SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2348

Mining hot calling contexts in small space‡

Daniele Cono D'Elia[1], Camil Demetrescu[1,*,†] and Irene Finocchi[2]

[1]*Department of Computer, Control, and Management Engineering, Sapienza University of Rome, Italy*
[2]*Department of Computer Science, Sapienza University of Rome, Italy*

SUMMARY

Calling context trees (CCTs) associate performance metrics with paths through a program's call graph, providing valuable information for program understanding and performance analysis. In real applications, however, CCTs might easily consist of tens of millions of nodes, making them difficult to analyze and also hurting execution times because of poor access locality. For performance analysis, accurately mining only *hot* calling contexts may be more useful than constructing an entire CCT with millions of uninteresting paths, because the distribution of context frequencies is typically very skewed. In this article, we show how to exploit this property to considerably reduce the CCT size, introducing a novel runtime data structure, called *hot CCT* (HCCT), in the spectrum of representations for interprocedural control flow. The HCCT includes only hot nodes and their ancestors in a CCT and can be constructed independently from it by using fast, space-efficient algorithms for mining frequent items in data streams. With this approach, we can distinguish between hot and cold contexts on the fly while obtaining very accurate frequency counts. We show, both theoretically and experimentally, that the HCCT achieves a similar precision as the CCT in a space that is several orders of magnitude smaller and roughly proportional to the number of hot contexts. Our approach can be effectively combined with previous context-sensitive profiling techniques, as we show for static bursting. We devise an implementation as a plug-in for the `gcc` compiler that incurs a slowdown competitive with the `gprof` call-graph profiler while collecting finer-grained profiles. Copyright © 2015 John Wiley & Sons, Ltd.

Received 26 May 2015; Revised 28 July 2015; Accepted 7 August 2015

KEY WORDS: performance profiling; dynamic program analysis; data streaming algorithms; frequent items; program instrumentation

1. INTRODUCTION

Context-sensitive profiling provides valuable information for program understanding, performance analysis, and runtime optimizations. Previous works demonstrated its effectiveness for tasks such as residual testing [1, 2], function inlining [3], statistical bug isolation [4, 5], performance bug detection [6], object allocation analysis [7], or anomaly-based intrusion detection [8]. A *calling context* is a sequence of routine calls that are concurrently active on the runtime stack and that lead to a program location. Collecting context information in modern object-oriented software is very challenging: application functionalities are divided into a large number of small routines, and the high frequency of function calls and returns might result in considerable profiling overhead, Heisenberg effects, and huge amounts of contexts to be analyzed by the programmer.

*Correspondence to: Camil Demetrescu, Department of Computer, Control, and Management Engineering, Sapienza University of Rome, Via Ariosto, 25 00185 Rome, Italy.

†E-mail: demetres@dis.uniroma1.it

‡This article is a significantly extended version of D'Elia D. C., Demetrescu C., and Finocchi I., 'Mining hot calling contexts in small space', in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '11), ACM, New York, NY, USA, 2011, 516–527. Work supported in part by PRIN project AMANDA ("Algorithmics for MAssive and Networked DAta"), funded by the Italian Ministry of University and Research (MIUR).

Several data structures have long been used to maintain information about interprocedural control flow. In a *call graph*, nodes represent routines, and arcs represent caller–callee relationships. The use of call-graph profiles was pioneered by `gprof` [9], which attributes the time for each routine to its callers by propagating times along edges of the call graph. Although very space and time efficient, this approach can lead to misleading results, as pointed out in [10, 11]. On the other side, each node of a *call tree* represents a different routine invocation. This yields very accurate context information but requires extremely large (possibly unbounded) space. The inaccuracy of call graphs and the huge size of call trees have motivated the introduction of *calling context trees* (CCT) [12]: unlike call trees, CCTs do not distinguish between different invocations of the same routine within the same context. While maintaining good accuracy, typically CCTs are several orders of magnitude smaller than call trees: the number of nodes may be large in the presence of recursion, but this is seldom the case in practice. The exhaustive approach to constructing a CCT is based on the instrumentation of each routine call and return, incurring considerable slowdown even using efficient instrumentation mechanisms [12, 13]. Sampled stack-walking trades accuracy for performance. The adaptive bursting mechanism proposed in [13] selectively inhibits redundant profiling, dramatically reducing the overhead over the exhaustive approach while preserving good profile accuracy.

As noticed in previous works [8, 13], even CCTs may be very large and difficult to analyze in several applications. Moreover, their sheer size might hurt execution time because of poor access locality during construction and query. As an example, in Table I, we report the number of nodes of the call graph, call tree, and CCT for a variety of off-the-shelf applications in a typical Linux distribution and for benchmarks from popular suites. These numbers have been

Table I. Number of nodes of call graph, call tree, calling context tree, and number of distinct call sites for different applications.

Application	|Call graph|	|Call sites|	|CCT|	|Call tree|
amarok	13 754	113 362	13 794 470	991 112 563
ark	9 933	76 547	8 171 612	216 881 324
audacity	6 895	79 656	13 131 115	924 534 168
bluefish	5 211	64 239	7 274 132	248 162 281
dolphin	10 744	84 152	11 667 974	390 134 028
firefox	6 756	145 883	30 294 063	625 133 218
gedit	5 063	57 774	4 183 946	407 906 721
ghex2	3 816	39 714	1 868 555	80 988 952
gimp	5 146	93 372	26 107 261	805 947 134
gwenview	11 436	86 609	9 987 922	494 753 038
inkscape	6 454	89 590	13 896 175	675 915 815
oocalc	30 807	394 913	48 310 585	551 472 065
ooimpress	16 980	256 848	43 068 214	730 115 446
oowriter	17 012	253 713	41 395 182	563 763 684
pidgin	7 195	80 028	10 743 073	404 787 763
quanta	13 263	113 850	27 426 654	602 409 403
sudoku	5 340	49 885	2 794 177	325 944 813
vlc	5 692	47 481	3 295 907	125 436 877
botan	3 388	27 114	308 550	26 272 804 980
cairo-perf-trace	1 408	3 696	137 920	15 976 619 734
crafty	107	516	36 434 095	10 403 074 070
fhourstones	18	32	OOM	39 272 563 944
gobmk	1 133	4 049	OOM	21 909 088 291
ice-labyrinth	2 335	8 050	2 160 052	1 637 076 406
mount-herring	2 318	8 269	3 733 120	3 311 257 932
overworld	14 173	50 394	3 774 937	4 112 679 880
scotland	13 932	51 206	1 813 368	5 982 612 379
sjeng	57	221	OOM	28 370 207 811

OOM stands for *out of memory*. Calling context tree (CCT) is too large to be constructed in main memory on a 32-bit machine with 4 GB of RAM.

DOI: 10.1002/spe

obtained from short runs of each application, which yield millions of calling contexts. The optimistic assumption that each CCT node requires 20 bytes (previous works use larger nodes [11, 12]) already results in almost 1 GB needed just to store a CCT with 48 million nodes for an application such as OpenOffice Calc. To cope with space issues, the approach proposed in [8] maintains a one-word probabilistically unique value per context, achieving minimal space overhead when all contexts have to be stored, but specific information about them is unnecessary. As noted in [8], even this approach, however, can run out of memory on large traces with tens of millions of contexts.

Although very useful, for example, in bug or intrusion detection applications, probabilistic calling context was not designed for profiling with the purpose of understanding and improving application performance, where it is crucial to maintain for each context the sequence of active routine calls along with performance metrics. In this scenario, only the most frequent contexts are of interest, because they represent the hot spots to which optimizations must be directed. As observed in [13]: 'Accurately collecting information about hot edges may be more useful than accurately constructing an entire CCT that includes rarely called paths.' Figure 1 shows that for different applications, only a small fraction of contexts are hot: in accordance with the Pareto principle, more than 90% of routine calls take place in only 10% of contexts. The skewness of context distribution suggests that space could be greatly reduced by keeping information about hot contexts only and discarding on-the-fly contexts that are likely to be cold, that is, to have low frequency or to require little time all over the entire execution.

Contributions. In this article, we introduce a novel runtime data structure, called *hot CCT* (HCCT), that compactly represents all the hot calling contexts encountered during a program's execution, offering an additional intermediate point in the spectrum of data structures for representing interprocedural control flow. The HCCT is a subtree of the CCT that includes only hot nodes and their ancestors, also maintaining estimates of performance metrics (e.g., frequency counts) for hot calling contexts. Our main contributions can be summarized as follows:

- We formalize the concept of HCCT, and we cast the problem of identifying the most frequent contexts into a data streaming setting: we show that the HCCT can be computed without storing the exact frequency of all calling contexts, by using fast and space-efficient algorithms for mining frequent items in data streams. With this approach, we can distinguish between hot and cold contexts on the fly while obtaining very accurate frequency counts.
- We implement space-efficient context-sensitive profilers based on the Space Saving [14] streaming algorithm, for which we devise a novel, highly tuned implementation that might be of independent interest.

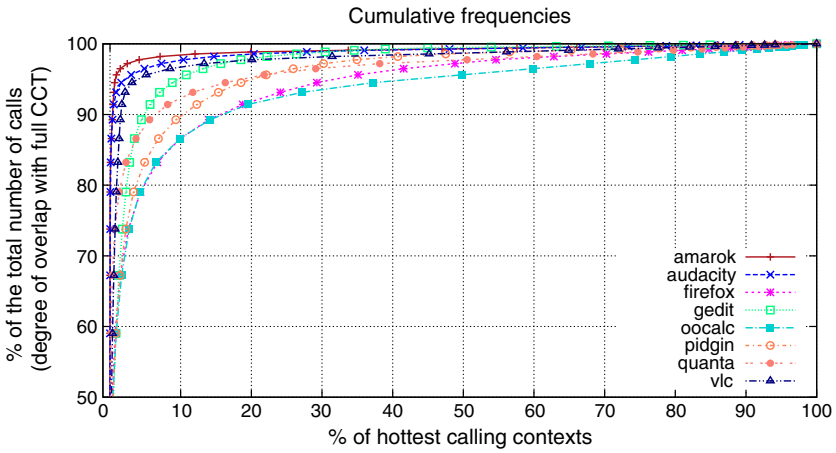


Figure 1. Skewness of calling contexts distribution on a representative subset of benchmarks. For instance, in `oocalc`, 10% of the hottest calling contexts account for more than 86% of all routine calls. CCT, calling context tree.

