical representation of the results. We have also plotted the longest MET (of $127,373$) so that the differences between them can be visualised.

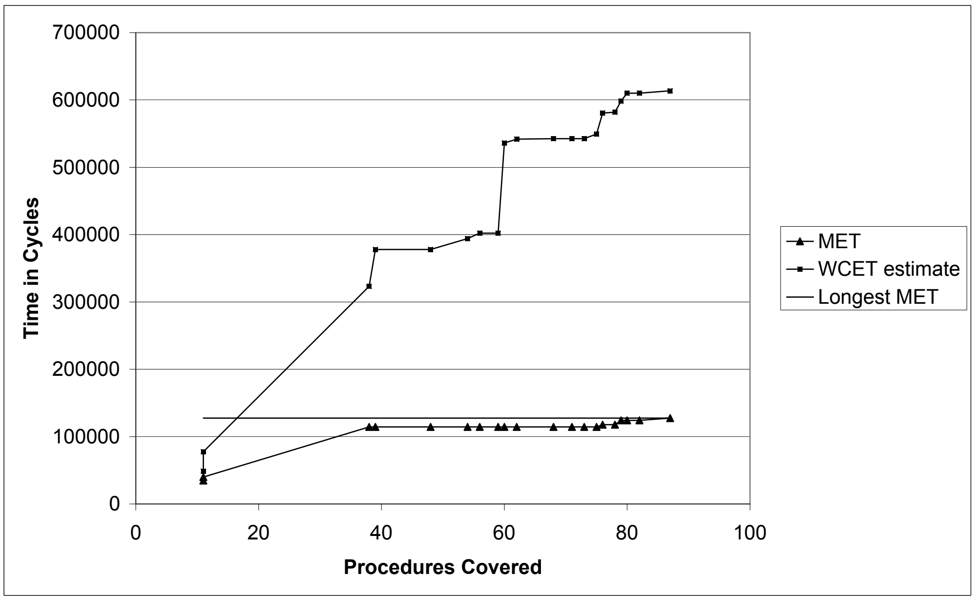


Figure 6.1. Graphical Representation of Experiment Four.

There are several interesting observations from these results. First, our HMB tool only underestimates whilst there is $\approx$ 13% procedure coverage (of those that can be covered) and $\approx$ 3% of the total number of test vectors have been employed. These are very small numbers, however, and it is unlikely that any system would be deployed with such little testing. On the other hand, end-to-end testing is much more sensitive to coverage as the longest MET is not discovered until 100% procedure coverage and $\approx$ 78% of the total number of test vectors have been employed.

Second, our HMB tool benefits much more from increasing coverage than does end-to-end testing. For example, the MET in the interval of $38 - 75$ covered procedures remains the same, but in the same interval, the WCET estimate changes considerably.

Especially note the difference between the WCET estimate at 59 covered procedures and at 60 covered procedures, i.e. where the covered procedures only differ by one. An interesting future direction of research, therefore, is to guide the test framework to procedures where execution time becomes concentrated. Clearly, this is not possible when only measuring the MET, which highlights another potential advantage of a HMB framework.

6.3.5 Discussion

Analogously to similar experiments reported elsewhere [26], the major bottleneck in our experiments was trace parsing, which took approximately 8 minutes on a Pentium 4 3.2 GHz processor with 512 MB of RAM. This is not entirely surprising since the trace file is 177 MB, thus just reading from the file (without parsing the data) consumes considerable time, about one third of the 8 minutes. Moreover, our implementation has not been optimised, and we therefore envisage marked improvements with a more thoughtful implementation.

Observe, however, that this motivates sparse instrumentation further as trace file sizes shrink and we can analyse more timing traces with the same overhead. Indeed, this was one of the motivations for the work in [8, 65], which developed an optimal instrumentation profile, i.e. fewest number of ipoints, whilst retaining path reconstructibility.

Regarding the calculation, we found that solution times dwarfed the complexity of trace parsing. That is, each modularised Itree or IPET calculation took milliseconds. For the IPET, this is because the ILP problems collapse to Linear Program (LP) problems, which have fast solutions as discussed in Chapter 5.

6.4 Summary

This chapter analysed a large-scale industrial application that had been executed using real test vectors on the target hardware. The analysis was automated by means of a prototype tool that implemented the techniques described in Chapters 3 through 5.

We performed four different experiments with the industrial application and concluded the following:

- When instrumentation is sparse and units of computation are extracted from a trace file, calculations on the IPG program model are much more accurate than those on conventional static analysis program models, such as the CFG. This is due to the fact that there is no pessimism in the unit of computation.
- For large applications, how contexts are handled is pivotal to the accuracy of WCET estimates. Calculations on procedures near the leaves of the call graph generally have very good precision but the overestimation widens as the calculation engine reaches the root due to an accumulation of pessimism.
- The tree-based calculation engine proposed in Chapter 4 can be competitive with the remodelled IPET described in Chapter 5. In particular, we confirmed the earlier finding (in Chapter 4) that the key to precise Itree calculations is well instrumented loops (in the structural sense).
- The biggest factor in decreasing the WCET estimate was the incorporation of frequency bounds extracted from trace parsing. Extra path information can therefore aid calculations on the IPG significantly.
- Increasing the amount of code coverage and test vectors does not necessarily increase confidence in the longest end-to-end WCET, but our HMB framework benefits considerably as the smaller units of computation (i.e. the ipoint transitions) are stressed further.
- The majority of the analysis time in HMB frameworks is locked in trace parsing. Therefore, it is desirable to reduce the size of timing traces by means of sparser instrumentation profiles, allowing more timing traces to be processed (without increasing overheads) and thereby increasing the confidence in the WCET estimate.

