

---

# Visualizing and Exploring Profiles with Calling Context Ring Charts



Philippe Moret,<sup>\*1</sup> Walter Binder,<sup>\*1</sup> Alex Villazón,<sup>2</sup>  
Danilo Ansaloni,<sup>1</sup> and Abbas Heydarnoori<sup>1</sup>

<sup>1</sup> *University of Lugano, Switzerland*

<sup>2</sup> *Centro de Investigaciones de Nuevas Tecnologías Informáticas (CINTI), Universidad Privada Boliviana (UPB), Cochabamba, Bolivia*

---

## SUMMARY

Calling context profiling is an important technique for analyzing the performance of object-oriented software with complex inter-procedural control flow. The Calling Context Tree (CCT) is a common data structure that stores dynamic metrics, such as CPU time, separately for each calling context. As CCTs may comprise millions of nodes, there is a need for a condensed visualization that eases the localization of performance bottlenecks. In this article, we discuss *Calling Context Ring Charts (CCRCs)*, a compact visualization for CCTs, where callee methods are represented in ring segments surrounding the caller's ring segment. In order to reveal hot methods, their callers, and callees, the ring segments can be sized according to a chosen dynamic metric. We describe two case studies where CCRCs help us to detect and fix performance problems in applications. A performance evaluation also confirms that our implementation can efficiently handle large CCTs.

KEY WORDS: Calling context profiles; Calling Context Tree (CCT); Calling Context Ring Charts (CCRC); Visualization; Dynamic metrics

## 1. Introduction

Nowadays, software systems have become more and more complex such that they may comprise many different components from various sources. Consequently, understanding the whole software systems has become more challenging for application developers, and pure static program analyzes have become less effective [9]. As a result, program profiling techniques are widely used by application developers to collect information about the behaviors of programs at runtime [42]. This

---

<sup>\*</sup>Correspondence to: P. Moret and W. Binder, Faculty of Informatics, University of Lugano, CH-6900 Lugano, Switzerland  
E-mail: philippe.moret@usi.ch, walter.binder@usi.ch

<sup>‡</sup>Contract/grant sponsor: The work presented in this article has been supported by the Swiss National Science Foundation. Part of the work presented in this article was conducted while A. Villazón was with the University of Lugano, Switzerland.

---

information can be used for instance to identify algorithmic bottlenecks or inefficient code [1]. A large body of work exists that benefits from profiling techniques in program comprehension tasks such as debugging [4, 18, 34], testing [10, 39], reverse engineering [28, 20], and optimizing software systems [11].

*Calling context profiling* is a common profiling technique that yields dynamic metrics separately for each calling context, such as the number of method invocations or the CPU time spent in a calling context. A calling context is a sequence of methods that were invoked but have not yet completed; that is, a calling context corresponds to the methods represented on the call stack at some moment during program execution. Calling context profiling is powerful since it helps us to analyze the dynamic inter-procedural control flow of applications. This technique is particularly important for understanding and optimizing modern object-oriented software, where polymorphism, dynamic binding, and reflection often hinder static analysis approaches.

The *Calling Context Tree (CCT)*, which was first introduced by Ammons et al. [1], is the prevailing data structure for representing calling context profiles. Each node in the CCT corresponds to a calling context and stores the measured dynamic metrics for that calling context; it also refers to a unique identifier of the method in which the metrics were collected. The parent of a CCT node represents the caller's context, while the children nodes correspond to the callee methods. If the same method is invoked in distinct calling contexts, the different invocations are represented by distinct nodes in the CCT. In contrast, if the same method is invoked multiple times in the same calling context, the dynamic metrics collected during the executions of that method are stored in the same CCT node. Hence, by distinguishing dynamic metrics for each distinct calling context, the CCT provides detailed information on dynamic program behavior which is required in many software engineering areas including profiling [1], debugging [4], testing [39], or reverse engineering [28]. There is a large body of work dealing with various techniques to generate CCTs [1, 3, 49, 42, 50, 7, 31], highlighting the importance of the CCTs for calling context profiling.

Prevailing techniques for generating CCTs typically illustrate them in the form of expandable rooted trees [46]. However, CCTs can become quite large, sometimes comprising millions of nodes. Moreover, the depth of CCTs can be high such that CCTs with 50–400 layers are common in practice. In this situation, navigating and exploring CCTs for a particular dynamic metric of interest (e.g., locating invocations of a particular method in various calling contexts) can be quite cumbersome when a simple expandable tree representation is used, and hence, better visualization techniques are required.

Visualizing hierarchical structures is an important area in the information visualization community that aim to provide better representations of hierarchical structures with enhanced navigation capabilities [16, 19]. One main category of these techniques are *radial* approaches [16] that illustrate hierarchical structures in circular patterns. Previous research has shown that these techniques are quite effective in aiding the users in exploring hierarchical structures [6, 43]. With respect to this discussion, this article introduces *Calling Context Ring Charts (CCRCs)* [29] as a way of visualizing and exploring CCTs. In a CCRC, the CCT root is represented as a circle in the center. Callee methods are represented by ring segments surrounding the caller's ring segment. With CCRCs, it is possible to display all calling contexts of a CCT in a single chart, preserving the caller/callee relationships conveyed in the CCT. For a detailed analysis of certain calling contexts, CCT subtrees can be selected to be visualized separately and the tree depth can be limited.

Currently, CCRCs support three different kinds of visualizations [29]. First, for each caller, the ring segments of the callees have the same size and completely surround the caller's ring segment. While

this representation eases the analysis of caller/callee relationships, it does not convey any information on dynamic metrics collected within different calling contexts. In the second visualization, the angle covered by each ring segment is proportional to the contribution of the corresponding calling context to a chosen dynamic metric, relative to the respective caller's contribution. For example, if CPU time is chosen as metric, this representation reveals the invoked hot methods, for each calling context. While in the first and second visualizations the width of each layer in the ring chart is the same, the third representation continuously reduces the width of outer layers such that the area occupied by each ring segment is proportional to the overall metric contribution of the corresponding calling context.

This article complements the above three visualizations with four tree manipulation operations on the CCT: *subtree selection* and *depth limitation* for handling large CCTs; *recursion elimination* to remove CCT nodes representing recursive calls; and, *dynamic metric aggregation* to determine the overall metric contribution of each method. We provide three implementations of the above CCRC visualizations and these tree manipulations based on JavaScript, Java Swing, and Eclipse SWT. We also report on two case studies where CCRCs have been successfully applied to locate performance problems in two different applications (a lexical analyzer generator and a cryptography library), which were optimized afterwards. These case studies are complemented with a performance evaluation on visualizing the CCTs resulting from the execution of the DaCapo benchmarks [8] for Java, confirming that our implementation can efficiently handle large CCTs.

In summary, this article presents (1) an introduction to CCRCs and the use of radial visualization techniques to represent the hierarchical structures of calling context trees, (2) an introduction to four tree manipulation operations on CCTs, (3) three implementations of the CCRC visualizations and tree manipulation operations, (4) two case studies where CCRCs were successfully applied to find performance issues in two different applications, which were optimized afterwards, and (5) an evaluation that confirms our implementation can efficiently handle large CCTs. With respect to earlier versions of the work that were presented in [29] and [30], the novel contributions of the article are in the items (2), (4), and (5).

The rest of this article is organized as follows. Section 2 presents some background information on the CCT. Section 3 shows the three types of CCT visualization supported by CCRC, while Section 4 discusses our various tree manipulation operations. Section 5 describes our implementations and Section 6 presents the two case studies. Section 7 discusses the results of our performance evaluation. Section 8 compares CCRCs with related work, and finally, Section 9 concludes.

## 2. Background: The Calling Context Tree (CCT)

The Calling Context Tree (CCT) [1] is a popular data structure for representing calling context profiles at runtime. Each node in the CCT corresponds to a calling context and stores the measured dynamic metrics for that particular calling context. The CCT does not restrict which and how many dynamic metrics are stored in the tree nodes. Common metrics include the number of method invocations, the CPU time, the number of cache misses, the number of object allocations, or the amount of allocated memory.