DOI: 10.1002/spe

- We integrate our space-efficient approach with previous techniques aimed at reducing time overhead: we focus, in particular, on static bursting [13], which offers very good time-accuracy trade-offs.
- We implement a full-fledged infrastructure for context-sensitive profiling of C/C++ Linux applications based on the `gcc` compiler and a set of libraries for address-to-symbol resolution and for the analysis and comparison of CCTs.
- We perform an extensive experimental analysis of performance and accuracy on a variety of prominent Linux applications. We test many different parameter settings and consider several metrics, including degree of overlap and hot edge coverage used in previous works [13, 15, 16]. The experiments not only confirm but also reinforce the theoretical prediction, showing that the HCCT represents the hot portions of the full CCT very well using only an extremely small percentage of the space required by the entire CCT. Even when the peak memory usage of our profilers is only 1% of standard context-sensitive profilers, we can show the following:
  - all the hottest calling contexts are always identified correctly (no false negatives);
  - frequency counters are close to the true values (average error less than 5%);
  - the number of false positives (cold contexts considered as hot) is small (0% to 5% for most benchmarks);
  - bursting [13] can be effectively combined with our approach;
  - while collecting profiles at a much finer grain, the slowdown is comparable with the widely used `gprof` call-graph profiler [9].

The rest of this article is organized as follows. Section 2 gives preliminary definitions about calling contexts and data stream algorithmics. Section 3 introduces the HCCT data structure and describes our approach. Section 4 focuses on implementation and engineering aspects, while Sections 5 and 6 present the methodology and the outcome of our experimental study, respectively. A comparison with related works is the subject of Section 7.

2. BACKGROUND

2.1. Calling context tree

The dynamic *calling context* of a routine invocation is the sequence of un-returned calls from the program's root function to the routine invocation. The CCT compactly represents all calling contexts encountered during the execution of a program. CCT nodes correspond to routines, and a path from a node v to the tree root represents the calling context of v. A routine with multiple contexts will appear more than once in a CCT, but each calling context is represented just once, and metrics for identical contexts are aggregated. Slightly extended definitions can be given to bound the depth of a CCT in the presence of recursion and to distinguish calls that take place at different call sites of the same calling procedure [12].

A CCT can be constructed on the fly during the execution of a program. Because in a tree data structure there is a unique path from the root for each of its nodes, it is rather intuitive to associate each calling context with exactly one node in the CCT. Let v be a cursor pointer that points to the current context, that is, to the node corresponding to the calling context of the currently active routine (v is initialized to the CCT root node). At each routine invocation, the algorithm checks whether v has a child associated with the called routine. If this is the case, the existing child is used, and its metrics are updated, if necessary. Otherwise, a new child of v is added to the CCT. In both cases, the cursor is moved to the callee. Upon routine termination, the cursor is moved back to the parent node in the CCT. This approach can be implemented either by instrumenting every routine call and return or by performing stack-walking if sampling is used to inhibit redundant profiling [13, 16, 17].

2.2. Frequent items in data streams

In recent years, there has been much interest in the design of algorithms able to perform near-real time analyses on massive data streams, where input data come at a very high rate and cannot be stored entirely because of their huge, possibly unbounded size [18, 19]. This line of research has

DOI: 10.1002/spe

been mainly motivated by networking and database applications: for instance, a relevant IP traffic analysis task consists of monitoring the packet log over a given link in order to estimate how many distinct IP addresses used that link in a given period of time. Because the stream may be very long and stream items may also be drawn from a very large universe (e.g., the set of source-destination IP address pairs), space-efficient data streaming algorithms can maintain a compact data structure that is dynamically updated upon arrival of new input data, supporting a variety of application-dependent queries. Approximate answers are allowed when it is impossible to obtain an exact solution using only limited space. Streaming algorithms are therefore designed to optimize space usage and update/query time while guaranteeing high solution quality.

The *frequent items* (a.k.a. heavy hitters) problem has been extensively studied in data streaming computational models. Given a frequency threshold $\phi \in [0, 1]$ and a stream of length N, the problem (in its simplest formulation) is to find all items that appear in the stream at least $\lfloor \phi N \rfloor$ times, that is, having frequency $\geqslant \lfloor \phi N \rfloor$. For instance, for $\phi = 0.1$, the problem seeks all items that appear in the stream at least 10% of the times. Notice that at most $1/\phi$ items can have frequency larger than $\lfloor \phi N \rfloor$. It can be proved that any algorithm that outputs an exact solution requires $\Omega(N)$ bits, even using randomization [19]. Hence, researchers focused on solving an approximate version of the problem.

Definition 2.1 ((ϕ, ε)-heavy-hitters problem)
Given two parameters $\phi, \varepsilon \in [0, 1]$, with $\varepsilon < \phi$, return all items with frequency $\geqslant \lfloor \phi N \rfloor$ and no item with frequency $\leqslant \lfloor (\phi - \varepsilon) N \rfloor$.

In the approximate solution, false negatives cannot exist; that is, all frequent items must be returned. Instead, some false positives are allowed, but their actual frequency is guaranteed to be at most εN-far from the threshold $\lfloor \phi N \rfloor$. Variants of the problem arise when, besides returning the heavy hitters, it is necessary to estimate accurately their true frequencies, when the stream length N is not known in advance or items are weighted.

Many different algorithms for computing (ϕ, ε)-heavy hitters have been proposed in the literature in the last 10 years. In this article, we focus on counter-based algorithms that, according to extensive experimental studies [20], have superior performance with respect to space, running time, and accuracy. Counter-based algorithms track a subset of items from the input and monitor counts associated with them. For each new arrival, the algorithms decide whether to store the item or not and, if so, what counts to associate with it. Update times are typically dominated by a small (constant) number of dictionary or heap operations.

3. HOT CALLING CONTEXT TREE

The execution trace of routine invocations and terminations can be naturally regarded as a stream of items. Each item is a triple containing routine name, call site, and event type. As shown in Table I, the number of distinct routines (i.e., the number of nodes of the call graph) is small compared with the stream length (i.e., to the number of nodes of the call tree), even for complex applications. Hence, non-contextual profilers – such as vertex profilers – can maintain a hash table of size proportional to the number of routines, using routine names as hash keys in order to update the corresponding metrics. This may be difficult in the case of contextual profiling, when the number of distinct calling contexts (i.e., the number of CCT nodes) is too large and hashing would be inefficient. Motivated by the fact that execution traces are typically very long and their items (calling contexts) are taken from a large universe, we cast the problem of identifying the most frequent contexts into a data streaming setting.

3.1. Definitions

Let N be the number of calling contexts encountered during a program's execution: N equals the number of nodes of the call tree, the sum of the frequency counts of CCT nodes, and the number of routine invocations in the execution trace.

DOI: 10.1002/spe

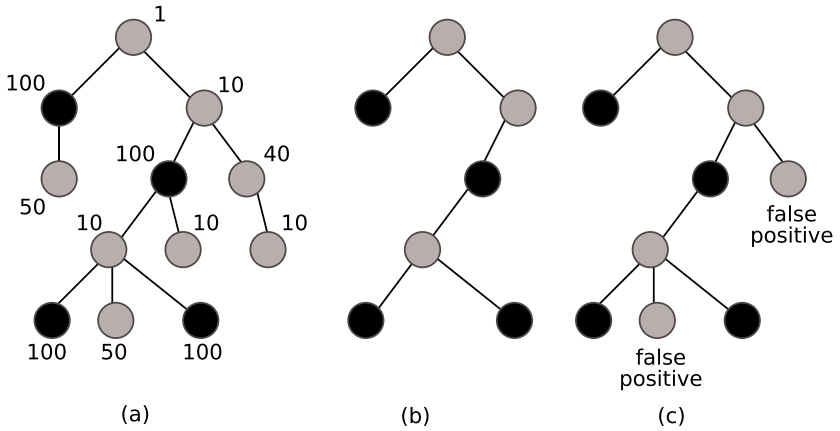


Figure 2. (a) Calling context tree (CCT) annotated with calling context frequency counts; (b) hot CCT (HCCT); and (c) (ϕ, ε)-HCCT. Hot nodes are black. In this example, $N = 581$, $\phi = 1/10$, and $\varepsilon = 1/30$: the approximate HCCT includes all contexts with frequency $\geqslant \lfloor \phi N \rfloor = 58$ and no context with frequency $\leqslant \lfloor (\phi - \varepsilon) N \rfloor = 38$.

Definition 3.1
A calling context is *hot* with respect to a frequency threshold $\phi \in [0, 1]$ if and only if the frequency count of its corresponding CCT node is $\geqslant \lfloor \phi N \rfloor$.

Any calling context that is not hot is said to be *cold*.

Definition 3.2
The HCCT is the (unique) subtree of the CCT obtained by pruning all cold nodes that are not ancestors of a hot node.

An example of HCCT is given in Figure 2(b). Note that the HCCT includes all the hot nodes and that all its leaves are necessarily hot. In graph theory, the HCCT corresponds to the Steiner tree of the CCT with hot nodes and the root used as terminals, that is, to the minimal connected subtree of the CCT spanning hot nodes and the root.

3.2. Relaxation and properties

The HCCT is the most compact data structure representing information about hot calling contexts. The space lower bound for the heavy-hitters problem (Section 2.2) extends to the problem of computing the HCCT, which cannot be calculated exactly in small space (in particular, using a space asymptotically smaller than the entire CCT). Hence, we relax the problem and compute an *approximate HCCT*, which we denote by (ϕ, ε)-*HCCT*, where $\varepsilon < \phi$ controls the degree of approximation. The (ϕ, ε)-HCCT contains all hot nodes (true positives) but may possibly contain some cold nodes without hot descendants (false positives). The true frequency of these false positives, however, is guaranteed to be at least $\lfloor (\phi - \varepsilon) N \rfloor$. Like the HCCT, the (ϕ, ε)-HCCT can be thought of as a minimal subtree of the CCT spanning a set of (ϕ, ε)-heavy hitters. Unlike the HCCT, a (ϕ, ε)-HCCT is not uniquely defined, because the set of (ϕ, ε)-heavy hitters is not unique: nodes with frequencies smaller than $\lfloor \phi N \rfloor$ and larger than $\lfloor (\phi - \varepsilon) N \rfloor$ may be either included in such a set or not.

