Università degli studi di Roma

La Sapienza

Facoltà di Ingegneria

Tesi di laurea in:

Ingegneria Informatica

IHPP
*An **I**ntraprocedural **H**ot **P**ath **P**rofiler*

Docente relatore:
Prof. Camil Demetrescu

Autore:
Vladislav K. Valtchev

Anno accademico: 2011/2012

Abstract

IHPP stands for *intra-procedural hot path profiler* and it is an innovative instrumentation profiler written in C++ and based on the Intel Pin framework. It allows *arbitrary* context-sensitive profiling at three different levels: at procedure level, at basic blocks *inside* procedures level and at basic blocks *among* procedures level. This is achieved using two novel data structures called k-slab forest and k-calling context forest created by four professors at the Sapienza University of Rome [1]. The idea on which are based these data structures is to generalize the concepts of *vertex* profiling, *edge* profiling and *full* context-sensitive profiling with the introduction of a k parameter which allows profiling in all intermediate points between *vertex* ($k = 0$) and *full* ($k = \infty$) context-sensitive profiling. The profiling at procedure level, called in IHPP *function mode*, allows one to understand which are the *hottest paths* or, in other terms, the most frequently activated procedure call chains in a program. The profiling of basic blocks *inside* procedures, called *intra-procedural mode*, instead, focus the study of *hot paths* on activations of basic block chains inside each procedure; in that mode, the concept of *calling context* of a basic block activation have to be reinterpreted as sequence of basic blocks (inside the same procedure) that has been activated before its activation. The *intra*-procedural mode has led to noticeable results, specially with the introduction of a *roll loops* option which allows full context-sensitive profiling ($k = \infty$) without having an unbounded output size since loops inside procedures are automatically recognized; in this way, the *intra-procedural mode* becomes a way for studying the behavior of algorithms. The last IHPP profiling level, called *inter-procedural mode*, uses the same ideas of the previously one but, this time, completely ignoring the concept of *procedure*: the hole analyzed program is considered as a set of basic blocks and their activation paths are built independently from their procedure calling context.

Contents

Chapter 1

Introduction

In a world like today's one where computers are everywhere and where Internet has more than 2.2 billion users, software has become more important than ever. Actual operating systems have to handle much many processes than in the past and every single process is often much more "heavy" than before due to many reasons including (but not limited to): the demand of users for much more complicated tasks (for example the multimedia related ones), the intensive use of software layers by the programmers, the use of high-level interpreted languages and many others. Therefore, even if performance of hardware is still growing following Moore's law and actual computers are thousands times faster than in the past, they are still *slow* sometimes and software still needs to be analyzed and optimized. One way of analyzing performance of programs is **profiling**: it consists substantially in running the target program in a controlled environment and collecting data about its behavior in order to discover possible bottlenecks in software. A basilar introduction of program analysis and various profiling techniques can be found in the chapter 2.

1.1 Actual profilers

There are many profiler types today and a discrete number of profilers for each category (as readers can see in the next chapter) but, *most of them* have two characteristics in common:

- They focus only on procedures: data such counters and timers are collected only on the procedure basis. This means that there is no way to understand what happened *inside* the procedures which caused, for example, a bottleneck.
- Data has almost no *calling context*, for example: it is possible to know that a function `a()` was called 10 times in a certain program, but it is not possible to know which were the *callers* of `a()`. Some profilers like **Valgrind** can produce data with 2 levels of context: this means it is possible to know, for example, that `a()` was called 10 times, 3 of them by `b()` and 7 of them by `c()`. This is better, but sometimes it is not enough.

Even if it is not too difficult to collect full context-sensitive data, for example by building a *calling context tree* (CCT), this is often unpractical because the amount of data collected grows too much specially due to functions recursion [1]: the CCT is often too big to be physically stored and copied in a easy way and it is too big to be human-analyzable. Therefore, in the last years a few researches in theoretical computer science proposed different approaches for solving this problem and collecting **not fully context-sensitive** data. IHPP, the project described in this paper, uses one of these approaches called ***k*-CCF** (after explained).

1.2 Motivations

The aim of creation of IHPP was to make something that allows the study of function calls with an arbitrary size of the calling context but also to go beyond the concept of procedure-only profiling: IHPP uses some of the ideas of procedure-profiling to analyze the activation of basic blocks chains *inside* procedures. This sometimes can be really useful when there is a bottleneck in the algorithm used for solving a problem: the programmer already knows which are the *slow functions* but he is still unaware of *where exactly* the problem is. An example can be a routine with 3 loop levels and many conditional statements: it is not trivial to understand where the problem is, neither to solve it. Nevertheless, even if a profiler *will not* tell the programmer *how* to solve the problem, it will tell *quickly* where is it, which is much better than nothing because it can save several human hours of work.

1.3 Contributions

In order to collect k-level context-sensitive data, IHPP uses some data structures and algorithms that *have not been* ideated by the author:

- k-SF *(k-Slab Forest)*
- k-SF construction algorithm
- k-CCF *(k-Calling Context Forest)*
- Forest *join* and inv_k operations[1]

For these great and innovative data structures and algorithms, the author warmly thanks *Giorgio Ausiello*, *Camil Demetrescu*, *Irene Finocchi* and *Donatella Firmani*, professors at the *Sapienza University of Rome*[2]. Their work, called *k-Calling Context Profiling*, has been accepted for the *27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2012).*

1.4 The document structure

In the chapter 2, the topic of *program analysis* is introduced in an encyclopedic form in order to explain basic concepts about *profiling* to non-expert readers. The chapter 3 briefly explains the k-SF and the k-CCF data structures and the algorithms for building them. The chapter 4 explains *what* IHPP does showing several example programs and their relative IHPP analysis output. The chapter 5 instead, assuming the reader has perfectly understood what IHPP does, explains *how* it is done showing various source code listings but, it also deals with some of the many complications occurred during the project developing. Finally, the chapter 6 contains only one page with the final work conclusions.

[1]These two operations should be intended as *theoretical* operations: in IHPP concrete algorithms have been developed by the author

[2]Officially, **Università degli studi di Roma "La Sapienza"**

Chapter 2

Program analysis

The short introduction chapter used the concept of *profiling* with no explanations. Instead, the goal of this chapter is to explain a *little more* about profiling contextualizing it in the more general concept of program analysis. This kind of *scientific background* has an encyclopedic form and absolutely has no claim to be a good and an exhaustive coverage about the subject since it is not the purpose of this paper.

Program analysis is the process of analyzing the behavior of computer programs. Main applications of program analysis are *program correctness checking* and *program optimization.* There are two main approaches in program analysis: static analysis and dynamic analysis. The main difference between them is that in *static* analysis nothing is executed: the analysis is conducted only by observing the program source code or the compiled program instructions; instead, the *dynamic* program analysis is based on executing the program and observing what is it doing, even in real time if possible.

2.1 Static analysis

A static analysis of a program can be done either by hand or by using another program. Information obtained by static analysis can be used in many ways, from highlighting possible coding errors to application of formal methods that mathematically prove some required properties of the algorithms used. It is necessary to say, even if this is not the right context, that, as the work of Alan Turing proved, there is no way to prove the absolute correctness of an arbitrary program because of the *halting problem* [3]. Nevertheless, there are many methods that produce estimated solutions with a good level of reliability. It is possible to mention four ways of doing static program analysis:

Model checking considers systems that have finite state or may be reduced to finite state by abstraction

Data-flow analysis is a lattice-based technique for gathering information about the possible set of values

Abstract interpretation models the effect that every statement has on the state of an abstract machine (i.e., it "executes" the software based on the mathematical properties of each statement and declaration)

Use of assertions in program code as first suggested by Hoare logic [2]

A more in deep explanation of these approaches goes too far away from the purpose of this paper.

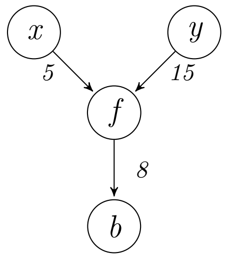


Figure 2.1: A call graph

2.2 Dynamic analysis

Dynamic program analysis is a form of analysis substantially based on the execution of the *target program* in a sort of “controlled environment”. This definition is a quite generic because there many different ways of doing this type of analysis as there are different objectives that who does the analysis wants to achieve. For example, it can be done in order to trace memory allocations (and discover memory leaks), to discover race conditions, memory corruption, security vulnerabilities and also to do a *performance analysis*, which is often called **profiling**. It is usual to refer as *profiling* when the final goal of the work is to improve the program *performance* and not, for example, to improve program *correctness*: a memory analysis is always a dynamic analysis but is not always a form of *profiling.*

2.2.1 The profiling

The profiling is probably the most common form of dynamic program analysis and its goal is, to analyze the performance of a program: the *performance* can be the amount of memory used, the frequency of certain instructions, the frequency and/or the time consumption of some *procedures / basic blocks* inside certain procedures. It is interesting here to focus the attention on the last kind of profilers; they can be classified in two ways: according to the type of output and according to the method of data acquisition. Using the first classification rule, the following distinctions can be done:

Flat profilers

These profilers count the number of function calls and/or average cpu time used by each function without keeping trace of the calling context of the function.

Call-graph profilers

These profilers do the same things flat profilers do but they also produce in output the call-chains involved on the callee, which means that it is possible to know, at the end of the execution of the target program, for every function, let it be $f()$, which functions have been called by $f()$, which functions $f()$ have called and how many times each function has called each other. With this information, it is possible to draw a *call-graph* like the one in figure 2.1; but, this is *not* context-sensitive profiling (which is what IHPP does) because, much information is missing. For example, the information produced by a *call-graph* profiler is sometime like this: $f()$ has called $b()$ 8 times and it has been called 20 times, 5 of them by $x()$ and 15 of them by $y()$; there is *not information* about how many times the distinct calling sequence $x() \rightarrow f() \rightarrow b()$ occurred and how many times the other calling sequence $y() \rightarrow f() \rightarrow b()$ occurred.

Instead, classifying profilers according to the method of data acquisition:

Event-based profilers

Some high-level languages and frameworks have they ad-hoc profilers based on events. For example, **Java** has **JVMPI** *(Java Virtual Machine Profiling Interface)*, while in **.NET** it is possible to attach a profiling agent as COM server to a .NET program using the *Profiling APIs*. These profilers are called *event-based* because statements (of the relative intermediate language) like function calls (or returns), object creations (and many others) have *traps* handled at low-level by the relative virtual machine which generates *events* and propagates these ones to the high-level user event-handlers.

Statistical profilers

This kind of profilers work by *sampling* at regular intervals the *instruction pointer* of the target program through *software interrupts*. This approach, of course, produces numerically approximate data, but allows the program to run at near full speed. Common profilers of that kind are **AMD CodeAnalyst**, **Apple Shark**, **Intel VTune** and **Intel Parallel Amplifier**.

Instrumentation profilers

This kind of profilers are used for *native programs*[1] and need to add binary instructions to the target program in order to "catch events" like function calls. Instrumentation profilers can be classified by the way they "add instructions" to the target program:

Manual This approach consists in modifying target program source code by adding *manually* additional statements in certain locations. For example, it is possible to add profiling statements at the beginning of a set of procedures and before every their `return` statement: this method of collecting data allows building function call-graphs, call context trees and much more. This method can be very reliable but it requires a considerable amount of work.

Automatic source level This approach is very similar to the last one but differs from it in the fact that profiling statements are added automatically by a tool according to an instrumentation policy.

Compiler assisted Using a *complier assisted* instrumentation means that the source code remains intact and is the compiler the one which adds profiling instructions at compile time. A practical example is the `gcc` compiler when used with `-pg` option: it produces an executable with profiling instructions but (in the specific case of `gcc`) they are executed only when the target program is executed in *profiling mode* by the specific tool `gprof`.

Binary translation This approach consist of adding instructions to an already compiled binary executable.

Runtime instrumentation In this case, the additional instructions are added at runtime after program is loaded in memory or a little before they are going to be executed by the cpu. In order to this to happen, another program which has *full control* of the target one is needed. This approach is used by tools like **Valgrind** and **Intel Pin**[2]; this last one is the tool used for the IHPP project.

[1]A native program is a program written in a compiled language like **C**, **C++**, **Pascal**: the result of the building is an executable containing architecture-specific instructions. Instead, non-native programs (often called managed programs) do not contain binary instructions: they contain intermediate-language (IL) instructions which only the relative virtual machine (VM) understand. In order to the program to run, their VM runtime compile IL instructions into machine specific instructions. **Java** and **.NET** technologies use intermediate languages called respectively **Java Bytecode** and **MSIL**.

[2]Pin's official page: `http://www.pintool.org`

Runtime injection This technique is based on the same idea of the last one but it has a more *lightweight* approach: substantially it consists of modifying the target program *text* adding unconditional branch instructions to helper functions. The tool which does this work does not have the *full control* of the target program but only partial. An example of tool which belongs to this category is **DynInst**.

Profiling through a hypervisor/simulator
This type of profilers analyze the target program by executing it with no changes in a kind of *virtual machine* which can have also some ad-hoc hardware support or it can work by literally emulating every single program instruction. This approach is not much used today. Two historical softwares which adopted this approach were IBM SIMMON and IBM OLIVER.

Chapter 3

Algorithms and data structures in IHPP

Since IHPP uses new and not yet published[1] data structures like k-SF and k-CCF, at least a basilar explanation of these ones is *strictly* necessary in this paper. Note: in order to remove every possible ambiguity about these data structures, formal definitions used in this chapter are literally taken from the original work.

operation	**curr. context**
start	⟨⟩
call(r)	⟨r⟩
call(a)	⟨r,a⟩
call(b)	⟨r,a,b⟩
return	⟨r,a⟩
return	⟨r⟩
call(c)	⟨r,c⟩
call(a)	⟨r,c,a⟩
call(b)	⟨r,c,a,b⟩
return	⟨r,c,a⟩
call(b)	⟨r,c,a,b⟩
return	⟨r,c,a⟩
return	⟨r,c⟩
return	⟨r⟩
return	⟨⟩

Figure 3.1: an execution trace

The figure 3.1 is an execution trace of a very simple imaginary program[2]; on its right part a very important information is shown: the current *calling context*. As mentioned in chapter 1, *vertex profiling* consists of counting the number of calls of a function; context-sensitive profiling instead, consists of counting the number of activations of a *path*. In order to clearly explain this concept, some formal definitions are necessary.

Definition 1: *k-calling context* [1]. Let $\pi = \langle r, ..., v\rangle$ be a calling context of v. The k-calling context of v in π is the maximal suffix of π of length at most $k + 1$.

For example, the 2-context of (r, c, a, b) is $\langle c, a, b\rangle$ and its 0-context is $\langle b\rangle$.

Definition 2: *Path activation* [1]. A path π of length q in the call graph of the program is activated by a `call(`v`)` operation if π is the q-context of v resulting from the `call` operation.

Definition 3: *k-calling context profiling* [1]. Given a trace of `call` and `return` operations, the k-(calling) context profiling problem consists of computing, for each activated path π of length $q \leq k$, the number $c(\pi)$ of times π is activated.

The figure 3.2 (below) shows, as a clarifying example, all k-contexts for $k = 0, 1, 2, 3$ of the execution trace illustrated by figure 3.1:

As it is possible to see, for various values of the k parameter there are more or less k-contexts with different values of $c(\pi)$: for $k = 0$ $c(\pi)$ is, simply, the number of times each function is called; for $k = 2$ instead, $c(\pi)$ is number of times an unique 3-elements-path like ⟨r,a,b⟩ is activated. This example is too simple to show that but, in general, as k grows, more context information is added, counter values decrease

[1]The conference in which will be presented the work *k-Calling Context Profiling* [1] will be held in October 2012, as written here: `http://splashcon.org/2012/`

[2]This example, like some other ones in this chapter, was taken, for simplicity, from the original work *k-Calling Context Profiling* [1].

k value	π *(k-context)*	$c(\pi)$ *(activation counter)*
0	⟨r⟩*	1
0	⟨a⟩	2
0	⟨b⟩	3
0	⟨c⟩	2
1	⟨r,a⟩*	1
1	⟨a,b⟩	3
1	⟨a,c⟩	1
1	⟨r,c⟩*	1
1	⟨c,a⟩	1
2	⟨r,a,b⟩*	1
2	⟨r,a,c⟩*	1
2	⟨r,c,a⟩*	1
2	⟨c,a,b⟩	2
3	⟨r,c,a,b⟩*	2

Figure 3.2: k-contexts for various values of k. Note: contexts marketed with (*) are *full* calling contexts.

and, for relatively big values of k, information made become too specific for any sort analysis so, it is a profiler's user task to choose the *right* k-value.