7 Conclusions and Future Work

In this final chapter we draw some conclusions and summarise the main contributions of the thesis. We also point out several areas of future work in the HMB direction.

7.1 Summary of Contributions

In Chapter 1, we motivated the development of a **Hybrid Measurement-Based** (HMB) framework by noting two key points:

1. Current static analysis techniques require *predictability* at all levels of the analysis. However, more advanced processors are infiltrating the embedded systems market, and these cause varying degrees of unpredictability. To manage complexity, the only solution is to make conservative assumptions about processor operation, which ultimately translates into an overestimation of the WCET.
2. Existing end-to-end testing techniques do not provide an accurate estimate of the WCET because they only target *functional* properties of the *software*. Therefore, the effect of hardware is neglected, which is especially deficient given the influence of advanced processors on the execution time of a program.

In Chapter 2, we gave a detailed account of the state-of-the-art in WCET analysis and reviewed existing testing and coverage practices. We observed that, although great strides have been taken in processor modelling, the effect of the operational interaction between disparate units (e.g. a branch predictor and a cache) largely remains untouched. Furthermore, most effort has been concentrated on modelling the CPU, neglecting the impact of peripheral devices, such as bus contention. For these reasons, some researchers are now advocating the design of more predictable hardware or are focusing their attention on reducing the WCET.

On the other hand, we also highlighted the deficiency of existing coverage metrics that are employed in end-to-end testing to compute a WCET estimate. As a result, HMB techniques have started to emerge. However, we noted that such techniques currently require very specific **instrumentation point** (ipoint) placement (to avoid pessimistic WCET estimates) because they use an existing static analysis program model — either the **Control Flow Graph** (CFG) or the **Abstract Syntax Tree** (AST) — in the calculation stage. Not only does this limit the type of instrumentation employed (i.e. software), it prevents usage of state-of-the-art instrumentation profiles that have been developed to reduce ipoint overheads.

Based on these observations, we motivated the development of a new HMB framework that allows for *arbitrary instrumentation*. In the context of this HMB framework, we made the following contributions:

- In Chapter 3, we introduced the **Instrumentation Point Graph** (IPG) as a novel program model, which models the transitions among ipoints instead of the transitions among basic blocks. This simple paradigm shift allows timing data extracted from timing traces (produced when the program is executed in the test phase) to be mapped directly onto the IPG, avoiding any overhead associated with basic blocks as a consequence.

  We showed how to construct and analyse structural properties of the IPG using the **CFG***, an intermediate form similar to the CFG. In particular, we demonstrated how to use the structural connection between a reducible CFG* and an IPG in order to identify arbitrary irreducible loops in the IPG. This relation also provides the mechanism by which loop bounds obtained through static analysis [47, 51] can be transferred onto the IPG. However, because such analyses can only ever be semi-automatic at best due to the Halting problem, we also presented a way to extract loop bounds from timing traces using properties of the IPG. Although the accuracy of such bounds is mainly tied to the amount of testing undertaken (bounds can be underestimated), we also showed that the instrumentation profile can be equally as influential (bounds can be overestimated).

The final contribution of this chapter was the analysis of programs with inter-procedural relations. We described how to virtually inline a subset of the ipoints from each callee into the caller (but none of the transitions) so that the **trace parser** has visibility to procedure calls and returns. We showed how to extract timing data from timing traces on a per context basis, as opposed to a per procedure basis, using the set of IPGs (one per procedure). This allows the precision of the analysis to be determined off-line as the calculation engine can unify or expand contexts as and when requested.

- Chapter 4 presented a tree-based calculation engine that operates on the **Itree**, a novel hierarchical representation of the IPG which models traditional high-level constructs such as sequence, selection, and iteration.

  We presented an algorithm to decompose the IPG into Itree form. For these purposes, we introduced the notion of **acyclic reducibility**, which basically decides if acyclic regions in the IPG can be decomposed hierarchically. This allows branch and merge vertices to be categorised either as acyclic-reducible or acyclic-irreducible, detection of which can prevent redundant traversals of the IPG that instead produces a forest of Itrees, an **Iforest**.

  We presented an algorithm that decomposes the IPG into an Iforest. The algorithm is not restricted to a particular class of IPGs; that is, it handles arbitrary instrumentation even if that produces arbitrary irreducible regions in the IPG. The only restriction is that the CFG* (from which the IPG is constructed) is reducible, but the algorithm supports multiple exits out of loops and multiple loop-back edges commonly associated with `break` and `continue` statements, respectively.

  We introduced the timing schema that drives the calculation over individual Itrees in the Iforest and then evaluated our tree-based calculation engine by considering an example program instrumented with two different (sparse) instrumentation profiles. We concluded that, when the atomic units of computation are derived from trace parsing, the Itree generates more accurate WCET estimates than the traditional AST-based calculation. Moreover, the locations of ipoints with respect to the program structure

Our tree-based calculation is sensitive to the locations of ipoints because of the problem of irreducibility. This forces the Itree into making a trade-off between the space overhead incurred and the precision of the analysis.

- Chapter 5 remodelled the **Implicit Path Enumeration Technique** (IPET) so that it applies to the IPG. In particular, we showed how to model **relative capacity constraints** for the IPG since these are needed to ensure that the execution counts returned by the ILP solver form a structurally feasible execution path.

  Finally, we evaluated the IPET using the same instrumentation profile as in Chapter 4 and concluded that, because it can handle arbitrary irreducibility without undue pessimism, the IPET should generally be the chosen calculation technique when the IPG is the program model.

- Chapter 6 described the implementation of the prototype tool developed to support the techniques described in Chapters 3 through 5. We used the prototype tool to analyse a large-scale industrial application that had been executed using real test vectors on the target hardware.

