k-Calling Context Profiling

Giorgio Ausiello
Dept. of Computer and System Sciences
Sapienza University of Rome
ausiello@dis.uniroma1.it

Camil Demetrescu
Dept. of Computer and System Sciences
Sapienza University of Rome
demetres@dis.uniroma1.it

Irene Finocchi
Dept. of Computer Science
Sapienza University of Rome
finocchi@di.uniroma1.it

Donatella Firmani
Dept. of Computer and System Sciences
Sapienza University of Rome
firmani@dis.uniroma1.it

Abstract

Calling context trees are one of the most fundamental data structures for representing the interprocedural control flow of a program, providing valuable information for program understanding and optimization. Nodes of a calling context tree associate performance metrics to whole distinct paths in the call graph starting from the root function. However, no explicit information is provided for detecting short hot sequences of activations, which may be a better optimization target in large modular programs where groups of related functions are reused in many different parts of the code. Furthermore, calling context trees can grow prohibitively large in some scenarios. Another classical approach, called edge profiling, collects performance metrics for caller-callee pairs in the call graph, allowing it to detect hot paths of fixed length one. We study a generalization of edge and context-sensitive profiles by introducing a novel data structure called k-calling context forest (k-CCF). Nodes in a k-CCF associate performance metrics to paths of length at most k that lead to each distinct routine of the program, providing edge profiles for $k = 1$, full context-sensitive profiles for $k = \infty$, as well as any other intermediate point in the spectrum. We study the properties of the k-CCF both theoretically and experimentally on a large suite of prominent Linux applications, showing how to construct it efficiently and discussing its relationships with the calling context tree. Our experiments show that the k-CCF can provide effective space-accuracy tradeoffs for interprocedural contextual profiling, yielding useful clues to the hot spots of a program that may be hidden in a calling context tree and using less space for small values of k, which appear to be the most interesting in practice.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics—performance measures; D.2.2 [*Software Engineering*]: Design Tools and Techniques—programmer workbench

General Terms Algorithms, Measurement, Performance.

Keywords Call graph, calling context tree, dynamic program analysis, edge profiling, profiling, vertex profiling.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1561-6/12/10...$10.00

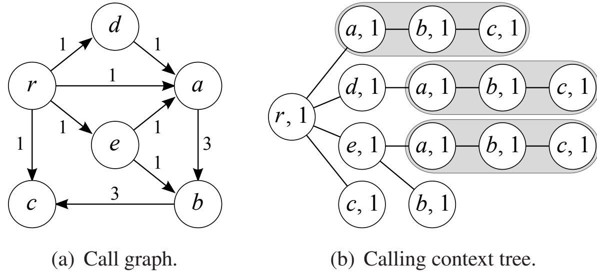
1. Introduction

Profilers attribute performance metrics to portions of a program's execution. Such metrics can be associated to isolated synctactic units, such as statements or procedures, or to richer contexts, such as paths in the control flow or the call graph. Accurate profiles can provide valuable information for program understanding, optimization, test coverage, and other software engineering tasks [20, 25, 33, 34].

In this paper we study profiling techniques for interprocedural control flow, where performance metrics are associated with paths in the call graph. In a spectrum of possible representations of calling behavior that trade space overhead for information accuracy, *vertex profiling* [15] provides the most compact solution. It maintains a performance counter for each routine of the program, but no information is given on the contexts in which hot routine invocations occur. This limitation can play a crucial role in profiling hardware events such as instruction stalls or cache misses, which can be very context-dependent [2]. *Edge profiling* [15] makes one step further by maintaining a single level of context sensitivity. However, since measurements are collected only for one-edge paths in the call graph, this can still yield misleading results [23, 28]. At the other end of the line, *dynamic call trees* record the complete history of routine invocations and can be used to maintain accurate profiling information, but they can grow prohibitively large for long-running programs.

Calling context trees (CCT) provide an intermediate solution by keeping track of all distinct *calling contexts* of a program, i.e., sequences of procedures that are active during intervals of a program's execution [2]. In a CCT, each node is labeled with the name of a routine and each path from the tree root to a node labeled with v represents a distinct calling context of v. Since there is no distinction between different invocations of the same routine within the same context, a CCT is typically smaller than a call tree, but it can still remain very large in many applications [12, 13, 27, 35]. The CCT encodes full information on the distinct call paths of a program's execution and supports *context-sensitive profiling* by associating performance measurements to calling contexts that start from the program's root. Unfortunately, calling context trees may fail to reveal certain relevant hot spots in the code, as discussed in the following example.

Motivating example. Consider the edge profiling report of Figure 1(a), which shows the call graph of a program along with the number of times each routine is called by another routine in a particular execution. It is easy to see that c, which is called 4 times,



(a) Call graph. (b) Calling context tree.

Figure 1. Motivating example for k-calling context profiling.

is one of the most frequently activated routines. To understand and optimize the program, we would like to trace back the context in which most invocations of c arise. By analyzing the edge weights of the call graph, we discover that c is called $1/4$ of the times by r and $3/4$ of the times by b. We notice however that it is impossible to determine from the call graph if a longer hot context of c exists, i.e., if most activations of c via b happened from e or from a. Figure 1(b) shows the calling context tree of the same execution. Observe that all distinct contexts of c are only reached once, which again yields no clues to the existence of a hot path longer than 1 that leads to c. A more careful analysis of the CCT reveals that call path $\langle a, b, c\rangle$, shown with a shaded background in Figure 1(b), appears three distinct times as a subtree of the CCT. This implies that $3/4$ of the calls of c arise via b when a is active, i.e., there is a call path of 2 edges that accounts for 75% of all calls of the hottest routine, whereas there is no interesting call path longer than 2. From this example, we argue that sometimes profiling short paths that lead to a hot routine may be more useful than considering whole contexts constrained to start from the program's root: this seems to be especially the case in large modular programs where groups of related functions are reused in many different parts of the code. Since the CCT provides frequency counters only for full contexts, extracting information on shorter paths may be difficult. Additionally, we notice that short paths may be repeated several distinct times in a CCT, offering the opportunity for more compact and space-efficient representations.

Profiling call paths of bounded length. Different authors have discussed the benefits of profiling call paths of bounded length, rather than entire paths constrained to start from the program's root. For instance, the N-level call encodings of Reiss and Renieris [24] were proposed in a different setting as a technique for compacting raw execution traces. When the size of the CCT is large, the `callgrind` cache simulator [31] exploits a reduced CCT_k, where two contexts are collapsed if the trailing k contexts of the corresponding call chains are identical. Ammons *et al.* [3] advocate the use of bounded-length calling contexts and show that they can be computed by analyzing offline a previously recorded call tree. While showing interesting practical results in applying their tool to improve the performance of several Java applications, their approach scales poorly to large programs, which may require to collect and process call trees of billions of nodes.

Our contributions. To answer the above concerns, we study a space and time-efficient profiling methodology that we call k-calling context profiling. The approach consists of computing performance metrics for all call paths of length up to k that lead to a program's routine. The k-calling context profiling problem is a generalization of other approaches, yielding vertex profiles for $k = 0$, edge profiles for $k = 1$, full context-sensitive profiles for $k = \infty$, as well as any other intermediate point in the spectrum. We develop this concept in several novel directions:

- We design a novel combinatorial structure for k-calling context profiling that we call k-calling context forest (k-CCF), showing that it has several relevant properties and discussing its space requirements (Section 4).
- We show that a k-CCF can be constructed from a CCT; we also design a new efficient data structure called k-slab forest (k-SF), which provides a more space-efficient alternative to the CCT for constructing the k-CCF when k is small (Section 5).
- We perform an extensive experimental evaluation of our data structures on a suite of large-scale Linux applications (Section 6 and Section 7), showing that:
  - hot call paths typically do not start from the program's root and are much shorter than the depth of the CCT, confirming that short paths that lead to a hot routine can be a better profiling target for detecting performance bottlenecks than the full calling contexts of a CCT;
  - it is possible to convert a CCT into a pruned version of the ∞-CCF that is extremely compact (on average, just 1.5% of the CCT on our benchmarks) and includes *explicit* profiles for *all* hot paths;
  - if room for storing an entire CCT is not available, small values of k can greatly reduce space at the price of a small increase of the running time, while still preserving useful information for detecting most hot call paths.