At this point, the problem is how to get (in terms of algorithms and data structures) all k-contexts of a program run for a given value of k: a solution is to extract the k-contexts from a CCT (*Calling Context Tree*) which is build "observing" the execution trace of the target program.

3.1 The Calling Context Tree

The CCT is a simple data structure for handling full context-sensitive information: it consists of a tree which has as root-node the *enter point* of the target program. Every procedure called is represented in the tree by a children-node of its callee. The CCT of the execution trace considered until now is shown in figure 3.3. It is self-evident that the CCT contains all k-contexts for all values of k even if it can be not obvious how a CCT is build or how k-calling contexts can be extracted from it.
An algorithm for building a CCT is the one in listing 3.1.

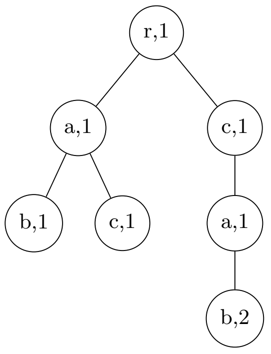


Figure 3.3: The CCT of figure 3.1. Nodes are labeled in this way: routine name, counter.

```
node treeRoot=null,currentNode=null

function func_call_event_handler(funcType func):

    if (currentNode == null) then
        currentNode = new node(func,1)
        treeRoot=currentNode
        return
    endif

    node temp = currentNode.findNodeByFuncInChildren(func)

    if (temp == null) then
        temp = new node(func,1)
        currentNode.addChildren(temp)
    else
        temp.incrementCounter()
    endif

    currentNode=temp

function func_ret_event_handler():
    currentNode=currentNode.getParent()
```

Listing 3.1: An algorithm for building a CCT

The above code should be self-explaining. The algorithm for extracting k-contexts from a CCT is a little more articulated and its pseudo-code will not be shown here but the idea is: given a CCT and a node n of it, it is possible to get the k-context of n taking the first (or at most) $k+1$ nodes from the path which joins n with the root node; doing this for every node and summing counters of all distinct paths is enough for collecting all k-contexts of a CCT.

3.2 The k-Calling Context Forest

As told in *k-Calling Context Profiling* [1], building (and handling) a CCT for a relatively long running program is unpractical and often useless. A much better data structure for handling k-contexts is *k-CCF*: the idea on which it is based is to have a *forest* formed by a tree (of at most k levels) *for each* function.

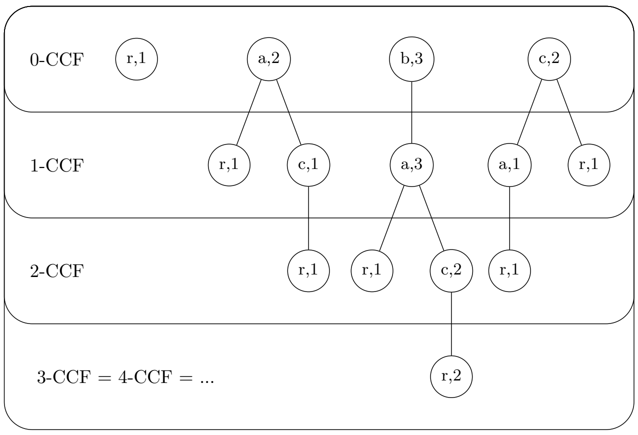


Figure 3.4: k-CCF relative to CCT in fig. 3.3 for various values of k.

The interpretation of fig. 3.4 is simple: for $k = 0$ there is no context information and only one node per function is present; its counter shows the number of times that function has been called during the program execution. For $k = 1$, every function has as children its callers, for example, $a()$ has been called 2 times: 1 by $r()$ and 1 by $c()$. This way of proceeding can be extended for greater values of k with the remark that there is always a maximum k-value which produces a full context-sensitive information (in this example, that value of k is 3): greater values of k have no effect on the output. Apart of these considerations, k-CCF have not been formally defined yet because its definition uses an operation called *tree join*.

3.2.1 The join operation

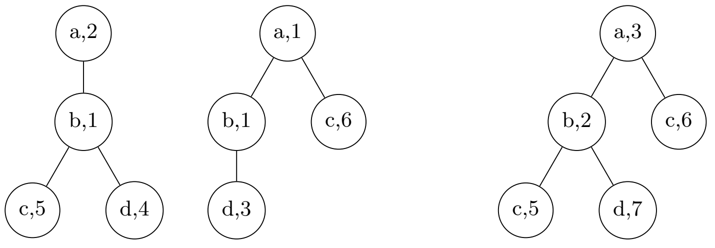
Definition 4 *Tree join* [1]. The join of two labeled trees T_1 and T_2, denoted as `join(`T_1`,`T_2`)`, is the minimal labeled forest F such that F contains a root-label path π if and only if T_1 or T_2 contains π.

Note: if T_1 and T_2 have different root labels, formally if $l(r_1) \neq l(r_2)$, then F will be simply a forest with two trees: T_1 and T_2. In order to deal with weighted trees, the join operation just defined have to be extended in this way:

Definition 4* *Weighted tree join* [1]. Let:

- T_1 and T_2 be two trees
- V_1 and V_2 be all nodes of T_1 and T_2
- $c(v)$ be a counter associated with each node v in V_1 and V_2
- F be the weighted tree join of T_1 and T_2
- z be a node of F
- π_z be the unique root-path that leads to z in F

$c(z)$ is defined as sum of all counters $c(v)$ of nodes v in $V_1 \cup V_2$ such that the root-path π_z that leads to v in T_1 or T_2 has the same sequence of labels as π_z.



(a) T_1 and T_2 (b) join(T_1,T_2)

Figure 3.5: An example of the tree join operation

Figure 3.5 shows an example that should clarify the above definition: T_1 and T_2 trees have a common root-node label, a, so they can be merged and both counters of a can be summed; also, in both trees is present a path $\langle a, b\rangle$ so the counter of node $a \to b$ in the resulting tree is the sum of the counters of that path in trees T_1 and T_2; the same logic can be used for node $a \to b \to d$. Instead, paths $\langle a, b, c\rangle$ and $\langle a, c\rangle$ are not common in both trees so they exist in the merged tree, but no counters are summed. The join operation can be extended for working with more than two trees. let (T_1, ..., T_h), with $h > 2$, be a set of trees, then: if all them have distinct root labels, the output forest F will be the same set of trees, otherwise, if for example T_1 and T_2 have the same root label, the following expression could be written:

`join(`T_1`, ..., `T_h`) = join(join(`T_1`,`T_2`),`T_3`, ..., `T_h`)`

3.2.2 The formal definition of k-CCF

At this point, k-CCF can be formally defined using its original definition.

Definition 5 *k-Calling Context Forest* [1]. The k-calling context forest of the execution of a program is a labeled forest defined as `join`(π_1^R, ..., π_s^R), where (π_1, ..., π_s) is the set of all distinct paths of length a most k+1 activated by the execution[3].

3.2.3 How to build a k-CCF

As previously stated, the CCT of a program execution contains all k-contexts for all possible values of k so, should exist a way to build a k-CCF from a CCT: in fact, this is true and can be obtained using a particular operation called k-inverse forest, formally defined below.

Definition 6 *k-inverse forest* [1]. Let F be a labeled forest with n nodes $v_1, \ldots, v_n$. For all $i \in [1, n]$, let π_i be the maximal suffix of length at most $k + 1$ of the unique root path that leads to v_i in F. The k-inverse forest of F, denoted as $inv_k(F)$, is the labeled forest obtained as `join`(π_1^R, ...,π_n^R).

Now, a k-CCF can be build from a CCT using the following property:

$$k\text{-CCF} = inv_k(\text{CCT})$$

The problem of this approach, as already stated before, is that building a CCT is *not* space-efficient: it contains all k-contexts for every value of k when, k-CCF needs only information about a specific value of k. Here comes another novel data structure called *k-Slab Forest*: instead of building a CCT from a program's execution trace, a k-SF can be build saving a great amount of space.

3.3 The k-Slab Forest

A good way to introduce the k-SF is by showing its original definition:

Definition 7 *k-Slab Forest* [1]. Let $v_1, \ldots, v_t$ be the t nodes at levels multiple of k in the CCT (including the root, which has level 0). For any $k > 0$ and each $i \in [1, t]$, let T_{v_i} be the maximal subtree of the CCT of depth at most $2k - 1$ rooted at v_i. The k-slab forest, denoted as k-SF, is the labeled forest defined by `join`(T_{v_1},...,T_{v_t}).

The above definition expresses k-SF in terms of how it can be build from a CCT. In a simpler terminology, considering only nodes of a CCT at levels multiple of k (like 0, k, 2k, ...) and their subtrees (of depth at most $2k - 1$), it is possible to obtain a k-SF by joining all them.

3.3.1 Building a k-SF from a CCT

Even if it has no practical applications, it is useful to know how to build a k-SF from a CCT in order to better understand what a k-SF is. Figure 3.6 illustrates a CCT and all its 5 subtrees at levels multiple of $k = 2$. Now, the 2-SF of this CCT can be obtained by computing `join`(T_1,...,T_5). As fig. 3.7 shows, the result is a forest with 4 trees: the first one is exactly T_1 because no other subtree has r as root-label; the second one is that obtained by `join`(T_3,T_4) and then last two ones are the one-node trees T_2 and T_5 which have not been joined because of their unique root-labels.

[3]The notation π^R is used for *reverse*-paths

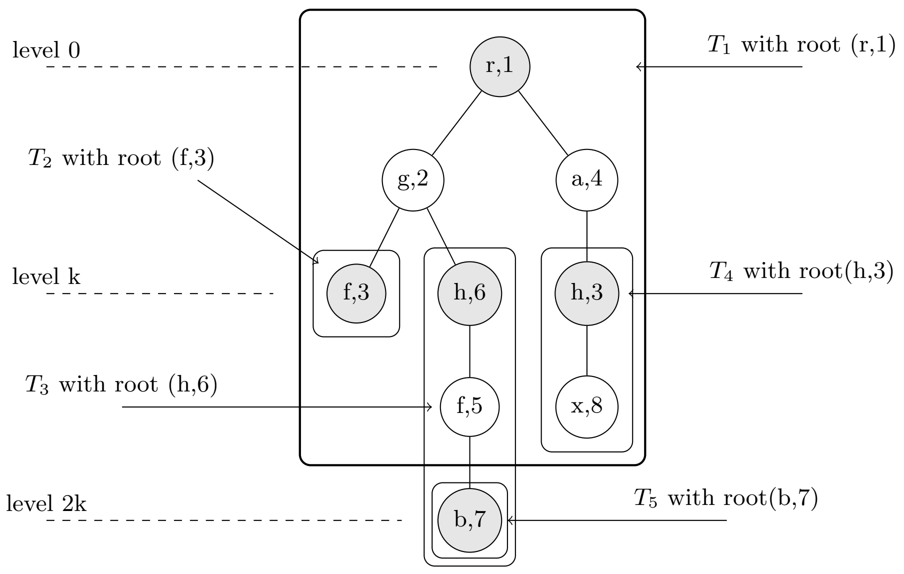


Figure 3.6: A CCT with its k-level subtrees in evidence

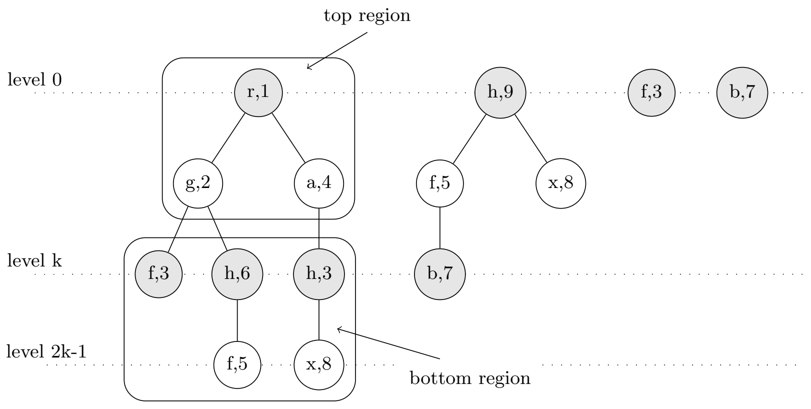


Figure 3.7: The 2-SF relative to fig. 3.6

3.3.2 Building a k-SF without a CCT

The algorithm for building *online* the k-SF is shown in listing 3.2; it is similar to the one purposed for building CCT but, instead of having one single `currentNode` pointer (and, of course, only one tree), it uses two pointers (`top` and `bottom`) that work at the same time on two different regions of the k-slab forest: the *top* region and the *bottom* region as shown in fig. 3.7. Besides the two pointers and the k-SF, the algorithm uses:

- `R`, a hashset which contains root pointers of all k-SF trees indexed by node label in a way that, given a label, it is possible to access its node-pointer in $O(1)$.
- A shadow stack `S` used for storing ⟨top,bottom⟩ pairs relative to each routine activation.

At the algorithm start, `S` contains a special pair of `null` pointers; at the first procedure call, the `if` condition on line 15 is verified: the algorithm creates the root-node, makes `top` pointing on it and adds also that pointer in `R`. In the following procedure calls, the `top` pointer is updated exactly as the `currentNode` pointer in the CCT building algorithm (lines 36-46) except when `S.size()-1` is a multiple of k: in that case (lines 16-34), `bottom` is moved to `top` and `top` is moved to a (possibly) new tree of the forest; when the "new" tree already exists (this information is provided by `R`), `top` is simply moved to its root (line 25). After the first time `top` is moved, the pointer `bottom` is not more `null` and it is updated as the `currentNode` pointer in CCT (lines 54-66).

```
 1
 2  stack S
 3  hashset R
 4  forest kSF
 5  node top=null,bottom=null
 6
 7  function init:
 8      S.push( (null,null) )
 9
10  function func_call_event_handler(funcType func):
11
12      (top,bottom) = S.top()
13
14      /* update top region */
15      if ( ( (S.size()-1) mod k ) == 0 ) then
16
17          /* bottom is moved to top and top to a "new" tree */
18
19          bottom=top
20          node temp = R.find( func )
21
22          if (temp == null) then
23
24              /* a tree with label "func" does not exist */
25              node n2 = new node( func, 0 )
26              kSF.addTree(n2)
27              R.add(n2)
28              top=n2
29          else
30
31              /* a tree with label "func" already exists */
32              top = temp
33          endif
34
35      else
36
37          /* regular top pointer update */
38
39          node temp = top.findNodeByFuncInChildren( func )
40
41          if (temp == null) then
42              temp = new node( func, 0 )
43              top.addChildren( temp )
44          endif
45
46          top = temp
47
48      endif
49
50      top.incrementCounter()
51
52      /* update bottom region */
```

```
    if (bottom != null) then

        /* regular bottom pointer update */

        node temp = bottom.findNodeByFuncInChildren( func )

        if (temp == null) then
            temp = new node( func, 0 )
            bottom.addChildren( temp )
        endif

        bottom = temp
        bottom.incrementCounter()

    endif

    /* top and bottom pointers are saved on the stack */
    S.push( (top, bottom) )

function func_ret_event_handler():

    /*
        the last procedure returned, so its record
        on the shadow stack is removed
    */

    S.pop()
```

Listing 3.2: An algorithm for building a k-SF

3.3.3 Building a k-CCF from a k-SF

A k-CCF can be build from a k-SF using the following fundamental property of k-SF:

$$k\text{-CCF} = inv_k(k\text{-SF})$$

This means that a k-CCF can be obtained from a k-SF using the *same* operation used on a CCT, as if k-SF were a generalization of CCT: this is true and in fact, for $k \to \infty$, k-SF = CCT [1]. There is only a problem in using the above property: the k-CCF build in that way has *wrong* counters. The cause of that is the intrinsic data duplication in k-SF when a new tree is added by moving the `top` pointer: `top` and `bottom` pointers build the same subtree in different places and both increment their counters. Of course, there is a solution: before applying inv_k to k-SF, all counters of top regions of k-SF, except for the first tree, must be cleared to zero. The fig. 3.8 shows the k-SF on fig. 3.7 after that operation has been applied on it.