The Venn diagram in Figure 3 summarizes some important relations:

- H $\subseteq$ A, where H is the set of hot calling contexts and A is a set of (ϕ, ε)-heavy hitters. Nodes in A\H are false positives.
- H $\subseteq$ HCCT. Nodes in HCCT\H are cold nodes having a descendant in H.
- A $\subseteq$ (ϕ, ε)-HCCT. Nodes in (ϕ, ε)-HCCT\A are cold nodes having a descendant in A.
- HCCT $\subseteq$ (ϕ, ε)-HCCT, as implied by the previous inclusions. Both of them are connected subtrees of the full CCT.

DOI: 10.1002/spe

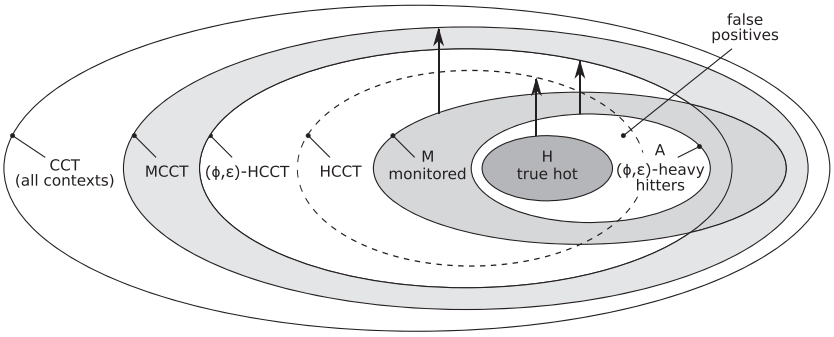


Figure 3. Tree data structures and calling contexts classification. We use graphical notation S ↑ T to indicate that T is the minimal subtree of the calling context tree (CCT) spanning all nodes in S. HCCT, hot CCT; MCCT, monitored calling context tree.

Figure 3 also introduces two additional structures: the set M of monitored nodes and the Monitored Calling Context Tree (MCCT), the subtree of the calling context tree spanning all nodes in M. In order to compute the set A of (ϕ, ε)-heavy hitters, counter-based streaming algorithms need to monitor a slightly larger set $M \supseteq A$ of elements. When a user query asks for the most frequent contexts, these algorithms prune M and return A.

Nodes in M\A can be either ancestors of nodes in A and thus already in the (ϕ, ε)-HCCT, or nodes not in the (ϕ, ε)-HCCT; for the latter category, we have to retain information about their ancestors as well, which might not be in the (ϕ, ε)-HCCT. Thus, in addition to M, our algorithm maintains the subtree MCCT of the CCT consisting of the nodes in M and of all their ancestors. At query time, the MCCT is appropriately pruned, and the (ϕ, ε)-HCCT $\subseteq$ MCCT is returned.

Discussion and example. To understand why the heavy hitters and the approximate HCCT are not maintained directly, but derived by pruning M and MCCT, respectively, we discuss a scenario where M is larger than the number of heavy hitters. Consider the following example: the execution trace contains the initial invocation of the `main` function, which in turn invokes once a routine p, and $N-2$ times a different routine q. Hence, we have three distinct calling contexts: `main`, `main`→`p`, and `main`→`q`. Assume that $N \geqslant 8$, $\varepsilon = 1/4$, and $\phi = 1/2$ and that the counter-based streaming subroutine can maintain three counters, one for each calling context. Then, only context `main`→`q` has frequency larger than $\lfloor(\phi-\varepsilon)N\rfloor$ and is a (ϕ, ε)-heavy hitter, but – as we assumed there is room in M for all contexts – a streaming algorithm can maintain the exact frequencies of both `main`→`p` and `main`→`q`. Because `main`→`p` has frequency 1, it would be an error returning it as a heavy hitter. For this reason, M needs to be post-processed in order to eliminate low-frequency items that may be included when there are more available counters than heavy hitters. Algorithmic details on updating and querying M and MCCT are given in Section 3.3.

3.3. Data structures update and query

At each function call, the set M of monitored contexts is updated by a counter-based streaming algorithm (Section 2.2). When M is changed, the subtree MCCT spanning nodes in M needs to be brought up to date as well. To describe how this happens, we assume that the interface of the streaming algorithm provides two main functions:

`update(x,M)`→`V` Given a calling context x, update M to reflect the new occurrence of x in the stream (e.g., if x was already monitored in M, its frequency count may be increased by one). This function might return a set V of *victim* contexts that were previously monitored in M and are evicted during the update (as a special case, x itself may be considered as a victim if the algorithm chooses not to monitor it).

`query(M)`→`A` Remove low-frequency items from M and return the subset A of (ϕ, ε)-heavy hitters (Figure 3).

As with the CCT, during the construction of the MCCT, we maintain a cursor pointer that points to the current calling context, creating a new node if the current context x is encountered for the first

DOI: 10.1002/spe

time (Section 2.1). Additionally, we prune the MCCT according to the victim contexts returned by the streaming `update` operation (these contexts are no longer monitored in M). The pseudocode of the pruning algorithm is given in Algorithm 1. Because the tree must remain connected, victims can

Input: MCCT; node x to be pruned.
Output: Pruned MCCT.

```
V ← update(x, M);
foreach context v ∈ V \{x} do
    while (v is a leaf in MCCT) ∧ (v ∉ M) do
        remove v from MCCT
        v ← parent(v)
    end
end
```

Algorithm 1: On-line pruning algorithm. MCCT, monitored calling context tree.

be removed from the MCCT only if they are leaves. Moreover, removing a victim might expose a path of unmonitored ancestors that no longer have descendants in M: these nodes are pruned as well. The node for the current context x is never removed from the MCCT, even if the context is not necessarily monitored in M. This guarantees that no node in the path from the tree root to x will be removed: these nodes have at least x as a descendant, and the leaf test (line 3 in Algorithm 1) will always fail. A similar pruning strategy can be used to compute the (ϕ, ε)-HCCT from the MCCT. The streaming `query` operation is first invoked on M, returning the support A of the (ϕ, ε)-HCCT. All MCCT nodes that have no descendant in A are then removed, following bottom-up path traversals as in the prune operation.

3.4. Discussion

Compared with the standard approach of maintaining the entire CCT, our solution requires storing the heavy-hitters data structure M and the subtree MCCT spanning nodes in M. The space required by M depends on the specific streaming algorithm that is used as a subroutine and is roughly proportional to $1/\varepsilon$ (Section 2.2). This space can be customized by appropriately choosing ε, for example, according to the amount of available memory. Such a choice appears to be crucial for the effectiveness of our approach: smaller values of ε guarantee more accurate results (i.e., fewer false positives and more precise counters) but imply a larger memory footprint. As we will see experimentally in Section 6, the high skewness of context frequency distribution guarantees the existence of convenient trade-offs between accuracy and space.

The MCCT consists of nodes corresponding to contexts monitored in M and of all their ancestors, which may be cold contexts without a corresponding entry in M. Hence, the space required by the MCCT dominates the space required by M. The number of cold ancestors is difficult to analyze theoretically: it depends on properties of the execution trace and on the structure of the CCT. In Section 6, we will show that, in practice, this amount is negligible compared with the size of M.

Updates of the MCCT can be performed very quickly. As we will see in Section 4, the streaming `update` operation requires constant time. Moreover, our implementation hinges upon very simple and cache-efficient data structures, with no need for time-consuming hashing. A simple amortized analysis arguments also show that the amortized running time of tree pruning is constant.

Unlike previous approaches such as adaptive bursting [13], the MCCT adapts automatically to the case where the hot calling contexts vary over time, and new calling patterns are not likely to be lost. Contexts that are growing more popular are added to the tree as they become more frequent, while contexts that lose their popularity are gradually replaced by hotter contexts and are finally discarded. This guarantees that heavy-hitters queries can be issued at any point in time and will always be able to return the set of hot contexts up to that time.

Our data streaming-based approach is orthogonal to previous techniques and can be integrated, for example, with sampled stack-walking [16, 17] or with more recent techniques such as static

DOI: 10.1002/spe

and adaptive bursting [13]. In addition to frequency counts, it can be extended to support arbitrary performance metrics (e.g., execution time, cache misses, and instruction stalls), exploiting the ability of some streaming algorithms to mine heavy hitters in weighted item sets.

4. IMPLEMENTATION

We implemented in C variants of the HCCT construction algorithm described in Section 3 using different streaming algorithms for the computation of frequent items: Space Saving (SS) [14], Sticky Sampling, and Lossy Counting [21]. All our implementations (including the construction of the entire CCT) are cast into a common framework in which different streaming algorithms can be plugged in. Sticky Sampling consistently proved itself to be less efficient and accurate than its competitors in our experiments. In an earlier version of this work [22], we presented a thorough experimental evaluation of Lossy Counting, resulting in similar accuracy but higher running time and memory usage compared with SS. We also showed that our implementation of SS based on a lazy priority queue is consistently more efficient than the solution based on an ordered bucket list proposed in [14]. We will therefore focus on SS and our new implementation of it in the remaining part of this article.

4.1. Data structures

We use a first-child, next-sibling representation for CCTs. Each MCCT node also contains a pointer to its parent, the routine ID, the call site, and the performance metrics. The first-child, next-sibling representation is space efficient and still guarantees that the children of each node can be explored in time proportional to their number. According to our experiments with several benchmarks, the average number of scanned children is a small constant around 2–3, so this representation turns out to be convenient also for checking whether a routine ID already appears among the children of a node. The parent field, which is needed to perform tree pruning efficiently (Algorithm 1), is not required in CCT nodes. As a routine ID, we use the routine address. Overall, CCT and MCCT nodes require 20 and 24 bytes, respectively, on 32-bit architectures. Using a simple bit packing technique [23], we also encode in one of the pointer fields a Boolean flag that tells if the calling context associated with the node is monitored in the streaming data structure M, without increasing the number of bytes per node. To improve time and space efficiency, we allocate nodes through a custom, page-based allocator, which maintains blocks of fixed size. Any additional algorithm-specific information needed to maintain the heavy hitters is stored as trailing fields within MCCT nodes.