  We performed four different experiments with the industrial application and concluded the following:

  - The IPG offers better precision than existing static analysis program models when instrumentation is sparse and units of computation must be gleaned from timing traces.
  - How contexts are handled plays a crucial role in the accuracy of WCET estimates generated from the IPG. In general, procedures near the leaves of the call graph have very good precision but the overestimation widens as the calculation engine reaches the root due to an accumulation of pessimism.
  - The tree-based calculation engine can be competitive with the remodelled IPET on an industrial-strength application. The key, in particular, is to place instrumentation with structural properties of the IPG in mind, i.e. to avoid irreducibility.

  - Confidence in the WCET estimate computed by our HMB framework increases with better code coverage and an increasing number of test vectors.
  - The biggest bottleneck in our HMB framework is attributed to trace parsing. This motivates a reduction in the size of timing traces by means of sparser instrumentation profiles so that more timing traces can be processed whilst keeping overheads low.

The more general conclusion that we may draw from the thesis is that the instrumentation profile employed largely determines the accuracy of WCET estimates computed through a HMB framework. On the one hand, sparse instrumentation places a greater burden on the testing front end because it is important to stress the WCETs of the units of computation (i.e. the ipoint transitions). Underestimation is always possible if coverage is insufficient because any analysis is tied to the accuracy of its input parameters.

On the other hand, WCET estimates have the potential to be overestimated if testing is good but ipoints are not well placed within loops. This is because loop bounds automatically derived from the properties of the IPG can be overestimated, whilst the tree-based calculation engine could produce a structurally infeasible path due to irreducibility.

However, what this thesis has contributed is a fully automatic HMB framework based on the IPG which allows programs to be instrumented as-is without causing any additional pessimism in the units of computation. Provided testing is good enough and loop bounds are accurate, WCET estimates will therefore be more accurate than those computed through existing static analysis program models.

7.2 Future Work

Each of the following is a future direction of work:

- One of the main assumptions of the thesis was that a suitable test framework is in place, including, in particular, an appropriate set of coverage metrics. To date, all such coverage metrics have targeted the functional properties of a program:

a stricter set of criteria, from the timing analysis perspective, would provide greater confidence in the WCET estimate computed through our HMB framework. We believe that any *WCET coverage* metric must, amongst others, take the following three key considerations into account:

1. The structural properties of the instrumentation profile, and specifically, whether it is path reconstructible or not. With the same number of ipoints, it is possible to instrument a program in two different ways such that there is a large disparity between the respective number of ipoint transitions to cover, e.g. by instrumenting each basic block that is a leaf in the pre-dominator tree [116]. Simply covering each edge in the IPG might therefore require different test cases.

   If the instrumentation profile is not path reconstructible, particular transitions could execute many different paths in the program. Not only must the test vectors attempt to exercise each such path, but they must also attempt to stress the path in its worst-case architectural state.
2. The properties of the processor in conjunction with the types of instructions on each ipoint transition due to the effect of the hardware architecture on the time each instruction takes to complete. For instance, transitions including floating-point instructions would require better coverage on processors that employed a separate floating-point pipeline as opposed to those which had no speed-up features (because all instructions would have fixed execution times).
3. The context of execution so that call contexts can be expanded with sufficient confidence in the WCET calculation stage. Obtaining poor context coverage forces the calculation to unify contexts, generally resulting in a more conservative WCET estimate.

Each of the above three points underline the message that WCET coverage metrics have very different requirements than existing functional criteria.

We believe in many cases that the IPG provides a suitable infrastructure for which attainment of WCET coverage metrics can be measured, primarily because it contains the program properties of interest, e.g. the section of code executed and the types of instructions.

- Regarding procedure calls, we assumed that recursion and function pointers were both absent. From our experience, there are a number of embedded systems that include these features, thus currently limiting our approach. We believe that function pointers can be handled by patching the call graph during trace parsing when the targets of calls become known.

- Our trace parsing mechanism required each IPG to be a Deterministic Finite Automata (DFA). This generally requires a larger number of trace identifiers and sometimes forces ipoints to particular locations (at procedure call sites) to avoid Non-Deterministic Finite Automatas (NDFA). However, the number of trace identifiers can be restricted by properties of the tracing mechanism, such as the number of pins available on an external port. Extending the analysis to handle NDFAs would, in theory, overcome these issues.

  Another future area of research linked to trace parsing is the derivation of non-functional properties from timing traces. Currently, we only retrieve loop bounds, but there is no reason to suggest that infeasible path information cannot also be gleaned. Indeed, such information could then be fed back into a static analysis tool, or related to the user, for subsequent verification.

- Concerning the IPET and static analysis. Many static analysis techniques derive quite sophisticated flow facts for the program under analysis, and generally, the IPET is the only calculation technique that can suitably model such constraints. Although the mapping is normally straightforward when the CFG program model is chosen, the IPG adds another layer of complexity because basic blocks appear on different ipoint transitions. The biggest factor in decreasing the WCET estimate was the incorporation of frequency bounds extracted from trace parsing. Extra path information can therefore aid calculations on the IPG significantly.

  A first step towards being able to transfer these data onto the IPG is to construct the path expressions of IPG edges, which we did not consider. We believe that the work in [102, 112] can be modified for these purposes as they have showed how to construct the path expressions on reducible flow graphs. However, the main difference between that problem and the path expression problem on the IPG is the latter only considers *ipoint-free* paths between ipoints, whereas the former considers all paths between vertices.