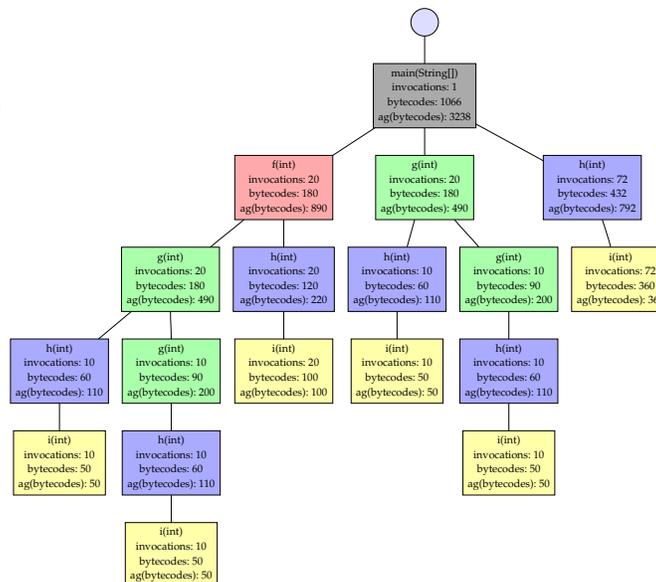
For some metrics  $M$ , it is useful to consider also the aggregated metrics  $ag(M)$  for subtrees of the CCT, in order to explore the overall costs of method executions, including the costs incurred by all direct and indirect callees. In other words, for each CCT node  $n$ ,  $ag(M_n)$  is computed as the sum of the

```

void main (String[] args) {
    for (int j=0; j<20; j++) {
        f(j);
        g(j);
        for (int k=1; k<j/2; k++)
            h(k);
    }
}
void f(int n) {
    int k = g(n);
    k=h(k)*k;
}
int g(int n) {
    if (n%2==0)
        return h(n/2);
    else
        return g(n+1);
}
void i(int n) {
    n=n*n;
}
int h(int n) {
    i(n);
    return n-1;
}

```

(a) Example code



(b) Generated CCT (conceptual representation)

Figure 1. Sample Java code and the CCT generated for one execution of method `main (String[])`. As dynamic metrics, each CCT node stores the number of method invocations and the number of executed bytecodes in the corresponding calling context. For each CCT node  $n$ , the aggregated metric  $ag(\text{bytecodes})$  sums up the executed bytecodes in the whole subtree rooted at node  $n$ .

metric values  $M_x$  of all nodes  $x$  in the CCT subtree rooted at node  $n$ . Aggregated metrics are usually not stored in the CCT nodes, but are computed on demand. For instance, for a given CCT node  $n$ , the metric  $CPU_n$  shows the CPU time spent executing the instructions within the body of the method  $m_n$  represented by the node  $n$ , excluding the CPU time spent in callees of  $m_n$ . In contrast, the metric  $ag(CPU_n)$  gives the overall CPU time spent in the calling context corresponding to node  $n$  and in all callees. Common metrics where aggregation for subtrees helps locate performance problems include the CPU time spent in a calling context or the amount of memory allocated in a calling context. Some other metrics, such as the number of method invocations, are rarely aggregated.

The example in Figure 1(b) shows a conceptual representation of the CCT resulting from executing the Java code sample in Figure 1(a); the used colors differentiate the distinct methods. The illustrated CCT represents one invocation of method `main (String[])`. For instance, such a CCT may be created with the profiler described in [7, 31]. In this example, we assume that two dynamic metrics are collected for each calling context, the number of method invocations and the number of executed bytecodes (in a Java Virtual Machine). Regarding the latter metric, which is a rather platform-independent alternative to the common CPU time metric, we are also computing the aggregated number of executed bytecodes  $ag(\text{bytecodes})$  for subtrees of the CCT.

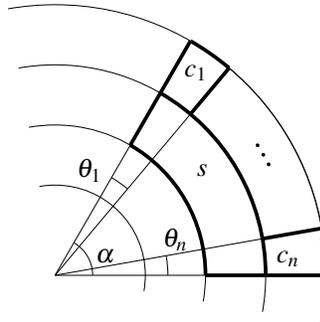


Figure 2. Representation of a CCT node as a ring segment  $s$

There are several variations of the CCT. For instance, calls to the same method from different call sites in a calling method may be represented by the same node or by different nodes in the CCT, depending on whether or not the user wants to distinguish between these call sites [1]. Furthermore, in the presence of recursions, the depth of the CCT may be unbounded, representing each recursive call to a method by a separate CCT node. As an alternative, dynamic metrics for recursive calls may be stored in the same node, effectively limiting the theoretical depth of the CCT (to the number of methods in the profiled program) and introducing back-edges in the CCT [1] (actually, the resulting CCT has not a tree structure).

### 3. CCT Visualization as Calling Context Ring Charts

This section presents three ways of visualizing CCTs as Calling Context Ring Charts. All three visualizations use an *onion-like* structure with circular layers in which each layer corresponds to a layer in the CCT. Beginning with the root node as the central ring, nodes are illustrated as ring segments, and children nodes (callees) are represented on the outer ring of their parent (caller).

#### 3.1. Ring Segments of Equal Length

Figure 2 illustrates a ring segment  $s$  and its  $n$  children nodes  $c_1, \dots, c_n$ . In this first visualization, given the  $\alpha$  as the angle covered by  $s$ , the angle  $\theta_i$  covered by its  $i^{\text{th}}$  child  $c_i$  is computed as follow:

$$\theta_i = \alpha \frac{1}{n} \quad (1)$$

Figure 3 shows this first visualization for our example code from Figure 1. As a consequence of equation (1), the children completely surround their parent. For instance, in the second layer the ring segments representing the callees of `main(String[])` (i.e., `f(int)`, `g(int)`, and `h(int)`) have the same length and completely surround the ring segment of `main(String[])`. The root node of the CCT is represented by the central circle.

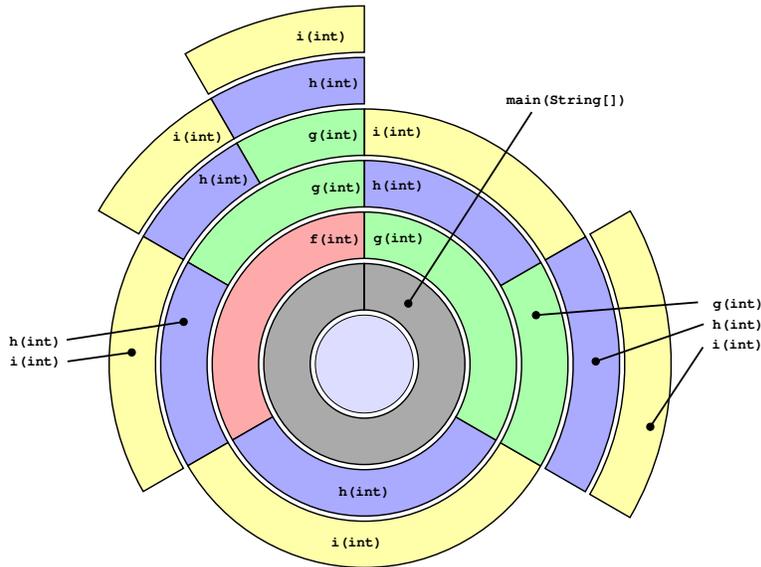


Figure 3. First CCT visualization: Callees are represented by ring segments of equal length

This visualization gives a condensed view of the overall CCT and eases the analysis of caller/callee relationships. However, the visualization does not convey any dynamic metric stored in the CCT.

### 3.2. Ring Segment Length Proportional to a Selected Aggregated Metric

To ease locating performance problems, we support a second visualization in which each ring segment is sized proportionally to an aggregated metric  $ag(M)$  chosen by the user. In this visualization,  $\theta_i$  is computed according to the following equation:

$$\theta_i = \alpha \frac{ag(M_{c_i})}{ag(M_s)} \quad (2)$$

Using equation (2), the sum of all  $\theta_i$  can be less than  $\alpha$ ; that is, the children ring segments do not entirely surround the parent ring segment. The remaining portion of the parent ring segment represents the metrics contribution of the parent excluding its callees.

Figure 4 presents a ring chart where ring segments are sized according to the aggregated metric  $ag(\text{bytecodes})$ . In this example, only about 67% of the ring segment of `main(String[])` are surrounded by the callees' ring segments, and the remaining 33% illustrate the contribution of method `main(String[])` to the overall bytecode execution.

This visualization helps locate hot calling contexts, since the length of ring segments is proportional to an aggregated metric. However, ring segments representing calling contexts with a low metric

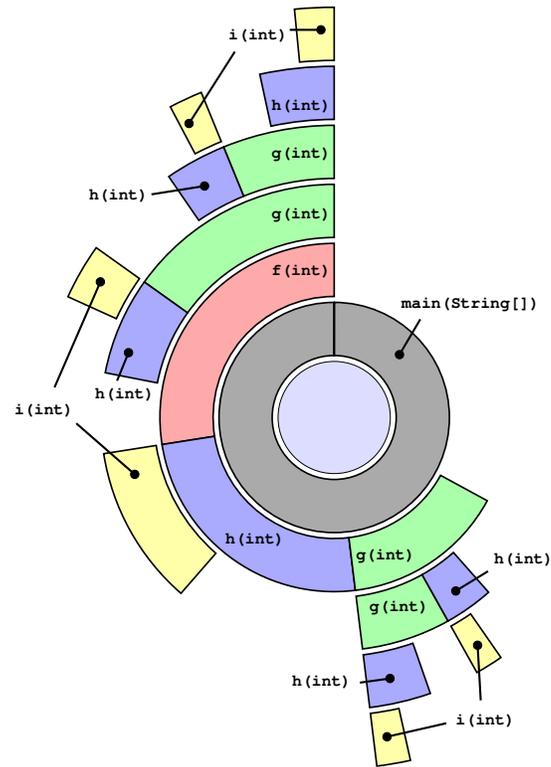


Figure 4. Second CCT visualization: Ring segment *length* is proportional to the aggregated bytecode metric

contribution can be very small such that the user may not be able to see all callees of a method. Hence, our different visualizations are complementary, and the user can switch between the different views.