In Section 2 and in Section 3 we introduce the k-calling context profiling problem and we provide some preliminary notation and tools used in the definition of our data structures. Related work is discussed in Section 8 and concluding remarks are given in Section 9.

2. k-Calling Context Profiling

In this section we formalize the *k-calling context profiling* problem, discussing its connections with other classical forms of interprocedural performance profiling.

Let $G = (V, E)$ be the (dynamic) call graph of a program, where V is the set of routine names and E is the set of caller-callee relationships, and let $r \in V$ be the root function of the program. We assume that the length of a path is its number of hops. At any time during the execution of the program, the *current execution context* is described by a path $\pi = \langle r, ..., u\rangle$ in G starting from r, representing the sequence of pending routine activations on the runtime stack.

The input of the profiler is a trace of `call`/`return` program operations, which update the current context starting from an empty context as follows:

- `call(v)` updates the current context $\pi = \langle r, ..., u\rangle$ by adding a node v s.t. $\langle u, v\rangle \in E$. The resulting context is $\pi' = \langle r, ..., u, v\rangle$.
- `return` updates the current context $\pi = \langle r, ..., u, v\rangle$ by deleting the last node v. The resulting context is $\pi' = \langle r, ..., u\rangle$.

Figure 2(a) shows an example of execution trace along with the current context after each operation.

We now define the *k-calling context* of a routine, which generalizes classical calling contexts [2] by focusing on a bounded number of the topmost consecutive activations on the runtime stack, rather than considering the whole path all the way down to the root function:

Definition 1 (k-calling context)**.** *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.*

operation	curr. context
start	$\langle\rangle$
call(r)	$\langle r \rangle$
call(a)	$\langle r, a \rangle$
call(b)	$\langle r, a, b \rangle$
return	$\langle r, a \rangle$
call(c)	$\langle r, a, c \rangle$
return	$\langle r, a \rangle$
return	$\langle r \rangle$
call(c)	$\langle r, c \rangle$
call(a)	$\langle r, c, a \rangle$
call(b)	$\langle r, c, a, b \rangle$
return	$\langle r, c, a \rangle$
call(b)	$\langle r, c, a, b \rangle$
return	$\langle r, c, a \rangle$
return	$\langle r, c \rangle$
return	$\langle r \rangle$
return	$\langle\rangle$

(a) Execution trace.

q	activated π	$c(\pi)$
0	$\langle r \rangle^*$	1
	$\langle a \rangle$	2
	$\langle b \rangle$	3
	$\langle c \rangle$	2
1	$\langle r, a \rangle^*$	1
	$\langle a, b \rangle$	3
	$\langle a, c \rangle$	1
	$\langle r, c \rangle^*$	1
	$\langle c, a \rangle$	1
2	$\langle r, a, b \rangle^*$	1
	$\langle r, a, c \rangle^*$	1
	$\langle r, c, a \rangle^*$	1
	$\langle c, a, b \rangle$	2
3	$\langle r, c, a, b \rangle^*$	2

(b) Distinct paths π of any length q activated by the trace of Figure 2(a). Counter $c(\pi)$ denotes the number of activations of path π in the execution trace.

Figure 2. Running example. The CCT corresponding to the execution trace is shown in Figure 4.

k-calling contexts correspond to the N-depth call sequences in [24]. For instance, if $\pi = \langle r, a, b, c, d \rangle$ is the calling context of d, its 2-context is $\langle b, c, d \rangle$. Another example is given in Figure 2(a), which shows as framed boxes all the 0, 1, and 2-contexts of the program's routines during the execution. As a special case, when k is larger than the length of π, the k-context of v coincides with its full context π.

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

In our example of Figure 2, all activations of length $q \leq 2$ are highlighted as framed boxes. The distinct activated paths are listed in Figure 2(b) grouped by their length.

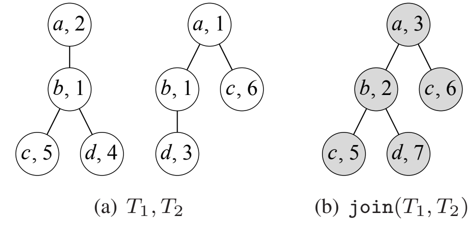
Definition 3 (k-calling context profiling)**.** *Given a trace of* `call` *and* `return` *operations, the k-*(calling) context profiling *problem consists of computing, for each activated path π of length $q \leq k$, the number $c(\pi)$ of times π is* activated*.*

Figure 2(b) shows all activation counters $c(\pi)$ computed by 3-context profiling for the sample trace of Figure 2(a). 2-context profiling would compute all counters for paths up to length $q = 2$. We notice that, since the longest activated path has length 3, k-context profiling for any $k > 3$ computes the same values as 3-context profiling. We remark that only counters of starred paths in Figure 2(b), which start from the root function, would be maintained in a CCT. Although we focus on counting routine activations, this approach can be adapted to deal with other performance metrics, including execution times, branch mispredictions, cache misses, etc.

The k-context profiling problem is a generalization of different classical performance profiling approaches at the interprocedural level. In particular:

- *vertex profiling* [15] is equivalent to k-context profiling with $k = 0$;
- *edge profiling* [15] is solved by k-context profiling with $k \geq 1$;
- *context-sensitive profiling* [2] is solved by k-context profiling with $k = \infty$.

In Section 8 we discuss further connections between this approach and previous work in the literature.



(a) T_1, T_2 (b) $\texttt{join}(T_1, T_2)$

Figure 3. Weighted tree join operation: for each node v we report its label $\ell(v)$ and its counter $c(v)$.

3. Preliminaries

In this section we introduce some preliminary notation and a useful operator on trees, called *tree join*, that will be used to define the data structures discussed in the paper.

Let T be a labeled tree where $\ell(v)$ denotes the label of node v. We assume that, for each tree node, all its children have distinct labels. Let r be the root of T. Throughout this paper we will assume that a tree root has level 0. A *root-path* is any path starting at r. A *root-to-leaf path* is a root-path ending on a leaf. We also say that a *label path* is the sequence of labels of a path in a labeled tree. A *root-label path* is the sequence of labels of a root-path and a *root-to-leaf label path* is the sequence of labels of a root-to-leaf path. We denote by $\pi^R = \langle z, \ldots, b, a \rangle$ the *reverse* of a path $\pi = \langle a, b, \ldots, z \rangle$.

Tree join. We define the tree join operation as follows:

Definition 4 (Tree join)**.** *The* join *of two labeled trees T_1 and T_2, denoted as* $\texttt{join}(T_1, T_2)$*, is the minimal labeled forest $\mathcal{F}$ such that $\mathcal{F}$ contains a root-label path π if and only if T_1 or T_2 contain π.*

By the minimality of $\mathcal{F}$, if two root-paths of T_1 and T_2 share a common prefix of labels, that prefix is represented in $\mathcal{F}$ only once. We can therefore distinguish two cases, depending on the labels of tree roots r_1 and r_2:

- if $\ell(r_1) \neq \ell(r_2)$, then $\mathcal{F}$ simply consists of T_1 and T_2;
- if $\ell(r_1) = \ell(r_2)$, then T_1 and T_2 are merged in $\mathcal{F}$ into a unique tree with root r and label $\ell(r) = \ell(r_1) = \ell(r_2)$.

Weighted tree join. The join operation can be easily extended to deal with the weighted case where a counter $c(v)$ is associated to each node v of T_1 and T_2. Let z be a node of $\mathcal{F}$ and let π_z the unique root-path that leads to z in $\mathcal{F}$. We define $c(z)$ as the sum of all counters $c(v)$ of nodes v in $V_1 \cup V_2$ such that the root-path π_v that leads to v in T_1 or T_2 has the same sequence of labels as π_z.