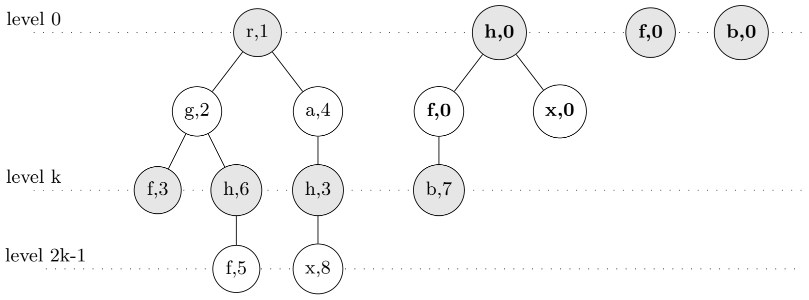


Figure 3.8: The k-SF of fig. 3.7 with cleared counters in top regions

3.4 Final remarks

In this chapter, k-CCF and k-SF data structures have been only *basically* explained: many of their properties and theorems about them have been voluntarily omitted; also, no proofs neither any kind of space/time analysis have been shown. This is because the goal of this chapter was only to explain what *practically* k-CCF and k-SF are and how they can be *technically* used in profilers such as IHPP.

The author invites readers who really want to understand these data structures and their theoretical implications to read the original work *k-Calling Context Profiling* [1].

Chapter 4

An Intraprocedural Hot Path Profiler

IHPP is a profiler written in C++. Technically it is not a program but a *plug-in*[1] for the tool Intel **Pin**[2]. This last one, is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures that allows the creation of dynamic program analysis tools. Pintools have, through pin, the full control of a target program: when this one is loaded in memory, all its routines and instructions are visible to the pintool: it can modify or simply *instrument* them. Instrumenting a routine means adding a `call` instruction to an analysis routine of the pintool at its beginning; also, it is possible to instrument `ret` instructions in order to catch `return` statements of the target program: in this way a tool can build, for example, the program's CCT. In this chapter, basics of IHPP such as its working modes and how can be used will be explained with a few technical considerations.

4.1 An overview of IHPP

IHPP has three main working modes: *function*-mode, *intra*-procedural mode and *inter*-procedural mode.

The function mode In this mode, IHPP builds a k-SF and a k-CCF for each thread of a program, with an option (called `joinThreads`) to join all k-SFs and produce a cumulative k-SF-k-CCF couple relative to the program execution. This working mode is almost a direct application of the concepts explained in chapter 3.

The intra-procedural mode This other mode instead, deals with *basic blocks* (from now called BBLs) inside procedures: the concept of *calling a function* is transformed in the *entering in a BBL*, while the concept of *returning from a function call* totally disappears in this working mode; BBLs have no stack information and so, it is impossible to "return" to a previously activated BBL instance: it is only possible to activate this BBL again though a loop or a `goto` statement. IHPP instruments all BBLs inside chosen functions of a program and produces in output a couple k-SF-k-CCF for each function executed within each thread. The `joinThreads` option is available also in this mode: it joins all k-SFs relative to a function generated by all threads in which that function has been called, for each function. Another important feature in *intra*-procedural mode is the `rollLoops` option: it substantially produces

[1]In the pin-specific slang, plug-ins like IHPP are called *pintools*.

[2]Official page: `http://www.pintool.org`

∞-SFs which do not grow due to loops inside functions because they are recognized and, as result, only counters of BBLs inside the loops are incremented.

The inter-procedural mode Despite of its name (*intra*procedural profiler), IHPP has also a particular and unique working mode that deals with BBLs but in which the concept of *procedure* is totally removed. In this working mode, a program is composed only by BBLs: there are no functions, so no calls and no returns. IHPP builds a couple of k-SF-k-CCF for each thread as other modes, but it is like the program has only one big procedure. This mode can be useful, for example, when one wants to analyze an algorithm implemented with two or more functions without considering the caller-callee relationships between them but instead he wants to focus the attention on the sequence of BBLs activated in the hole algorithm. In the case when only one function is instrumented, even if it *seems* that there is no difference between *inter*- and *intra*-procedural modes, in fact there is a *big* difference: *inter*-procedural mode ignores the function activation *context* in which a basic block is activated. This concept will be, of course, explained later.

4.1.1 Technical considerations

Like Pin itself, IHPP works both on Microsoft **Windows** and **Linux**-based systems but, output data sometimes is different and the reason is the compiler: under Linux systems the compiler `gcc` is supported instead, under Windows only the compiler `cl` included in the software **Visual Studio** is supported. This means that both the IHPP and the target program have to be compiled with the same compiler under a specific platform. Under Windows systems, target programs built with `gcc` portings or other compilers are not supported due to problems with debug information: practically, IHPP fetches the debug information of a program through Pin which accepts only `pdb` debug info format under Windows and only `dwarf` format under Linux-based systems; nevertheless, the problematic difference between the two compilers is another: they compile C/C++ code in a different way and often they do not respect the *standard calling conventions*, specially in auxiliary routines inserted into the target program. Because of this, a huge amount of work has been done in IHPP to overcome these problems, specially under Windows systems.

4.2 The *function* mode

As said before, *function* mode (internally called `funcMode`) is a way of profiling very similar to the one theoretically explained in chapter 3. A concrete example will be helpful to show this.

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void a(int x); void b(); void c();
void d(); void e(); void f();

void a(int x) {

    if (!x) {
        b(); c();
    } else {
        b(); f();
    }
}
```

```

void b() { }
void c() { }
void f() { }
void d() { c(); c(); }
void e() { d(); c(); a(0); }

void *thread2(void *arg) { a(1); return 0; }

int main(int argc, char ** argv) {

    int r;
    pthread_t th;

    a(0); e(); e();
    r = pthread_create(&th, NULL, thread2, NULL);
    pthread_join(th, NULL);
    return 0;
}
```

Listing 4.1: prog1.c, a simple multi-threaded program

Compiling the program in listing 4.1 under a 32-bit Linux system with:

```
gcc -g prog1.c -o prog1 -g -lpthread
```

and starting IHPP analysis with[3]:

```
pin -t ihpp.so -ksf -kinf -funcMode -outfile out1.txt -- ./prog1
```

will produce as output a file called `out1.txt`[4]:

```
K value: INFINITE
=============================================================================
Thread ID: 4294967296
=============================================================================
-----------------------------------------------------------------------------
DUMP of K-SF
-----------------------------------------------------------------------------
| _start(),1
   | __libc_csu_init(),1
      | __i686.get_pc_thunk.bx(),1
      | _init(),1
         | frame_dummy(),1
         | __do_global_ctors_aux(),1
   | main(),1
      | a(),1
         | b(),1
         | c(),1
      | e(),2
         | d(),2
            | c(),4
         | c(),2
         | a(),2
            | b(),2
            | c(),2
   | _fini(),1
```

[3]Due to permission problems under Linux pin have to be run with the `-injection child` option before the `-t` option. See its official user manual for an explanation.

[4]Some lines have been removed for compactness.

```
      | __do_global_dtors_aux(),1
=============================================================================
Thread ID: 8589934593
=============================================================================
-----------------------------------------------------------------------------
DUMP of K-SF
-----------------------------------------------------------------------------

| thread2_func(),1
   | a(),1
      | b(),1
      | f(),1
```

Listing 4.2: file out1.txt

A few considerations:

– The program is multi-threaded and IHPP correctly shows this producing two different k-SFs.
– Only k-SFs are present in output because IHPP has been called without the `-kccf` option.
– k-SFs shown are in fact CCTs because `-kinf` option is used.
– The are some compiler routines that almost always will not be an object of interest for a IHPP user. Note: programs compiled with `cl` under Windows have *hundreds* of compiler/library routines even if, of course, the program *is not* compiled with *static* linking of standard libraries.

Instead, executing IHPP on the same program but specifying this time the list of functions to instrument with `-funcs a,b,c,d,e,f` produces in output:

```
K value: INFINITE
=============================================================================
Thread ID: 4294967296
=============================================================================
-----------------------------------------------------------------------------
DUMP of K-SF
-----------------------------------------------------------------------------
| __root__(),1
   | a(),1
      | b(),1
      | c(),1
   | e(),2
      | d(),2
         | c(),4
      | c(),2
      | a(),2
         | b(),2
         | c(),2
=============================================================================
Thread ID: 8589934593
=============================================================================
-----------------------------------------------------------------------------
DUMP of K-SF
-----------------------------------------------------------------------------
| __root__(),1
   | a(),1
      | b(),1
      | f(),1
```

Listing 4.3: The output of IHPP run with selective tracing

It is clear that reading this output is easier than reading output in listing 4.2 but, another thing comes out: there is `__root__`, a "new" function not shown before. It is not a really function but only a *fake* symbol that IHPP automatically adds as root of the first tree for each thread. The reason is simple: when an user specifies a list of functions, there is absolutely no guarantee that among these, one is a relative root function. One criticism towards this solution can be to say that k-SF is a *forest*, not a tree, so the fake root node can be avoided: the answer is that criticism is wrong because of two reasons:

1. ∞-SF is a CCT *by definition*: without the fake root ∞-SF will still be a forest and not a *calling context tree.*
2. Since k-SF will be wrong without a fake root (for all values of k greater than a certain $\bar{k}$), a correct k-CCF could not be generated: as explained in chapter 3, before the inv_k operation, counters of all nodes in top regions must be cleared with the exception of the first three, which has as root node, the root of the CCT relative to the program execution trace. In the example above, neither `a()` nor `e()` is eligible as CCT's root.

4.2.1 The *joinThreads* option

When invoking IHPP with the `-joinThreads` option, after the target program has ended, IHPP joins all k-SFs together before (eventually) building the k-CCF. The output of IHPP after profiling the program in listing 4.1 with this options is:

```
-------------------------------------------------------------------------------
DUMP of K-SF
-------------------------------------------------------------------------------

| __root__(),2
   | a(),2
      | b(),2
      | c(),1
      | f(),1
   | e(),2
      | d(),2
         | c(),4
      | c(),2
      | a(),2
         | b(),2
         | c(),2
```

Listing 4.4: the output of IHPP with the joinThreads option

Observing listing 4.4 should be clear that the k-SF of the second thread has been joined: `a()` and `b()` counters of the first tree has been incremented and the `f()` node has been added to the cumulative k-SF.

4.2.2 The *kccf* option

Using the `-kccf` option simply makes IHPP to produce also a k-CCF relative to each k-SF. It can be used with every value of k specified with the option `-k <value>` or it can be combined with the `-kinf` option even if this is often useless (because it is useless to compute the k-CCF of a ∞-SF which is a CCT). On the next page it is shown the *full* output of IHPP running on the program of listing 4.1 with `-joinThreads -ksf -kccf -k 2 -funcs a,b,c,d,e,f` options.

```
K value: 2
Functions count: 18
Maximum number of different threads: 2
Nodes count of k slab forest: 20
Nodes count of kCCF: 25
-----------------------------------------------------------------------------
DUMP of K-SF
-----------------------------------------------------------------------------
| __root__(),2
   | a(),2
      | b(),2
      | c(),1
      | f(),1
   | e(),2
      | d(),2
         | c(),4
      | c(),2
      | a(),2
         | b(),2
         | c(),2
| b(),2
| c(),3
| d(),2
   | c(),4
| a(),2
   | b(),2
   | c(),2
| f(),1
-----------------------------------------------------------------------------
DUMP of K-CCF
-----------------------------------------------------------------------------
| __root__(),2
| a(),4
   | __root__(),2
   | e(),2
      | __root__(),2
| b(),4
   | a(),4
      | __root__(),2
      | e(),2
| c(),9
   | a(),3
      | __root__(),1
      | e(),2
   | d(),4
      | e(),4
   | e(),2
      | __root__(),2
| f(),1
   | a(),1
      | __root__(),1
| e(),2
   | __root__(),2
| d(),2
   | e(),2
      | __root__(),2
```

Listing 4.5: a full IHPP output

Note: it is obvious that for finite values of k, IHPP produces as k-SF a *real* forest and the fake node `__root__` is the root *only* of the first tree.

4.2.3 The *unrollSimpleRec* option

Actually, with the options explained since now, IHPP does not produce an exactly standard CCT or k-SF as a profiler which uses the algorithm for building a k-SF shown in chapter 3 will do. IHPP, by default, compresses (rolls) subtrees generated by recursions of single functions. Listing 4.6 shows an appropriate example:

```c

void foo(); void bar();

void recFunc(int n) {

    if (!n) { bar(); return; }
    recFunc(n-1);
}

void foo() { recFunc(5); }
void bar() { }

int main(int argc, char ** argv) { foo(); return 0; }
```

Listing 4.6: prog2.c, a simple program that uses recursion

The ∞-SF of program in listing 4.6 produced by IHPP is:

```
| __root__(),1
   | main(),1
      | foo(),1
         | recFunc(),6
            | bar(),1
```

Listing 4.7: an example of k-SF *without* the `unrollSimpleRec` option

It is clearly evident that recursion of `recFunc()` is not explicitly shown. Using the `-unrollSimpleRec` option instead:

```
| __root__(),1
   | main(),1
      | foo(),1
         | recFunc(),1
            | recFunc(),1
               | recFunc(),1
                  | recFunc(),1
                     | recFunc(),1
                        | recFunc(),1
                           | bar(),1
```

Listing 4.8: an example of k-SF *with* the `unrollSimpleRec` option

The output shown in listing 4.8 is the *real* CCT of the program in listing 4.6 but it is not more useful than the one in listing 4.7 even if it is longer: readers would imagine what will happen if `foo()` had called `recFunc()` with $n = 10000$. Of course, using a finite value of k will solve the problem but some information will be lost. An important consideration: when `-unrollSimpleRec` is not used, *only* recursions like the one just shown will be rolled, not the more complex recursions like those generated by the program in listing 4.9 (on the next page).

```

void bar(int n);

void foo(int n) {

    if (!n) return;
    bar(n-1);
}

void bar(int n) {

    if (!n) return;
    foo(n-1);
}

int main(int argc, char ** argv) { foo(5);  return 0; }
```

Listing 4.9: prog3.c, a program that uses complex recursions

Indeed, independently of the option `-unrollSimpleRec`, IHPP produces this output for `prog3`:

```
| __root__(),1
   | main(),1
      | foo(),1
         | bar(),1
            | foo(),1
               | bar(),1
                  | foo(),1
                     | bar(),1
```

Listing 4.10: IHPP partial output for `prog3`

4.3 The *intra*-procedural mode

In the *intra*-procedural mode (internally called `intraMode`), IHPP traces BBLs (basic blocks) activations inside procedures building a couple k-SF-k-CCF for each procedure. Nevertheless, before proceeding, a BBL definition is necessary because the concept of BBL used in IHPP is a quite different from the classical one.

4.3.1 The basic blocks

The classical definition of BBL that can be found in textbooks is: *a BBL is a single entrance, single exit sequence of instructions.* This definition can be perfectly applied to the concept of BBL used in IHPP, even if IHPP's BBLs are different from the classical ones. The following example will explain why.

```

    switch(condition_var)
    {
        case 3: other_var++;
        case 2: other_var++;
        case 1: other_var++;
        case 0:
        default: break;
    }
```

Listing 4.11: a switch statement

The `switch` statement with fall-through cases in listing 4.11 will be usually compiled (for the IA-32 architecture) as something like this:

```
.L10:
    add dword [ebp-4], 1
.L9:
    add dword [ebp-4], 1
.L8:
    add dword [ebp-4], 1
```

Listing 4.12: assembly code relative to listing 4.11

According to the classical definition of BBL, listing 4.12 contains three distinct BBLs, one for each instruction. Instead, in IHPP BBLs are still three but the first one, starting at `.L10`, contains all the three instructions; the second one, starting at `.L9`, contains the last two `add` instructions and the last one is formed only by the last `add` instruction. The reason of this different BBL concept is that IHPP is based on **Pin** and uses BBLs as provided by it. Pin, instead, has no way to "extract" classic BBLs from a compiled program because they can be only discovered at runtime: for example, when a branch (that could be indirect) sets the *instruction pointer* to the second `add` at `.L9`, Pin has no way to know in advance that in the future (and this is not certain) some other branch will set the instruction pointer to `.L8` so, the BBL starting at `.L9` contains both `add` instructions.

4.3.2 *Intra*Mode basics

When run with the `-intraMode` option, IHPP traces BBLs inside all functions or inside the chosen ones, like in the *function* mode. Every BBL is identified in output by:

- the function's name which the BBL belongs to
- the offset BBL address - function address in bytes
- the line and the column of the first statement in program source code that belongs to the BBL[5].

Listing 4.13 shows a program that is going to be analyzed.