### 3.3. Ring Segment Area Proportional to a Selected Aggregated Metric

In the previous visualization, we considered sizing ring segments with respect to the angle, which can be misleading: A node close to the center (i.e., with a low depth in the CCT) will appear smaller in terms of area than a node with the same aggregated metric contribution but in a deeper CCT layer. The third visualization presented below compensates for this bias by reducing the width of outer layers such that any two nodes with the same aggregated metric contribution will be represented by ring segments of the same area.

We call  $r_i$  the radius of the  $i^{\text{th}}$  concentric circle,  $r_0$  being the radius of the central circle. In the two previous visualizations, we implicitly defined  $r_i = r_{i-1} + c$ , where  $c$  is the constant width of ring

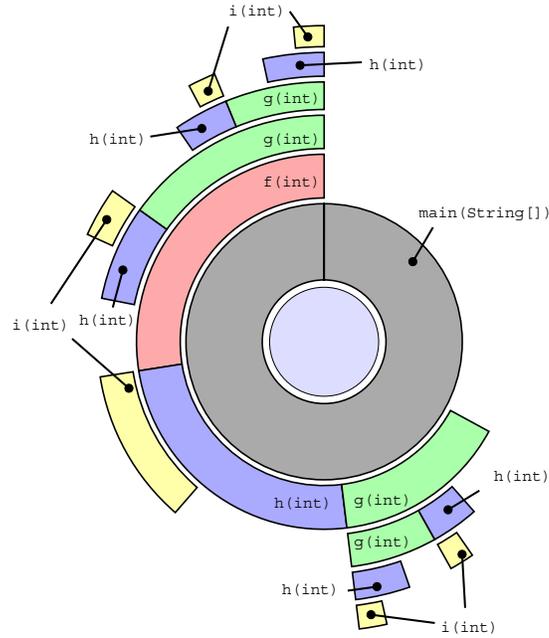


Figure 5. Third CCT visualization: Ring segment *area* is proportional to the aggregated bytecode metric

segments. Since the area of a circle sector is  $\frac{\alpha}{2}r^2$  (where  $r$  is the radius and  $\alpha$  is measured in radians), the area  $A_s$  of the ring segment  $s$  in layer  $i$  is given by:

$$A_s = \frac{\alpha}{2}(r_i^2 - r_{i-1}^2) \quad (3)$$

For an area-proportional visualization of a calling context according to its aggregated metric contribution, but independently of its depth in the CCT, the term  $(r_i^2 - r_{i-1}^2)$  in equation (3) must be constant. This is achieved by redefining  $r_i$  according to equation 4 in which  $K$  is a constant corresponding to the desired overall area of a ring (layer) in the chart:

$$r_i = \sqrt{r_{i-1}^2 + \frac{K}{\pi}} \quad (4)$$

We illustrate this visualization in Figure 5. While this visualization better supports locating hotspots, in a deep tree outer rings may become too thin to be easily visible. Thus, it is important to let the user toggle between the different views.

## 4. CCT Manipulation Operations

In this section, we discuss four tree manipulation operations on the CCT that complement our visualizations and help exploring calling context profiles. The first two CCT manipulations [29], *subtree selection* and *depth limitation*, are essential for handling large CCTs. The third manipulation, *recursion elimination*, removes CCT nodes representing recursive calls. This operation can significantly reduce the CCT size in the presence of recursions. Finally, the fourth operation, *dynamic metric aggregation*, aggregates metrics for each method, completely discarding calling context information.

### 4.1. Subtree Selection

This manipulation allows the selection of any CCT node as root, hence resulting in the visualization of a subtree as a CCRC. In this way, the user can focus on a particular calling context of interest, together with the direct and indirect callees.

Our implementation supports interactive subtree selection by clicking on the ring segment to become the new root. The history of selected subtrees is preserved, allowing the user to revert to previous views of the CCT. When clicking on the central circle representing the root node of the currently visualized CCT subtree, the previously displayed subtree is restored. Clicking on the root of the overall CCT has no effect.

Figure 6(a) shows an example of the subtree selection operation, where the ring segment corresponding to `f(int)` is selected as the new root (cp. the original CCT in Figure 1). In this example, we use our second visualization where the length of ring segments is proportional to the metric *ag*(bytecodes).

### 4.2. Depth Limitation

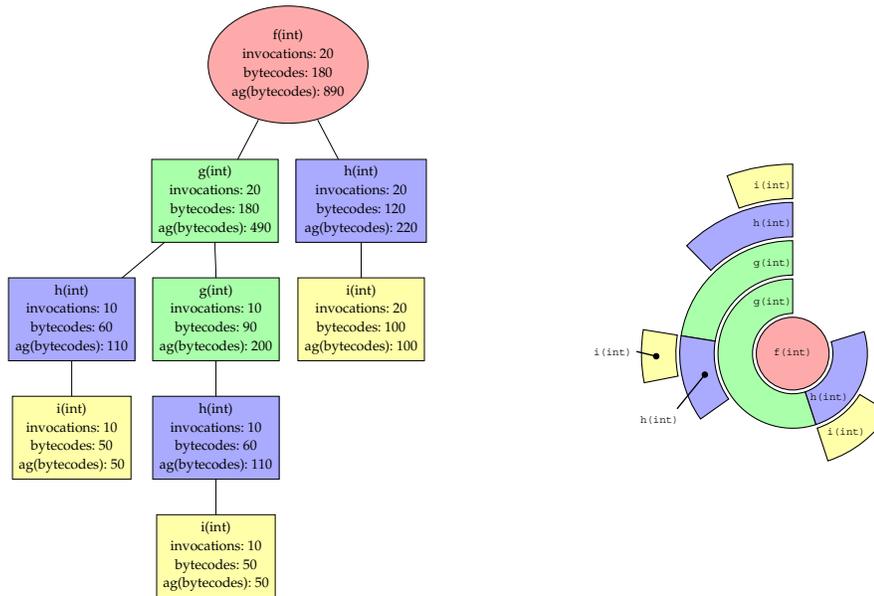
The second CCT manipulation, limiting the depth of the visualized CCT, also helps analyze large CCTs with deep calling contexts. By reducing the CCT depth, the remaining layers in the CCT can be visualized in wider rings.

As in the case of subtree selection, also depth limitation can be applied interactively, either by entering a new maximum depth value or by using the scroll wheel on the mouse. In particular, the latter possibility gives the user an easy way of (un)zooming the center of the displayed CCT. By successively applying subtree selection and adapting the depth limit, the user can quickly navigate in the CCT. With the aid of subtree selection, the interesting part of program execution can be chosen, and depth limitation allows the user to adjust the details of callees to be displayed.

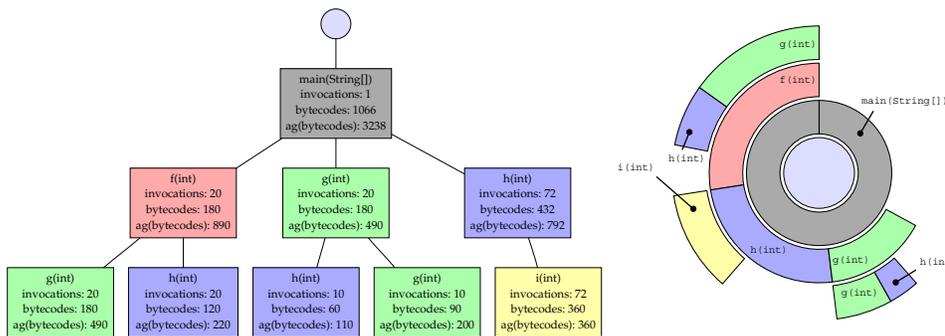
Figure 6(b) illustrates the effect of limiting the CCT depth to 3, i.e., displaying only 3 layers of the CCT. Note that the aggregated metrics *ag*(*M*) (for any metric *M*) are not recomputed. Aggregated metrics are initially computed on the complete CCT.