Forest join. The join operator can be also generalized to arbitrary sets of trees $\{T_1, ..., T_h\}$, with $h \geq 3$: if all trees have distinct root labels, then $\mathcal{F}$ coincides with the input forest $\{T_1, ..., T_h\}$. Otherwise, let T_1 and T_2 be two labeled trees with the same root labels, then:

$$\texttt{join}(T_1, ..., T_h) = \texttt{join}(\texttt{join}(T_1, T_2), T_3, ..., T_h).$$

Examples. An example of weighted join application is shown in Figure 3. As another example, we notice that the join operator can be used to provide an alternate definition of the CCT, compared to the classical formulation based on an equivalence relation on pairs of nodes of the call tree [2]. Let T be a call tree, let h be its number of leaves, and let $\pi_1, ..., \pi_h$ be the root-to-leaf paths of T: the CCT of the program's execution encoded by T is exactly the result of the $\texttt{join}(\pi_1, ..., \pi_h)$ operation. Since all paths share the same root, they have a non-empty common prefix and the join will return a unique tree (the CCT) instead of a forest.

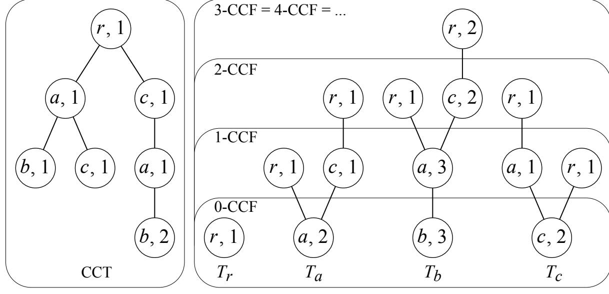


Figure 4. CCT and k-CCF for the example of Figure 2.

4. k-Calling Context Forest

In this section we study data structures for k-context profiling. In particular, we introduce a novel data structure called *k-calling context forest*, discussing its properties and space requirements.

4.1 Definition and Properties

A k-calling context forest maintains information on all activated paths of length up to k and can be simply defined as the join of the reverse of those paths:

Definition 5 (k-calling context forest)**.** *The k-calling context forest (k-CCF) of the execution of a program is a labeled forest defined as* $\texttt{join}(\pi_1^R, \ldots, \pi_s^R)$*, where $\{\pi_1, \ldots, \pi_s\}$ is the set of all distinct paths of length at most k activated by the execution.*

Figure 4 shows the k-CCF for the running example of Figure 2(a) for different values of k. k-calling context forests have the following relevant properties:

1. each node ω of the forest is labeled with a routine name $\ell(\omega)$;
2. there is a tree T_u of depth at most k for each distinct routine u in the program; the root of T_u is labeled with u; there is a one-to-one mapping between nodes of the call graph and roots of the k-CCF;
3. there is a one-to-one mapping between k-CCF nodes and distinct call paths of length up to k activated during the program's execution; in particular, there is a root-label path $\langle a, b \ldots, z\rangle$ of length $q \leq k$ in T_a if and only if $\langle z, \ldots, b, a\rangle$ is an activated q-context of a;
4. 0-CCF $\subseteq$ 1-CCF $\subseteq$ 2-CCF $\subseteq \ldots \subseteq \infty$-CCF;
5. for all $k \geq d$, k-CCF $=$ d-CCF $=$ ∞-CCF, where d is the length of the longest distinct context (i.e., the depth of the CCT);
6. all leaves of the ∞-CCF are labeled with the name r of the root function of the program;
7. each leaf of the ∞-CCF corresponds to a distinct context of the program; therefore, there is a one-to-one mapping between CCT nodes and ∞-CCF leaves.

Performance metrics. We now extend the definition of the k-CCF to include the number of activations of each call path represented in the forest. We assume that each node ω of the k-CCF is equipped with a counter $c(\omega)$. Let $T_{\ell(\alpha)}$ be the k-CCF tree containing ω, and let $\langle \alpha, \beta \ldots, \omega\rangle$ be the unique root-path that leads to ω in $T_{\ell(\alpha)}$. Then $c(\omega)$ is the number of activations of the call path $\pi = \langle \ell(\omega), \ldots, \ell(\beta), \ell(\alpha)\rangle$, i.e., $c(\omega) = c(\pi)$. Therefore, building a k-CCF solves the k-calling context profiling problem.

Counters associated to k-CCF nodes have some relevant properties:

(a) let υ be a node in a k-CCF and let ν be its parent. Then $c(\nu) \geq c(\upsilon)$;

(b) let ν be a node in a k-CCF and let $\upsilon_1, \ldots, \upsilon_h$ be the children of ν. For any program trace such that the root function is never recursively called, $c(\nu) = \sum_{i=1}^{h} c(\upsilon_i)$;

(c) let L_k be the set of leaves of a k-CCF. Then $\sum_{\nu \in L_k} c(\nu)$ is equal to the number of `call` operations in the execution trace.

4.2 Space Analysis

In this section we provide upper bounds on the size of a k-CCF in terms of the sizes of the CCT and the call graph and show that the bounds are tight in the worst case. We also show that the k-CCF can be considerably smaller than the CCT in some scenarios.

Theorem 1. *The number of nodes of a k-CCF is bounded by $O(\min\{kn, dn, |V|^{k+1}\})$ in the worst case, where n is the number of nodes of the CCT, d is the depth of the CCT, and $|V|$ is the number of routines of the program.*

Proof. We first bound the size of a tree T_u of the k-CCF. By Definition 5, T_u is obtained by joining the reverse of all activated q-contexts of u of length $q \leq k$. The number of such contexts cannot exceed the number n_u of nodes of the CCT labeled with u. Since each q-context is not longer than $\min\{k, d\}$, where d is the length of the longest distinct context (i.e., the depth of the CCT), then the number of nodes of T_u is $O(n_u \cdot \min\{k, d\})$. The claim follows from the observation that $\sum_{u \in V} n_u = n$, where n is the number of nodes of the CCT, and from the fact that there can be at most $|V|^{q+1}$ distinct q-contexts for each $q \leq k$. ☐

As special cases, we notice that a 0-CCF contains $|V|$ nodes and a 1-CCF contains $|V| + |E|$ nodes, where $|E|$ is the number of edges of the dynamic call graph.

Discussion. The worst-case space bound proved in Theorem 1 is tight. Consider for instance a CCT of $n = |V|$ nodes consisting of a chain $\langle v_1, v_2, ..., v_n\rangle$ in which all node labels are distinct (i.e., $\ell(v_i) \neq \ell(v_j)$ for all $i \neq j$). For each node v_i, the k-CCF contains one chain rooted at v_i, which includes the (at most k) predecessors of v_i in the CCT, taken in reverse order. Hence, the k-CCF contains $(n - k)(k + 1) + k(k + 1)/2$ nodes. If $k = n/2$, the k-CCF forest has quadratic size $\Theta(n^2)$ with respect to the CCT.

In Section 7 we will provide experimental evidence that the size of the k-CCF for typical real-world applications can be considerably smaller than the size of the CCT for small values of k. We conclude this section by showing a scenario that illustrates why a k-CCF can be more compact than the CCT in some cases.

Theorem 1 and the chain example described above show that the k-CCF can be up to k times larger than the CCT in the worst case. However, in that example each k-context appears in the CCT only once (all node labels are distinct). On the other hand, all repeated occurrences of the same k-context in the CCT will appear only once in the k-CCF, and in that case we can expect a space reduction. Consider the example in Figure 5, in which $k = 2$ and $n = t^2 + 1$ for any appropriate integer $t \geq 2$: in this example, any two CCT nodes at the same distance from the root have the same label, except for the children of the root. Hence, each 2-context ending in a node at level ≥ 4 appears exactly t times in the CCT, but only once in the 2-CCF. It is not difficult to see that in this scenario the number of nodes of the k-CCF is $O(t) = O(\sqrt{n})$ for any constant value of k, i.e., quadratically smaller than the CCT.

5. Construction Algorithms

In this section we discuss algorithms for building the k-CCF. We first observe that, since each `call` event activates up to k paths si-

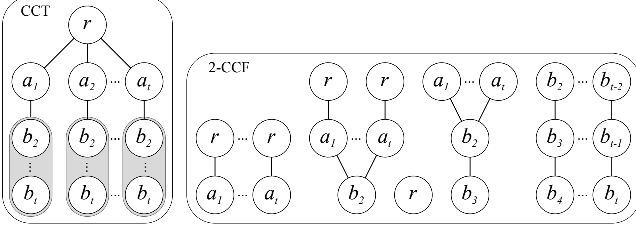


Figure 5. Example showing that the k-CCF can be asymptotically smaller than the CCT. Shaded label paths of length $t-2$ are duplicated t times in the CCT.