7.3 Final Remarks

This thesis did not set out to suggest that HMB analysis can provide *upper bounds* on the WCET. Rather, the starting point of the thesis was that, because modelling advanced processors is complex, existing static analysis techniques can greatly over-estimate the WCET. With this motivation in mind, we developed a HMB framework based on the IPG that combines the relative strengths of static analysis and existing testing practices, thus avoiding processor modelling altogether. This thesis contends that the IPG is the most suitable program model when computing *WCET estimates* through a HMB approach.

A Terminology and Notation

Here we review core graph and tree terminology and notation since they are not standardised in the literature. We also clarify some set notation used throughout the text.

A.1 Basic Set Notation

For a set S, we denote its cardinal as $|S|$. The **singleton set** S has a unique element; we sometimes abuse notation and use S to indicate the unique element $s \in S$. The **power set** of a set S, i.e. the set of all subsets of S, is denoted 2^S.

A **multiset** is an unordered collection of elements where an element can occur as a member more than once. The **multiplicity** of an element s_i in a multiset S is the number of times s_i occurs in S.

A.2 Basic Graph Terminology

A **graph** $G = \langle V_G, E_G \rangle$ is a pair of finite sets V_G and E_G, called **vertices** and **edges**, respectively. We will denote the vertex and edge sets by $V(G)$ and $E(G)$ when context does not disambiguate these sets amongst several graphs. There are two types of graphs:

- An **undirected** graph $G = \langle V_G, E_G \rangle$ has an edge set $E_G = \{\{u,v\} | u,v \in V_G\}$.
- A (forward) **directed** graph (**digraph**) $G = \langle V_G, E_G \rangle$ has an edge set $E_G = \{(u,v) | u,v \in V_G\}$. We sometimes denote a directed edge (u,v) as $u \rightarrow v$ and say that u and v are **adjacent**, u is the **source**, and v is the **destination**. For any $u \in V_G$:

- $pred(u) = \{v | (v,u) \in E_G\}$ denotes the set of **(immediate) predecessors**. A **merge** vertex u is one for which $|pred(u)| > 1$.
- $succ(u) = \{v | (u,v) \in E_G\}$ denotes the set of **(immediate) successors**. A **branch** vertex u is one for which $|succ(u)| > 1$.

Following are two derivatives of a digraph $G = \langle V_G, E_G \rangle$:

- The **underlying undirected graph** $G' = \langle V_G, E_{G'} \rangle$ has an edge set $E_{G'} = \{\{u,v\} | (u,v) \in E_G \vee (v,u) \in E_G\}$.
- The **reverse** digraph $G' = \langle V_G, E_{G'} \rangle$ has an edge set $E_{G'} = \{(u,v) | (v,u) \in E_G\}$.

A **multi-digraph** $G = \langle V_G, E_G \rangle$ is a digraph such that E_G is a multiset. Any edge in E_G appearing more than once is termed a **multi-edge**.

A **flow graph** is a weakly connected digraph $G = \langle V_G, E_G, \mathtt{s}, \mathtt{t} \rangle$ such that:

- $\mathtt{s}, \mathtt{t} \in V_G$ are distinguished (dummy) vertices in which $|pred(\mathtt{s})| = 0$ and $|succ(\mathtt{t})| = 0$; $\mathtt{s}$ is the **entry** vertex and $\mathtt{t}$ the **exit** vertex.
- Every vertex v is reachable from $\mathtt{s}$ and can reach $\mathtt{t}$.

Let $G = \langle V_G, E_G \rangle$ and $G' = \langle V_{G'}, E_{G'} \rangle$ be two graphs. If $V_{G'} \subseteq V_G$ and $E_{G'} \subseteq E_G$ then G' is a **subgraph** of G, written as $G' \subseteq G$. Moreover, G' is an **induced** subgraph of G whenever $G' \subseteq G$ and either:

- $E_{G'} = \{(u,v) \in E_G | u, v \in V_{G'}\}$ if G is directed, or
- $E_{G'} = \{\{u,v\} \in E_G | u, v \in V_{G'}\}$ if G is undirected.

For a graph $G = \langle V_G, E_G \rangle$, a **path** p of length $m-1$ is a sequence $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_{m-1} \rightarrow v_m$ such that, for all $1 \leq i < m+1$, $v_i \rightarrow v_{i+1} \in E_G$. The notation $u \xrightarrow{*} v$ denotes a path of length zero or more, whereas $u \xrightarrow{+} v$ denotes a path of length one or more. We say that a vertex v is **reachable** from vertex a u (or alternatively, u can **reach** v) if there is at least one path $u \xrightarrow{*} v$. We denote the set of all paths from u to v as $Paths(u,v)$. Two paths $p : u \xrightarrow{*} v$ and $q : u \xrightarrow{*} v$ are **vertex disjoint** if u and v are the only vertices on both p and q. A **cycle** is a path $u = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_{m-1} \rightarrow v_m = u$ such that $m > 1$, i.e. it cannot be empty; a digraph is **acyclic** if there are no cycles.

We are often interested in the connected property of a graph as follows:

- An undirected graph $G = \langle V_G, E_G \rangle$ is **connected** if there is a path $u \xrightarrow{*} v$ between any $u, v \in V_G$.
- A digraph $G = \langle V_G, E_G \rangle$ is **weakly connected** if its underlying undirected graph $G' = \langle V_G, E_{G'} \rangle$ is connected.
- A **Strongly Connected Component** (SCC) of a digraph $G = \langle V_G, E_G \rangle$ is a maximal subgraph $G' = \langle V_{G'}, E_{G'} \rangle$ such that, for every $u, v \in V_{G'}$, there are paths $u \xrightarrow{+} v$ and $v \xrightarrow{+} u$. G is **strongly connected** whenever $V_G = V_{G'}$.