### 4.3. Recursion Elimination

In Section 2 we discussed different variations of the CCT in which recursive calls may be represented either by the same CCT node or by different CCT nodes. In the former case, back-edges are introduced

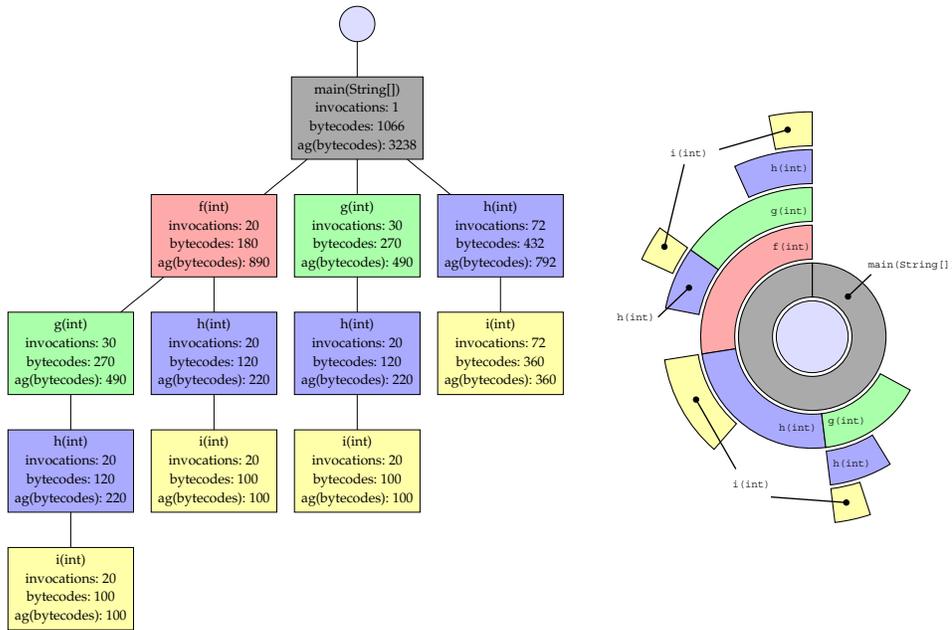


(a) Subtree selection: The ring segment of method `f (int)` has been selected as root

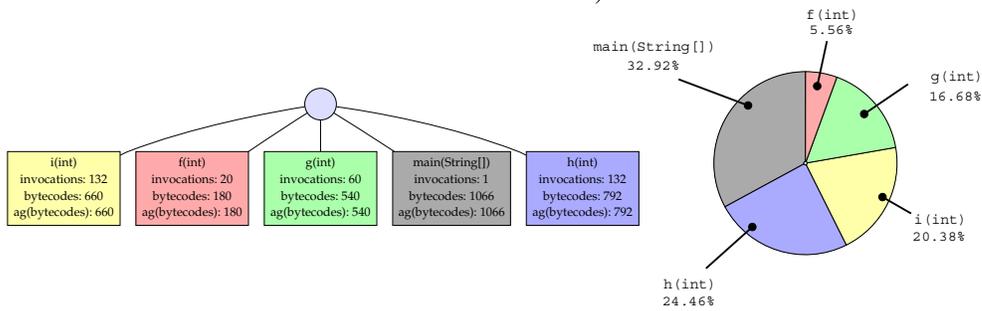


(b) Depth limitation: The CCT depth has been limited to 3 layers

Figure 6. Subtree selection and Depth limitation operations (the original CCT is depicted in Figure 1)



(a) Recursion elimination: CCT nodes representing recursive calls have been removed (back-edges are not shown)



(b) Metrics aggregation by method: Calling context information has been discarded

Figure 7. Recursion elimination and Metrics aggregation operations (the original CCT is depicted in Figure 1)

```

Node eliminateRecurSIONS(Node rootIn) {
    Node rootOut = new Node();
    eliminateRec(rootIn, rootOut);
    computeAggregatedMetrics(rootOut);
    return rootOut;
}

void eliminateRec(Node in, Node out) {
    for (Node nIn: in.children()) {
        Node nOut = findMatchingAncestor(out, nIn.methodID());
        if (nOut == null)
            nOut = out.getOrCreateChild(nIn.methodID());
        nOut.metrics().add(nIn.metrics());
        eliminateRec(nIn, nOut);
    }
}

Node findMatchingAncestor(Node n, String id) {
    if (n == null) return null;
    else if (n.methodID().equals(id)) return n;
    else return findMatchingAncestor(n.parent(), id);
}

```

Figure 8. Pseudo-code for eliminating recursive structures in a CCT

in the CCTs [1]. Nevertheless, the CCTs can become very deep in the latter case when representing executions of recursive programs. To address this issue, this section provides a CCT transformation that takes a (potentially very deep) CCT where recursive calls are represented by separate nodes and produces a corresponding (more compact) CCT where the metrics collected in recursive method executions are stored within the same node. Figure 7(a) illustrates this transformation with an example.

The transformation is specified using Java-style pseudo-code in Figure 8. The `eliminateRecurSIONS(...)` method takes the root node of a CCT and returns the root of a new CCT (not sharing any nodes with the input CCT) without any recursive structures. To this end, `eliminateRec(...)` traverses the input CCT in pre-order (depth-first) and builds the corresponding output CCT. Before creating a new node in the output CCT, `eliminateRec(...)` checks whether a matching node is on the current path to the root (`findMatchingAncestor(...)`). In that case, no new node is created in the output CCT. Since back-edges cannot be visualized, they are not created in the output CCT.

The pseudo-code assumes that CCT nodes are represented by the type `Node`, which supports the operations `methodID()` (returning a string uniquely identifying the method for which the node stores dynamic metrics), `parent()` (returning the parent node in the CCT or `null` for the root node), `children()` (returning the set of children nodes), and `metrics()` (returning the dynamic metrics for the node). The `getOrCreateChild(String calleeName)` method returns the `Node` instance representing the callee with the given method name. If a child node for the given callee name does not yet exist, a new node is created. The dynamic metrics collected for a node are represented by the type `Metrics`, which supports, amongst others, the operation

`add(Metrics from)`, incrementing the metric values by the values from the argument `Metrics` instance. Finally, `computeAggregatedMetrics(Node root)` computes all desired aggregated metrics in a CCT.

#### 4.4. Dynamic Metrics Aggregation for Each Method

While the CCT gives detailed information for each calling context, it is difficult to estimate the overall metric contribution of a method if it is used in many different calling contexts. In our example (see Figure 1), `i(int)` appears in six CCT nodes, which makes it hard to evaluate the total metric contribution of that method in comparison to other methods. Hence, we provide a transformation that discards all calling context information. Each method occurring in the original CCT is represented by a single node that stores the summed up metrics for that method (disregarding any callees).

Figure 7(b) illustrates an example of this operation. Since all calling context information is discarded, the resulting tree has a depth of one and can be represented as a conventional pie chart. While such pie charts are not unique to our approach (many prevailing profilers provide similar views), they provide a useful, complementary means for performance analysis. The transformation is also applicable for CCT subtrees (in conjunction with subtree selection).

### 5. Implementation

We implemented three versions of our CCRC visualization tool. The first version is based on JavaScript and on the standard XML-based Scalable Vector Graphics (SVG)<sup>†</sup> format, whereas the second version is a Java application using the Swing toolkit. The third version is also implemented in Java by using the Eclipse's Standard Widget Toolkit (SWT)<sup>‡</sup>.

Our first implementation uses JavaScript to create and dynamically transform the CCT, which is represented as an XML structure. This implementation allows us to visualize CCRCs directly in standard web browsers. The rendered page consists of two parts, one part is showing the ring chart, while the other part presents detailed calling context information of the currently selected CCT node (a node is automatically selected when it is pointed to by the mouse), such as the complete call stack and collected dynamic metrics. The navigation in the CCT representation is implemented using events which are handled by the browser and trigger calls to JavaScript routines that update the displayed information. On one hand, this implementation is well suited for integration into web-based collaboration tools. On the other hand, JavaScript execution engines embedded in currently available browsers are often unable to handle very large structures as those required for representing CCTs. Moreover, the limited performance of Javascript for complex tree manipulations can result in poor user experience.

In our second Java-based implementation, the GUI has the same structure and similar behavior as in the first implementation. However, in contrast to the first implementation, it has been designed to handle very large CCTs. Additional features were added, such as an advanced search tool to find and

---

<sup>†</sup><http://www.w3.org/Graphics/SVG/>

<sup>‡</sup><http://www.eclipse.org/swt/>

highlight CCT nodes with certain properties. For instance, the user can search for calling contexts representing execution in a particular package, class, or method. It is also possible to quickly locate CCT nodes where the collected metrics meet given constraints (e.g., exceeding some thresholds).

Our third implementation, which is based on Eclipse SWT, has the same advanced features as the Swing-based version. In addition, it can be used in the Eclipse IDE. We integrated our CCRC visualization in the Senseo Eclipse plugin [38], which enriches Eclipse's static source view with dynamic metrics from a running application. Senseo provides the developer with various dynamic metrics, such as runtime receiver, argument, and return types of invoked methods, the number of allocated objects, and an estimation of total memory consumption. The integration of CCRC enhances Senseo with complete calling context information. It also supports interactions between the CCRC and the corresponding methods in the source code, such as highlighting the calling contexts in the CCRC corresponding to a chosen method, or showing the method source code of a selected calling context in the CCRC.