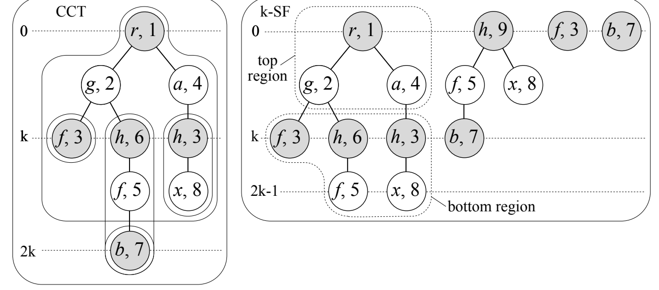


Figure 6. Example of CCT and k-SF with $k = 2$ computed over the same execution trace.

multaneously, maintaining explicitly the k-CCF concurrently with program execution would require updating up to k nodes at each routine invocation. This may considerably slow down the program even for small values of k. We therefore discuss offline solutions that construct the k-CCF at the end of the execution. We first show that the k-CCF can be constructed starting from the CCT, and then we propose a more space-efficient approach based on a new data structure that can be maintained efficiently during the execution of the program.

5.1 Building the k-CCF from the CCT

Since a CCT represents all distinct contexts of a program, by Definition 1 it includes all distinct q-contexts for any value of q. Hence, the CCT contains implicitly all the information needed to construct the k-CCF.

In this section we show that the k-CCF can be constructed starting from the CCT by joining the maximal reversed suffixes of length at most k of all its root paths. We abstract this operation by introducing a useful forest inversion primitive and show how to implement it.

Definition 6 (k-inverse forest)**.** *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 of the unique root path that leads to v_i in F. The k-inverse of F, denoted as $inv_k(F)$, is the labeled forest obtained as* $\texttt{join}(\pi_1^R, \ldots, \pi_n^R)$.

The proof of the following lemma provides a simple algorithm for computing the k-inverse of a labeled forest.

Lemma 1. *Given a labeled forest F with n nodes, the inverse forest $inv_k(F)$ can be computed in $O(kn)$ time.*

Proof. To build the inverse forest, we first compute for each distinct label x of F a list L_x of all nodes $v \in F$ such that $\ell(v) = x$. This can be done in $O(n)$ time. We then construct one output tree T_x at a time: for each label x, we start from an empty tree T_x and for each node $v \in L_x$ we trace at most k ancestors back in F, joining with T_x the path of scanned nodes. To trace all ancestors in $O(k)$ time, we assume that there is a pointer in F from each node to its parent. Since joining a tree with a path of length at most k requires $O(k)$ time and we have $|L_x|$ paths to join, each tree can be constructed in $O(k|L_x|)$ time. The bound follows from the observation that $\sum_x |L_x| = n$. □

The following claim follows directly by Definition 5 and Definition 6:

Property 1. *k-CCF $= inv_k($CCT$)$.*

By Lemma 1 and Property 1, the k-CCF can be computed from a CCT with n nodes in $O(kn)$ time.

5.2 A More Space-efficient Approach

The approach discussed in Section 5.1 has the disadvantage that an entire CCT has to be constructed in order to produce a k-CCF. We notice that the CCT contains all the information sufficient to build a k-CCF for any value of k. This may be superfluous when the k-CCF is smaller than the CCT. The question that we tackle in this section is how to design a data structure that:

1. maintains information on all activated paths of length up to k, and thus can be used for constructing the k-CCF;
2. is more compact than the CCT, at least for small k;
3. can be constructed efficiently on the fly during program execution.

To this aim, we propose a novel data structure called *k-slab forest* (k-SF), which can be defined in terms of the CCT as follows:

Definition 7 (k-slab forest)**.** *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* $\texttt{join}(T_{v_1}, \ldots, T_{v_t})$.

If nodes at levels multiple of k have z distinct node labels in the CCT, then the k-SF will contain exactly z trees, one per label. An example of k-SF for $k = 2$ is illustrated in Figure 6. The example shows that k-SF trees are not necessarily subtrees of the CCT, due to the join operation. For instance, the k-SF tree rooted at the node with label h has no corresponding subtree in the CCT, as it is obtained by joining the two CCT subtrees with labels $\langle h, f, b\rangle$ and $\langle h, x\rangle$.

Properties. By Definition 7, k-SF $=$ CCT for all $k > d$, where d is the depth of the CCT. Therefore, the k-SF can be regarded as a generalization of the CCT.

To analyze the k-SF, we consider the CCT to be conceptually divided into *slabs* of height $k-1$, where the i-th slab consists of nodes with level $\in [ik, (i+1)k)$. We will say that a node is a *slab boundary node* if its level is a multiple of k (see shaded CCT nodes in Figure 6). Each tree T_j of the k-SF can be also divided into a *top* region and a *bottom* region: the top region consists of nodes in levels up to $k-1$, i.e., $top(T_j) = \{u \in T_j \mid \text{level}(u) < k\}$, while the bottom region contain all the remaining nodes: $bottom(T_j) = \{u \in T_j \mid k \le \text{level}(u) < 2k\}$. Each tree T_j of the k-SF spans two consecutive slabs of T: the top and bottom regions contain nodes belonging to the i-th and the $(i+1)$-th slabs, respectively, for some i.

Lemma 2. *The size of the k-SF is upper bounded by $\min\{2n, |V|^{2k}\}$, where n if the number of nodes of the CCT and $|V|$ is the number of routines of the program.*

Proof. Each slab of the CCT (except for slab 0) appears both in the top region and in the bottom region of some k-SF tree. This implies that each node of the CCT appears at most twice in the k-SF, and therefore $|k\text{-SF}| \leq 2n$. Since the join operation preserves the tree height, the depth of any k-SF tree is at most $2k-1$. The number of distinct labeled paths of length $2k-1$ starting from a node with a given label is at most $|V|^{2k-1}$, and any two occurrences of the same path in the CCT are joined. The inequality $|k\text{-SF}| \leq |V|^{2k}$ follows by summing up over all distinct labels. □

When k is not too large and the number $|V|$ of routines is small compared to the CCT size n, the k-SF can be considerably smaller than the CCT, as we will see in Section 7.

We now show that the k-SF includes all activated paths of length up to k, and thus can be used for constructing the k-CCF:

Lemma 3. *For each activated path $\pi = \langle v_1, ..., v_q \rangle$ of length $q \leq k$, there is a tree in k-SF containing π.*

Proof. Let v be the deepest slab boundary node that is an ancestor of v_1. Since the distance between v and v_1 in the CCT is smaller than k, π has length at most k, and the k-SF trees have depth $2k-1$, then π is fully contained in the k-SF tree with root label $\ell(v)$. □

Property 2. *k-CCF $= inv_k(k\text{-SF})$.*

Proof. By Lemma 3 any q-context in the CCT with $q \leq k$ also appears in some k-SF tree. Since the inverse is obtained by joining reversed q-contexts, it follows that $inv_k(\text{CCT}) \subseteq inv_k(k\text{-SF})$. The opposite inclusion can be proved by observing that any root-to-leaf label path in the k-SF is also a label path starting at some slab boundary node of the CCT, and thus $inv_k(k\text{-SF}) \subseteq inv_k(\text{CCT})$. The claim follows by Property 1. □

By Lemma 1 and Property 2, the k-CCF can be computed from a k-SF with n nodes in $O(kn)$ time. As a technical note, we observe that computing correct frequency counters in the k-CCF using the inv_k primitive requires all counters of nodes in the top regions of the k-SF, except for the tree with root label r, to be first cleared to zero.

Algorithm. We now show an online algorithm for building the k-SF in constant worst-case time per trace operation. The approach is similar to the classical CCT construction algorithm [2]. The main difference is that, instead of having one current location as in the CCT, in the k-SF we have two current locations, working on two trees at a time. One location t points to a node in a top region (*top location*); the other, denoted by b, points to a node in a bottom region (*bottom location*). Both locations are locally updated in the same way as the CCT. The algorithm starts with an empty k-SF and maintains the following data:

- A set R of roots of k-SF trees. Since each tree has a distinct root label, R is stored in a mapping table so that, given a label x, it is possible to find in $O(1)$ time the tree associated with x: we denote this operation `find`(R, x).
- The top and bottom pointers t and b.
- A shadow stack S containing, for each pending routine activation, the corresponding $\langle t, b \rangle$ pair of locations.