4.2. Engineering Space Saving

Space Saving monitors a set of $1/\varepsilon = |M|$ pairs of the form $(item, count)$, initialized by the first $1/\varepsilon$ distinct items and their exact counts. After the init phase, when a context c is observed in the stream, the `update` operation (Section 3.3) works as follows:

1. if c is monitored, the corresponding counter is incremented;
2. if c is not monitored, the $(item, count)$ pair with the smallest count is chosen as a victim and has its item replaced with c and its count incremented.

The update time is bounded by the dictionary operation of checking whether an item is monitored and by the operations of finding and maintaining the item with minimum count. In our setting, we can avoid the dictionary operation using the cursor pointer to the MCCT: using this pointer, we can directly access the `monitored` flag of the MCCT node associated with the current context. Heavy-hitters queries are answered by returning entries in M such that count $\geqslant \lfloor \phi N \rfloor$.

In [14], it is suggested to use an ordered bucket list, where each bucket points to a list of items with the same count, and buckets are ordered by increasing count values. We devised a more efficient variant based on a lazy priority queue that uses an unordered array M of size $1/\varepsilon$, where each entry

DOI: 10.1002/spe

Input: M, `min`, and `min-idx`.
Output: Min value in the lazy priority queue.

while $(M[\textit{min-idx}] \neq \textit{min}) \wedge (\textit{min-idx} \leqslant M)$ **do**
    min-idx $\leftarrow$ min-idx $+1$
end
if $\textit{min-idx} > M$ **then**
    min $\leftarrow$ minimum in M
    min-idx $\leftarrow$ smallest index j s.t. $M[j] =$ min
end
return *min*

Algorithm 2: `find-min` operation used in *lazy* Space Saving.

points to an MCCT node. The queue supports two operations, `find-min` and `increment`, which return the item with minimum count and increment a counter, respectively.

We (lazily) maintain the value `min` of the minimum counter and the smallest index `min-idx` of an array entry that points to a monitored node with counter equal to `min`. The `increment` operation does not change M, because counters are stored directly inside MCCT nodes. However, `min` and `min-idx` may become temporarily out of date after an `increment`: this is why we call the approach lazy. The `find-min` operation described in Algorithm 2 restores the invariant property on `min` and `min-idx`: it finds the next index in M with counter equal to `min`. If such an index does not exist, it completely rescans M in order to find a new `min` value and its corresponding `min-idx`.

Theorem 4.1
After a `find-min` query, the lazy priority queue correctly returns the minimum counter value in O(1) amortized time.

Proof
Counters are never decremented. Hence, at any time, if a monitored item with counter equal to `min` exists, it must be found in a position larger than or equal to `min-idx`. This yields correctness.

To analyze the running time, let Δ be the value of `min` after k `find-min` and `increment` operations. Because there are $|M|$ counters $\geqslant \Delta$, counters are initialized to 0, and each `increment` operation adds 1 to the value of a single counter; it must be $k \geqslant |M| \cdot \Delta$. For each distinct value assumed by `min`, the array is scanned twice. We therefore have at most $2 \cdot \Delta$ array scans each of length $|M|$, and the total cost of `find-min` operations throughout the whole sequence of operations is upper bounded by $2 \cdot |M| \cdot \Delta$. It follows that the amortized cost is $(2 \cdot |M| \cdot \Delta) / k \leqslant 2$. □

4.3. A context-sensitive profiling plug-in for `gcc`

The `gcc` compiler provides an instrumentation infrastructure to emit calls to analysis routines at the beginning and at the end of each function, passing as arguments the address of the current function and its calling site. On top of these two primitives, we have built a full-fledged infrastructure for context-sensitive profiling of multi-threaded Linux C/C++ applications that ships as a plug-in[‡] for the GNU compiler collection.

Our plug-in provides native support for techniques aimed at reducing runtime overhead, such as sampling and bursting, and does not require modifications to the existing `gcc` installation or to the program to be analyzed (except for its `Makefile`).

When a program is compiled, instrumentation is injected into the code by the compiler, and the executable is eventually linked against a generic profiling library named `libhcct`. When a user wants to analyze the behavior of an instrumented program, it is possible to switch between different

[‡]Source code and documentation are available at: `https://github.com/dcdelia/hcct`

DOI: 10.1002/spe

techniques – including the canonical CCT construction – or parameter settings with no need to further recompile the code.

As part of our infrastructure, we have developed two additional pieces of software that might be of independent interest: a library for resolving addresses to symbols and a set of tools for the analysis and comparison of CCTs from distinct executions. In general, even for deterministic benchmarks, it might not be trivial to line up nodes from two executions, as technical aspects such as address space randomization and dynamic loading of libraries program addresses can change. In some cases, it is not always possible to resolve addresses offline up to a source-file line-number granularity, but the available information is only partial (e.g., we know only the source file where the method is defined). If this happens for two or more sibling nodes that have identical frequency counters, lining them up with tree nodes from another execution requires a similarity analysis of their spanned subtrees. We observed similar scenarios frequently in our experiments, both for hot and cold calling contexts. Because spanned subtrees for CCT nodes can be large, rapid, and accurate, heuristics are required to summarize the subtrees and compute their similarity; accuracy of heuristics is even more crucial when comparing a CCT with a HCCT, as spanned subtrees in the latter might have been partially or entirely pruned. Using combinatorial techniques and ad-hoc heuristics based on topological properties of the trees, we were able to quickly (i.e., in a few minutes) reconstruct for all our experiments a full and accurate mapping between pairs of different trees.

5. METHODOLOGY AND EXPERIMENTAL SETUP

In this section, we present an extensive experimental study of our data streaming-based profiling mechanism. We implemented several variants of space-efficient context-sensitive profilers, and we analyzed their performance and the accuracy of the produced (ϕ, ε)-HCCT with respect to several metrics and using many different parameter settings. Besides the exhaustive approach, where each routine call and return is instrumented, we integrate our implementations with previous techniques aimed at reducing time overhead: we focus, in particular, on static bursting [13], which offers convenient time-accuracy trade-offs. The experimental analysis not only confirms but also reinforces the theoretical prediction: the (ϕ, ε)-HCCT represents the hot portions of the full CCT very well using only an extremely small percentage of the space required by the entire CCT: all the hottest calling contexts are always identified correctly, their counters are very accurate, and the number of false positives is rather small. With bursting, the running time overhead can be kept under control without affecting accuracy in a substantial way. Before discussing the results, we present the details of our experimental methodology, focusing on benchmarks and accuracy metrics, and we describe how the parameters of the streaming algorithms can be tuned.

5.1. Benchmarks

Tests were performed on a variety of large-scale Linux applications and on a set benchmarks drawn from the `Phoronix PTS` and the `SPEC CPU2006` test suites. To ensure deterministic replay of the execution of the interactive applications, we used the Pin dynamic instrumentation framework [24] to record timestamped execution traces for typical usage sessions of approximately 15 min.

Interactive applications include graphics programs (`inkscape` and `gimp`), a hexadecimal file viewer (`ghex2`), audio players/editors (`amarok` and `audacity`), an archiver (`ark`), an Internet browser (`firefox`), an HTML editor (`quanta`), a chat program (`pidgin`), the OpenOffice suite for word processing (`oowriter`), spreadsheets (`oocalc`), and drawing (`ooimpress`). The remaining benchmarks include a cryptographic library (`botan`), a 2D graphics library (`cairo-perf-trace`), advanced chess engines (`crafty` and `sjeng`), connect-4 (`fhourstones`) and go (`gobmk`) games, and 3D games run in demo mode (PlanetPenguin Racer in the `ice-labyrinth` and `mount-herring` scenarios and SuperTuxKart on the `overworld` and `scotland` tracks).

Statistical information about test sets is shown in Table I: even short sessions result in CCTs consisting of tens of millions of calling contexts, whereas the call graph has only a few thousand

DOI: 10.1002/spe

nodes. We also observe that the number of distinct call sites is roughly one order of magnitude larger than the call graph.

5.2. Metrics

Besides memory usage and time consumption of our profiler, we test the accuracy of the (ϕ, ε)-HCCT according to a variety of metrics.

1. Degree of overlap [13, 15, 16] measures the completeness of the (ϕ, ε)-HCCT with respect to the full CCT:

$$overlap((\phi, \varepsilon)\text{-HCCT}, \text{CCT}) = \frac{1}{N} \sum_{\text{arcs } e \in (\phi,\varepsilon)\text{-HCCT}} w(e)$$

   where N is the total number of routine activations (corresponding to the CCT total weight) and $w(e)$ is the true frequency of the target node of arc e in the CCT.
2. Hot edge coverage [13] measures the percentage of CCT hot edges covered by the (ϕ, ε)-HCCT, using an edge-weight threshold $\tau \in [0, 1]$ to determine hotness. Because (ϕ, ε)-HCCT $\subseteq$ CCT, hot edge coverage can be defined as follows:

$$cover((\phi, \varepsilon)\text{-HCCT}, \text{CCT}, \tau) = \frac{|\{e \in (\phi, \varepsilon)\text{-HCCT}: w(e) \geqslant \tau \cdot w_{\max}\}|}{|\{e \in \text{CCT}: w(e) \geqslant \tau \cdot w_{\max}\}|}$$

   where $w_{\max}$ is the weight of the hottest CCT arc.
3. Maximum hotness of uncovered calling contexts, where a context is uncovered if is not included in the (ϕ, ε)-HCCT:

$$\max Uncov((\phi, \varepsilon)\text{-HCCT}, \text{CCT}) = \max_{e \in \text{CCT} \setminus (\phi,\varepsilon)\text{-HCCT}} \frac{w(e)}{w_{\max}} \times 100$$

   Average hotness of uncovered contexts is defined similarly.
4. Number of false positives, that is, $|A \setminus H|$: the smaller this number, the better the (ϕ, ε)-HCCT approximates the exact HCCT obtained from CCT pruning.
5. Maximum counter error, that is, maximum error in the frequency counters of (ϕ, ε)-HCCT nodes with respect to their true value in the full CCT:

$$\max Error((\phi, \varepsilon)\text{-HCCT}) = \max_{e \in (\phi,\varepsilon)\text{-HCCT}} \frac{|w(e) - \tilde{w}(e)|}{w(e)} \times 100$$

   where $w(e)$ and $\tilde{w}(e)$ are the true and the estimated frequency of context e, respectively. Average counter error is defined similarly.