## 6. Case Studies

This section presents two case studies where CCRCs have been successfully applied to locate performance problems in applications, which were optimized afterwards. In both case studies, the calling context profiles are collected with the profiler described in [7, 31], which collects the number of executed bytecodes in each calling context (in addition to other dynamic metrics); this metric is largely platform-independent. This profiler has the advantage of profiling overall application execution, including the execution of methods in the Java class library. Recursions are fully represented in the generated profiles, that is, the CCTs do not have any back-edges.

For CCRC visualization, we use our Java Swing-based implementation. We do not remove any recursions in the presented CCRCs. For the CCRCs in this case study, we use our second kind of visualizations, where the ring segment length is proportional to the aggregated bytecode metric (see Section 3.2). The CCRCs show the `main(String[])` method as root in the center, that is, we use subtree selection in order to concentrate on application execution, disregarding the execution of system threads, which is also conveyed in the profiles generated by our profiler. For all shown CCRCs, we limit the depth to 10 layers.

The primary contribution of these case studies is to show that CCRCs help quickly locate hotspots in calling context profiles. As secondary contribution, the case studies also demonstrate that profiling using a platform-independent metric (i.e., the number of executed bytecodes) – instead of the common CPU time metric – is a useful technique for detecting hotspots and for optimizing applications. The resulting optimized applications not only execute less bytecodes, but they also perform significantly better in terms of execution time.

### 6.1. Case Study 1: JLex

Our first case study is inspired by a paper describing the profiler generator ProfBuilder [12]. The authors of ProfBuilder analyzed and detected a hotspot in the lexical analyzer generator JLex [17]. In this case study, we show that CCRCs ease locating such a hotspot. We generated the CCT for the execution of a sample grammar included in the JLex distribution.

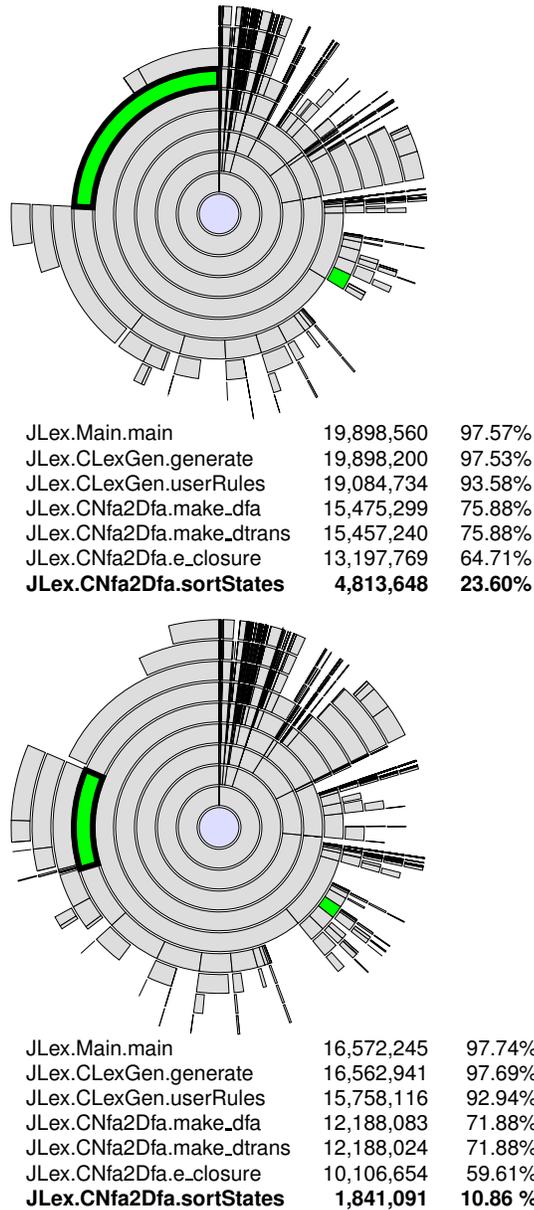


Figure 9. CCRCs for JLex with the original (top) and optimized (bottom) sort algorithms

Figure 9 (top) shows a CCRC representing the execution of the original, unmodified JLex. Looking at the aggregated bytecodes, we were able to locate a hot calling context, an invocation of the `sortStates(...)` method. This method is executed in two different calling contexts; in one of them, it executes a high number of bytecodes, contributing 23.6% of the overall executed bytecodes. The large ring segment corresponding to this calling context is highlighted (both in color and in bold) in Figure 9 (top). The corresponding call stack (together with the aggregated bytecode metric) is also shown in Figure 9. The other calling context, where `sortStates(...)` is executed, is marked in color (but not in bold); its contribution to the overall executed bytecodes is relatively insignificant.

The `sortStates(...)` method uses a primitive selection sort algorithm of complexity  $O(n^2)$ . In order to optimize the code, we replace it with the merge sort algorithm of complexity  $O(n \log n)$ . Figure 9 (bottom) shows the resulting CCRC after the optimization. The number of executed bytecodes in the previously hot calling context representing the `sortStates(...)` method is reduced to 10.86% of the overall executed bytecodes.

In order to confirm that our optimization based on the bytecode metric also results in a speedup in execution time, we run both versions of JLex (original and optimized) on an Intel Core 2 Duo 2.33GHz computer with 2GB RAM (Linux Fedora 10), using Sun JDK 1.6.0\_12. Regarding the execution time, we present the median of 15 runs. The original JLex executes 20,393,685 bytecodes in 37.04ms (including the profiled bytecodes executed by system threads, which are not shown in Figure 9), whereas the optimized JLex executes 16,955,185 bytecodes in 27.7ms. In the number of executed bytecodes metric, the optimized version is 20.3% “faster”, while in the CPU time metric, the optimized version is 33.7% faster.<sup>§</sup>

## 6.2. Case Study 2: Shamir’s Secret Sharing (SSS)

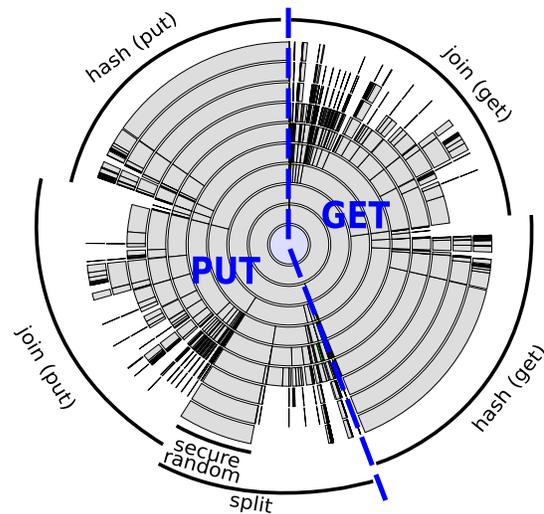
Our second case study analyzes an implementation of the Shamir’s Secret Sharing (SSS) cryptography algorithm [40] done by Ricardo Padilha at the University of Lugano. We asked the developer to analyze his implementation with CCRCs in order to validate his design choices and to optimize his code.

Figure 10 illustrates a CCRC corresponding to one hundred thousand *put* and *get* operations. It also represents the bytecode distribution of different computations. In this implementation of the SSS algorithm, a secret is divided into parts (by using the *split* operation) and the parts are distributed among the participants (by using the *hash(put)* operation). Then, all the parts or some of them are fetched (by using the *hash(get)* operation) in order to reconstruct the secret (by using the *join(get)* operation). The *join(put)* operation is also needed because of the API which was used in programming the putting of shares. According to this API, when replacing a stored value, the old value must be returned, forcing the join to be calculated also upon putting the shares. In this figure, a symmetry between the calculations of putting the shares and getting them back can be seen as well.

After analyzing the CCRC in Figure 10, the developer first observed that the execution of the *split* operation was dominated by the use of the `java.security.SecureRandom` class. As an optimization, he decided to use the non-secure class `java.util.Random` instead since the security of his implementation was not dependent on the randomly generated coefficients [35, 25]. Figure 11

---

<sup>§</sup>While the number of executed bytecodes is exactly reproducible in each run, the measured execution time varies considerably, but the optimized version is consistently faster.



Total number of executed bytecodes: 29,568,435.996

Figure 10. CCRC for the initial implementation of SSS algorithm

represents the CCRCs before and after the change. The highlighted segments in Figure 11 (top and bottom) correspond to the `BigInteger.randomBits()` method, which is the common code executed when using the `SecureRandom` and `Random` classes. Thanks to this optimization, the calculation of the random coefficients in the SSS formula [40] was reduced from 5.96% to 0.43% of the overall executed bytecodes.

Second, the developer noticed that a high percentage of executed bytecodes were used to calculate the hash function (see Figure 10). He decided to revise his implementation of the SSS algorithm and remove the use of hash function. Figure 12 shows the CCRC for the final SSS implementation. Compared to the original implementation (see Figure 10), the total number of executed bytecodes has been reduced from 29,568,435,996 to 15,129,000,509.