Figure 7 shows how to update the k-SF at each `call` and `return` event. The stack S is initialized with the special pair $\langle$`null`, `null`$\rangle$. When the root function r is called, the algorithm creates the first root of the k-SF, pointed to by t (line 6). Let $d = |S| - 1$ denote the length of the current calling context. As long as $d < k$, $b =$ `null` and we work only on the top region of the tree rooted at r, adding nodes and updating counters at location t (lines 9–14). When $d \geq k$, both t and b locations are updated (lines 2–16 and 17–25, respectively). Every time d reaches a positive multiple of k (line 6), t enters a bottom region of the forest, and thus we move b to t and raise t back to the top by letting it point to a (possibly new) tree root in the k-SF (lines 3–7).

If lookups at lines 9 and 18 are implemented with a hash table, the algorithm requires constant time per `call` event.

procedure `call`(x):
1: $\langle t, b \rangle \leftarrow$ top of stack S
    // update top region
2: **if** $(|S| - 1)$ **mod** $k = 0$ **then**
3:     $b \leftarrow t$
4:     $t \leftarrow$ `find`(R, x)
5:     **if** $t =$ `null` **then**
6:         add root t with $\ell(t) = x$ and $c(t) = 0$ to k-SF and R
7:     **end if**
8: **else**
9:     find child w of node t with label $\ell(w) = x$
10:     **if** $w =$ `null` **then**
11:         add node w with $\ell(w) = x$ and $c(w) = 0$ to k-SF
12:         add arc (t, w) to k-SF
13:     **end if**
14:     $t \leftarrow w$
15: **end if**
16: increase $c(t)$ by 1
    // update bottom region
17: **if** $b \neq$ `null` **then**
18:     find child u of node b with label $\ell(u) = x$
19:     **if** $u =$ `null` **then**
20:         add node u with $\ell(u) = x$ and $c(u) = 0$ to k-SF
21:         add arc (b, u) to k-SF
22:     **end if**
23:     $b \leftarrow u$
24:     increase $c(b)$ by 1
25: **end if**
26: push $\langle t, b \rangle$ onto stack S

procedure `return`:
1: pop stack S

Figure 7. k-SF construction algorithm.

Exhaustive Instrumentation vs. Sampling. For the sake of simplicity, the algorithm we presented for on-line construction of the k-SF assumes all call/return events are traced using exhaustive instrumentation. Notice that full event tracing is required in all applications where perfect accuracy is needed, e.g., debugging or intrusion detection [12]. If we can settle for approximate results, a simpler alternative to the k-SF could be to just walk up to k frames on the top of the stack periodically, joining the traced k-contexts to an initially empty k-CCF. In this way, one could build directly a k-CCF on the fly, without having to first construct a k-SF. However, as discussed in [12, 35], a low-overhead solution using sampled stack-walking alone can be rather inaccurate. To overcome this problem, the authors of [35] use an effective technique called *bursting*, where stack-walking is immediately followed by a short period, called a burst, during which the profiler traces each and every routine call and return. This substantially increases accuracy while keeping the running time reasonably low. We remark that building a k-CCF on the fly during a burst can be rather expensive, requiring $O(k)$ time per call. A natural way to take advantage of the benefits of bursting in collecting k-context profiles consists of feeding the k-SF algorithm with an event trace generated by sampled bursting. By suitably tuning the sampling frequency and the burst duration, this approach can yield effective time/accuracy tradeoffs. We analyze this method experimentally in Section 7.

Application	Call graph	CCT	CCT (call sites)	Call tree	CCT depth
amarok	13 957	6 212 090	13 794 470	1 254 852 066	210
ark	10 081	3 859 081	8 171 612	238 179 120	192
audacity	6 880	5 264 498	13 131 115	960 243 960	305
dolphin	9 349	3 451 678	11 667 974	185 785 164	160
evince	4 869	2 003 454	6 772 430	92 775 397	226
firefox	6 076	12 670 331	30 294 063	617 296 926	433
gedit	5 934	3 927 949	4 183 946	419 970 719	263
ghex2	3 778	960 781	1 868 555	145 049 522	158
gimp	5 275	11 350 729	26 107 261	819 811 432	291
gwenview	11 510	4 970 270	9 987 922	459 196 696	281
inkscape	6 426	5 854 809	13 896 175	624 835 038	299
kile	12 280	5 450 415	12 936 360	484 246 000	230
oowriter	16 541	16 371 565	41 395 182	657 189 223	268
sudoku	5 347	1 335 864	2 794 177	186 644 308	266
vlc	5 808	1 844 857	3 295 907	122 164 908	173

Table 1. Number of nodes of call graph, CCT (both with and without call sites), call tree, and depth of the CCT for different Linux applications.

Handling Recursion. A major problem related to the construction of the CCT is that its depth may be unbounded in the presence of recursion. To overcome this problem, Ammons *et al.* [2] define a vertex equivalence relation on the call tree so that all occurrences of a given procedure name on a same path from the root to a leaf are equivalent. While this allows it to bound the depth of the CCT to the number of distinct routines in the program, it implies adding back-edges, which break the tree structure of the CCT and its context-uniqueness property with respect to the dynamic call tree [2]. In contrast, by Definition 7 the k-SF has a depth bounded by $2k - 1$ and maintains the context-uniqueness property by avoiding back-edges.

6. Experimental Setup

In this section we give some details on our implementation, benchmarks, and experimental methodology.

Tree implementation. We implemented our data structures in C in a common framework that makes the different versions directly comparable. We used two different representations for tree nodes:

- a standard first-child, next-sibling representation, which is very space-efficient and guarantees that the existence of a node child can be checked in time proportional to the number of children;
- an *indexed representation* obtained by statically analyzing the code of each routine and associating an integer key to each called subroutine. Each tree node stores its children in a direct-access array, which is indexed by the integer keys. With this representation we can check the existence of a node child in constant time with just one table lookup. The indexing works only for direct calls: indirect calls are added to a separate list, similarly to the standard representation.

According to our experiments with several benchmarks, the average degree of CCT nodes is a small constant around 2-3, making the classical first-child, next-sibling tree representation a simple and efficient solution for maintaining a CCT. Conversely, since the k-SF tends to have larger degrees due to the tree join operations (see Definition 7), the indexed representation can deliver substantial speedups especially for small values of k. We will discuss this issue in more depth in Section 7.

Data structures. In addition to the CCT, k-CCF, and k-SF, we also considered variants of the k-CCF obtained by pruning nodes according to two different rules:

- *pruned k-CCF*. Obtained by removing from the k-CCF all chain subtrees that lead to a leaf. For each such chain, we maintain in the partial k-CCF only its first node, along with a pointer to a corresponding k-SF or CCT node from which the pruned chain can be univocally reconstructed. This pruning rule is motivated by the worst-case example of Section 4: in a CCT chain with distinct node labels, each node can be represented in $\Theta(k)$ trees of the k-CCF, while these repeated occurrences are eliminated by chain pruning.
- *$x\%$-similar ∞-CCF*. Contains *all* calling contexts of maximal length in which most invocations of each routine occur. These paths are the most interesting from a performance profiling perspective and, as we will see in Section 7, they tend to be rather short. The $x\%$-similar ∞-CCF is formally defined as follows. We say that a node u in a k-CCF tree with root r is *$x\%$-similar* to r if its counter is sufficiently large w.r.t. the counter of r, namely if $c(u) \geq (x/100)\, c(r)$. Otherwise, the node is called *$x\%$-dissimilar*. Monotonicity of counters discussed in Section 4 guarantees that all descendants of a dissimilar node are also dissimilar. The $x\%$-similar ∞-CCF is obtained by removing from the ∞-CCF all $x\%$-dissimilar nodes.

We also considered two scenarios for constructing the CCT and the k-SF: exhaustive instrumentation and static bursting with a 2 msec sampling interval and a 0.2 msec burst length (see Section 5.2 and [13, 35]).