An accurate solution should maximize (1) and (2) and minimize the remaining metrics.

5.3. Parameter tuning

Before describing our experimental findings, we discuss how to choose parameters ϕ and ε to be provided as input to the streaming algorithms. According to the theoretical analysis, an accurate choice of ϕ and ε might greatly affect the space used by the algorithms and the accuracy of the solution. In our study, we considered many different choices of ϕ and ε across rather heterogeneous sets of benchmarks and execution traces, always obtaining similar results that we summarize in the following.

A rule of thumb about ϕ and ε validated by previous experimental studies [20] suggests that it is sufficient to choose $\varepsilon = \phi/10$ in order to obtain high counter accuracy and a small number of false positives. We found this choice overly pessimistic in our scenario: the extremely skewed cumulative distribution of calling context frequencies shown in Figure 1 makes it possible to use much larger

DOI: 10.1002/spe

Table II. Typical thresholds.

Benchmark	HCCT nodes $\phi = 10^{-3}$	HCCT nodes $\phi = 10^{-5}$	HCCT nodes $\phi = 10^{-7}$	CCT nodes
audacity	112	9 181	233 362	13 131 115
dolphin	97	14 563	978 544	11 667 974
gimp	96	15 330	963 708	26 107 261
ice-labyrinth	93	9 413	529 945	2 160 052
inkscape	80	16 713	830 191	13 896 175
oocalc	136	13 414	1 339 752	48 310 585
quanta	94	13 881	812 098	27 426 654

CCT, calling context tree; HCCT, hot CCT.

values of ε without sacrificing accuracy. This yields substantial benefits on the space usage, which is roughly proportional to $1/\varepsilon$. Unless otherwise stated, in all our experiments, we used $\varepsilon = \phi/5$.

Let us now consider the choice of ϕ: ϕ is the hotness threshold with respect to the stream length N, that is, to the number of routine enter events. However, N is unknown a priori during profiling, and thus, choosing ϕ appropriately may appear to be difficult: too large values might result in returning very few hot calling contexts (even no context at all in some extreme cases), while too small values might result in using too much space and returning too many contexts without being able to discriminate accurately which of them are actually hot. Our experiments suggest that an appropriate choice of ϕ is mostly independent of the specific benchmark and of the stream length: as shown in Table II, different benchmarks have HCCT sizes of the same order of magnitude when using the same ϕ threshold (results for omitted benchmarks are similar). This is a consequence of the skewness of context frequency distribution and greatly simplifies the choice of ϕ in practice. Unless otherwise stated, in our experiments, we used $\phi = 10^{-4}$, which corresponds to mining roughly the hottest 1000 calling contexts independently of the benchmark.

5.4. Platform

Experiments were performed on a 2.53-GHz Intel Core2 Duo T9400 with 128KB of L1 data cache, 6 MB of L2 cache, and 4 GB of main memory DDR3 1066, running Ubuntu 12.04, Linux Kernel 3.5.0, gcc 4.7.2, 32 bit. Performance measurements were collected with negligible background activity, running multiple trials for each benchmark/tool combination and reporting confidence intervals stated at 95% confidence level.

6. EXPERIMENTAL RESULTS

6.1. Memory usage

We first evaluate how much space can be saved by our approach, reporting the size of the MCCT constructed by the SS algorithm compared with the size of the full CCT as a function of the hotness threshold ϕ. Figure 4 shows the results for a subset of our benchmarks. Notice that the size of the MCCT, hence the space used by the algorithm, decreases with ϕ. For values of $\phi \geq 10^{-4}$, that is, contexts that appear at least 0.01% of the times, space usage remains less than 1% than the CCT size for most benchmarks, with a worst case of about 4.1% over all our experiments.

As a second experiment, we study the actual memory footprint of our profilers considering both SS and the combination of SS with bursting techniques. We focus, in particular, on *static bursting*, a technique that allows the application to run unhindered between sampling points, collecting at each of these points a sequence (i.e., a *burst*) of events for a given interval (*burst length*). Figure 5 plots the peak memory usage of our profilers as a percentage of the full CCT. We recall that during the computation, we store the minimal subtree MCCT of the CCT spanning all monitored contexts. This subtree is eventually pruned to obtain the (ϕ, ε)-HCCT (Section 3.2). The peak memory usage is proportional to the number of MCCT nodes, which is typically much larger than the actual number of hot contexts obtained after pruning.

DOI: 10.1002/spe

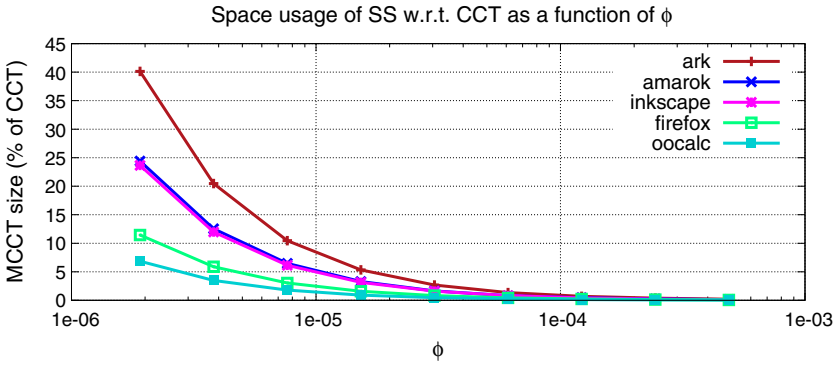


Figure 4. Space usage as a function of the hotness threshold ϕ. SS, Space Saving; CCT, calling context tree.

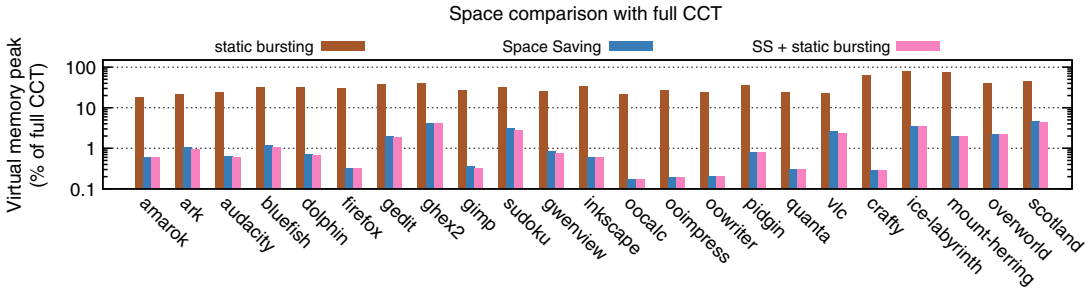


Figure 5. Space analysis on several benchmarks of static bursting [13] (left), Space Saving (SS) (center), and SS combined with static bursting (right) with sampling interval 2 ms and burst length 0.2 ms. CCT, calling context tree.

Quite surprisingly, static bursting also improves space usage. This depends on the fact that sampling reduces the variance of calling context frequencies: MCCT cold nodes that have a hot descendant are more likely to become hot when sampling is active, and monitoring these nodes reduces the total MCCT size. The histogram also shows that static bursting alone (i.e., without streaming) is not sufficient to substantially reduce space: in addition to hot contexts, a large fraction of cold contexts is also sampled and included in the CCT. We also observed that the larger the applications, the larger the space reduction of our approach over bursting alone.

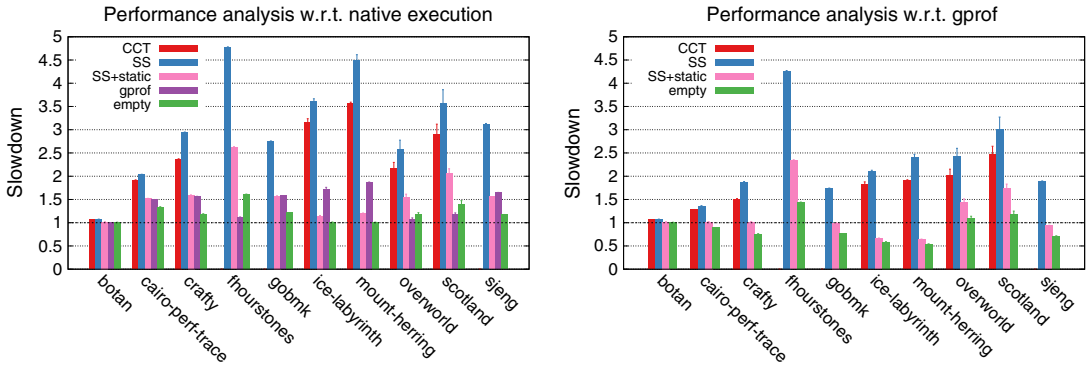
Because the average node degree is a small constant, cold HCCT nodes are typically a fraction of the total number of nodes, as shown in Figure 9 for $\phi = 10^{-4}$. In our experiments, we observed that this fraction strongly depends on the hotness threshold ϕ and, in particular, decreases with ϕ: cold nodes that have a hot descendant are indeed more likely to become hot when ϕ is smaller.

6.2. *Time overhead*

We now discuss the time overhead of our approach, both alone and in combination with static bursting. We compare it with native execution, with the widely used call-graph profiler `gprof` [9], and with the CCT construction. To assess the instrumentation overhead, we also compare with *empty* instrumentation (i.e., when no analysis is performed).