In order to confirm that the optimizations based on profiles with bytecode metrics also result in a speedup, we measured the execution times of the three different versions (initial version, version without secure random, and final version with neither secure random nor hash), using the same settings as in the previous case study. The original implementation took 9150ms. The implementation without secure random took 7579ms (speedup of 20.7%). Finally, the implementation with neither secure random nor hash took 5557ms (speedup of 64.7%).

In summary, our case studies demonstrate that CCRCs are useful in optimizing real-world applications.

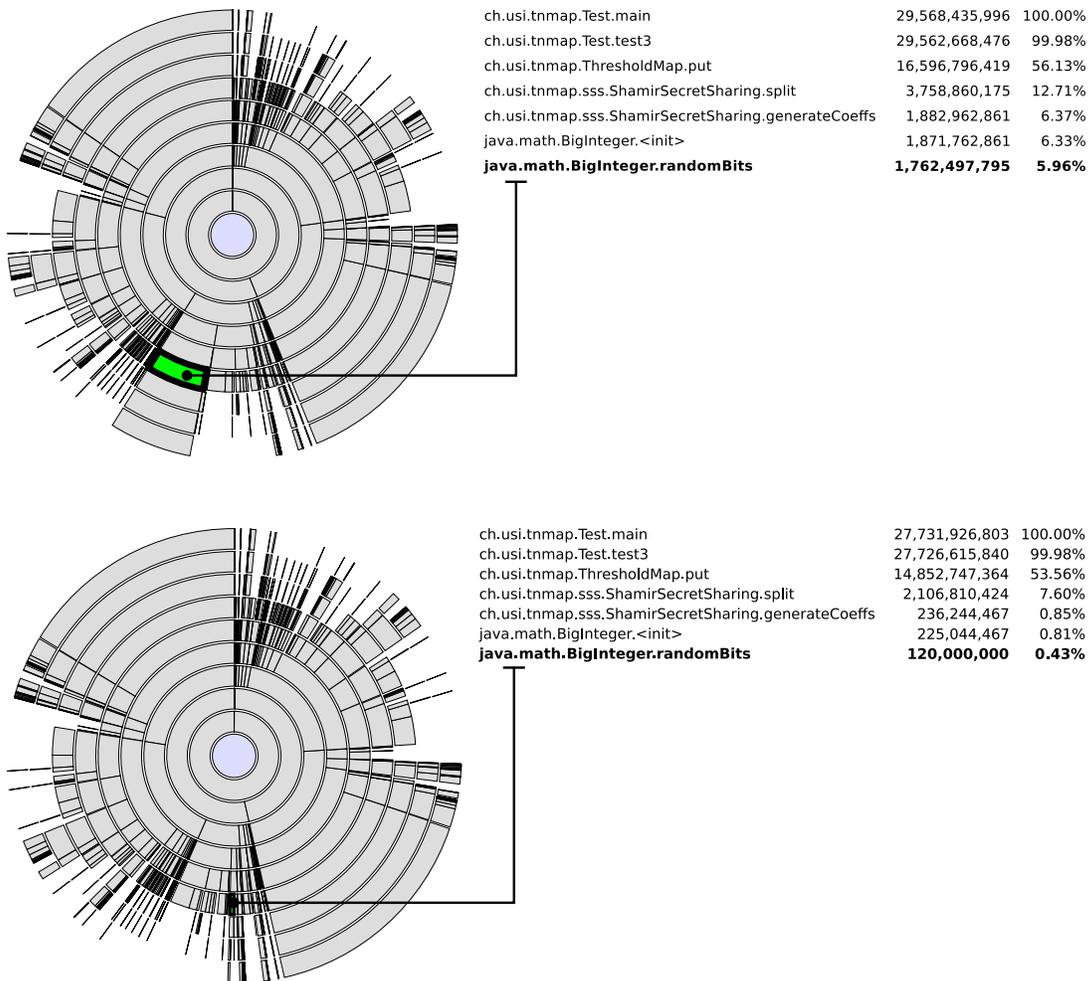
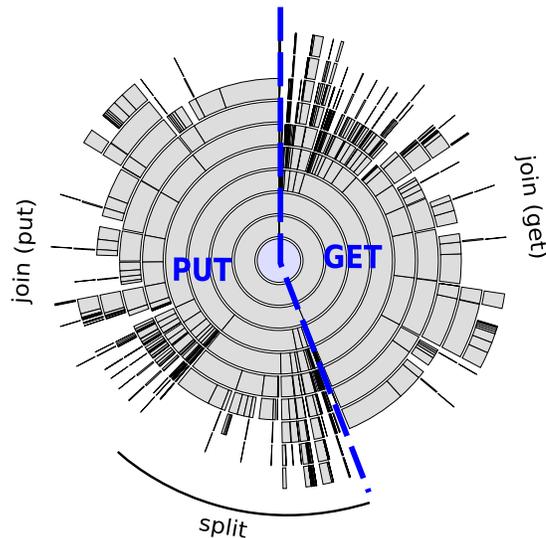


Figure 11. SSS implementation using `java.security.SecureRandom` (top) versus SSS implementation using `java.util.Random` (bottom)

## 7. Evaluation

To validate our implementation of CCRC visualizations with very large profiles, we generated and visualized the CCTs for the standard DaCapo benchmark suite [8]. In the following, Section 7.1 presents some statistics on the generated CCTs that correspond to the execution of one run of each benchmark with its default settings. Then, these CCTs were rendered and visualized using our Java Swing-based implementation, with different tree depth settings. Section 7.2 discusses the results of measuring these renderings and shows the relationship between the number of visualized tree layers with the rendering time and the number of rendered segments. All measurements were collected with



Total number of executed bytecodes: 15,129,000.509

Figure 12. CCRC of the final implementation of SSS (without hashing and without secure random)

Sun JDK 1.6\_18 running on a laptop with an Intel Core 2 Duo 2.33Ghz processor, 3GB of memory, and the Linux Fedora 12 operating system.

### 7.1. Statistics of Generated CCTs

This section presents some statistics of the CCTs generated for the standard DaCapo benchmark suite. To generate these CCTs, we benefitted from the same calling context profiler [7, 31] that was used in our two case studies in Section 6. We also transformed the CCTs by removing recursive structures in order to explore how much recursion is used in the benchmarks and to see to which extend the algorithm presented in Figure 8 is able to reduce the size of the CCTs.

Table I indicates the statistics computed for the original CCTs and the transformed CCTs. In this table, column “#Method Calls” indicates the total number of method invocations (in all CCT nodes), which can be very high (e.g., over 1 billion for the “bloat” benchmark). Column “#Unique Methods” illustrates the number of unique methods that were invoked at least once. These two statistics are not affected by the transformation that removes recursions.

The columns “Original CCTs” and “Transformed CCTs” respectively illustrate the number of nodes (column “#Nodes”) and the maximum depth of trees (column “Max Depth”) for the originally generated CCTs and for the transformed ones after removing all recursions. The two statistics in the column “Original CCTs”, i.e., “#Nodes” and “Max Depth”, give an idea of the sizes of the CCTs to be

DaCapo Benchmark	#Method Calls	#Unique Methods	Original CCTs		Transformed CCTs	
			#Nodes	Max Depth	#Nodes	Max Depth
antlr	165,465,913	2,498	627,577	164	254,289	39
bloat	1,055,927,864	3,033	645,316	128	212,927	48
chart	404,140,995	4,554	92,272	65	75,266	55
eclipse	941,528,217	11,555	2,166,169	131	977,531	56
fop	43,143,714	4,013	149,492	76	100,590	55
hsqldb	214,764,027	2,507	66,391	62	60,533	60
jython	945,580,102	4,574	1,941,246	223	553,124	76
luindex	443,252,127	1,969	58,834	52	46,155	38
lusearch	502,521,414	1,768	33,189	63	28,414	35
pmd	462,558,862	3,663	800,071	416	106,841	79
xalan	636,834,930	3,679	211,213	79	97,547	56

Table I. Statistics of original CCTs and the transformed ones after removing all recursions

visualized. For instance, for the “eclipse” benchmark, the original CCT consists of more than 2 million nodes, and for the “pmd” benchmark, the CCT depth exceeds 400 layers.

Considering the statistics for the transformed CCTs indicates a significant reduction in the sizes of the CCTs compared to the original ones. In particular, the transformed CCTs have less than 1 million nodes and the maximum depth of the trees do not exceed 80 layers. This data shows that recursive algorithms play an important role in typical Java workloads. It is worth mentioning that recursive algorithms are not only found in the DaCapo benchmark suite, but also in the standard Java class library. Prominent examples include classloading (classloaders use a delegation model by default) and balanced tree algorithms in the `java.util` package.

## 7.2. CCT Visualization Performance

We visualized the original CCTs generated in the previous subsection using our Java Swing-based implementation and our second type of visualization (see Section 3.2) on a screen resolution of  $1440 \times 900$  pixels. Then, we varied the number of displayed tree layers from 2 to 150 using the tree depth limitation operation, and measured the relationship between the number of visualized tree layers in the CCRC with rendering time and the number of visualized ring segments. For each setting, we took the median of 15 runs, so as to avoid perturbations such as garbage collection that may be triggered during the renderings.