Benchmarks. Tests were performed on a variety of large-scale Linux applications, including an Internet browser (`firefox`), graphics programs (`inkscape` and `gimp`), an archiver (`ark`), an hexadecimal file viewer (`ghex2`), audio players/editors (`amarok` and `audacity`), and the Open Office word processor (`oowriter`). To ensure deterministic replay of the execution of the interactive applications, following [13] we used recorded execution traces of typical usage sessions. Statistical information about test sets is shown in Table 1. In our experiments, we considered the simplest scenario where distinct call sites within the same routine are regarded as equivalent. We notice that, even in this case, the calling context trees of the benchmarks we analyzed contain several million nodes. By maintaining distinct nodes for distinct call sites, the number of CCT nodes grows by a factor between about 2 and 3 in our test suite, making the quest for space-efficient techniques even more important (see also [27]).

Platform. Running times were measured on a 2.8 GHz Intel Core i7 with 3 GB of main memory, running Debian 6.0.4, Linux Kernel 2.6.32, 32 bit.

7. Experimental Results

In this section we present an experimental analysis of our data structures. The experiments aim at studying accuracy of k-context profiling, at quantifying running time and space requirements of our data structures, and at tuning parameter k in practical scenarios.

Accuracy of k-context profiles. The motivating example in Section 1 shows that relevant profiling information may be missing in the call graph and hidden in the CCT. For the example we discussed, a 2-context profile appears to be more informative than both edge profiles and full context sensitive profiles. In general, the most interesting call paths that lead to a routine appear to be those of *maximal length* in which most invocations of the routine occur. We notice that the notion of similarity introduced in Section 6 characterizes such interesting paths in a natural way: the interesting paths are precisely those that lead to a leaf in the $x\%$-similar ∞-CCF. The question we address here is which values of k are large enough in practice so that the k-CCF contains most such paths.

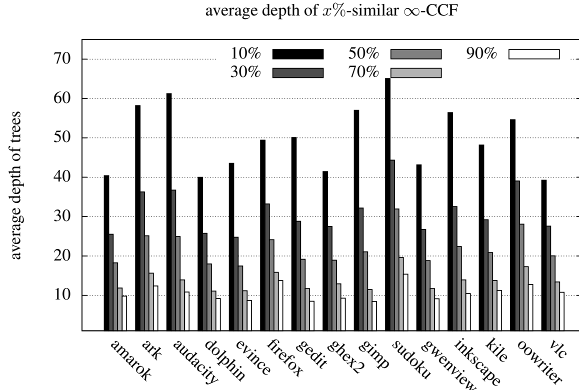


Figure 8. Average depth of trees in the x%-similar ∞-CCF for different similarity thresholds x.

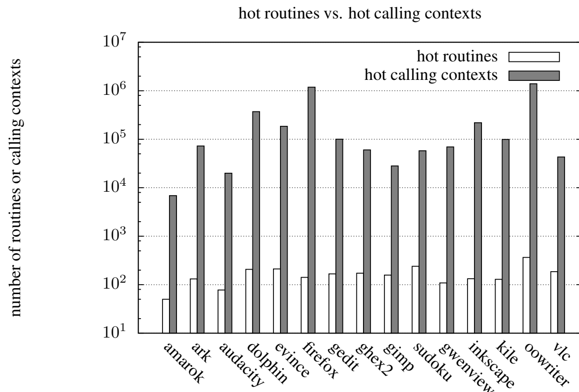


Figure 9. Number of routines and calling contexts with the highest activation counters accounting for 90% of the calls (hot routines and hot calling contexts).

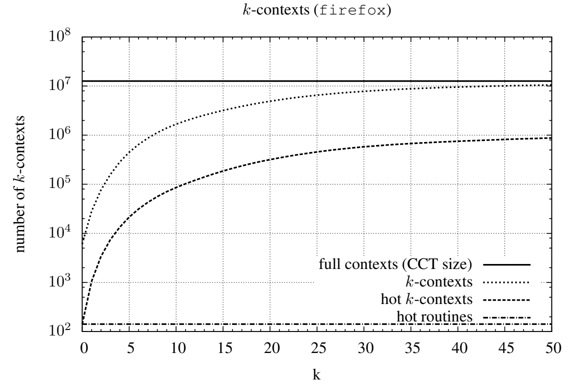


Figure 10. Number of k-contexts (k-CCF leaves) and number of k-contexts with the highest activation counters accounting for 90% of the calls (hot k-contexts) for $k \leq 50$ on the `firefox` benchmark.

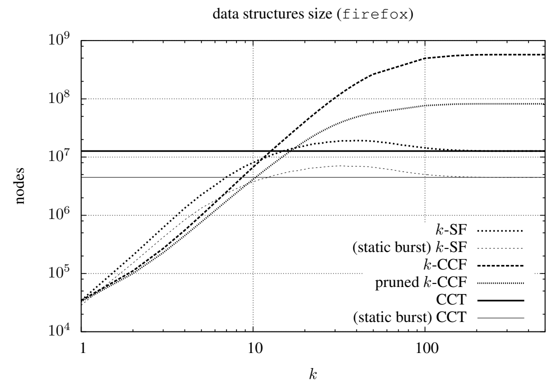


Figure 11. Data structures size as a function of k on the `firefox` benchmark.

In Figure 8 we report the outcome of an experiment in which we evaluate the average depth of $x\%$-similar ∞-CCF trees for different similarity thresholds x. As expected by the definition of similarity, the depth decreases as we increase the threshold: the larger x, the more nodes are pruned from ∞-CCF trees. In practice, the depth is a small constant, which depends on x but is quite similar across the different benchmarks. Comparing these values with the CCT depths reported in Table 1, the experiment suggests that rather small values of k are sufficient on average to detect the hottest call paths, without the need to build an entire CCT. For instance, by choosing $k \in [10, 20]$, the k-CCF records for each routine v most partial contexts of v whose counters differ by at most 10% ($x = 90\%$) from $c(v)$, independently of the specific benchmark. Although not reported in Figure 8, we also observed that the average depth would be even smaller by focusing on hot trees only, i.e., by omitting from the average depth computation trees whose root counter is small enough. Such hot trees are the natural target of context sensitive profiling.

Size and skewness of k-context profiles. Figure 9 compares the numbers of hot routines and hot distinct calling contexts, showing that these numbers can differ by up to four orders of magnitude. This suggests that full individual calling contexts tend to be substantially colder than individual routines in typical applications, motivating our effort in exploring partial contexts. In our experiment, hot items (routines or contexts) are those accounting for 90% of the total number of calls. In Figure 10 we compare the number of k-contexts and hot k-contexts for $k \leq 50$ on the `firefox` benchmark. Values of k larger than 50 up to the CCT depth (433) exhibit negligible variations and are not plotted. The chart shows an initial exponential growth followed by a slow convergence, confirming that small values of k are the most interesting in practice. As a frame of comparison, we also report the number of hot routines and the number of full distinct calling contexts (CCT size) for the same benchmark.

Space usage. We first evaluate the space required by our data structures as a function of k. Figure 11 plots the number of nodes in the CCT, k-CCF, k-SF, and pruned k-CCF for $k \in [1, 500]$ for the `firefox` benchmark (notice that 500 is larger than the CCT depth reported in Table 1 and Property 5 of Section 4 holds). While the number of CCT nodes is obviously constant, the sizes of the other data structures can differ by up to five orders of magnitude and largely benefit from small values of k. When $k \leq 12$, all our data structures are smaller than the CCT, suggesting that the space bound given in Theorem 1 is overly pessimistic. For larger values of k, the k-CCF can be much larger than the CCT, but the chain pruning rule is very effective and reduces the size considerably. The k-SF curve has an increasing trend up to a maximum value roughly equal to $1.6n$ (the theoretical bound is $2n$, see Lemma 2). Then, it converges to the CCT size: the larger k, the smaller the number of trees in the k-SF. For $k = \infty$ the forest consists of a single tree, equal to the CCT. Figure 11 also shows the size of the

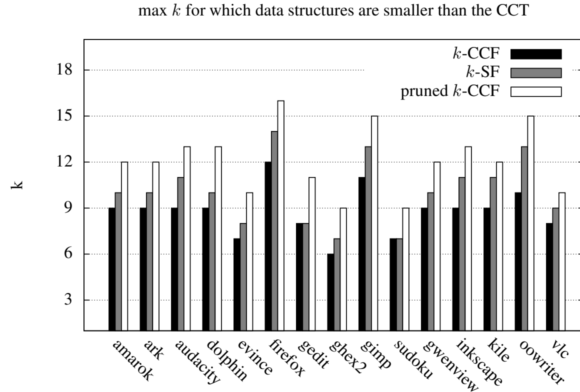


Figure 12. Max values of k for which k-CCF, k-SF, and pruned k-CCF are smaller than the CCT.