Figure 6(a) shows the overheads of the different profilers normalized against the performance of a native execution. The average slowdown for the CCT construction is 2.45×, with a peak of 3.56× for `mount-herring`. Note that data for benchmarks `fhourstones`, `gobmk`, and `sjeng` are not reported for the CCT profiler as it ran out of memory. We observe that the construction of the (ϕ, ε)-HCCT incurs an average slowdown of 2.9× (3.09× considering also out-of-memory benchmarks) and is 16.28% slower than the CCT profiler, with a peak of 26.08% for `mount-herring`. Given the previously discussed memory usage reduction, this represents an interesting space-time trade-off.

DOI: 10.1002/spe



(a) (b)

Figure 6. Runtime analysis for calling context tree (CCT) and (ϕ, ε)-hot calling context tree (HCCT) construction compared with (a) native executions and (b) executions under `gprof`. The *empty* bars measure the cost of instrumenting function calls and returns. Space Saving (SS) with stating bursting has been executed with sampling interval 20 ms and burst length 2 ms.

The integration with static bursting reduces the average overhead of our approach to 1.58× for the whole set of benchmarks, which is not far from the 1.21× slowdown introduced by the `gcc` instrumentation itself. We observe a peak of 2.62× on the benchmark `fhourstones`: we believe this is due to the particular structure of its source code, which contains very frequently invoked tiny functions that could be replaced with macros.

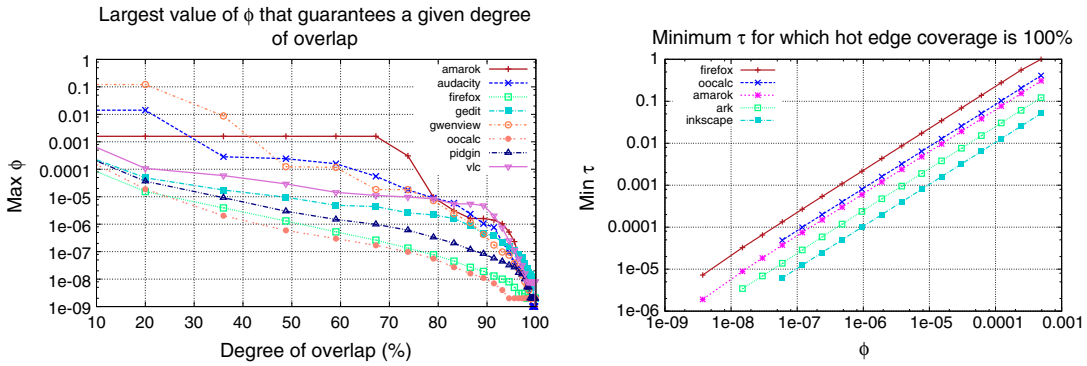
In Figure 6(b), we have normalized the runtime overheads against an execution under `gprof`. The combination of our approach with static bursting is very effective, as it is on average 18% (5.16% if we exclude `fhourstones`) slower than `gprof`. On five out of 10 benchmarks, the two tools achieve nearly identical slowdowns. We observe 2.34×, 1.44×, and 1.74× slowdowns on `fhourstones`, `overworld`, and `scotland`, respectively. Notice that for all these benchmarks, the cost of the instrumentation inserted by `gcc` is already greater than the slowdown introduced by `gprof`. On the other hand, we observe appreciable speedups on `ice-labyrinth` and `mount-herring`, for which SS combined with static bursting is 1.51× and 1.55× faster than `gprof`, respectively.

6.3. Accuracy: exact HCCT

We first discuss the accuracy of the exact HCCT with respect to the full CCT. Because the HCCT is a subtree of the (ϕ, ε)-HCCT computed by our algorithms, the results described in this section apply to the (ϕ, ε)-HCCT as well: the values of degree of overlap and hot edge coverage on the HCCT are a lower bound to the corresponding values in the (ϕ, ε)-HCCT, while the frequency of uncovered contexts is an upper bound.

It is not difficult to see that the cumulative distribution of calling context frequencies shown in Figure 1 corresponds exactly to the degree of overlap with the full CCT. This distribution roughly satisfies the 10–90% rule; hence, with only 10% of hot contexts, we have a degree of overlap around 90% on all benchmarks. Figure 7(a) illustrates the relation between degree of overlap and hotness threshold, plotting the value $\widetilde{\phi}$ of the largest hotness threshold for which a given degree of overlap d can be achieved: using any $\phi \leqslant \widetilde{\phi}$, the achieved degree of overlap will be larger than or equal to d. The value of $\widetilde{\phi}$ decreases as d increases: if we want to achieve a larger degree of overlap, we must include in the HCCT a larger number of nodes, which corresponds to choosing a smaller hotness threshold. However, when computing the (ϕ, ε)-HCCT, the value of ϕ indirectly affects the space used by the algorithm and in practice cannot be too small (Section 5.3). By analyzing hot edge coverage and uncovered frequency, we show that even when the degree of overlap is not particularly large, the HCCT and the (ϕ, ε)-HCCT are nevertheless good approximations of the full CCT. As shown in Figures 4 and 7, for values of ϕ as small as 10^{-4}, the space usage is less than 1%

DOI: 10.1002/spe



(a) (b)

Figure 7. (a) Relation between ϕ and degree of overlap between the exact hot calling context tree (HCCT) and the full CCT on a representative subset of benchmarks. (b) Hot edge coverage of the exact HCCT: relation between ϕ and edge-weight threshold τ.

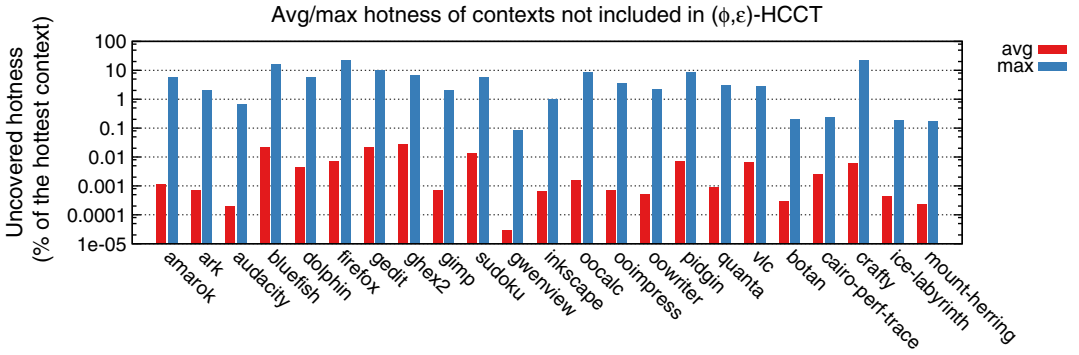


Figure 8. Maximum and average hotness of calling contexts not included in the (ϕ, ε)-hot calling context tree (HCCT).

of the full CCT while guaranteeing 100% coverage for all edges with hotness at least 10% on most benchmarks. Smaller values of ϕ increase space and improve the degree of overlap but are unlikely to be interesting in applications that require mining hot calling contexts.

Notice that $\phi = 10^{-4}$ yields a degree of overlap as small as 10% on two of the less skewed benchmarks (`oocalc` and `firefox`), which seems to be a bad scenario. However, Figure 8 analyzes how the remaining 90% of the total CCT weight is distributed among uncovered contexts: the average frequency of uncovered contexts is about 0.01% of the frequency of the hottest context, and the maximum frequency is typically less than 10%. This suggests that uncovered contexts are likely to be uninteresting with respect to the hottest contexts and that the distribution of calling context frequencies obeys a 'long-tail, heavy-tail' phenomenon: the CCT contains a huge number of calling contexts that rarely become executed, but overall, these low-frequency contexts account for a significant fraction of the total CCT weight.

Figure 7(b) confirms this intuition, showing that the HCCT represents the hot portions of the full CCT remarkably well even for values of ϕ for which the degree of overlap may be small. The figure plots, as a function of ϕ, the smallest value $\tilde{\tau}$ of the hotness threshold τ for which hot edge coverage of the HCCT is 100%. Results are shown only on some of the less skewed and thus more difficult benchmarks. Note that $\tilde{\tau}$ is directly proportional to and roughly one order of magnitude larger than ϕ. This is because the HCCT contains all contexts with frequency $\geq \lfloor \phi N \rfloor$ and always contains the hottest context, which has weight $w_{\max}$ as in the definition of hot edge coverage in Section 5. Hence, the hot edge coverage is 100% as long as $\lfloor \phi N \rfloor \geq \tau \cdot w_{\max}$, which yields $\tilde{\tau} = \lfloor \phi N \rfloor / w_{\max}$. The experiment shows that 100% hot edge coverage is always obtained for $\tau \geq 0.01$. As a frame of

DOI: 10.1002/spe

comparison, notice that the τ thresholds used in [13] to analyze hot edge coverage are always larger than 0.05, and for those values, we always guarantee total coverage.

6.4. *Accuracy:* (ϕ, ε)*-HCCT*

We now discuss the accuracy of the (ϕ, ε)-HCCT compared with the exact HCCT. Figure 9 shows the percentages of cold nodes, true hots, and false positives in the (ϕ, ε)-HCCT using $\phi = 10^{-4}$ and $\varepsilon = \phi/5$. We observe that SS includes in the tree only very few false positives: less than 10% of the total number of tree nodes in the worst case and between 0% and 5% for the large majority of the benchmarks. The percentage of cold nodes strictly depends on the characteristics of the single benchmark and is not remarkably influenced by the number of false positives, which is small.

In Figure 10, we show how the number of false positives decreases considerably as we decrease ε, becoming very close to 0% on most benchmarks when the ratio ϕ/ε is larger than 10. The choice of assigning $\varepsilon = \phi/5$ thus offers a good trade-off between quality of the solution (i.e., number of false positives) and space usage.

An interesting feature of our approach is that counter estimates are very close to the true frequencies, as shown in Figure 11. A comparison between average and maximum errors suggests that just a few nodes are appreciably overestimated. The average counter error computed for hot contexts is actually greater than 4% only for `crafty` and smaller than 2% for the majority of the benchmarks.

It is worth noticing that when integrating SS with sampling-based approaches, the theoretical guarantees of the algorithm apply only to the stream of sampled events, and not to the full stream of routine calls and returns from the execution. For this reason, we analyze the impact of static bursting on the quality of the solution.