```
#include <stdio.h>

void p(int n) {

    printf("%d\n", n);
}

void foo(int n) {

    int i;

    if (n) {

        p(0);

        for (i=0; i < 3; i++) {
            p(1);
        }

```

[5]In order all this data to be obtained from the executable, it *must* be build with debug info as told in the chapter's beginning: programs without debug info cannot be analyzed by IHPP, independently from the working mode.

```
20          foo(0);
21
22      } else {
23          p(2);
24      }
25  }
26
27  int main(int argc, char ** argv) { foo(1); return 0; }
```

Listing 4.13: prog4.c, another simple program

Running IHPP with `-intraMode -ksf -k 100 -funcs foo,p` produces:

```
==============================================================================
Function: p()
------------------------------------------------------------------------------
DUMP of K-SF
------------------------------------------------------------------------------
| {p+0 at 3,0},5
   | {p+26 at 6,0},5
==============================================================================
Function: foo()
------------------------------------------------------------------------------
DUMP of K-SF
------------------------------------------------------------------------------
| {foo+0 at 8,0},2
   | {foo+12 at 14,0},1
      | {foo+24 at 16,0},1
         | {foo+49 at 16,0},1
            | {foo+33 at 17,0},1
               | {foo+45 at 16,0},1
                  | {foo+33 at 17,0},1
                     | {foo+45 at 16,0},1
                        | {foo+33 at 17,0},1
                           | {foo+45 at 16,0},1
                              | {foo+55 at 20,0},1
                                 | {foo+67 at 20,0},1
                                    | {foo+81 at 25,0},1
   | {foo+69 at 23,0},1
      | {foo+81 at 25,0},1
```

Listing 4.14: IHPP partial output for `prog4`

A few preliminary considerations about listing 4.14:

- All column values are zero: this happens when column information is not available but it is not necessarily a whole program problem: some BBLs could have this info and others could not, depending from the compiler.
- In this example, as in many others, only k-SFs with big values of k are shown because they provide full information analysis. This approach is the best for simple programs that practically allows it. However, in this particular example there is a specific reason to use $k = 100$ instead of $k = \infty$: IHPP does not allow `-kinf` option alone in *intra*Mode because output size is potentially unbounded. The only way for using `-kinf` option is combined with `-rollLoops` which will be explained later.

An interpretation Comparing the output in listing 4.14 with the program in listing 4.13 it is possible to observe a good matching between them, for example, the loop in lines 16-17 is clearly visible; also, the behavior of the `if-then-else` statement in `foo()` is evident. In particular, recognizing this last one is not trivial: IHPP has to understand that a recursion has occurred and lines 23-25 are not executed after line 20 but instead, in a new function activation. From a theoretically point of view, this is *equivalent* to building a different k-SF for *each* function activation and then joining all them. Practically, IHPP achieves it in a better way: it substantially uses a *shadow stack* to trace function activations in which a couple of pointers (`top`,`bottom`) is stored for each function activation. Details of this approach will be explained in the next chapter.

Attentive readers would have probably noticed that there are two BBLs on line 16 and other two ones on line 20: even if it is clear that they are distinct BBLs due to their function offset, it is not obvious what they exactly do. Since there is no other way than looking machine instructions to understand what a BBL do, IHPP provides an option to do this.

4.3.3 Inside a BBL: the disassembly utility in IHPP

Using the option `-blocksDisasm` together with the option `-showBlocks` when calling IHPP, makes it to write at the end of the output file a table that contains for each BBL, four columns: its string name (used in k-SFs), its memory address, the absolute number of times it has been called (vertex profiling) and a kind of "special" disassembly. This last column is a post-elaborated form of the Intel-style disassembly that Pin provides to IHPP: usually, in disassembly no labels are used in call and branch instructions and they appear something like this: `jle 0x8048400`. IHPP, in order to help the analyst who uses it, replaces all addresses with human readable strings. The listing 4.15 shows that.

```
------------------------------------------------------------------------------
All basic blocks
------------------------------------------------------------------------------


{foo+24 at 16,0} addr: 0x8048418 counter: 1
Disassembly:
                                             mov dword ptr [ebp-0xc], 0x0
                                             jmp {foo+49 at 16,0}

{foo+33 at 17,0} addr: 0x8048421 counter: 3
Disassembly:
                                             mov dword ptr [esp], 0x1
                                             call p

{foo+45 at 16,0} addr: 0x804842d counter: 3
Disassembly:
                                             add dword ptr [ebp-0xc], 0x1
                                             cmp dword ptr [ebp-0xc], 0x2
                                             jle {foo+33 at 17,0}

{foo+49 at 16,0} addr: 0x8048431 counter: 1
Disassembly:
                                             cmp dword ptr [ebp-0xc], 0x2
                                             jle {foo+33 at 17,0}

{foo+55 at 20,0} addr: 0x8048437 counter: 1
Disassembly:
                                             mov dword ptr [esp], 0x0
                                             call foo
```

```
{foo+67 at 20,0} addr: 0x8048443 counter: 1
Disassembly:
                                        jmp {foo+81 at 25,0}
```

Listing 4.15: a part of "all basic blocks" table

Observing the listing 4.15, the reason why there are two BBLs on line 16 comes out: the BBL at `foo+24` is the `i=0` initialization statement of the `for` loop, instead, the BBL at `foo+45` contains the `i++` statement and a conditional branch (which makes the loop to continue) taken when `i <= 2`. In this case, as it happens very often, machine instructions do not follow the order of `C` statements. BBLs on line 20 instead, are contiguous but are separated due to a `call` instruction at the end of the first one.

Advanced features The IHPP disassembly system has an advanced feature that can be shown in the analysis of the program in listing 4.13 run under Windows systems; the disassembly code relative to the Linux BBL at `{foo+55}` that it produces is:

```
{foo+56 at 20,0} addr: 0xcc1088 counter: 1
Disassembly:
                                    push 0x0
                                    call .text+10 --> jmp {foo+0 at 8,0}
```

Listing 4.16: an example of a two-step call

This means that `foo()` does not call `p()` function *directly*, but instead it calls an *not well-defined* area of the `.text` section of the program situated ten bytes from the beginning; in this location, a direct jump instruction makes the program to run the `p()` routine. Since practically all user functions are called in this indirect way when compiling with `cl` compiler programs with debug info, the implementation of this feature was strictly necessary. IHPP can handle also the hypothetical situation in which a call is done through two or more indirect jumps.

The full disassembly of routines IHPP offers also the opportunity to show at the end of the output file a table containing all routines using the option `-showFuncs` and their disassembly code adding also the `-funcsDisasm` option. In *intra-* and *inter-*procedural modes, both options (`showFuncs` and `showBlocks`) can be used, instead, in *function* mode only the `showFuncs` option can be used.

4.3.4 The *rollLoops* option

Running IHPP with the `-rollLoops` option on the program in the listing 4.13 produces for the `foo()` function the k-SF in the listing 4.17 (below):

```
---------------------------------------------------------------------------
| {foo+0 at 8,0},2
   | {foo+12 at 14,0},1
      | {foo+24 at 16,0},1
         | {foo+49 at 16,0},1
            | {foo+33 at 17,0},3
               | {foo+45 at 16,0},3
                  | {foo+55 at 20,0},1
                     | {foo+67 at 20,0},1
                        | {foo+81 at 25,0},1
   | {foo+69 at 23,0},1
      | {foo+81 at 25,0},1
```

Listing 4.17: an output of IHPP with rollLoops option

It is clearly visible that the `for` loop on lines 16-17 has been compressed or, in the specific slang, *rolled*. The `-rollLoops` option, implies the `-kinf` option that means $k = \infty$ but, with the guarantee that the output size is limited.

4.3.5 The *joinThreads* option in *intraMode*

The concept of the `-joinThreads` option is the same as in the *function* mode even if an example will make it clear.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *foo(void *arg) {

    int i=0;

    while(1) {

        if (arg) {

            if (i >= 3)
                break;
        } else {

            if (i >= 9)
                break;
        }

        printf("%i\n",i);
        i++;
    }

    return 0;
}

int main(int argc, char ** argv) {

    pthread_t th;

    foo((void*)1);

    pthread_create(&th, NULL, foo, NULL);
    pthread_join(th, NULL);

    return 0;
}
```

Listing 4.18: prog5.c, an example program

The IHPP analysis of the program in listing 4.18 with the options `-intraMode -rollLoops -func foo` produces the following output[6]:

```
===============================================================================
Thread ID: 4294967296
===============================================================================
```

[6]A great deal of output lines have been omitted

```
| {foo+0 at 5,0},1
   | {foo+19 at 13,0},4
      | {foo+33 at 21,0},3
         | {foo+53 at 22,0},3
            | {foo+13 at 11,0},3
      | {foo+25 at 14,0},1
         | {foo+60 at 25,0},1

============================================================================
Thread ID: 8589934593
============================================================================

| {foo+0 at 5,0},1
   | {foo+27 at 17,0},10
      | {foo+33 at 21,0},9
         | {foo+53 at 22,0},9
            | {foo+13 at 11,0},9
      | {foo+59 at 18,0},1
```

Listing 4.19: the output of IHPP analyzing `prog5`

Instead, using the `-joinThreads` option, the output is:

```
============================================================================
Function: foo()
============================================================================
| {foo+0 at 5,0},2
   | {foo+19 at 13,0},4
      | {foo+33 at 21,0},3
         | {foo+53 at 22,0},3
            | {foo+13 at 11,0},3
      | {foo+25 at 14,0},1
         | {foo+60 at 25,0},1
   | {foo+27 at 17,0},10
      | {foo+33 at 21,0},9
         | {foo+53 at 22,0},9
            | {foo+13 at 11,0},9
      | {foo+59 at 18,0},1
```

Listing 4.20: the output of IHPP analyzing `prog5` with `-joinThreads`

Even if the interpretation of the outputs at the point *should* be trivial, some complications not correlated with the *joinThreads* option has came out. The function `foo()` is executed once per thread; in the main thread, the loop does 3 iterations and then exists going to line 14, instead in the second one, the loop does 9 iterations and then exists going to line 18. A first problem: it is unclear why BBL at line 11 is activated *after* and not *before* BBL at line 22. The answer can be found only by looking at the disassembly code in listing 4.21.

```
{foo+0 at 5,0} addr: 0x80484c4 counter: 2
Disassembly:
                                                push ebp
                                                mov ebp, esp
                                                sub esp, 0x28
                                                mov dword ptr [ebp-0xc], 0x0
                                                cmp dword ptr [ebp+0x8], 0x0
                                                jz {foo+27 at 17,0}
{foo+13 at 11,0} addr: 0x80484d1 counter: 12
Disassembly:
                                                cmp dword ptr [ebp+0x8], 0x0
```

```
                                                  jz {foo+27 at 17,0}

{foo+19 at 13,0} addr: 0x80484d7 counter: 4
Disassembly:
                                                  cmp dword ptr [ebp-0xc], 0x2
                                                  jle {foo+33 at 21,0}

{foo+25 at 14,0} addr: 0x80484dd counter: 1
Disassembly:
                                                  jmp {foo+60 at 25,0}

{foo+27 at 17,0} addr: 0x80484df counter: 10
Disassembly:
                                                  cmp dword ptr [ebp-0xc], 0x8
                                                  jnle {foo+59 at 18,0}

{foo+59 at 18,0} addr: 0x80484ff counter: 1
Disassembly:
                                                  nop
                                                  mov eax, 0x0
                                                  leave
                                                  ret

{foo+60 at 25,0} addr: 0x8048500 counter: 1
Disassembly:
                                                  mov eax, 0x0
                                                  leave
                                                  ret
```

Listing 4.21: a part of "all basic blocks" table of `prog5`

The BBL `{foo+0 at 5,0}` includes in itself the BBL `{foo+13 at 11,0}`: that is the reason why BBL on line 11 is not activated suddenly after the BBL on line 5; instead, it is activated at every loop iteration after BBL on line 22 (not shown in the listing 4.21). The same "problem" occurs for BBLs on lines 18 and 25: as already explained, the origin of this unusual concept of BBL is the real-time BBL identification: when the program was running at `foo+13` (included in BBL at `foo+0`), **Pin** had no way to understand that in the future a BBL at `foo+53` will branch to it.

Apart of these considerations about BBLs, as it can be seen in listing 4.20, the behavior of the *joinThreads* option is exactly what one expects: it produces as output the forest join of the k-SFs related to each thread, or better, in this particular case, the *tree* join of the two CCTs.

4.4 The *inter*-procedural mode

When launched with the `-intraMode` option[7], IHPP instruments only BBLs and builds *inter*-procedural k-SFs, one for each thread. Each k-SF, contains BBLs which belong to chosen functions or, eventually, to all functions of the main image of the target program. The below listings show an example program and its relative IHPP analysis output for $k = 100$.

```
void bar(int n); void f3() { }

void foo() {

   bar(0);
```

[7]The inter-procedural mode is often called internally in IHPP `interProcMode` instead of `interMode` to avoid misunderstanding problems correlated with `intraMode`.

```c
    f3();
    bar(1);
}

void bar(int n) {

    if (n)
        f3();
}
int main(int argc, char ** argv) { foo(); return 0; }
```

Listing 4.22: prog6.c, an example program

```
| {__root__+0 at 0,0},1
   | {foo+0 at 3,0},1
      | {bar+0 at 10,0},1
         | {bar+14 at 14,0},1
            | {foo+18 at 6,0},1
               | {f3+0 at 1,0},1
                  | {foo+23 at 7,0},1
                     | {bar+0 at 10,0},1
                        | {bar+9 at 13,0},1
                           | {f3+0 at 1,0},1
                              | {bar+14 at 14,0},1
                                 | {foo+35 at 8,0},1
```

Listing 4.23: partial output of IHPP analysis in `interProcMode` of `prog6`

Observing the output in listing 4.23 should be enough to understand what the *inter*-procedural mode does: a *k*-SF is build as if there were no routines at all. An important consideration about this IHPP working mode is that even if only one produce is instrumented, the output will be different from the one made by `intraMode`; a proof for this is given by listing 4.24 which shows the IHPP analysis output of program in listing 4.13, already analyzed in `intraMode` in the listing 4.14.

```
-----------------------------------------------------------------------------
DUMP of K-SF
-----------------------------------------------------------------------------

| {__root__+0 at 0,0},1
   | {foo+0 at 8,0},1
      | {foo+12 at 14,0},1
         | {foo+24 at 16,0},1
            | {foo+49 at 16,0},1
               | {foo+33 at 17,0},1
                  | {foo+45 at 16,0},1
                     | {foo+33 at 17,0},1
                        | {foo+45 at 16,0},1
                           | {foo+33 at 17,0},1
                              | {foo+45 at 16,0},1
                                 | {foo+55 at 20,0},1
                                    | {foo+0 at 8,0},1
                                       | {foo+69 at 23,0},1
                                          | {foo+81 at 25,0},1
                                             | {foo+67 at 20,0},1
                                                | {foo+81 at 25,0},1
```

Listing 4.24: partial output of IHPP analysis in `interProcMode` of `prog4`

It is evident that the recursion of `foo()` has absolutely no effect: the BBL `{foo+0 at 8,0}` follows the BBL `{foo+55 at 20,0}` as if it was a simply loop iteration.

A last note: IHPP supports also *rollLoops* and *joinThreads* options in `interProcMode`, even if no examples are provided here.

4.5 The *XML* output feature

IHPP has been designed to be *extensible* and since data exchange is on the basis of that principle, IHPP provides an alternative output format.