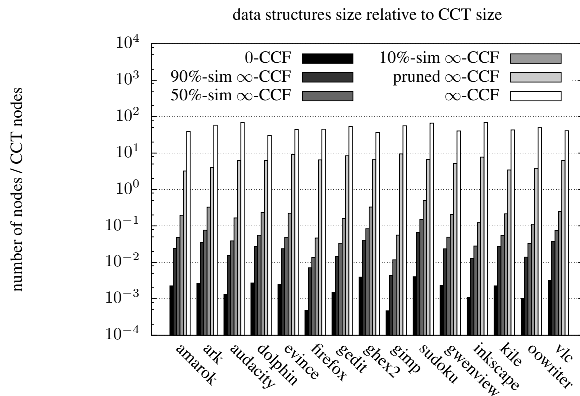


Figure 13. Minimum and maximum k-CCF size and size of x%-similar ∞-CCF for different similarity thresholds x, normalized over the number of CCT nodes.

CCT and of the k-SF as a function of k constructed on a call/return event stream generated with static bursting: notice that the size of the CCT with static busting is about 30% of the size resulting from exhaustive instrumentation. The bursting technique yields a similar space reduction for the k-SF, showing that it can be effectively combined with our method.

Similar results are confirmed on the entire set of benchmarks. This is shown in Figure 12, which plots the largest value of k for which our data structures are smaller than the CCT, and in Figure 13, which plots the best-case ratio (corresponding to the case $k = 0$) and the worst-case ratio (corresponding to the case $k = \infty$, both with and without pruning) between the numbers of k-CCF and CCT nodes.

Finally, we have measured the number of nodes of the x%-similar ∞-CCF for different similarity thresholds x. The results, reported in Figure 13, show that this number is very small compared to the size of the CCT even for small values of x, providing an extremely compact explicit representation of *all* hot paths of a program, with no length restrictions.

Running time. As a first experiment, we analyzed the impact of k on the performance of the construction algorithms of the different data structures. Since our approach is independent of any specific instrumentation mechanisms, and different techniques for tracing routine enter and exit events might incur rather different overheads in practice, we focus on the time required by the analysis routines only, omitting instrumentation times. Our tests, whose

Application	CCT (∞-SF)	1-SF	1-SF (indexed)	∞-SF (indexed)	k
amarok	11.66	2 029.90	33.05	14.52	70
ark	2.29	178.18	12.90	3.33	43
audacity	11.05	118.84	25.41	13.36	40
dolphin	1.64	45.11	6.16	2.40	21
evince	1.96	8.76	2.64	2.28	14
firefox	5.33	55.38	7.07	6.85	38
gedit	4.50	34.85	8.19	5.78	13
ghex2	1.48	6.31	2.14	1.93	6
gimp	9.61	93.73	30.45	12.18	15
gwenview	4.12	381.16	26.21	6.20	45
inkscape	4.89	25.47	6.82	6.43	5
kile	4.12	315.80	17.13	5.99	41
oowriter	6.05	38.21	9.57	7.56	10
sudoku	2.34	11.43	2.87	2.60	13
vlc	1.11	49.58	1.60	1.50	28

Table 2. Comparison of running times.

outcome is exemplified by Figure 14a on the `firefox` benchmark, show that the time required by the indexed k-SF is barely affected by varying k, while the running time decreases steeply for the standard representation (the time spent for constructing the CCT is independent of k, and therefore constant). This depends on the fact that the average degree of internal nodes of the k-SF decreases when k becomes larger as shown in Figure 14b, and the indexed representation is more effective on large degrees.

In Table 2 we summarize performance figures of our data structures on all benchmarks, reporting the running times for constructing CCT and k-SF for $k = 1$ and $k = \infty$. We remark that $k = 1$ yields the largest construction times for the k-SF and that CCT and ∞-SF are exactly the same data structure. The indexed implementations of the CCT and of the 1-SF proved to be respectively slower and much faster than the standard implementations: on average, the construction algorithm for the indexed 1-SF is $2.8x$ slower than the CCT, but can save more than two orders of magnitudes in space compared to maintaining the CCT (see, e.g., Figure 11). The table also reports the smallest value of k for which the standard implementation of the 1-SF becomes preferable to the indexed version: for almost all benchmarks, this breakpoint value is rather large, proving the usefulness of supporting direct access to tree node children. We remark that performance can be greatly improved by using sampled bursting techniques (see [13]) along with our data structures.

8. Related Work

There is a vast literature related to software profiling. In this section we survey research on context-sensitive profiling that appears to be most relevant to our work.

Call path profiling. Call graph profiles produced by gprof introduce for the first time a form of context sensitivity [15], by associating procedure timing with caller-callee pairs and relating it with edges of the call graph (hence the alternative name edge profiling). It has been later observed that a single level of context sensitivity may yield to several inaccuracies [23, 28]. To overcome these issues, Goldberg and Hall [17] propose call path profiles: a call path is a sequence of function pairs in a caller-callee relationship, and the profile is a sorted list of call paths along with their metrics. The space usage with this approach, however, can be prohibitive. Limiting the length of profiled call paths as we do is mentioned in [24, 31] as a possible useful call tree/CCT compaction technique. However, differently from our work, in [24, 31] the implications of the basic idea are not further explored and no data structures for short paths are provided. The work in [3] introduces "Bottlenecks", an interactive tool for helping developers identify hot call-

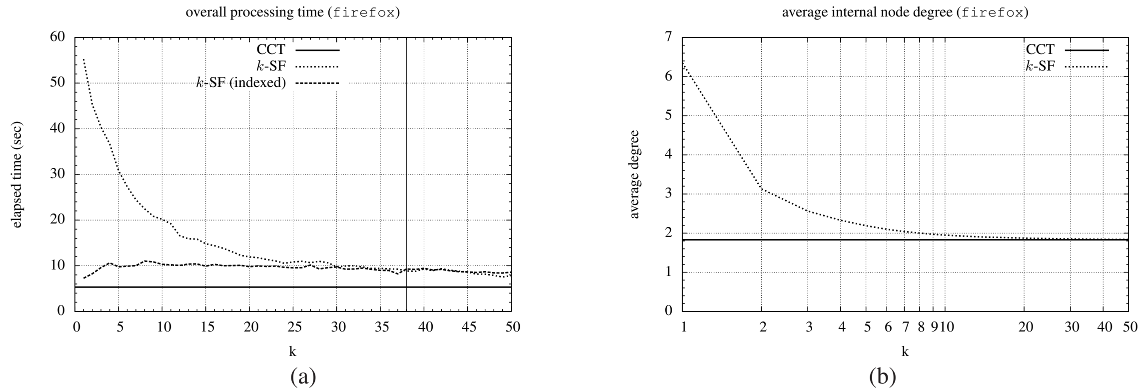


Figure 14. Running time of the k-SF construction algorithms and average internal degree of CCT and k-SF as a function of k.

ing contexts of bounded length. Differently from our approach, the Bottlenecks tool works offline on a previously recorded call tree allowing users to interactively mine hot call paths. Their approach scales poorly to the large C/C++ applications we considered, which produce call trees up to billion nodes. They do not provide any time/space bounds, neither theoretically nor experimentally. Our k-SF data structure is constructed online with provable time bounds, is orders of magnitude smaller than a call tree, and supports fully automatic identification of hot call paths using a k-CCF.