  For a digraph G consisting of SCCs $S_1, S_2, \ldots, S_n$, the **component graph** G' of G is built by collapsing all S_i into abstract vertices u_{S_i}; any entry edge $v \longrightarrow w$ in G, $w \in S_i$, has u_{S_i} as its destination in G'; any exit edge $w \longrightarrow v$ in G, $w' \in S_i$, has u_{S_i} as its source in G'.

A.3 Basic Tree Terminology

A **spanning tree** $T = \langle V_G, E_T \rangle$ of a connected undirected graph $G = \langle V_G, E_G \rangle$ is an induced acyclic subgraph of G.

A **rooted (directed) tree** $T = \langle V_T, E_T, r \rangle$ is a connected, acyclic graph with the following properties:

- r is a distinguished vertex called the **root** such that $|pred(r)| = 0$. We sometimes denote r as r_T.
- For each $u \in V_T - \{r\}$, $|pred(u)| = 1$. Further, $p \in pred(u)$ is the **parent** of u, denoted $parent_T(u)$.
- u is an **internal** vertex whenever $|succ(u)| \geq 1$; each $s \in succ(u)$ is a **child** of u.
- u is a **leaf** whenever $|succ(u)| = 0$.
- The **level** of $u \in V_T$, denoted $level(u)$, is the length of the unique path $r \xrightarrow{*} u$.
- The **height** of T, denoted $height(T)$, is defined as $max\{level(u) | u \in V_T\}$.
- If a vertex v appears on the unique path $r \xrightarrow{*} u$ then v is an **ancestor** of u and u a **descendant** of v; when $v \neq u$, v is a **proper** ancestor of u and u a **proper** descendant of v.

For a rooted tree $T = \langle V_T, E_T, r \rangle$, the following additional notation with respect to vertices u, v is sometimes used:

- $u \; [\overset{*}{\rightarrow}) \; v$ to represent the path from u to v in T, excluding v; if $u = v$, then $u \; [\overset{*}{\rightarrow}) \; v$ is the empty path.
- $u \; (\overset{*}{\rightarrow}] \; v$ to represent the path from u to v in T, excluding u; if $u = v$, then $u \; (\overset{*}{\rightarrow}] \; v$ is the empty path.
- $u \; (\overset{*}{\rightarrow}) \; v$ to represent the path from u to v in T, excluding both u and v; if $v \in succ(u)$, then $u \; (\overset{*}{\rightarrow}) \; v$ is the empty path.

An **ordered** (rooted) tree $T = \langle V_T, E_T, r \rangle$ is one in which, for each internal vertex u, there is a total order relation $\langle succ(u), \preceq \rangle$.

For a rooted tree $T = \langle V_T, E_T, r \rangle$, we say that a vertex w is the **least common ancestor** of vertices u and v, denoted $w = lca_T(u, v)$, if w is on both paths $p : r \overset{*}{\rightarrow} u$ and $q : r \overset{*}{\rightarrow} v$ and there is no $y \neq w$ on both p and q with $level(y) > level(w)$. The notion also extends to a set of vertices $V'_T \subseteq V_T$.

A **forest** is a disjoint union of a set of trees $\{T_1, T_2, \ldots, T_n\}$.

A.4 Depth-first Search

A **depth-first search** (DFS) of a graph $G = \langle V_G, E_G \rangle$ searches G by choosing to visit the unexplored successors of the most recently explored vertex. The DFS of G produces a forest $F = \{T_1, T_2, \ldots, T_n\}$ of depth-first trees $T_i = (V_i, E_i, r_i)$ where: r_i is a vertex from which a DFS was initiated; for each $v \in V_i - \{r_i\}$, $parent(v)$ is the vertex from which v was discovered. There is a unique tree in F whenever G is a weakly connected digraph and the search is initiated at a vertex v from which all vertices are reachable. In such cases, we refer to the unique tree in the forest as the **DFS spanning tree**.

The DFS imparts a classification on the set of edges E_G as follows:

- (u, v) is a *DFS tree* edge if $u = parent(v)$ in F.
- (u, v) is a *DFS back* edge if v is an ancestor of u in F.

- (u,v) is a *DFS forward* edge if $u \neq parent(v)$ is an ancestor of v in F.
- (u,v) is a *DFS cross* edge if there is no ancestor-descendant relation between u and v in F.

A **pre-order** (**post-order**) numbering of V_G is the order in which vertices were first (last) visited during the DFS. A reverse post-order of an acyclic digraph G is also termed a **topological sort** of G in which u appears before v in the ordering if there is a path $u \overset{+}{\rightarrow} v$ in G.

A.5 The Dominance Relations

The dominance relations are used extensively at the optimisation stage of a compiler, especially to analyse the structural properties of flow graphs.

For a flow graph G, a vertex u **pre-dominates** a vertex v if every path from $\mathtt{s}$ to v includes u. In addition, a vertex u **post-dominates** a vertex v if every path from v to $\mathtt{t}$ includes u. Following is the notation used with respect to the dominance relations:

- $u \trianglerighteq v$ (respectively $u \trianglelefteq v$) to denote that u pre-dominates v (respectively u post-dominates v).
- $u \triangleright v$ (respectively $u \triangleleft v$) to denote that u **strictly** pre-dominates v (respectively u **strictly** post-dominates v); that is, $u \trianglerighteq v$ and $u \neq v$.
- $ipre(v) = u$ (respectively $ipost(v) = u$) to denote that u is the **immediate** pre-dominator of v (respectively u is the **immediate** post-dominator of v), i.e. $u \triangleright v$ and there is no vertex $y \neq u$ such that $u \triangleright y \triangleright v$.