Using the `-xml` option, IHPP will provide its output in **XML** format which is nowadays one of most supported data exchange languages. The IHPP-specific XML language is defined in a computer interpretable form too, using the **XML Schema** language. The specific **XSD** file that contains the output language definition is called `outputschema.xsd` and it is located in the `doc/` directory of the project. A short extract from the `outputshema.xsd` document is shown in the listing 4.25 below.

```
<xs:element name="threads">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="thread" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="id"/>
            <xs:element name="kSF" type="forest" minOccurs="0"/>
            <xs:element name="kCCF" type="forest" minOccurs="0"/>
            <xs:element name="intraMode_ctx" minOccurs="0">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="func_ctx" maxOccurs="unbounded">
                    <xs:complexType>
                      <xs:sequence>
                        <xs:element name="funcAddr" type="addrType"/>
                        <xs:element name="kSF" type="forest" minOccurs="0"/>
                        <xs:element name="kCCF" type="forest" minOccurs="0"/>
                      </xs:sequence>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Listing 4.25: an extract from `outputschema.xsd`

Chapter 5

The implementation

Actually, the IHPP project is written in about 4,900 lines of `C++` code[1]: even only copying down its full source code will take about 84 pages using this font. Therefore, the goal of this chapter is not to explain all IHPP source code line by line, but instead is to give to the reader a panoramic overview of the project implementation, showing its architecture, its key-files and the approaches adopted for solving problems occurred. Of course, there will be source code listings, but almost always the code will be *purged* from the lines unessential for specific context in which the code is presented.

5.1 A general overview of the project

As can be seen in fig. 5.1, IHPP has been implemented in 26 files, grouped by areas of competence. The `main.cpp` file is where the enter point of the profiler is located and contains, in addition to the initialization code, all *instrumentation routines.* These last ones substantially place *callbacks* in the target program to functions located in files of group (3): functions in these files use context objects defined in files of groups (4), tracing objects defined in files of group (2) and specially the `traceObject()` function implemented in `traceObjFunc.cpp`. This last one, is substantially the function that implements the algorithm for building a k-SF from an execution trace shown in listing 3.2: when called, it *traces* the activation of a generic `TracingObject<T>` in the given *context.* The *context* depends from the working mode: in `funcMode`, for example, the *context* is pointer to a `ThreadContext` object; instead, in `IntraMode` the *context* is a pointer to a `IntraModeContext` object. In order to simplify everything, *abstraction* is used: both `ThreadContext` and `IntraModeContext` classes are specializations of the `GenericTraceContext` class, exactly as `FunctionObj` and `BasicBlock` classes are specializations of the generic `TracingObject<T>` class.

[1]All source code lines are included in this calculation, even ones which contain comments and the empty ones.

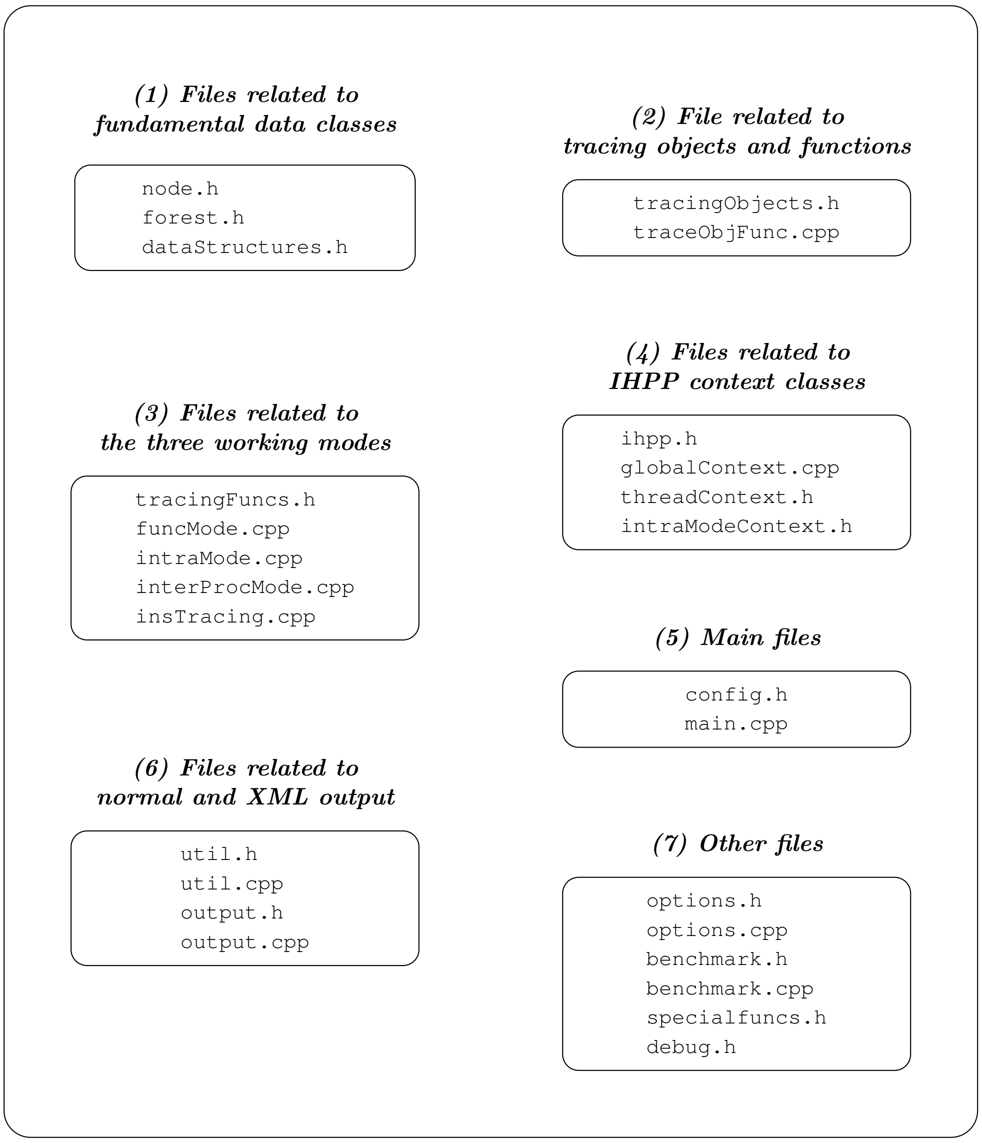


Figure 5.1: File organization of the project

The `traceObject()` function mentioned before, works with basilar objects defined in files of group (1). The first two files in this last group use on their basis container classes for `ObjectWithKey<T>` objects defined and implemented in the file `dataStructures.h`. `ObjectWithKey<T>` is the most abstract class in IHPP since it is a generalization of the abstract class `TracingObject<T>`. Nodes and forests data structures are implemented respectively in `node.h` and `forest.h` files. The concept of *tree* is not explicitly modeled in IHPP, but it is modeled inside the `node<T>` class instead.

A technical note Files in groups (1) and (2) are mainly *headers* because they contain *template* classes which cannot be complied like others; the `traceObject()` function instead, is not a template function because of an optimization strategy: it is a function pointer which is assigned at runtime to the *faster* `traceObject` implementation, specially to speeding up the $k = \infty$ case.

Figure 5.2 shows an UML-like diagram which summarizes some of the class relations just explained.

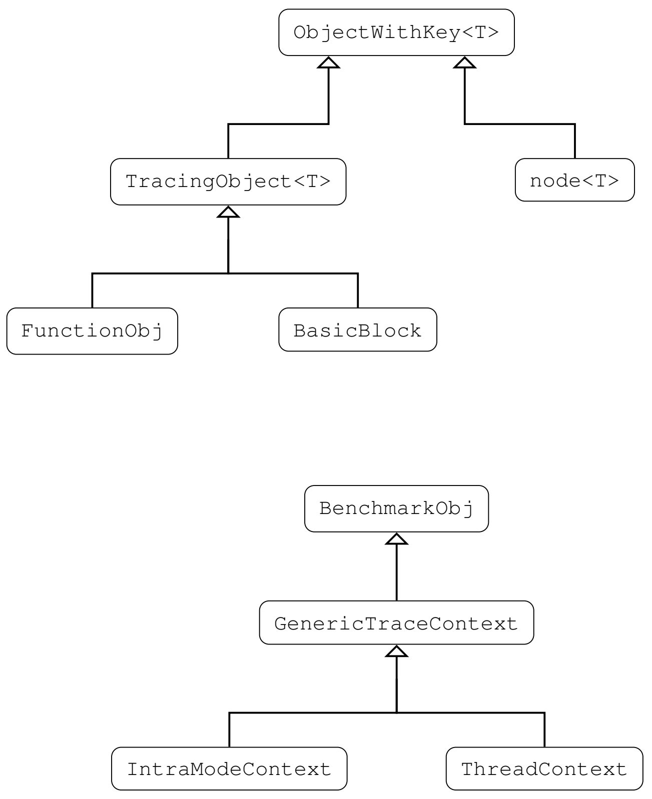


Figure 5.2: some important class relationships

The output generation When the target program ends, the `Fini()` *callback* routine located in `output.cpp` is called. It writes the output, as shown in chapter 4, using supporting routines in `output.cpp` and `util.cpp` files. When the `-kccf` option is used, supporting routines in `output.cpp` use the `inverseK()` method of class `forest<T>` to produce a k-CCF from each k-SF. The `-joinThreads` option makes output routines to do a preventive *join* operation on all k-SFs, leaving the result in the first thread context.

The *instruction* tracing mode The shortly called *insTracing* mode is a hidden part of IHPP never mentioned before now. It handles the instrumenting of *all* machine instructions contained in chosen functions in order to improve correctness of outputs produced by `funcMode` and `intraMode` in *extreme* conditions, when *calling conventions* are not respected by the compiler, for example when a call to a routine is done with a `jmp` instruction instead of a `call` one, or when particular functions like `longjmp()` are used. This *feature* consists of a *set* of several approaches to the problem of correctly understanding the execution trace of the program but, since this is really *heavier* than the simply function instrumentation, it is active only when IHPP is launched with the `-insTracing` option.

Other features IHPP has implemented in it a tiny benchmark framework which allows to count the number of nodes and forests created and copied (during the target program execution) within each thread or even within each function in each thread. For that reason, the `GenericTraceContext` class derives from the `BenchmarkObj` class. Of course, benchmark is a *heavy* feature too (disabled by default) which can be enabled in the `config.h` file. Also, IHPP provides a *debug* mode which writes selective debug information to the `stderr` output stream.

5.2 The context classes

The knowledge of context classes is essential to understand the IHPP implementation.

5.2.1 The *GlobalContext* class

The main IHPP context class is `GlobalContext`: it contains all globally shared variables in the program and it is instanced only once. Even if the *singleton* design pattern could had be used for it, for performance reasons it is instanced as a simply global symbol called `globalSharedContext` defined in `ihpp.h`. Listing 5.1 shows a great part of the `GlobalContext` implementation.

```
class GlobalContext {

    PIN_LOCK lock;
    /* code omitted [...] */

public:

    ofstream OutFile;

    set<string> funcsToTrace;
    set<ADDRINT> funcAddrsToTrace;

    BlocksMap allBlocks;
    FuncsMap allFuncs;
    vector<ThreadContext*> threadContexts;

    /* code omitted [...] */

    optionsClass options;
    specialAttrs spAttrs;

    GlobalContext(WorkingModeType wm, unsigned kval, optionsClass options);
    ThreadContext *getThreadCtx(PIN_THREAD_UID tid);

    WorkingModeType WorkingMode() { return _WorkingMode; }
    unsigned int kval() { return _K_CCF_VAL; }

    inline bool hasToTrace(ADDRINT funcAddr);

    /* code omitted [...] */
};
```

Listing 5.1: partial definition of `GlobalContext`

As told in the chapter introduction, source code will not be explained line by line. The most important properties and methods in this case are briefly explained below.

allBlocks is a *hashmap* of all BBLs (`BasicBlock` objects) indexed by their address (of type `ADDRINT`, a Pin macro that stands for a pointer-size unsigned integer)

allFuncs is a *hashmap* containing all functions of the main image of the program, not only the ones that have to be traced

funcsAddrsToTrace is a set of all functions that have to be traced, indexed by their address. When all functions of the main image of the program have to be traced, this set, like the set `funcsToTrace`, will have no elements

threadContexts is a vector containing pointers to all `ThreadContext` objects. A `std::vector<T>` is used intentionally instead of a `std::map` because usually programs have a small number of threads: iterating over all them is better than using a *hashmap* which works in $O(1)$ but has a big hidden constant

getThreadCtx(...) is the fundamental method of this class used in all working modes: given the current *thread_id*, it returns a pointer to its relative thread context object or, if this one does not exists, creates a new thread context object and then returns it.

5.2.2 The *GenericTraceContext* class

The `GenericTraceContext` class is the basic context used by the `traceObject()` function. Listing 5.2 shows its definition.

```
class GenericTraceContext: public BenchmarkObj {

public:

    ADDRINT rootKey;
    ihppForest kSlabForest;
    unsigned int counter;

    ihppStack<ShadowStackItemType> shadowStack;

    GenericTraceContext() : rootKey(0), counter(0)
    {
        //code omitted [...]
    }

};
```

Listing 5.2: partial definition of `GenericTraceContext`

It contains the essential elements explained in chapter 3 for building a couple k-SF - k-CCF: a k-slab forest, a shadowStack and a `rootKey` variable used to identify the first tree of the k-SF. The `R` map, explained in chapter 3, is not necessary here because the `forest<T>` class used as the type[2] of `kSlabForest` uses itself a hashmap for tree roots searches. The `counter` variable, instead, is a new concept: it is used by the `traceObject()` function as a replacement of the expression `S.size()-1` used in listing 3.2 on line 15. The reason of `counter` is that the `shadowStack` size does not grow neither in `intraMode` nor in `interMode` when new BBLs are activated but the algorithm needs a growing variable. Actually, `counter` is always incremented by every `traceObject()` invocation; also `funcMode` uses this counter instead of `shadowStack.size()` but, its value is decreased on stack `pop()` actions.

5.2.3 The *IntraModeContext* class

The `IntraModeContext` class is no more than a `GenericTraceContext` with an attribute `functionAddr`. Its full definition is shown in listing 5.3.

[2]`ihppForest` is simply a `typedef` of `forest<T>` with `T = ADDRINT`.

```
class IntraModeContext : public GenericTraceContext {

    ADDRINT funcAddr;

public:

    IntraModeContext(ADDRINT functionAddr) :
        GenericTraceContext(), funcAddr(functionAddr) { }

    ADDRINT getFunctionAddr() { return funcAddr; }
};
```

Listing 5.3: definition of `IntraModeContext`

5.2.4 The *ThreadContext* class

The `ThreadContext` class, as mentioned before, is the main tracing context dedicated to each thread of the target program; a partial definition of it is shown in the listing 5.4 below; note: this class has many properties, specially Windows-specific and related to *insTracing* ones, which are not shown in here. It is a derived class of `GenericTraceContext` because it *is* a tracing context from the `funcMode`'s routines point of view. Instead, in `intraMode` it is used properly as a *thread context* and the *right* intraMode context is obtained by invoking its `getCurrentFunctionCtx()` method. An important concept to be understood is that for each thread there is always a *current function* in which the thread in running: the address of that function is kept by `ThreadContext` in the variable `currentFunction` which is *mainly* set by `funcMode` routines. This means that a `funcMode` layer runs also in `intraMode`, in order to help it to correctly understand which is the current function instance that executes a specific BBL.

The `treeTop` and `treeBottom` properties are used, as a comment says, only by `InterProcMode` routines: as it will be explained later, `InterProcMode` does not need a *shadow stack* but instead a context that provides only three elements: top and bottom pointers and a k-SF object.

```
class ThreadContext : public GenericTraceContext {

    ADDRINT currentFunction;
    PIN_THREAD_UID threadID;

public:

    //WM_InterProcMode properties
    ihppNode *treeTop;
    ihppNode *treeBottom;

    //code omitted [...]

    //WM_IntraMode properties
    map<ADDRINT, IntraModeContext*> intraModeContexts;

    //Methods
    ThreadContext(PIN_THREAD_UID tid, /* code omitted [...] */);

    //code omitted [...]