Figures 13–16 illustrate the results of our measurements. As can be seen in these diagrams, the number of displayed segments in the CCRCs ranges from 2 to 4206, and the rendering time is between 14 and 204 milliseconds for varying number of displayed tree layers. As experimented in [13], users will not notice delays in their mouse interactions when the response time is up to 195 milliseconds. So, these diagrams support our claim that rapid navigation of CCRCs is easily possible, allowing us to explore large calling context profiles smoothly.

The above interesting results are achieved by some optimizations in the rendering engine in which the segments that get too small to be visualized (e.g., their sizes would be less than a pixel) are replaced by a line. If the user is interested in this subtree, he can select a parent node as the new root and specify a desired tree depth accordingly for further visualization. These optimizations are also the reason why

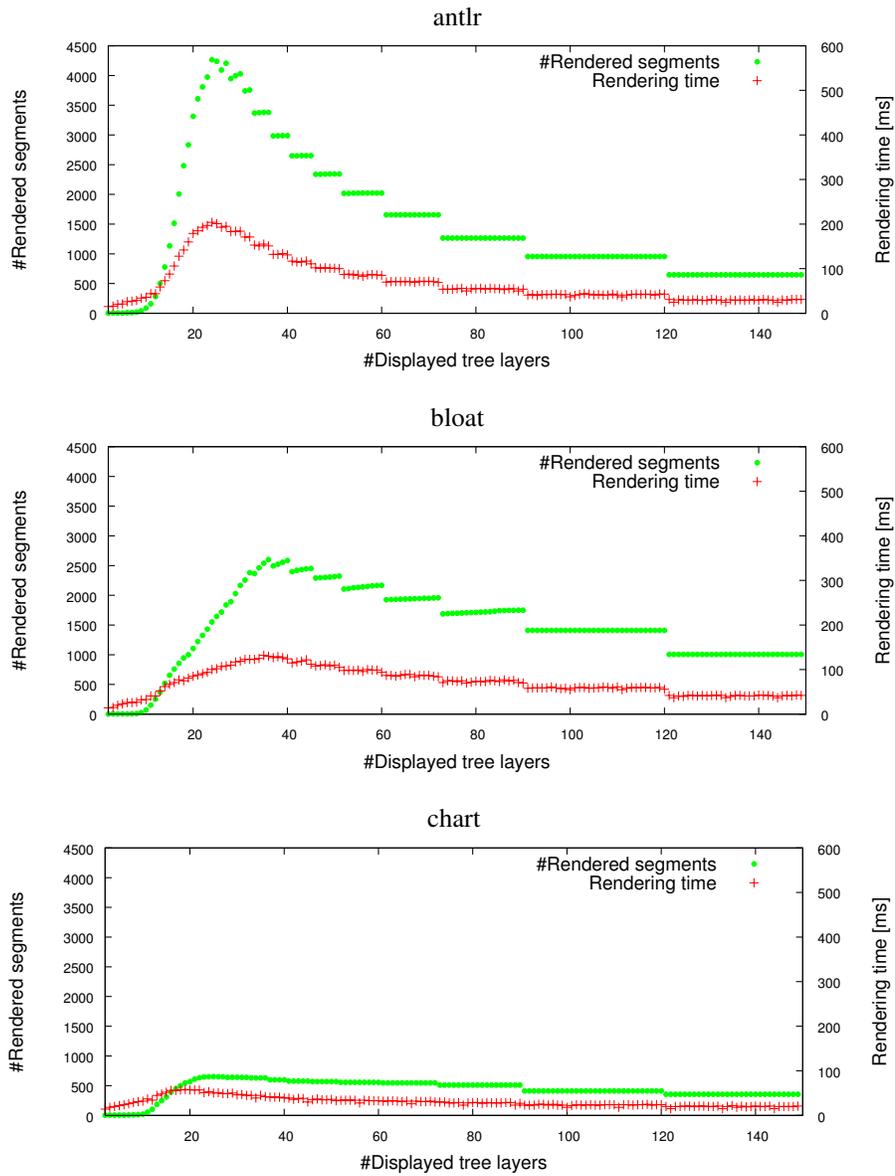


Figure 13. Relationship between the number of visualized tree layers in the CCRC and the rendering time and the number of visualized ring segments for the benchmarks “antlr”, “bloat”, and “chart”

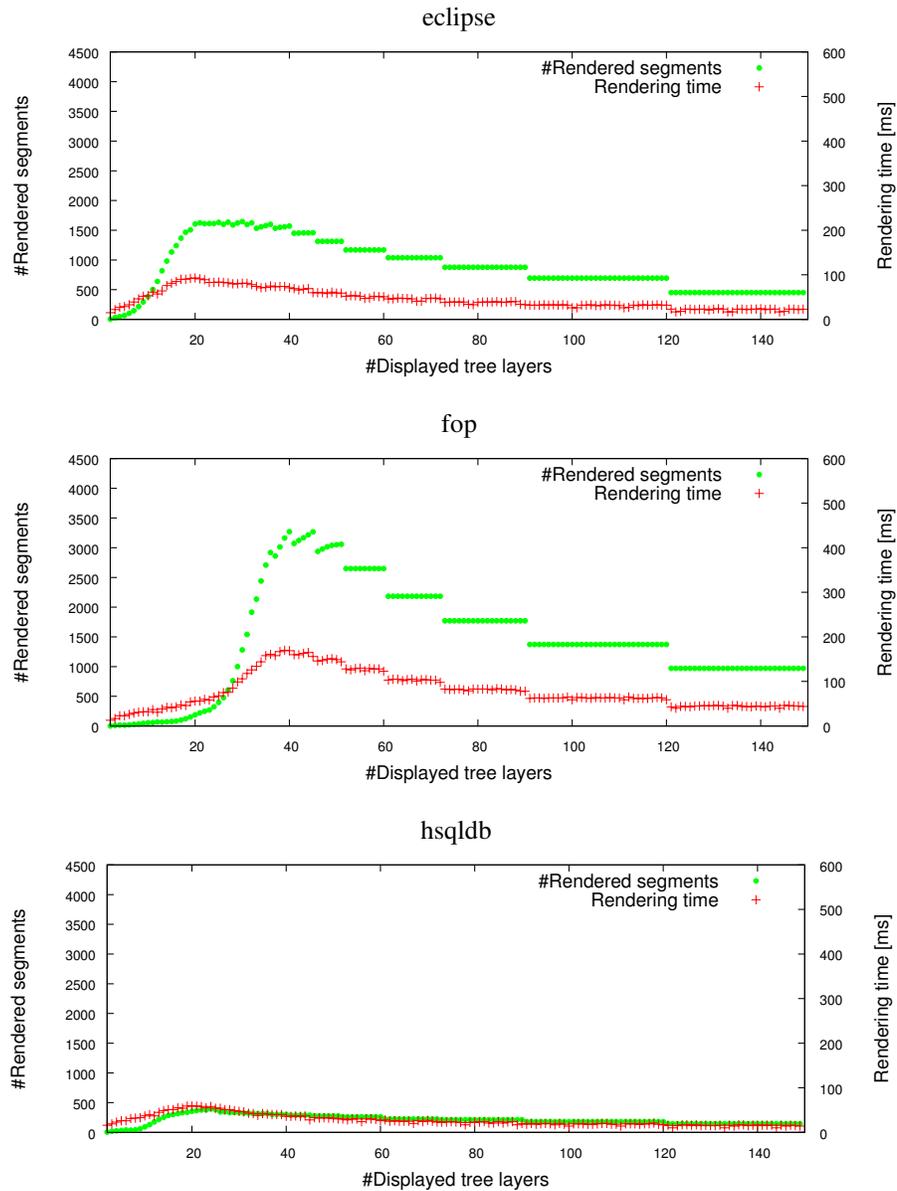


Figure 14. Relationship between the number of visualized tree layers in the CCRC and the rendering time and the number of visualized ring segments for the benchmarks “eclipse”, “fop”, and “hsqldb”