The pre- and post-dominance relations can be succinctly represented in respective trees:

Definition 14. *The* ***pre-dominator tree*** $T^G_{pre} = \langle V_G, E_T, \mathtt{s} \rangle$ *of a flow graph G is a rooted directed tree such that:*

- $E_T = \{(ipre(u), u) | u \in V_G - \{\mathtt{s}\}\}$.

Definition 15. *The* ***post-dominator tree*** $T^G_{post} = \langle V_G, E_T, \mathtt{t} \rangle$ *of a flow graph G is a rooted directed tree such that:*

- $E_T = \{(ipost(u), u) | u \in V_G - \{\mathtt{t}\}\}$.

Construction of the dominator trees is a well studied problem. The first near linear time algorithm was proposed in [66], which has since been improved to make it run in linear time [4].

We can extend the notion of pre-dominance (and post-dominance) to relate the sets of vertices and edges of G. That is, a vertex u pre-dominates an edge $v \longrightarrow w$ if every path from s to $v \longrightarrow w$ passes through u, or alternatively, $v \longrightarrow w$ pre-dominates u if every path from s to u passes through $v \longrightarrow w$.

Another important relation used in compiler optimisation is the dominance frontier relation [32]:

Definition 16. *For a flow graph G:*

- *The **pre-dominance frontier** of a vertex v, denoted $DF_{pre}(v)$, is the set $\{y | (\exists p \in pred(y))(v \rhd p \wedge v \not\rhd y)\}$.*
- *The **post-dominance frontier** of a vertex v, denoted $DF_{post}(v)$, is the set $\{y | (\exists s \in succ(y))(v \lhd s \wedge v \not\lhd y)\}$.*

A.6 Regular Expressions

We use terminology and notation consistent with that of [111, 112].

A **regular expression** over a finite alphabet Σ is constructed from the following rules:

- λ and $\emptyset$ are atomic regular expressions denoting the empty string and the empty set, respectively. For each $a \in \Sigma$, a is an atomic regular expression.
- If R_1 and R_2 are regular expressions then $(R_1 \cup R_2)$, $(R_1 \cdot R_2)$, and (R_1^*) are compound regular expressions denoting set union, concatenation, and reflexive, transitive closure under concatenation, respectively.

The regular expressions obtained from this definition are fully parenthesized. However, parentheses are usually relaxed using the operator precedence of $*$ over $\cdot$ over $\cup$. Notation is sometimes abused: $(a \cdot b)$ is written (ab); $((a^*)a)$ is written a^+.

Each regular expression R over Σ thus represents a set $\sigma(R)$ of strings over Σ as follows:

- $\sigma(\lambda) = \{\lambda\}$; $\sigma(\emptyset) = \emptyset$; $\sigma(a) = \{a\}$ for $a \in \Sigma$.
- $\sigma(R_1 \cup R_2) = \sigma(R_1) \cup \sigma(R_2) = \{w | w \in \sigma(R_1)$ or $w \in \sigma(R_2)\}$.
- $\sigma(R_1 \cdot R_2) = \sigma(R_1) \cdot \sigma(R_2) = \{w | w \in \sigma(R_1)$ and $w \in \sigma(R_2)\}$.
- $\sigma(R*) = \cup_{k=0}^{\infty} \sigma(R)^k$, where $\sigma(R)^0 = \{\lambda\}$ and $\sigma(R)^i = \sigma(R)^{i-1} \cdot \sigma(R)$.

List of References

[1] The Nexus 5001^{TM}Forum. http://www.nexus5001.org, May 2008.

[2] H. Agrawal. Dominators, super blocks, and program coverage. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, January 1994.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principle, Techniques and Tools*. Addison-Wesley, 1986.

[4] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal of Computing*, 28(6):2117–2132, June 1999.

[5] AMD®. http://www.amd.com, May 2008.

[6] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the 15th Real-Time Systems Symposium (RTSS'94)*, December 1994.

[7] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[8] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, February 1992.

[9] I. Bate and D. Kazakov. Towards new methods for developing real-time systems: Automatically deriving loop bounds using machine learning. In *Proceedings of the 11th Conference on Emerging Technologies and Factory Automation (ETFA'06)*, September 2006.

[10] I. Bate and R. Reutemann. Worst-case execution time analysis for dynamic

branch predictors. In *Proceedings of the 16th Euromicro Conference of Real-Time Systems (ECRTS'04)*, July 2004.

[11] I. Bate and R. Reutemann. Efficient integration of bimodal branch prediction and pipeline analysis. In *Proceedings of the 11th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, August 2005.

[12] M. A. Bender and M. Farach-Colton. The lca problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN'00)*, April 2000.

[13] M. R. C. M. Berkelaar. `lp_solve` - (mixed integer) linear programming problem solver. ftp://ftp.es.ele.tue.nl/pub/lp_solve.

[14] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *Proceedings of the 25th IFAC Workshop on Real-Time Programming*, May 2000.

[15] G. Bernat, A. Colin, and S. M. Petters. Wcet analysis of probabilistic hard real-time systems. In *Proceedings of the 23rd Real-Time Systems Symposium (RTSS'02)*, December 2002.

[16] G. Bernat, M.J. Newby, and A. Burns. Probabilistic timing analysis: an approach using copulas. *Journal of Embedded Computing*, 1(2):179–194, November 2005.

[17] A. Betts, G. Bernat, Raimund Kirner, Peter Puschner, and Ingomar Wenzel. Wcet coverage for pipelines. Technical report, University of York, August 2006.

[18] F. Bodin and I. Puaut. A wcet-oriented static branch prediction scheme for real-time systems. In *Proceedings of the 17th Euromicro Conference of Real-Time Systems (ECRTS'05)*, July 2005.