Figure 12(a) shows the average counter error among hot calling contexts. In order to compare them with the exact frequencies from the corresponding CCTs, we adjusted the counters by dividing

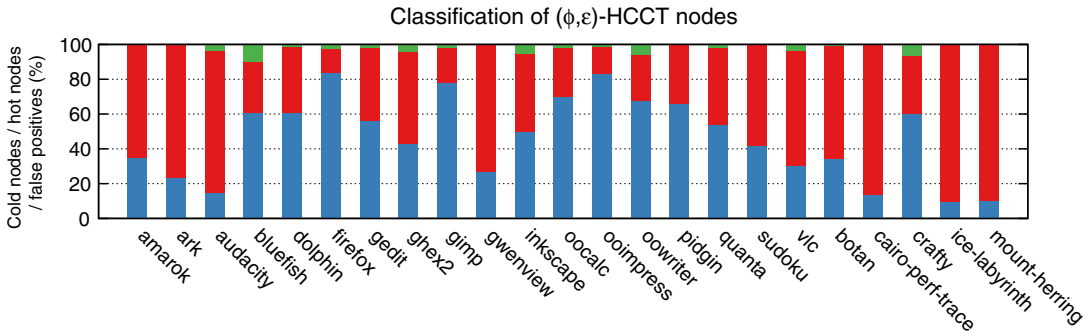


Figure 9. Partition of (ϕ, ε)-hot calling context tree (HCCT) nodes into: cold (bottom bar), hot (middle bar), and false positives (top bar).

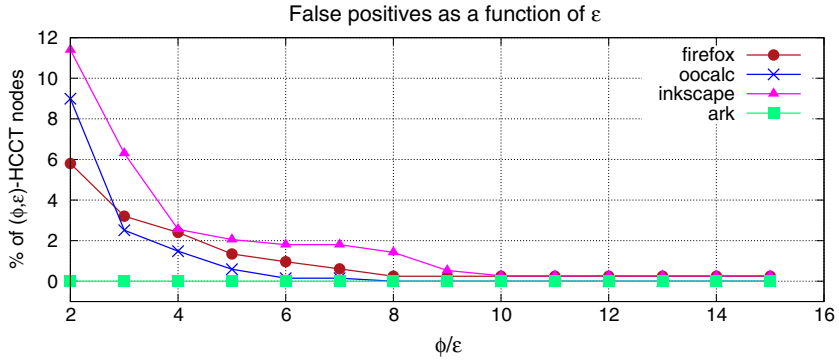


Figure 10. False positives in the (ϕ, ε)-hot calling context tree (HCCT) as a function of ε. The value of ϕ is fixed to 10^{-4}.

DOI: 10.1002/spe

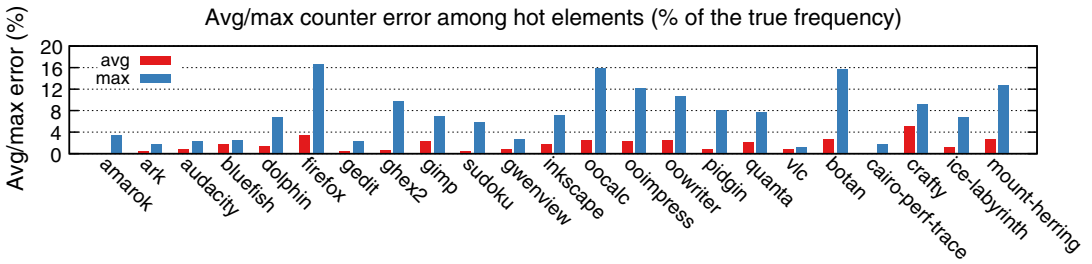
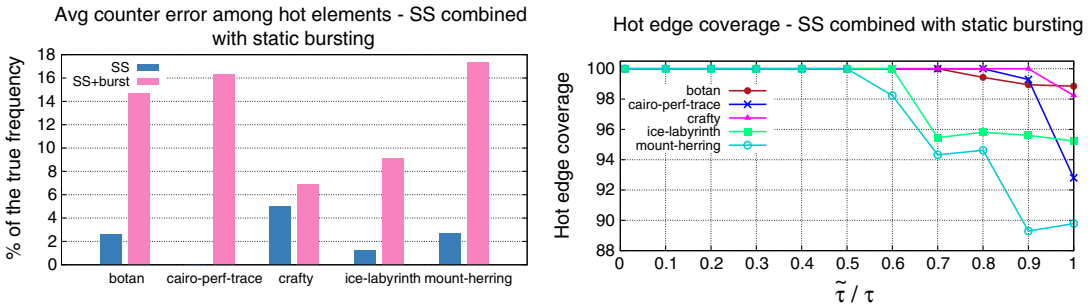


Figure 11. Accuracy of context frequencies measured on hot contexts included in the (ϕ, ε)-hot calling context tree (HCCT).



(a) (b)

Figure 12. (a) Accuracy of frequencies measured on hot calling contexts when Space Saving (SS) is combined with static bursting. (b) Hot edge coverage for decreasing values of the hotness threshold τ. As pointed out in Section 6.3, $\tilde{\tau}$ is the minimum threshold for which SS guarantees 100% coverage when static bursting is disabled. We chose a representative subset of benchmarks for both charts; sampling interval has been set to 20 ms and burst length to 2 ms.

them by the fraction of sampled events in the stream of function calls and returns. Note that if the stream is uniformly distributed in time, this fraction is equal to the ratio between burst length and sampling interval. While processing roughly only one-tenth of the whole stream, we observe that the average counter error ranges from 6.89% to 17.31%. An analysis of the results from the integration of static bursting with the canonical CCT construction shows very similar numbers, thus suggesting that SS does not degrade the quality of the solution.

The adoption of sampling-based techniques may cause the algorithm to miss some of the contexts with frequency very close to $\lfloor \phi N \rfloor$, leading to some false negatives. However, our analysis of hot edge coverage reported in Figure 12(b) shows that static bursting does not appreciably degrade the quality of the solution. Given the smallest $\tilde{\tau}$ value for which SS guarantees 100% coverage, in all of our experiments, we achieve 100% coverage for any $\tau \geqslant 2\tilde{\tau}$ (i.e., when $\tilde{\tau}/\tau \leqslant 0.5$), and only in one case (`mount-herring` benchmark), hot edge coverage drops below 90%.

7. RELATED WORK

This section describes research on context-sensitive profiling: it focuses on CCTs and briefly considers other forms of profiling at both interprocedural and intraprocedural levels.

Early approaches. The utility of calling context information was already clear in the 80s: `gprof` [9] approximates context-sensitive profiles by associating procedure timing with caller–callee pairs rather than with single procedure. This single level of context sensitivity, however, may yield to several inaccuracies [10, 11]. In [25], the authors introduce call-path profiles of monotonic program resources and show how they can be computed in Unix processes using interval-based sampling: a call path is a sequence of function pairs in a caller–callee relationship, and the profile is a sorted

DOI: 10.1002/spe

list of call paths and of their performance metrics. The space usage with this approach can be prohibitive, because at each sample point, metrics are recorded along with the entire call stack.

Calling context trees. CCTs have been introduced in [12] as a practical data structure to associate performance metrics with paths through a program's call graph: Ammons, Ball, and Larus suggest to build a CCT by instrumenting procedure code and to compute metrics by exploiting hardware counters available in modern processors. It has been later observed, however, that exhaustive instrumentation can incur large slowdowns.

Reducing overhead. To reduce overhead, in [26], the authors generate path profiles including only methods of interest, while statistical profilers [16, 17, 25, 27] attribute metrics to calling contexts through periodic sampling of the call stack. For call-intensive programs, sample-driven stack-walking can be orders of magnitude faster than exhaustive instrumentation but may incur significant loss of accuracy with respect to the complete CCT: sampling guarantees neither high coverage [8] nor accuracy of performance metrics [13], and its results may be highly inconsistent in different executions.

A variety of works explores the combination of sampling with bursting [13, 15, 28]. Most recently, Zhuang *et al.* suggest to perform stack-walking followed by a burst during which the profiler traces every routine call and return [13]: experiments show that adaptive bursting can yield very accurate results. In [29], the profiler infrequently collects small call traces that are merged afterwards to build large CCTs: ambiguities might emerge during this process, and the lack of information about where the partial CCTs should be merged to does not allow reconstructing the entire CCT univocally. The main goal of all these works is to reduce profiling overhead without incurring significant loss of accuracy. Our approach is orthogonal to this line of research and regards space efficiency as an additional resource optimization criterion besides profile accuracy and time efficiency. When the purpose of profiling is to identify hot contexts, exhaustive instrumentation, sampling, and bursting might all be combined with our approach and benefit of our space reduction technique.

In a recent work [30], a novel data structure is proposed to avoid the node lookup operation in dynamic bug detectors. A new node is instead allocated for each context, and the costs of allocations are mitigated by extending the garbage collector not only to collect unused node but also to merge duplicate ones lazily.

Reducing space. A few previous works have addressed techniques to reduce profile data (or at least the amount of data presented to the user) in context-sensitive profiling. Incremental call-path profiling lets the user choose a subset of routines to be analyzed [26]. Call-path refinement helps users focus the attention on performance bottlenecks by limiting and aggregating the information revealed to the user [31]. These works are quite different in spirit from our approach, where only hot contexts are profiled and identified automatically during program's execution.

Probabilistic calling contexts have been introduced as an extremely compact representation (just a 32-bit value per context), especially useful for tasks such as residual testing, statistical bug isolation, and anomaly-based intrusion detection [8]. Bond and McKinley target applications where coverage of both hot and cold contexts is necessary, but their inspection is unnecessary. This is not the case in performance analysis, where identifying and understanding a few hot contexts are typically sufficient to guide code optimization. Hence, although sharing with Bond and McKinley [8] the common goal of space reduction, our approach targets a rather different application context.

Somner *et al.* proposed a technique called *precise calling context encoding* (PCCE) that encodes acyclic paths in the call graph of a program into one number, while recursive call paths are divided into acyclic subsequences and encoded independently [32]. Different calling contexts are guaranteed to have different IDs that can be faithfully decoded, and experiments on a prototype implementation for C programs show negligible overhead. However, PCCE would not work in the presence of virtual methods and dynamic class loading in object-oriented languages, and the encoding scheme shows scalability problems when handling large-scale software [33, 34]. These limitations, which

DOI: 10.1002/spe

are absent from our solution, have been recently addressed in [34], and it would be interesting to investigate whether their approach can be extended to collect performance metrics and in turn filter the collected data on the fly using a data streaming approach as we do in this work.