Calling context trees. Calling context trees have been introduced in [2] as a compact data structure to associate performance metrics with paths through a program's call graph: Ammons, Ball, and Larus show how to build a CCT by instrumenting procedure code and to compute metrics by exploiting hardware counters available in modern processors. Since exhaustive instrumentation can lead to a large slowdown, a variety of techniques have been proposed to reduce the time overhead of context-sensitive profilers. Bernat and Miller [9] generate path profiles including only a subset of methods of interest, while statistical profilers [6, 14, 17, 32] attribute metrics to calling contexts through periodic sampling of the call stack. Although much faster than exhaustive instrumentation, sample-driven stack-walking might incur significant loss of accuracy with respect to the complete CCT [12, 35]. More accurate results can be obtained by combining sampling with bursting [5, 18, 35], i.e., by performing stack-walking followed by a burst during which the profiler traces every routine call and return. As we have discussed in Section 5.2, sampling and bursting can be integrated with our algorithms to reduce profiling overhead.

Space issues in context-sensitive profiling. Although the CCT compactly represents all distinct calling contexts encountered during the execution of a program, it has been noticed in previous works that it could be still very large and difficult to analyze [12, 13, 35], especially if collected profiles are not only complete, but also call-site aware [27]. Different techniques have been thus proposed to reduce either profile data or the amount of data presented to the user. Most previous works focus on performance analysis applications, where identifying a few hot contexts is typically sufficient to guide code optimization. Incremental call-path profiling lets the user choose a subset of routines to be analyzed [9], while call path refinement helps users focus the attention on performance bottlenecks by limiting and aggregating the information revealed to the user [16]. To this end, in [27] the authors provide a dumper that serializes the CCT as a dynamic call graph. The hot calling context tree proposed in [13] is a subtree of the CCT that includes only hot nodes and their ancestors, compactly representing the hot calling contexts encountered during a program's execution. Our data structures provide another solution to the profile data abundance problem, allowing programmers to control the space usage by appropriately choosing the value of k.

Other approaches are based on compactly encoding the calling contexts encountered during the execution of a program [12, 29], which is especially useful for tasks such as residual testing, optimization, statistical bug isolation, and anomaly-based intrusion detection. In contrast, our work focuses on explicit context encoding for performance profiling applications.

Flow-sensitive profiling. At the intraprocedural level, the seminal work of Ball and Larus [7] has spawned much research on flow-sensitive profiling [1, 2, 4, 8, 10, 11, 19, 21, 22, 30]. A path profile determines how many times each path in the control flow graph executes: path profiles terminate either at backedges or at procedure exits, and are thus acyclic. Though useful at driving many compiler optimizations, it has been observed in [26] that more opportunities can be exploited in the presence of information about longer paths: k-iteration paths are intraprocedural cyclic paths spanning up to k loop iterations, and can be computed efficiently by generalizing the Ball-Larus profiling algorithm. The work on k-iteration paths [26] has some similarities with our approach, although the focus there is on intraprocedural profiles.

9. Conclusions

In this paper we have studied a generalization of the classical calling context profiling approach, motivated by a key observation: full calling contexts starting from the program's root can be substantially colder than individual routines. In some cases, there may even be hot routines having no hot calling context at all. On the other hand, performance bottlenecks may still arise in short sequences of calls leading to a hot routine, making them a useful profiling target.

We have shown that profiles for call paths of length up to k can be compactly represented using a simple but rich data structure that we called k-calling context forest. Our experiments confirm that this data structure can provide effective space-accuracy tradeoffs for interprocedural performance profiling.

We regard it as an interesting open question how to construct a $x\%$-similar ∞-CCF for profiling all hot call paths using less space than the CCT. We expect that the techniques developed in this paper could be successfully applied to intraprocedural profiling as well, allowing it to detect hot paths of bounded length in the control flow graph.

Acknowledgments. We are indebted to Daniele Cono D'Elia for his help in generating some of the test suites of this paper and

to Fabrizio D'Amore, Bruno Ciciani, and Giuseppe F. Italiano for allowing us to use their computing equipment for our experiments. We would also like to thank the anonymous reviewers and the Program Committee for many useful comments on our work.

References

[1] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. *SIGPLAN Not.*, 39(4):568–582, 2004.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, 1997. ISSN 0362-1340.

[3] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP*, volume 3086 of *LNCS*, pages 170–194, 2004.

[4] T. Apiwattanapong and M. J. Harrold. Selective path profiling. In *Proc. ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 35–42. ACM, 2002.

[5] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *PLDI*, pages 168–179. ACM, 2001.

[6] M. Arnold and P. Sweeney. Approximating the calling context tree via sampling. Technical Report RC 21789, IBM Research, 2000.

[7] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, 1996.

[8] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: the showdown. In *POPL*, pages 134–148. ACM, 1998.

[9] A. R. Bernat and B. P. Miller. Incremental call-path profiling. Technical report, University of Wisconsin, 2004.

[10] M. D. Bond and K. S. McKinley. Continuous path and edge profiling. In *Proc. 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 130–140. IEEE Computer Society, 2005.

[11] M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizers. In *CGO*, pages 205–216. IEEE Computer Society, 2005.

[12] M. D. Bond and K. S. McKinley. Probabilistic calling context. *SIGPLAN Not. (proceedings of the 2007 OOPSLA conference)*, 42(10): 97–112, 2007.

[13] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining hot calling contexts in small space. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 516–527. ACM, 2011.

[14] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proc. 19th Annual International Conf. on Supercomputing*, pages 81–90. ACM, 2005.

[15] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler (with retrospective). In K. S. McKinley, editor, *Best of PLDI*, pages 49–57. ACM, 1982.

[16] R. J. Hall. Call path refinement profiles. *IEEE Trans. Softw. Eng.*, 21 (6):481–496, 1995.

[17] R. J. Hall and A. J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proc. Summer 1993 USENIX Technical Conference*, pages 1–19. USENIX Association, 1993.

[18] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Proc. 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[19] R. Joshi, M. D. Bond, and C. Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *CGO*, pages 239–250, 2004.

[20] Z. Lai, S.-C. Cheung, and W. K. Chan. Inter-context control-flow and data-flow test adequacy criteria for nesc applications. In M. J. Harrold and G. C. Murphy, editors, *SIGSOFT FSE*, pages 94–104. ACM, 2008. ISBN 978-1-59593-995-1.

[21] J. R. Larus. Whole program paths. *SIGPLAN Not.*, 34(5):259–269, 1999.

[22] D. Melski and T. W. Reps. Interprocedural path profiling. In S. Jähnichen, editor, *CC*, volume 1575 of *Lecture Notes in Computer Science*, pages 47–62. Springer, 1999. ISBN 3-540-65717-7.

[23] C. Ponder and R. J. Fateman. Inaccuracies in program profilers. *Softw., Pract. Exper.*, 18(5):459–467, 1988.

[24] S. P. Reiss and M. Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 221–230, 2001.

[25] T. W. Reps, T. Ball, M. Das, and J. R. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In M. Jazayeri and H. Schauer, editors, *ESEC / SIGSOFT FSE*, volume 1301 of *Lecture Notes in Computer Science*, pages 432–449. Springer, 1997. ISBN 3-540-63531-9.

[26] S. Roy and Y. N. Srikant. Profiling k-iteration paths: A generalization of the ball-larus profiling algorithm. In *CGO*, pages 70–80, 2009.

[27] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, and M. Mezini. JP2: Call-site aware calling context profiling for the java virtual machine. *Science of Computer Programming*, 2012. ISSN 0167-6423. doi: 10.1016/j.scico.2011.11.003.

[28] J. M. Spivey. Fast, accurate call graph profiling. *Softw., Pract. Exper.*, 34(3):249–264, 2004.

[29] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *ICSE*, pages 525–534. ACM, 2010. ISBN 978-1-60558-719-6.

[30] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: compactly numbering interesting paths. In *POPL*, pages 351–362. ACM, 2007.

[31] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. In *International Conference on Computational Science*, volume 3038 of *LNCS*, pages 440–447, 2004.

[32] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 78–87. ACM Press, 2000.

[33] M. R. Woodward, D. Hedley, and M. A. Hennell. Experience with path analysis and testing of programs. *IEEE Trans. Software Eng.*, 6 (3):278–286, 1980.

[34] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In M. Young and P. T. Devanbu, editors, *SIGSOFT FSE*, pages 81–91. ACM, 2006. ISBN 1-59593-468-5.

[35] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI*, pages 263–271, 2006.