[19] C. Burguière, C. Rochange, and P. Sainrat. A case for static branch prediction in real-time systems. In *Proceedings of the 11th conference on Embedded and Real-Time Computing Systems and Applications*, August 2005.

[20] A. Burns, G. Bernat, and I. Broster. A probabilistic framework for schedulability analysis. In *Proceedings of the 3rd international embedded software conference (EMSOFT'03)*, October 2003.

[21] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Link, 1997.

[22] R. Chapman. *Static Timing Analysis and Program Proof*. PhD thesis, University of York, March 1995.

[23] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, September 1994.

[24] Hella KGaA Hueck & Co. http://www.hella.com, May 2008.

[25] A. Colin and G. Bernat. Scope-tree: a program representation for symbolic worst-case execution time analysis. In *Proceedings of the 14th Euromicro Conference of Real-Time Systems (ECRTS'02)*, July 2002.

[26] A. Colin and S. M. Petters. Experimental evaluation of code properties for wcet analysis. In *Proceedings of the 24th Real-Time Systems Symposium (RTSS'03)*, December 2003.

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

[28] A. Colin and I. Puaut. A modular & retargetable framework for tree-based wcet analysis. In *Proceedings of the 13th Euromicro Conference of Real-Time Systems (ECRTS'01)*, July 2001.

[29] K. D. Cooper, T. J. Harvey, and K. Kennedy. Iterative dataflow analysis, revisited. In *Proceedings of the (PLDI'02)*, November 2002.

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[31] P. Cousot and Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by contruction or appromimation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL)*, January 1977.

[32] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, October 1991.

[33] ARM development tools. http://www.arm.com, May 2008.

[34] J. F. Deverge and I. Puaut. Wcet-directed dynamic scratchpad memory allocation of data. In *Proceedings of the 19th Euromicro Conference of Real-Time Systems (ECRTS'07)*, July 2007.

[35] J. L. Diaz, D. F. Garcia, K. Kim, C. Lee, L. L. Bello, J. M. Lopez, S. L. Min, and O. Mirabella. Stochastic analysis of periodic real-time systems. In *Proceedings of the 23rd Real-Time Systems Symposium (RTSS'02)*, December 2002.

[36] A. El-Haj-Mahmoud, A. S. AL-Zawawi, A. Anantaraman, and E. Rotenberg. Virtual multiprocessor: an analyzable, high-performance architecture for real-time computing. In *Proceedings of the international conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05)*, September 2005.

[37] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, April 2002.

[38] J. Engblom. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *Proceedings of the 9th Real-Time Technology and Applications Symposium (RTAS'03)*, May 2003.

[39] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, December 1999.

[40] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, June 2003.

[41] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of the International Euro-Par Conference on Parallel Processing*, August 1997.

[42] A. Ermedahl, F. Stappert, and J. Engblom. Clustered calculation of worst-case execution times. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems (CASES'03)*, November 2003.

[43] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise wcet determination for a real-life processor. In *Proceedings of the 1st International Workshop on Embedded Software*, October 2001.

[44] G. Frantz. Digital signal processor trends. *IEEE Micro*, 20(6):52–59, November 2000.

[45] AbsInt Angewandte Informatik GmbH. http://www.absint.com, May 2008.

[46] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system c programs. In *Proceedings of the 10th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'05)*, February 2005.

[47] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *Proceedings of the 27th Real-Time Systems Symposium (RTSS'06)*, December 2006.

[48] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, May 1984.

[49] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):557–567, July 1997.

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

[51] C. A. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proceedings of the 4th Real-Time Technology and Applications Symposium (RTAS'98)*, June 1998.

[52] C. A. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting

timing analysis by automatic bounding of loops iterations. *Real-Time Systems*, 18(2-3):129–156, May 2000.

[53] J. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Mogran Kaufmann Publishers, 2003.

[54] Intel®. http://www.intel.com, May 2008.

[55] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: computing control regions in linear time. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'94)*, June 1994.

[56] N. P. Jouppi and S. J. E. Wilton. Tradeoffs in two-level on-chip caching. In *Proceedings of the 21st symposium on Computer architecture*, April 1994.

[57] D. Jungnickel. *Graphs, Networks and Algorithms*. Springer Verlag, second edition, 2004.

[58] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 1:158–171, January 1976.

[59] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the first ACM Symposium on Principles of Programming Languages*, October 1973.

[60] S-K. Kim, R. Ha, and S. L. Min. Efficient worst case timing analysis of data caching. In *Proceedings of the 2nd Real-Time Technology and Applications Symposium (RTAS'96)*, June 1996.

[61] S-K. Kim, R. Ha, and S. L. Min. Analysis of the impacts of overestimation sources on the accuracy of worst case timing analysis. In *Proceedings of the 20th Real-Time Systems Symposium (RTSS'99)*, December 1999.

[62] R. Kirner and P. Puschner. Transformation of path information for wcet analysis during compilation. In *Proceedings of the 13th Euromicro Conference of Real-Time Systems (ECRTS'01)*, June 2001.

[63] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner. Using measurements as a complement to static worst-case execution time analysis. *Intelligent Systems at the Service of Mankind*, 2:205–226, January 2006.

[64] P. A. Laplante. *Real-Time Systems Design and Analysis*. Wiley Publishers, fourth edition, 2004.

[65] J. R. Larus. Efficient program tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[66] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flow-graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, July 1979.

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

[68] X. Li, T. Mitra, and A. Roychoudbury. Modeling control speculation for timing analysis. *Real-Time Systems*, 29(1):27–58, January 2005.

[69] X. Li, A. Roychoudbury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *Proceedings of the 25th Real-Time Systems Symposium (RTSS'04)*, December 2004.