Path profiling. At the intraprocedural level, the seminal work of Ball and Larus [35] has spawned much research on flow-sensitive profiling [2, 12, 36–44]. The Ball–Larus path profiling computes a unique number through each possible path in the control flow graph [35]: a path profile determines how many times each acyclic path in a routine executes, extending the more common basic block and edge profiling. Melski and Reps have proposed interprocedural path profiling in order to capture both interprocedural and intraprocedural control flows [45]. However, their approach does not scale because of the large number of statically possible paths existing across procedure boundaries.

Workload sensitivity. All the aforementioned works aim at associating performance metrics with distinct paths traversed either in the call graph or in the control flow graph during a program's execution. A different line of research explores input-sensitivity issues, with the goal of discovering workload-dependent performance bottlenecks (e.g., [46–50]). Context and input sensitivities represent two orthogonal aspects of program profiles, and the techniques described in this paper could be combined with input-sensitive profilers to pinpoint routines' calling contexts characterized by superlinear running times.

8. CONCLUSIONS

Calling context trees offer a compact representation of all calling contexts encountered during a program's execution. Even for short runs of medium-sized applications, CCTs can be rather large and difficult to analyze. Motivated by the observation that only a very small fraction of calling contexts are hot, in this article we have presented a novel technique for compactly representing frequent calling contexts without sacrificing accuracy. By adapting modern data mining techniques, we have devised an algorithm for interprocedural contextual profiling that can discard on-the-fly cold contexts and appears to be practical. We have evaluated our approach on several large-scale Linux applications and benchmarks, showing significant SS with respect to the full CCT and a slowdown competitive with `gprof`.

We believe that a careful use of data mining techniques has the potential benefit of enabling some previously impossible dynamic program analysis tasks, which would otherwise be too costly. In particular, our techniques could be applied to certain forms of path profiling: for example, they could help leverage the scalability problems encountered when collecting performance metrics about interprocedural paths (i.e., acyclic paths that may cross procedure boundaries) [45].

REFERENCES

1. Pavlopoulou C, Young M. Residual test coverage monitoring. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99. ACM: New York, NY, USA, 1999; 277–284.
2. Vaswani K, Nori AV, Chilimbi TM. Preferential path profiling: compactly numbering interesting paths. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07. ACM: New York, NY, USA, 2007; 351–362.
3. Chang PP, Mahlke SA, Chen WY, Hwu WmW. Profile-guided automatic inline expansion for c programs. *Software: Practice and Experience* 1992; **22**(5):349–369. DOI:10.1002/spe.4380220502.
4. Feng HH, Kolesnikov OM, Fogla P, Lee W, Gong W. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03. IEEE Computer Society: Washington, DC, USA, 2003; 62–75.
5. Liblit B, Aiken A, Zheng AX, Jordan MI. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03. ACM: New York, NY, USA, 2003; 141–154.
6. Nistor A, Jiang T, Tan L. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13. IEEE Press: Piscataway, NJ, USA, 2013; 237–246.
7. Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07. ACM: New York, NY, USA, 2007; 89–100.

DOI: 10.1002/spe

8. Bond MD, McKinley KS. Probabilistic calling context. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07. ACM: New York, NY, USA, 2007; 97–112.
9. Graham SL, Kessler PB, Mckusick MK. Gprof: a call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82. ACM: New York, NY, USA, 1982; 120–126.
10. Ponder C, Fateman RJ. Inaccuracies in program profilers. *Software: Practice and Experience* 1988; **18**(5):459–467. DOI: 10.1002/spe.4380180506.
11. Spivey JM. Fast, accurate call graph profiling. *Software: Practice and Experience* 2004; **34**(3):249–264. DOI:10.1002/spe.562.
12. Ammons G, Ball T, Larus JR. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97. ACM: New York, NY, USA, 1997; 85–96.
13. Zhuang X, Serrano MJ, Cain HW, Choi JD. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06. ACM: New York, NY, USA, 2006; 263–271.
14. Metwally A, Agrawal D, Abbadi AE. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Transactions on Database Systems* 2006; **31**(3):1095–1133. DOI:10.1145/1166074.1166084.
15. Arnold M, Ryder BG. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01. ACM: New York, NY, USA, 2001; 168–179.
16. Arnold M, Sweeney PF. Approximating the calling context tree via sampling. *Technical Report RC 21789*, IBM Research: Yorktown Heights, New York, USA, 2000.
17. Whaley J. A portable sampling-based profiler for java virtual machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00. ACM: New York, NY, USA, 2000; 78–87.
18. Demetrescu C, Finocchi I. Algorithms for data streams. In *Handbook of Applied Algorithms: Solving Scientific, Engineering, and Practical Problems*, Vol. 241. John Wiley and Sons: Hoboken, NJ, USA, 2007; 239–267.
19. Muthukrishnan S. Data streams: algorithms and applications. *Foundations and Trends in Theoretical Computer Science* 2005; **1**(2):117–236. DOI:10.1561/0400000002.
20. Cormode G, Hadjieleftheriou M. Finding frequent items in data streams. *Proceedings of the VLDB Endowment* 2008; **1**(2):1530–1541. DOI:10.14778/1454159.1454225.
21. Manku GS, Motwani R. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2002; 346–357.
22. D'Elia DC, Demetrescu C, Finocchi I. Mining hot calling contexts in small space. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11. ACM: New York, NY, USA, 2011; 516–527.
23. Standish TA. *Data Structure Techniques*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1980.
24. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05. ACM: New York, NY, USA, 2005; 190–200.
25. Hall RJ, Goldberg AJ. Call path profiling of monotonic program resources in unix. In *Proceedings of the USENIX Summer 1993 Technical Conference on Summer Technical Conference – volume 1*, Usenix-stc'93. USENIX Association: Berkeley, CA, USA, 1993; 1:1–1:19.
26. Bernat AR, Miller BP. Incremental call-path profiling. *Concurrency and Computation: Practice and Experience* 2007; **19**(11):1533–1547. DOI:10.1002/cpe.v19:11.
27. Froyd N, Mellor-Crummey J, Fowler R. Low-overhead call path profiling of unmodified, optimized code. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05. ACM: New York, NY, USA, 2005; 81–90.
28. Hirzel M, Chilimbi T. Bursty tracing: a framework for low-overhead temporal profiling. *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, Austin, Texas, USA, 2001; 117–126.
29. Serrano M, Zhuang X. Building approximate calling context from partial call traces. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09. IEEE Computer Society: Washington, DC, USA, 2009; 221–230.
30. Huang J, Bond MD. Efficient context sensitivity for dynamic analyses via calling context uptrees and customized memory management. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13. ACM: New York, NY, USA, 2013; 53–72.
31. Hall RJ. Call path refinement profiles. *Transactions on Software Engineering* 1995; **21**(6):481–496. DOI:10.1109/32.391375.
32. Sumner WN, Zheng Y, Weeratunge D, Zhang X. Precise calling context encoding. *Transactions on Software Engineering* 2012; **38**(5):1160–1177. DOI:10.1109/TSE.2011.70.
33. Bond MD, Baker GZ, Guyer SZ. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10. ACM: New York, NY, USA, 2010; 13–24.

34. Zeng Q, Rhee J, Zhang H, Arora N, Jiang G, Liu P. Deltapath: precise and scalable calling context encoding. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14. ACM: New York, NY, USA, 2014; 109:109–109:119.
35. Ball T, Larus JR. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29. IEEE Computer Society: Washington, DC, USA, 1996; 46–57.
36. Ball T, Mataga P, Sagiv M. Edge profiling versus path profiling: the showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98. ACM: New York, NY, USA, 1998; 134–148.
37. Larus JR. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99. ACM: New York, NY, USA, 1999; 259–269.
38. Apiwattanapong T, Harrold MJ. Selective path profiling. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '02. ACM: New York, NY, USA, 2002; 35–42.
39. Ammons G, Larus JR. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98. ACM: New York, NY, USA, 1998; 72–84.
40. Joshi R, Bond MD, Zilles C. Targeted path profiling: lower overhead path profiling for staged dynamic optimization systems. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04. IEEE Computer Society: Washington, DC, USA, 2004; 239–250.
41. Bond MD, McKinley KS. Continuous path and edge profiling. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38. IEEE Computer Society: Washington, DC, USA, 2005; 130–140.
42. Bond MD, McKinley KS. Practical path profiling for dynamic optimizers. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05. IEEE Computer Society: Washington, DC, USA, 2005; 205–216.
43. Roy S, Srikant YN. Profiling k-iteration paths: a generalization of the Ball–Larus profiling algorithm. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09. IEEE Computer Society: Washington, DC, USA, 2009; 70–80.
44. D'Elia DC, Demetrescu C. Ball–Larus path profiling across multiple loop iterations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13. ACM: New York, NY, USA, 2013; 373–390.
45. Melski D, Reps TW. Interprocedural path profiling. In *Proceedings of the 8th International Conference on Compiler Construction, Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, CC '99. Springer-Verlag: London, UK, UK, 1999; 47–62.
46. Coppa E, Demetrescu C, Finocchi I. Input-sensitive profiling. *Transactions on Software Engineering* 2014; **40**(12):1185–1205.
47. Coppa E, Demetrescu C, Finocchi I, Marotta R. Estimating the empirical cost function of routines with dynamic workloads. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14. ACM: New York, NY, USA, 2014; 230:230–230:239.
48. Goldsmith SF, Aiken AS, Wilkerson DS. Measuring empirical computational complexity. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC-FSE '07. ACM: New York, NY, USA, 2007; 395–404.
49. Xiao X, Han S, Zhang D, Xie T. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013. ACM: New York, NY, USA, 2013; 90–100.
50. Zaparanuks D, Hauswirth M. Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12. ACM: New York, NY, USA, 2012; 67–76.

DOI: 10.1002/spe