    IntraModeContext *getFunctionCtx(ADDRINT funcAddr);
    IntraModeContext *getCurrentFunctionCtx();
    ADDRINT getCurrentFunction();
    IntraModeContext *setCurrentFunction(ADDRINT currFunc);
    string getCurrentFunctionName();

    PIN_THREAD_UID getThreadID() { return threadID; }
};
```

Listing 5.4: partial definition of `ThreadContext` class

5.3 Nodes and forests

5.3.1 The *node* class

A fundamental IHPP class is `node<T>` or better, `node<keyT>`: it is the basic element used in data structures such as k-SFs and k-CCFs. A `node` object is a *container* for *one* value-object of type `ObjectWithKey<keyT>*` and *one* integer counter. Besides, a `node<keyT>` object is identified among other nodes by a *key* of type `keyT`, a *template* parameter. In IHPP, `keyT` is always `ADDRINT` which is, as already told, a Pin macro that stands for a pointer size unsigned integer, even if, `keyT` can be any type, for example also a `std::string`.

The relationship between `TracingObject<keyT>` and `node<keyT>` classes is that the first ones are always used as node *values*, thus practically nodes are a *transport* for *tracing objects* inside trees and forests.

Since `node` objects act as *trees* in IHPP, they have to transport also their *children* nodes. As can be seen in listing 5.5, *children* nodes are contained using a `ihppNodeChildrenContainer<keyT,valueT>` object.

```
template <typename keyT>
class node : public ObjectWithKey<keyT> {

private:
    node<keyT> *parent;

protected:

    ObjectWithKey<keyT> *val;
    obj_counter_t counter;

    ihppNodeChildrenContainer< keyT, node<keyT> > children;

    //code omitted [...]

public:

    typedef typename
    ihppNodeChildrenContainer< keyT,
                            node<keyT> >::iterator nodesIterator;

    //code omitted [...]

    node(keyT key, ObjectWithKey<keyT>* val,
                                obj_counter_t counter=0);

    node<keyT>* getChildRef(keyT k);
    node<keyT>* addChild(node<keyT> &n);
    node<keyT>* getParentRef() { /* omitted */ }
    //code omitted [...]

    size_t childrenCount() { return children.size(); }

    //code omitted [...]

    keyT getKey() { /* omitted */ }
    ObjectWithKey<keyT> * getValue() { /* omitted */ }

    nodesIterator getNodesIteratorBegin() { /* omitted */ }
    nodesIterator getNodesIteratorEnd() { /* omitted */ }

    void incCounter() { counter++; }
    obj_counter_t getCounter() { return counter; }
    void setCounter(obj_counter_t c) { counter=c; }

    node<keyT> kpathR(unsigned int k);

    //code omitted [...]
    node<keyT> &operator=(node<keyT> n);
};
```

Listing 5.5: partial definition of `node<keyT>` class

Technically, `ihppNodeChildrenContainer` is not a real type but a macro which stands for `ihppNodeChildrenContainerList1`. This last one, is one of the actually *three* node container classes defined in `dataStructures.h`. Other container classes has exactly the same interface but store data internally in other ways, for example using a hashmap. The linked list is the preferred data structure for children nodes because they are almost always only a few per node: hashmap, as can be empirically proved, is slower the linked list in this particular case. Containers that *relocate* data such as *dynamic arrays* cannot be used as children containers for `node<keyT>` class because it stores `node` objects into the container (and not *pointers* to node objects) and then uses a node-*pointer* for the *parent* node: if the *parent* node is moved in the memory by the children nodes container, all its children will have an invalid `parent` reference. This approach, apart of this little problem, is better than manually using pointers to `node` objects because fully delegates the memory management problem to the container: when a `node` object is destroyed the `children` object is automatically destroyed too with all its nodes; therefore, full tree copies are very simply in this way since the `node<keyT>` class has its overload of `operator=()` like the `ihppNodeChildrenContainer` has its own: no explicit memory allocations in `node<keyT>` are necessary.

A technical note. Methods directly implemented in class definitions like the ones in `node<keyT>` class, are *implicit inline* methods with *no overhead* compared to *public* properties.

5.3.2 The *forest* class

The `forest<keyT>` class is very similar to `node<keyT>`, with the main difference that it is not identified by a key, neither transports an object pointer as value. It is substantially a container for nodes and can use exactly the same children nodes containers used by `node<keyT>`. Actually, it uses `ihppNodeChildrenContainerMap` as container because for small values of *k* in `funcMode`, a forest can have hundreds trees and a hashmap will have better performance then a linked list. Listing 5.6 (below) shows a partial definition of the `forest<keyT>` class.

```
template <typename keyT>
class forest {
    ihppNodeChildrenContainerMap< keyT, node<keyT> > trees;
    static void joinSubtrees(node<keyT> &t1, node<keyT> &t2);

public:

    //code omitted [...]

    typedef typename ihppNodeChildrenContainerMap< keyT,
                        node<keyT>  >::iterator treesIterator;

    forest();
    forest(const forest &f);
    forest(node<keyT> n);

    node<keyT> * getTreeRef(keyT k);

    //code omitted [...]

    inline node<keyT> * addTree(node<keyT> &n);

    //code omitted [...]
    treesIterator getTreesIteratorBegin() { /* omitted */ }
    treesIterator getTreesIteratorEnd() { /* omitted */ }

    //code omitted [...]

    forest<keyT> inverseK(unsigned int k);

    void local_join(node<keyT> &t2);
    void local_join(forest<keyT> &f2);
    void local_joinByVal(node<keyT> t2) { local_join(t2); }

    //code omitted [...]

    inline forest<keyT> &operator=(forest<keyT> f);
};
```

Listing 5.6: partial definition of `forest<keyT>` class

5.3.3 The inv_k operation

The inv_k operation described in chapter 3 is implemented in the `inverseK()` method of the `forest<keyT>` class and uses the *join* operation implemented in the `local_join()` method which belongs to the same class too. Listings 5.7 and 5.8 shows full code of *join* and inv_k operations with no explanations. Only a remark: `kpathR()` method of `node` class returns a *degenerated* tree (a list) made by the first (or at most) k ancestors of the node on which the method is called.

```
template <typename keyT>
void forest<keyT>::joinSubtrees(node<keyT> &t1, node<keyT> &t2) {

    node<keyT> *t;
    typename node<keyT>::nodesIterator it;

    t1.setCounter(t1.getCounter() + t2.getCounter());

    for (it = t2.getNodesIteratorBegin();
            it != t2.getNodesIteratorEnd(); it++)
    {

        t = t1.getChildRef(it->getKey());

        if (!t)
            t1.addChild(*it);
        else
            joinSubtrees(*t, *it);
    }
}

template <typename keyT>
void forest<keyT>::local_join(node<keyT> &t2) {

    node<keyT> *t;
    t = getTreeRef(t2.getKey());

    if (t)
        joinSubtrees(*t, t2);
    else
        addTree(t2);
}
```

Listing 5.7: the implementation of the *join* operation

```
template <typename keyT>
forest<keyT> forest<keyT>::inverseK(unsigned int k) {

    treesIterator it;
    forest<keyT> res;
    vector< node<keyT> * > tmp;
    typename vector< node<keyT> * >::iterator it2;

    for (it = getTreesIteratorBegin();
            it != getTreesIteratorEnd(); it++)
    {
        it->autoSetParents();
        tmp = it->getAllTreeNodesRef();

        for (it2 = tmp.begin(); it2 != tmp.end(); it2++)
            res.local_joinByVal((*it2)->kpathR(k));
    }

    return res;
}
```

Listing 5.8: the implementation of the inv_k operation

5.4 Basics of *function* mode

5.4.1 The instrumentation

As the reader would know at this point, everything starts in the `main()` routine where function pointers to instrumentation routines are given to `Pin`; listing 5.9 shows these lines of code.

```
int main(int argc, char ** argv) {
    //much code omitted [...]

    if (globalSharedContext->WorkingMode() != WM_FuncMode) {
        TRACE_AddInstrumentFunction(BlockTraceInstrumentation, 0);
    }

    IMG_AddInstrumentFunction(ImageLoad, 0);

    //much code omitted [...]
}
```

Listing 5.9: a fragment of main() routine

As the above code shows, only the `ImageLoad()` function is added to instrumentation functions in `funcMode`. Listing 5.10 shows a fragment of the `ImageLoad()` function which is called by **Pin** every time the target program loads a new image[3].

```
void ImageLoad(IMG img, void *) {

    GlobalContext *ctx = globalSharedContext;

    //code omitted [...]

    for (SEC sec = IMG_SecHead(img); SEC_Valid(sec); sec = SEC_Next(sec))
    {

        for (RTN rtn = SEC_RtnHead(sec); RTN_Valid(rtn); rtn = RTN_Next(rtn))
        {

            ADDRINT funcAddr = RTN_Address(rtn);
            string funcName = RTN_Name(rtn);

            //code omitted [...]

            fc = new FunctionObj(funcAddr, funcName, fileName);
            ctx->allFuncs[funcAddr]=fc;

            bool trace = ctx->hasToTraceByName(funcName, funcAddr);

            if (trace || FUNC_IS_TEXT_N(funcName)) {
                imageLoad_doInsInstrumentation(img, rtn, fc);
            }

            if (!trace)
                continue;

            if (ctx->WorkingMode() != WM_InterProcMode)
            {
                RTN_Open(rtn);
                RTN_InsertCall(rtn,
                    IPOINT_BEFORE, AFUNPTR(FunctionObjTrace),
                    IARG_CALL_ORDER, CALL_ORDER_FIRST,
                    IARG_PTR, (ADDRINT)fc,
                    /* code about stack ptr omitted */
                    IARG_END);
                RTN_Close(rtn);
            }
```

[3]An *image* is a file with compiled symbols (routines and constant variables), for example a `.so` (or `DLL` under Windows) dynamic library. An executable file is the *main* image of a program.

```
        }
    }

    //code omitted [...]
}
```

Listing 5.10: a fragment of ImageLoad() routine

The *instrumentation* of the function on address `funcAddr` is achieved by calling the Pin `RTN_InsertCall()` routine. As it is clearly evident, a `FunctionObj` object pointer is also passed as argument to the `RTN_InsertCall()` routine. Now, after the end of `ImageLoad()`, every time the target program enters in the `funcName` routine, the `FunctionObjTrace()` function will be called with the `fc` pointer passed as first argument. The function *return* is caught by instrumenting all `ret` instructions within it: `ImageLoad` calls `imageLoad_doInsInstrumentation()` function to do this.

```
void imageLoad_doInsInstrumentation(IMG &img,
                            RTN &rtn, FunctionObj *fc)
{
    //code omitted [...]
    RTN_Open(rtn);
    for( INS ins = RTN_InsHead(rtn);
            INS_Valid(ins); ins = INS_Next(ins) )
    {
        if (ctx->WorkingMode() != WM_InterProcMode)
            insInstrumentation(rtn, ins);

        //much code omitted [...]
    }
    RTN_Close(rtn);
}
```

Listing 5.11: a fragment of `imageLoad_doInsInstrumentation()` routine

The above function in listing 5.11 calls `insInstrumentation()` at every routine instruction and, even if it not shown above, it does many other operations related to saving disassembly code of instructions.

```
void insInstrumentation(RTN rtn, INS ins) {

    if (INS_IsRet(ins) && !FUNC_IS_TEXT(RTN_Address(rtn))) {

        INS_InsertPredicatedCall(ins,
                IPOINT_BEFORE, (AFUNPTR)funcMode_ret,
                IARG_CALL_ORDER, CALL_ORDER_FIRST, IARG_END);
    }

#if ENABLE_INS_TRACING
    if (!globalSharedContext->options.insTracing)
        return;

    //much code omitted [...]
#endif
}
```

Listing 5.12: a fragment of `insInstrumentation()` routine

As listing 5.12 shows, when `ins` variable is a `ret` instruction, it is *instrumented* with the `funcMode_ret()` routine.

5.4.2 The *funcMode* routines

Once instrumentation is finished, only two routines are called[4] at runtime in `funcMode`: `FunctionObjTrace()` and `funcMode_ret()`.

```
void FunctionObjTrace(FunctionObj *fc, /* code omitted */) {

    ihppNode *treeTop=0;
    ihppNode *treeBottom=0;
    ThreadContext *ctx;
    GlobalContext *globalCtx = globalSharedContext;

    ctx = globalCtx->getThreadCtx(PIN_ThreadUid());

    //code omitted [...]

    ctx->setCurrentFunction(fc->functionAddress());
    fc->incSimpleCounter();

    //code omitted [...]

    /* code about stack ptr check omitted */

    if (ctx->shadowStack.size()) {
        treeTop = ctx->shadowStack.top().treeTop;
        treeBottom = ctx->shadowStack.top().treeBottom;
    }

    //code omitted [...]

    traceObject(fc, ctx, treeTop, treeBottom);

    //code omitted [...]

    ctx->shadowStack.push(ShadowStackItemType(treeTop,treeBottom));

    /* code about forward jump recognition omitted */
}
```

Listing 5.13: a fragment of `FunctionObjTrace()` routine

As can be seen in listing 5.13, the *essence* of `FunctionObjTrace()` function is simply: for first the current thread context is obtained then, the current function is set and the `simpleCounter` property of `fc` is incremented[5]; then *top* and *bottom* pointers are loaded from the *shadow stack* if it is not empty and then finally the fundamental `traceObject()` function is called passing to it all strictly necessary parameters: the pointer to the `FunctionObj` to be traced, the current context and the two *top* and *bottom* pointers. After that, the *top* and *bottom* pointers are stored to the *shadow stack*. The implementation of the `traceObject()` function will not be shown here because it is very similar to the pseudo-code in listing 3.2.

The `funcMode_ret()` routine implementation is conceptually very simple.

```
void funcMode_ret()
{
    ThreadContext *ctx;
```

[4]Instrumentation routines are technically called by the target program which instructions have been modified by Pin in order call them

[5]This counter is a sort of "absolutely reliable" counter variable contained in each `TracingObject`: it has the benefit to be totally independent from k-SFs, k-CCFs and their join operations.

```
    ctx = globalSharedContext->getThreadCtx(PIN_ThreadUid());

    /* code about forward jump recognition omitted */
    //other code omitted [...]

    ctx->shadowStack.pop();

    if (globalCtx->WorkingMode() == WM_IntraMode)
        intraMode_ret();

    //code omitted [...]

    if (ctx->shadowStack.size())
        ctx->setCurrentFunction(
            ctx->shadowStack.top().treeTop->getValue()->getKey()
        );
}
```

Listing 5.14: a fragment of `funcMode_ret()` routine

Listing 5.14 shows a small fragment of `funcMode_ret()`: it substantially pops the *shadow stack* and sets as current function the previously one.

These would be really the full implementations of `funcMode` routines only in a "perfect world": practically in IHPP they are much more bigger because they have to handle *unusual* situations like the one which is just going to be explained.

5.4.3 The *long jump* problem

The `ANSI C` standard library provides a couple of functions called `setjmp()` and `longjmp()` that allows to change the program *context*: they are used mainly for handling exceptions because the `C` language has no a built-in method for doing this. When a *long jump* occurs, the *context* saved by `setjmp()` is restored and the program continues its execution from the point on which `setjmp()` was called. Listing 5.15 shows an example program that uses *long jumps*.

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf jump_buffer;

void testJmp(int par) {

    if (par)
        longjmp(jump_buffer, 1);

    printf("par is zero!\n");
}
int main(int argc, char ** argv) {

    if (!setjmp(jump_buffer)) {

        testJmp(1);
        printf("this message will not be printed!\n");
    } else {
        testJmp(0);
    }
    return 0;
}
```

Listing 5.15: source code of `prog7.c`

At the beginning, the `jump_buffer` is set in `main()` by `setjmp()` which returns `0` (the first time); then `testJmp(1)` is called which makes a *long jump* again to the `if` condition in `main()` but, this time `setjmp()` returns `1` since a *jump* is happened and the code in `else` statement is executed. The second call of `testJmp()` is a normal function call. Therefore, the `main()` function has called `testJmp()` twice but, since no `ret` instruction has been executed by `testJmp()` when `par` was one, if run the `funcMode` analysis as described since now, it will produce in output this:

```
-----------------------------------------------------------------------------
DUMP of K-SF
-----------------------------------------------------------------------------

| __root__(),1
   | main(),1
      | testJmp(),1
         | testJmp(),1
```

Listing 5.16: an example of the *longjmp* problem

The profiler has assumed *wrongly* that since no `ret` instruction has been executed by `testJmp()`, the second call is a recursion of it. Results like this one can be obtained with IHPP using `-unrollSingleRec` option after rebuilding it with all advanced checks in `config.h` disabled.