[70] X. Li, A. Roychoudbury, and T. Mitra. Modeling out-of-order processors for wcet analysis. *Real-Time Systems*, 34(3):195–227, November 2006.

[71] Y-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM/IEEE conference on Design Automation*, June 1995.

[72] Y-T. S. Li and S. Malik. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(3):257–279, July 1995.

[73] Y-T. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[74] Y-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*, December 1996.

[75] S. Lim, Y. Bae, G. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim. An accurate worst case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[76] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(21):46–61, January 1973.

[77] Rapita Systems Ltd. http://www.rapitasystems.com/, May 2008.

[78] T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, June 1998.

[79] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th Real-Time Systems Symposium (RTSS'99)*, December 1999.

[80] Sun MicroSystems. http://www.amd.com, May 2008.

[81] A. Milenkovic, M. Milenkovic, and N. Barnes. A performance evaluation of memory hierarchy in embedded systems. In *Proceedings of the 35th Southeastern Symposium on System Theory*, March 2003.

[82] T. Mitra, A. Roychoudhury, and X. Li. Timing analysis of embedded software for speculative processors. In *Proceedings of the 15th international symposium on System Synthesis (ISSS '02)*, October 2002.

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

[84] F. Mueller. Timing predictions for multi-level caches. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems (LCTES'97)*, June 1997.

[85] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2-3):217–247, May 2000.

[86] G. Ottosson and M. Sjödin. Worst case execution time analysis for modern hardware architectures. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, June 1997.

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

[88] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, May 1991.

[89] S. M. Petters. Bounding the execution time of real-time tasks on modern processors. In *Proceedings of the 7th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'00)*, December 2000.

[90] S. M. Petters and G. Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Proceedings of the 6th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'99)*, December 1999.

[91] Mälardalen University WCET project homepage. http://www.mrtc.mdh.se/projects/wcet, May 2008.

[92] I. Puaut. Wcet-centric software-controlled instruction caches for hard real-time systems. In *Proceedings of the 18th Euromicro Conference of Real-Time Systems (ECRTS'06)*, July 2006.

[93] P. Puschner. Is worst-case execution-time analysis a non-problem? - towards new software and hardware architectures. In *Proceedings of the Euromicro International Workshop on WCET Analysis (ECRTS'02)*, June 2002.

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

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

[96] G. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):175–188, March 1999.

[97] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(5):455–490, September 2002.

[98] C. Rochange and P. Sainrat. A time-predictable execution mode for superscalar

pipelines with instruction prescheduling. In *Proceedings of the 2nd conference on Computing frontiers (CF'05)*, May 2005.

[99] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper. Faster wcet flow analysis by program slicing. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'06)*, June 2006.

[100] M. Schlett. Trends in embedded microprocessor design. *IEEE Computer*, 31(8):44–49, August 1998.

[101] J. Schneider and C. Ferdinand. Pipeline behaviour prediction for superscalar processors by abstract interpretation. In *Proceedings of the ACM SIGPLAN workshop on Languages, compilers, and tools for embedded systems (LCTES'99)*, May 1999.

[102] B. Scholtz and J. Blieberger. A new elimination-based data flow analysis framework using annotated decomposition trees. In *Proceedings of the 16th International Conference on Compiler Construction (CC'07)*, March 2007.

[103] The Eclipse SDK. http://www.eclipse.org, May 2008.

[104] V. C. Sreedhar. *Efficient Program Analysis using DJ Graphs*. PhD thesis, McGill University, 1995.

[105] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, February 2000.

[106] F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems (CASES'01)*, November 2001.

[107] J. Staschulat and R. Ernst. Worst case timing analysis of input dependent data cache behavior. In *Proceedings of the 18th Euromicro Conference of Real-Time Systems (ECRTS'06)*, July 2006.

[108] V. Suhendra, T. Mitra, R. Roychoudhury, and T. Chen. Wcet centric data al-

location to scratchpad memory. In *Proceedings of the 26th Real-Time Systems Symposium (RTSS'05)*, December 2005.

[109] V. Suhendra, T. Mitra, R. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Proceedings of the 43rd annual conference on Design automation*, July 2006.

[110] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, December 1974.

[111] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[112] R. E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, July 1981.

[113] H. Theiling. Ilp-based interprocedural path analysis. In *Proceedings of the Second International Conference on Embedded Software (EMSOFT '02)*, October 2002.

[114] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2-3):157–179, May 2000.

[115] M. Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 142(2):159–181, May 1998.

[116] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2002.

[117] N. Tracey. *A Search-Based Automated Test-Generation Framework for Safety-Critical Software*. PhD thesis, University of York, July 2000.

[118] uDraw(Graph). http://www.informatik.uni-bremen.de/uDrawGraph/, May 2008.

[119] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. *SIAM Journal of Computing*, 11(2):289–313, May 1982.

[120] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems (LCTES'01)*, August 2001.

[121] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by mean of evolutionary testing. *Real-Time Systems*, 15(3):275–298, November 1998.

[122] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, November 2001.

[123] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proceedings of the 5th International Conference on Quality Software*, September 2005.

[124] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by cfg partitioning and model checking. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)*, March 2005.

[125] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the 3rd Real-Time Technology and Applications Symposium (RTAS'97)*, June 1997.

[126] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2-3):209–233, November 1999.

[127] C. Young, N. Gloy, and M. D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[128] W. Zhao, W. Kreahling, D. Whalley, C. Healy, and F. Mueller. Improving wcet by applying worst-case path optimizations. *Real-Time Systems*, 34(2):129–152, October 2006.

[129] H. Zhu, P. A. V Hall, and J. H. R May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.