A first good solution

The first solution implemented in IHPP to solve problems like these, consisted of storing in the *shadow stack* also the value of the *stack pointer*[6] and comparing it with the actual *stack pointer* value every time the function `FunctionObjTrace()` is called using this logic: since the *stack* grows towards zero and `call` instructions make it to grow because they push on it the current *instruction pointer* value, the current *stack pointer* value should be *lesser* then the previously stored one[7]. If the condition just stated is false, than something like a *long jump* should be happen so, the solution is to `pop()` the `shadowStack` until a greater value of *stack pointer* is found.

This is implemented in `FunctionObjTrace()` by calling (in a point where code is omitted in listing 5.13) an *inline* function shown below:

```
#if ENABLE_RELY_ON_SP_CHECK
inline void funcMode_sp_check(ThreadContext *ctx, ADDRINT reg_sp)
{
    if (reg_sp >= FUNCMODE_TOP_STACKPTR())
    {
        while (ctx->shadowStack.size() > 1 &&
                reg_sp >= FUNCMODE_TOP_STACKPTR())
        {
            if (!ctx->popShadowStack())
                break;
        }
    }
}
#endif
```

Listing 5.17: implementation of `funcMode_sp_check()`

However, even if this approach to the problem is *really fast* compared to others later explained, it works perfectly only under two conditions. The first condition simply requires that routines *must* be called with the `call` instruction or an equivalent

[6]register `ESP` on IA-32 or register `RSP` on amd64 architectures
[7]obtained by `shadowStack.top().reg_sp`

sequence which pushes the *instruction pointer* on the stack, decreases its value and then jumps to the function address; even if it seems very strange, there are situations in which this does not happen: these not so unusual situations will be shown later.

The second condition is: if the compiler of the target program *strictly* follows the `cdecl` calling convention for *variadic* functions (which means passing arguments in inverse order using the `push` instruction), they must not use *long jumps*. The program in listing 5.18 shows a situation in which the *stack pointer check* method does not work with some compilers.

```c
#include <stdio.h>
#include <setjmp.h>
#include <stdarg.h>

jmp_buf jump_buffer;

void testJmp(int par, ...) {

    va_list ap;

    if (par)
        longjmp(jump_buffer, 1);

    printf("par is zero!\n");

    va_start(ap, par);

    while (par != -1) {

        par = va_arg(ap, int);
        printf("arg: %i\n", par);
    }
    va_end(ap);
}

int main(int argc, char ** argv) {

    if (!setjmp(jump_buffer)) {

        testJmp(1);
        printf("this message will not be printed!\n");

    } else {

        testJmp(0, 25, 100, 386, -1);
    }

    return 0;
}
```

Listing 5.18: source code of `prog8.c`

The reason why the *stack pointer check* method does not work is that the *stack pointer* value in the second call of `testJmp()` is *lesser* than (instead of be *greater* or equals to) its value in the first one because it has been called with 4 more 32-bit arguments which subtracted 16 bytes from the *stack pointer* so, the check does not find nothing strange and the situation is equivalent (from the point of view of the stack pointer movement) as if `testJmp()` had been called itself with 3 arguments. This would not have happened if `testJmp()` was also called the first time with the same number of arguments because in that case, the two *stack pointer* values would have been equals and the *shadow stack* would had been popped by `funcMode_sp_check()`. Note: even if this situation can happen also on x86-64 architectures, it is more improbable because the standard calling convention used on these architectures is to pass the first six arguments through registers and only the seventh argument and the following ones on the stack; this means that *variadic* routines called with less than seven arguments move the *stack pointer* only by 8 bytes and this problem does not occur.

An interesting consideration related to this problem is that, as told before, the problem occurs when the compiler follows the `cdecl` convention for *variadic* functions: the `cl` compiler included with the software package *Microsoft Visual Studio 2010* which often uses particular optimizations (also in debug mode) for built-in routines, compiles (in debug mode, with no other options) user functions in the `prog8.c` exactly as one expects using `push` instructions and so the problem just shown occurs. The below listing shows the `main()` routine compiled by `cl`.

```
1001dd0    push ebp
1001dd1    mov ebp, esp
1001dd3    push 0x0
1001dd5    push 0x100ee40
1001dda    call __setjmp3
1001ddf    add esp, 0x8
1001de2    test eax, eax
1001de4    jnz main+47
1001de6    push 0x1
1001de8    call .text+5 --> jmp testJmp
1001ded    add esp, 0x4
1001df0    push 0x100d01c
1001df5    call printf
1001dfa    add esp, 0x4
1001dfd    jmp main+68
1001dff    push 0xffffffff
1001e01    push 0x182
1001e06    push 0x64
1001e08    push 0x19
1001e0a    push 0x0
1001e0c    call .text+5 --> jmp testJmp
1001e11    add esp, 0x14
1001e14    xor eax, eax
1001e16    pop ebp
1001e17    ret
```

Listing 5.19: `main()` disassembly of `prog8.c` compiled with `cl`

Instead, the `gcc 4.6` compiler does not use `push` instructions, but writes the parameters directly to the *stack red zone* with `mov` instructions as the listing 5.20 shows.

```
80484d4    push ebp
80484d5    mov ebp, esp
80484d7    and esp, 0xfffffff0
80484da    sub esp, 0x20
80484dd    mov dword ptr [esp], 0x804a040
80484e4    call .plt+48
80484e9    test eax, eax
80484eb    jnz main+51
80484ed    mov dword ptr [esp], 0x1
80484f4    call testJmp
80484f9    mov dword ptr [esp], 0x8048628
8048500    call .plt+64
8048505    jmp main+95
8048507    mov dword ptr [esp+0x10], 0xffffffff
804850f    mov dword ptr [esp+0xc], 0x182
```

```
8048517    mov dword ptr [esp+0x8], 0x64
804851f    mov dword ptr [esp+0x4], 0x19
8048527    mov dword ptr [esp], 0x0
804852e    call testJmp
8048533    mov eax, 0x0
8048538    leave
8048539    ret
```

Listing 5.20: `main()` disassembly of `prog8.c` compiled with `gcc`

Paradoxically, in this situation the *stack pointer check* method works great because the *stack pointer* value is the same in both times `testJmp()` is called: this means that the second `testJmp()` call cannot had be done by `testJmp()` itself so, a `long jump` has occurred.

Conclusions

It is possible to say that the *stack pointer check* is a good and *light* method for catching particular events such as *long jumps* and that actually it *should* always work for programs compiled with `gcc`. Instead, it could produce wrong results with Windows programs compiled with `cl` in rare situations (such as the one just shown) when only user routines are instrumented but, when *all* routines of a program compiled with `cl` are instrumented (IHPP called without the `-funcs` option) this method produces *totally* wrong results: the reason is that, as will be explained later, built-in routines in `cl`-compiled programs calls each other sometimes using simply the `jmp` instruction and also `ret` instructions can be replaced with unconditional jumps; in addition to all this, there are cases of fall-though in routines which means that the program enters in a routine *without* any form of *branch*, by simply executing the next instruction. In order to *partially* solve these problems, a mode called `insTracing` has been implemented in IHPP which instruments all machine instructions of the program: even if neither in this way a 100% correct k-SF can be build, results are much better than without it and a percentage of correctness about 90% is achieved.

5.5 The *intra*-procedural mode

As explained for the `funcMode`, everything starts with the *instrumentation*: as can be seen in listing 5.9, the instrumentation routine for `IntraMode` and `InterProcMode` is `BlockTraceInstrumentation()`. The main difference between the function instrumentation and the BBL instrumentation is that this last one is done only when the target program is *already* running because BBLs cannot be identified before. The listing 5.21 (below) shows a partial implementation of the instrumentation function.

```
void BlockTraceInstrumentation(TRACE trace, void *)
{
    //code omitted [...]
    map<ADDRINT,BasicBlock*>::iterator it;

    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        PIN_LockClient();
        blockPtr = BBL_Address(bbl);

        /* code that stores in funcAddr the address of
            the routine that contains the BBL, omitted */

        /* code that checks if the BBL should be traced or not, omitted */

        it = ctx->allBlocks.find(blockPtr);
        if (it == ctx->allBlocks.end()) {

            INS ins = BBL_InsTail(bbl);
            ADDRINT lastAddr = INS_Address(ins);
            FunctionObj *fc = ctx->allFuncs.find(funcAddr)->second;

            bb = new BasicBlock(blockPtr, fc, lastAddr, row, col);
            ctx->allBlocks[blockPtr]=bb;

        } else {
            bb = it->second;
        }

        if (ctx->WorkingMode() == WM_InterProcMode)
            BBL_InsertCall(bbl,
                                IPOINT_BEFORE, AFUNPTR(interModeBlockTrace),
                                IARG_PTR, bb,
                                IARG_END);

        if (ctx->WorkingMode() == WM_IntraMode)
            BBL_InsertCall(bbl,
                                IPOINT_BEFORE, AFUNPTR(intraModeBlockTrace),
                                IARG_CALL_ORDER, CALL_ORDER_LAST,
                                IARG_PTR, bb,
                                /* code about stack ptr check omitted */
                                IARG_END);

        PIN_UnlockClient();
    }
}
```

Listing 5.21: partial implementation of `BlockTraceInstrumentation()`

As it can be seen, the instrumentation is not very different from the one of *routines* but, this time instead of iterating over *sectors* of an *image* and then over *routines*, there is directly an iteration over BBLs of a *trace*[8]. Differently from the function instrumentation, here it can happen that a more than one trace have a BBL on the same address so, a preliminary check is done to avoid the creation of a new `BasicBlock` object *but* the instrumentation of the BBL *must* be done again because Pin makes distinction between the same BBL on different traces; technically the logic BBL is the same but the `bbl` object is different.

Another peculiarity that differentiates the `intraMode` from the `funcMode` is that only one `intraMode` routine is instrumented: `intraModeBlockTrace()`; nevertheless, `intraMode` has a *ret* routine too but it is not directly instrumented: as already explained, `intraMode` is conceptually a *layer* over `funcMode` so, is this last one that notices `intraMode` routines when a function "returnes". As already seen, the concept of *function return* is a quite "virtual": sometimes it is not referred to a *real* `ret` instruction but instead to a *long jump* or something like that.

5.5.1 How *intraModeBlockTrace()* works

Listing 5.22 (below) shows a partial implementation of `intraModeBlockTrace()`.

```
void intraModeBlockTrace(BasicBlock *bb, /* omitted */) {

    IntraModeContext *intraCtx;
```

[8]A *trace* is a single entrance, multiple exit sequence of instructions.

```
    ihppNode *treeTop=0
    ihppNode *treeBottom=0;
    ThreadContext *ctx = globalSharedContext->getThreadCtx(PIN_ThreadUid());

    bb->incSimpleCounter();

    //code omitted [...]
    //code about stack ptr check omitted

    if ( ctx->getCurrentFunction() &&
            ctx->getCurrentFunction() != bb->functionAddr() )
    {
        intraMode_ret();
        ctx->setCurrentFunction(bb->functionAddr());
    }

    intraCtx = ctx->getCurrentFunctionCtx();

    if (!bb->isFirstBlock()) {

        //code omitted [...]

        INTRAMODE_LOAD_TOP_BOTTOM();

        //code about stack ptr check omitted

        if (!globalSharedContext->options.rollLoops) {

            traceObject(bb, intraCtx, treeTop, treeBottom);

        } else {

            bool found=false;
            ihppNode *parent = treeTop->getParentRef();

            while (parent) {

                if (parent->getKey() == bb->getKey()) {

                    treeTop=parent; treeBottom=0;
                    found=true;
                    treeTop->incCounter();
                    break;
                }

                parent = parent->getParentRef();
            }

            if (!found) {
                traceObject(bb, intraCtx, treeTop, treeBottom);
            }
        }

        INTRAMODE_REPLACE_TOP_BOTTOM();
        return;
    }

    //The first basic block of a function is met:
    //it has the same address of the function which belongs to
```

```
63
64     if (!intraCtx->rootKey) {
65
66         //Rootkey is null, so this is the first time
67         //the function is called (in this thread): everything is very simple.
68
69         treeTop=0; treeBottom=0;
70         traceObject(bb, intraCtx, treeTop, treeBottom);
71
72         //asserts omitted
73
74         INTRAMODE_STORE_TOP_BOTTOM();
75         return;
76     }
77
78     //The BBL is a first block but this is NOT the first time
79     //this function is called (in this thread)
80
81     if (intraCtx->shadowStack.size())
82         INTRAMODE_LOAD_TOP_BOTTOM();
83
84     if (!INTRAMODE_TOP_BOTTOM_ARE_POINTING_TO_ROOT())
85     {
86         INTRAMODE_SET_TOP_BOTTOM_TO_ROOT();
87         INTRAMODE_STORE_TOP_BOTTOM();
88     }
89
90     treeTop->incCounter();
91 }
```

Listing 5.22: partial implementation of `intraModeBlockTrace()`

It is convenient to explain the above routine following the chronological order of events. Let be `foo()` the traced function and `th` the only thread in the target program. The first time `foo()` is called two IHPP tracing routines are activated: `FunctionObjTrace()` that records the event in the *thread context* related to `th` and `intraModeBlockTrace()` which have to trace the activation of the first BBL of `foo()`; this happens in this way:

1. The current *thread context* is obtained on line 6
2. The `intraMode` context is obtained by "asking" to the `th`-context which is the current running function (set by `funcMode` routines)
3. The condition on line 22 fails, since `bb` is the first block of `foo()`
4. The condition on line 64 results true since this is the first time `foo()` is called in `th` so, *top* and *bottom* pointers are cleared and `traceObject()` routine is called.
5. The `traceObject()` routine traces the *event* and sets the `rootKey` variable
6. *Top* and *bottom* pointers are stored to the `intraMode` *shadow stack* using the macro `INTRAMODE_STORE_TOP_BOTTOM()`
7. The tracing routine returns

Other BBLs in `foo()` are traced in this other way:

1. Contexts are obtained as before
2. The condition on line 22 becomes true

3. *Top* and *bottom* pointers are loaded from the *top* of `intraCtx.shadowStack` using the `INTRAMODE_LOAD_TOP_BOTTOM()` macro
4. If *roll loops* option is not enabled:
   (a) The BBL activation is traced
5. Otherwise:
   (a) The parent node container of the last BBL activated is obtained
   (b) A loop iterating over all its ancestors is performed searching for an ancestor with same address as the current BBL's one
   (c) If that ancestor is found:
      i. *Top* and *bottom* pointers are moved to it or, to better say, only the *top* pointer is moved since the *bottom* pointer is not used in *roll loops* mode because k is infinite
      ii. The counter of the *top* pointer is incremented without calling no other routine.
   (d) Else, `traceObject()` is simply called
6. The top of the *shadow stack* is *replaced* with the new values of *top* and *bottom* pointers using a macro
7. The tracing routine returns

5.5.2 How *intraMode_ret()* works

Now, before dealing with the last case of routine in listing 5.22 which starts on line 78, it is useful to understand what happens when `foo()` returns. Listing 5.23 shows a partial implementation of the `intraMode_ret()` routine.

```
 1  void intraMode_ret()
 2  {
 3      ThreadContext *ctx = globalSharedContext->getThreadCtx(PIN_ThreadUid());
 4      IntraModeContext *intraCtx = ctx->getCurrentFunctionCtx();
 5
 6      //Win32-specific code omitted [...]
 7
 8      //code about strange situations omitted [...]
 9
10      intraCtx->shadowStack.pop();
11
12      if (!intraCtx->shadowStack.size()) {
13
14          ihppNode *treeTop,*treeBottom;
15
16          //code about stack ptr check omitted
17
18          INTRAMODE_SET_TOP_BOTTOM_TO_ROOT();
19          INTRAMODE_STORE_TOP_BOTTOM();
20      }
21  }
```

Listing 5.23: partial implementation of `intraMode_ret()`

As it can be seen, the main purpose of the `intraMode_ret()` routine is the statement on line 10 which pops the *shadow stack*. When a function like `foo()` returns, if there have been no recursion or if it is has ended, the *shadow stack* remains empty after the `pop()` on line 10 so, the `if` on line 12 checks this case and if true, moves back *top* pointer to the *node* in the *k*-SF which has as key (BBL address) the value stored in `intraCtx->rootKey` and sets *bottom* pointer to *zero*. After that, it stores them into the *shadow stack*.

Now is possible to explain the last case of routine in listing 5.22: when `foo()` is not called for the first time in the thread `th`. In this situation, the BBL is the first of `foo()` but the `rootKey` has been already set (the *k*-slab forest for `foo()` is not empty) also, the *shadow stack* should not be *empty* because `intraMode_ret()` takes care this to never happen. Therefore, since *strange* situations can happen, `intraModeBlockTrace()` tries to load the *top* and *bottom* pointers from the *shadow stack* on line 82 and checks again if `treeTop` points to the *root* node and `treeBottom` to zero and, if it is not true, fixes the situation. As last and main operation, the counter of the node pointed by `treeTop` is incremented (line 90).
For completeness, definitions of macros used are shown in the listening below:

```
INTRAMODE_LOAD_TOP_BOTTOM():
treeTop = intraCtx->shadowStack.top().treeTop;
treeBottom = intraCtx->shadowStack.top().treeBottom;

INTRAMODE_STORE_TOP_BOTTOM():
intraCtx->shadowStack.push(ShadowStackItemType(treeTop,treeBottom));

INTRAMODE_REPLACE_TOP_BOTTOM():
intraCtx->shadowStack.pop();
intraCtx->shadowStack.push(ShadowStackItemType(treeTop,treeBottom));

INTRAMODE_SET_TOP_BOTTOM_TO_ROOT():
intraCtx->counter=1;
treeTop=intraCtx->kSlabForest.getTreeRef(intraCtx->rootKey);
treeBottom=0;

INTRAMODE_TOP_BOTTOM_ARE_POINTING_TO_ROOT():
(treeTop==intraCtx->kSlabForest.getTreeRef(intraCtx->rootKey) && !treeBottom)
```

Listing 5.24: partial implementation of `intraMode_ret()`

5.6 The *inter*-procedural mode

The `intraProcMode` is the simplest working mode implemented in IHPP: is consists of only one tracing procedure called `interModeBlockTrace()` which is instrumented in the same place where the main `intraMode` routine is instrumented as can be seen in listing 5.21. Its only procedure is implemented in `interProcMode.cpp` and its partial implementation is show in the listing 5.25 below.

```
void interModeBlockTrace(TracingObject<ADDRINT> *to) {

    ThreadContext *ctx;
    BasicBlock *bb = static_cast<BasicBlock*>(to);

    ctx = globalSharedContext->getThreadCtx(PIN_ThreadUid());

    //code omitted [...]

    bb->incSimpleCounter();

    if (!globalSharedContext->options.rollLoops) {

        traceObject(bb, ctx, ctx->treeTop, ctx->treeBottom);
        return;
    }

    ihppNode *parent = ctx->treeTop->getParentRef();

    while (parent) {

        if (parent->getKey() == bb->getKey()) {

            ctx->treeTop=parent;
            ctx->treeBottom=0;
            ctx->treeTop->incCounter();
            return;
        }

        parent = parent->getParentRef();
    }

    //parent NOT found
    traceObject(bb, ctx, ctx->treeTop, ctx->treeBottom);
}
```

Listing 5.25: partial implementation of `interModeBlockTrace`

As listing 5.25 shows, practically the only operation which `interModeBlockTrace()` does is to call `traceObject()` using as *top* and *bottom* pointers the ones defined in `ThreadContext` and nothing other, when the *roll loops* option is not enabled; instead, when it is, a logic identical to the one used in `intraMode` is implemented.

5.7 Advanced *function* mode: *insTracing*

The *instruction tracing* mode has been implemented manly to improve the correctness of k-SFs in `funcMode` for Windows programs compiled with the `cl` compiler. As already explained, when only a set of *user* functions are instrumented using the `-funcs` option and the *stack pointer check* is enabled, k-SFs produced are often correct with the exception of the case shown in listing 5.18 but, when all routines are instrumented, k-SFs produced in that way become wrong because `cl` put in the main image of the program various library and system-related routines which sometimes are not called using the `call` instruction: in that situations, trying to understand what happened between one routine activation and another using the *stack pointer* as a reference point does not make sense since it remains unaltered. Therefore, the *insTracing* approach is to totally ignore the *stack pointer* value and to reconstruct the *execution trace* analyzing each instruction.

The instrumentation When IHPP is compiled with *insTracing* support[9] and it is is called with the `-insTracing` option, the three *insTracing* routines are instrumented; this is done in the point where code is omitted in listing 5.12. Briefly, two of them are called before a branch or a call instruction is executed (one handles the direct branch (or call) case and the other the indirect one) and the third one is called before every other instruction is executed.

5.7.1 A definitive solution to the *long jump* problem

The first goal of *insTracing* mode was to solve the *long jump* problem in particular situations like the one shown in listing 5.18 at least in the simpler case when only user

[9]This can be done setting by `ENABLE_INS_TRACING` to 1 in `config.h`. Under Windows systems, *insTracing* is supported by default.

functions are instrumented. The main idea is: the routine address related to *every* branch (or call) destination address is stored in a variable of the current *thread context*; before every instruction is executed, a routine checks if the instruction which is going to be executed belongs or not to the *target routine* previously stored in the *thread context*: if it does not, then something like a *long jump* is happened using instructions not instrumented so, substantially the `funcMode_ret()` routine is invoked. The just described approach is *theoretically* infallible only when one *strong* condition is verified: the set of instrumented functions must be an *ihpp-layer*.
An *ihpp-layer* is a set of functions which follow these rules:

1. Every function in the *ihpp-layer* can call every other function in this layer
2. Every function of above layers can call functions inside the *ihpp-layer*
3. Every function in the *ihpp-layer* can call *only* other functions belonging to below layers which absolutely *never* calls functions in the *ihpp-layer*

A clarifying example: there are three routines, `a()`, `b()` and `c()`. The routine `a()` calls `b()` and this one calls `c()`. The set $\langle a, c \rangle$ is *not* an *ihpp-layer* because `b()` which should belong to a *below* layer calls `c()` which belongs to the *ihpp-layer*. If a concrete program containing these routines were compiled and IHPP executed to analyze it with `-funcs a,c` option, a *false long jump* from `a()` to `c()` will be wrongly found.

Technically, the implementation of this approach is more complicated then as just explained because programs built with `cl /Zi` uses a sort of two-step call system for user routines: a caller uses a `call` instruction which jumps to an area of `.text` section which has a direct `jmp` instruction to the destination routine. Therefore, in order to handle this, all instructions which belongs to the *no routine area* of `.text` are instrumented and two jump target addresses are stored in the *thread context*.

5.7.2 The *forward jump recognition* approach

In the last section it was described a good approach to the *long jump* problem which works for a limited set of routines. As stated many times, when all routines of a program compiled with `cl` are instrumented, wrong k-SFs are often produced because of the presence of built-in routines, indistinguishable from the user ones, which sometimes call each other using simply `jmp` instructions. Furthermore, `jmp` instructions are used also to return to the "caller" since the *instruction pointer* has not been stored on the *stack* by a `call`. Thus, when a `jmp` instruction is found, it is not obvious if the `jmp` is a sort of *call-jump* or a *ret-jump* instead.

Since the output of any full instrumented program is prohibitively large for this paper, an ad-hoc **masm** assembly program that emulates the behavior of "evil" routines has been written[10]; the listing 5.26 below shows its full code.

```
.model  flat
INCLUDELIB LIBCMT

_DATA   SEGMENT

$hello_main     DB  'hello from main()', 0aH, 00H
$rfoo_called    DB  'real_foo() called', 0aH, 00H
$foo_base       DB  'foo() base called', 0aH, 00H

_DATA   ENDS

PUBLIC _main, foo, real_foo, bar
EXTRN   _printf:PROC
```

[10]The program is built using the **Microsoft Macro Assembler** in this way: `ml /Zi prog9.asm`

```

_TEXT   SEGMENT

real_foo:
    push OFFSET $rfoo_called
    call _printf
    add esp, 4
    jmp after_foo

foo:
    push OFFSET $foo_base
    call _printf
    add esp, 4
    jmp real_foo

bar:
    push ebp
    mov ebp, esp
    ; do nothing
    leave
    ret

_main:
    push ebp
    mov ebp, esp

    jmp foo

    after_foo:

    push OFFSET $hello_main
    call _printf
    add esp, 4

    call bar

    mov eax, 0
    leave
    ret

_TEXT   ENDS
END
```

Listing 5.26: prog9.asm, a tricky program

It is clear that `foo()` and `real_foo()` are *not* normal routines like `_main()` and `bar()`. If they were not *public symbols*, they would be completely transparent to IHPP so there would be no problem but, they are public and there is no way to ignore them. Running IHPP analysis on `prog9` with `-funcs _main,foo,real_foo,bar,printf` and without the `-insTracing` option, produces the output shown in listing 5.27.

```
----------------------------------------------------------------------------
DUMP of K-SF
----------------------------------------------------------------------------

| __root__(),1
   | _main(),1
      | foo(),1
         | printf(),1
      | real_foo(),1
```

```
         | printf(),2
         | bar(),1
```

Listing 5.27: IHPP output without *insTracing*

As can be seen, the output in listing 5.27 is totally wrong: `real_foo()` was not called by `_main()` and it did not called twice `printf()` and once `bar()`; instead, `printf()` has been called once by `real_foo()` and once by `_main()`, which also has called `bar()`. Using the `-insTracing` option, instead, produces the *correct* output:

```
-----------------------------------------------------------------------------
DUMP of K-SF
-----------------------------------------------------------------------------
| __root__(),1
   | _main(),1
      | foo(),1
         | printf(),1
         | real_foo(),1
            | printf(),1
      | printf(),1
      | bar(),1
```

Listing 5.28: IHPP output with *insTracing*

How the *forward jump recognition* works The heart of the idea is to add an integer variable `fjmps` to each record of the *shadow stack* and using it in this way: every time an *unconditional* branch which target address is outside the current routine is found, iterate over the *shadow stack* searching for the target routine of the jump: if it is found, assume that the jump is a *ret-jump* and pop the *shadow stack*; otherwise, the jump is a *call-jump*, or better a *forward jump*, so set a flag *forward jump happened* in the *thread context*. After the jump happened, the `FunctionObjTrace()` routine clears the flag and sets the `fjmps` variable related to the current *shadow stack* record to the value of `fjmps` on the previous record plus one. Using this logic, when `real_foo()` is traced, `fjmps` variable of the *shadow stack* top record assumes value 2. When on line 22 `real_foo()` jumps back to the label `after_foo` in `_main()`, the *forward jump recognition* code founds the jump-target routine `_main()` into the *shadow stack* and assuming that the jump is a *ret-jump*, pops the *shadow stack* a number of times equals to `fjmps`.

Considerations The approach just explained has no *theoretical* reasons to be valid: it makes assumptions that are not in general always true but, it *empirically* produces much better results than the *stack pointer check* when the program contains "evil" routines such as the ones shown in listing 5.26. Nevertheless, the implementation of *insTracing* mode has a considerable number of other `cl`-specific checks due to various complications; this is main reason for the total absence of code listings in this section. Interested readers are invited to read the source code for a total comprehension of the topic.

5.8 The performance of IHPP

As every other non-statistical profiler, IHPP considerably reduces the execution speed of the *analyzed program*. This speed reduction is caused by two different reasons: the computational cost of the instrumentation itself and the computational cost of the analysis routines. The first one depends by the way Pin adds additional instructions to the program and can be measured by instrumenting the program's routines (or BBLs) with *empty* analysis routines. The second one instead, depends only from the code inside the analysis routines and is the *net* slowdown caused by IHPP; therefore, the slowdown measured should be always compared to the *empty analysis* slowdown. This last performance indicator strictly depends from the analyzed program type: for example, a program that makes an intensive use of *recursion* has a bigger slowdown in `funcMode` than a program which uses mainly loops for heavy tasks. In order to discover the slowdown in IHPP, some timers has been added to it with the purpose to isolate the Pin load time, the program and its shared libraries load time and the instrumentation process from the *real* running time of the program so, the slowdown factors are calculated dividing the mean net running time of the program (which starts after all libraries has been loaded) by the mean net running time of program when profiled with IHPP. The fig. 5.3 (below) shows indicative values of the slowdown caused by an instrumentation with *empty* routines analysis routines, or simply an "empty instrumentation".

	insTracing OFF	insTracing ON	relative insTracing slowdown
funcMode	2.5x - 4.0x	60x - 110x	15x - 44x
intraMode	15x - 26x	70x - 120x	2.7x - 8x
interMode	13x - 24x	–	–

Figure 5.3: Slowdown caused by *empty* analysis routines

As can be seen, the only "empty instrumentation" of *insTracing* mode seriously reduces the execution speed of the target program since before every single simple instruction like a `mov` there is a *heavy* function call. The fig. 5.4 instead, shows the overall slowdown of IHPP using actual instrumentation routines.

	insTracing OFF	insTracing ON	relative insTracing slowdown
funcMode	9x - 12x	190x - 206x	16x - 23x
intraMode	75x - 87x	247x - 270x	2.8x - 3.6x
interMode	43x - 54x	–	–

Figure 5.4: Overall IHPP slowdown

The results shown in the figure above, are obtained by measuring IHPP performance in various situations (different programs and options). Every working mode has been tested using several values for the k parameter and other mode-specific options such as the *rollLoops* option in `intraMode`. An interesting result is that the k parameter value has no a big influence on the slowdown factor: bigger values of k make the profiled program to run a little faster but the gap is under 10% from $k = 3$ to $k = 10$. Another noticeable result is that the *rollLoops* option in `intraMode` has good performance: it has a slowdown that can be compared to values of k about 7-8; only values of k about 10 or higher produce some little slowdown improvements compared to it.

Considerations It is evident that *insTracing* cannot be used to profile interactive programs but is necessary to say that is really needed only in particular conditions under Windows systems; instead, in other cases non cpu-intensive interactive programs can be profiled with IHPP in `funcMode`. The `intraMode` (like the `interMode`) even without *insTracing* is still slow for profiling GUI interactive programs, but it has an acceptable slowdown for many textual tools: this should not be a *real* problem because the goal of the *intra*-procedural profiling is not to analyze big and complex GUI programs but instead to analyze small programs that implements various algorithms which the user of IHPP would like to improve; the *intra*-procedural profiling is a way to *better understand algorithms.*

Chapter 6

Conclusions

IHPP is a new project and even if much work has been done to develop it, an amount of work even bigger should be done to extend and improve it. It would need a *graphical user interface* in order to be easily used (for this reason XML output support has been also implemented in it); also, actually it is totally Pin-dependent and for that reason it is not a real GPL program; the *huge* software layer implemented by Pin should be rewritten one day if the project have to be distributed as a Linux program.

Nevertheless, this is a start: IHPP maybe is not something absolutely exceptional in the world, but it really offers some *additional* analysis information that the greatest part of other profilers (if not almost all) does not actually offer. The *intra*-procedural mode, in particular, is a way for studying new algorithms and improving the actual ones through a deep internal inspection of them which can be, of course, done by hand but that often programmers avoid because of the great deal of time it takes.

Sometimes, due to the considerable gap between the C and assembly layer (on which BBLs are defined), the matching between the C source code and the k-SFs built by `intraMode` analysis becomes really hard to be done, but if the assembly code of the procedure is considered instead of the C one, everything becomes simpler. Studying algorithms in assembly should not be thought as a "back to the past" solution, but as a "back to the basis" approach because where there is the need of performance, a programmer must know what his code is really doing and this can never be done by looking the C or C++ source code of a program. A programmer should never forget that computers do not understand *objects*, *virtual classes*, *polymorphism* and not even the simply *procedures* because they work only with *machine instructions*.

Bibliography

[1] Giorgio Ausiello, Camil Demetrescu, Irene Finocchi, and Donatella Firmani. k-Calling Context Profiling. In *27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, October 2012.

[2] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, October 1969.

[3] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42, pages 230–265, 1936.