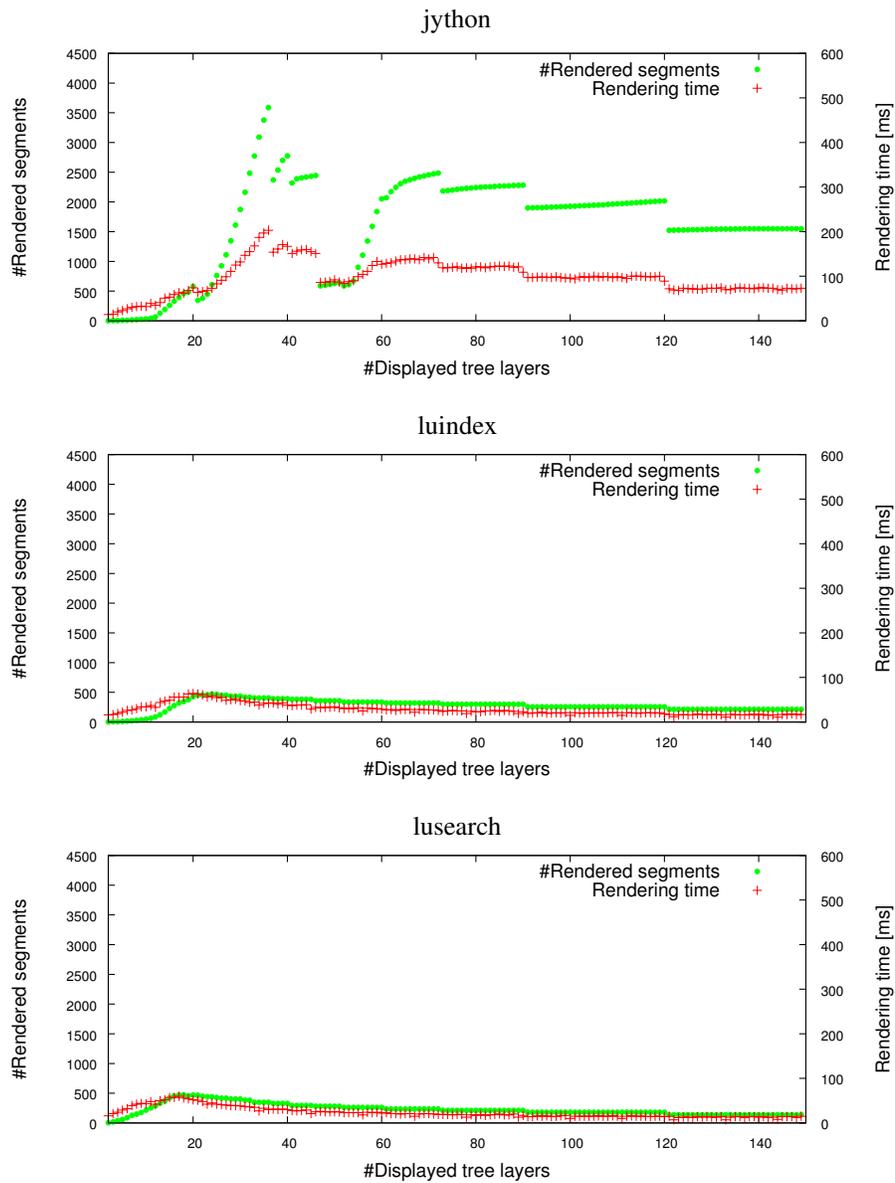


Figure 15. Relationship between the number of visualized tree layers in the CCRC and the rendering time and the number of visualized ring segments for the benchmarks “jython”, “luindex”, and “lusearch”

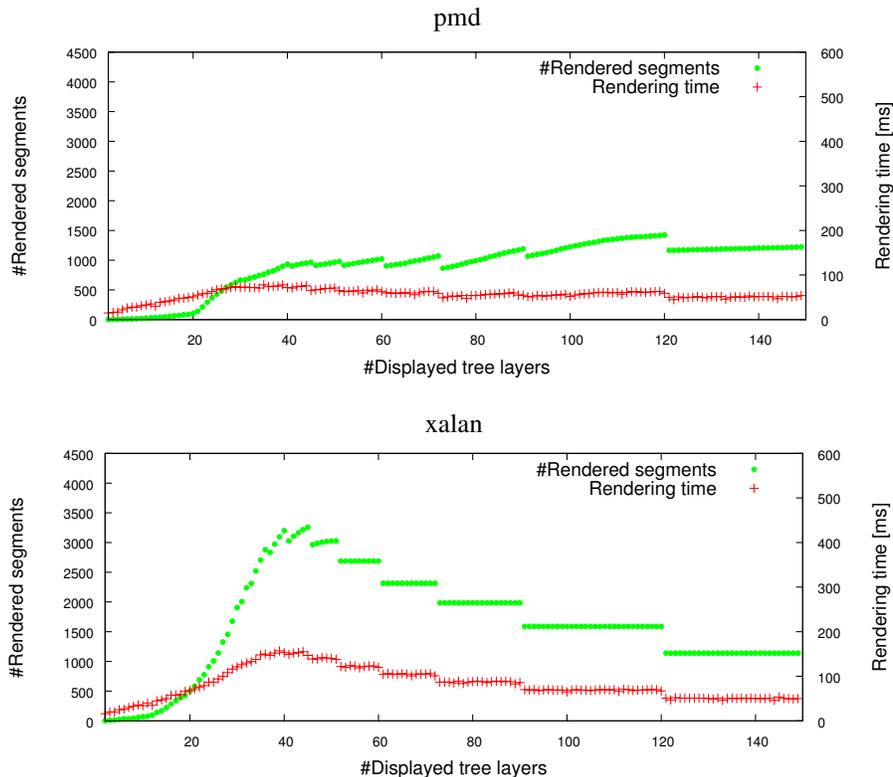


Figure 16. Relationship between the number of visualized tree layers in the CCRC and the rendering time and the number of visualized ring segments for the benchmarks “pmd” and “xalan”

the diagrams are not continuous lines, and why the number of displayed segments decrease after a while as the number of layers increase (i.e., the subtrees that would not be visible anyway are not rendered). In the case of “jython” and “pmd”, a different pattern is seen in their corresponding diagrams: the number of rendered segments increases as the number of displayed tree layers increases (we go deeper into their CCTs). The reason for this situation in these two particular diagrams is that some branches appear only when more tree layers are visualized.

In summary, our performance evaluation confirms that CCRC rendering is perceived almost instantaneously by the users, even for very large CCTs.

## 8. Related Work

This section considers related work in the areas of visualizing hierarchical structures and visualizing profiling data as follows.

### 8.1. Visualizing Hierarchical Structures

Since CCTs have hierarchical structures, different techniques for visualizing hierarchical structures are of interest in this article. A large body of work exists for visualizing hierarchical structures, most of which focus on the challenge to display large hierarchies in a comprehensible form [14]. One can refer to [19] for a detailed survey of different techniques. Examples include *SunBurst views* [43, 44], *tree-maps* [24], *rooted trees* [46], *cone trees* [37], *radial trees* [19, 46], *reconfigurable disk trees* [23], *botanical trees* [26], *collapsible cylindrical trees* [14], *hyperbolic layouts* [27, 32], and *SHriMP views* [45]. One main category of these approaches is space-filling techniques that aim to utilize the display space very efficiently.

One of the most well-known space-filling techniques is the tree-map approach [24] and its variations [5, 41, 47]. In tree-maps, nodes are represented as rectangular areas sized proportionally to a metric. Although tree-maps cover 100% of the display space, only leaf nodes are clearly identifiable, which complicates understanding of the hierarchical structure. Moreover, the original layout, called the slice and dice, tends to create outstretched rectangles which are difficult to see, as pointed out in [5, 41]. Several improvements to the original tree-maps have been proposed, such as changing the layout policy [41] to obtain better proportioned rectangle nodes while keeping ordering information, such that a slight change in the tree does not cause a major change in the tree-map. In [5] polygons are used instead of rectangles so as to provide a way to distinguish different branches. While CCRCs do not cover the entire display space due to their circular shape, they offer a convenient way to view the hierarchical structure of a CCT, to navigate in the tree, and to identify calling contexts with a significant contribution to a selected dynamic metric.

Another category of space-filling techniques are radial methods [16]. An example of radial methods are SunBurst views [43, 44] which highly resemble CCRCs. In SunBurst views, the items in a hierarchy are laid out radially, with the top of the hierarchy at the center and deeper layers further away from the center. The sweep angle of an item is proportional to a desired metric and its color may correspond to some desired attributes. Previous empirical studies, such as [6] and [43], have illustrated that radial visualization techniques work better for navigating and exploring hierarchical structures compared to rectangular space filling techniques such as tree-maps. However, they suffer from this inherent difficulty that slices may become too small to distinguish. This issue is addressed by context+focus techniques, such as the ones in [44], or *information slices* [2] in which two semi-circular areas are used as a form of two-level overview and detail. While CCRCs deal with hierarchies that may include millions of elements, the paper on SunBurst views [44] considers a hierarchy that comprises only 500 elements. The large size of CCTs requires specific navigation techniques. To address this requirement, CCRCs support zooming by depth limitation as well as subtree selection for exploring different parts of a CCT, while SunBurst navigation offers different zooming features that show the whole hierarchy and a zoomed part within the same view.

---

## 8.2. Visualizing Profiling Data

Tree-maps have been used to visualize profiling data. For instance, in [48] the authors present KCacheGrind, a front end to a simulator-based cache profiling tool, using a combination of tree-maps and call graphs to visualize the data. In contrast, CCRCs use a specialized representation that is suited both for understanding caller/callee relationships and for locating hotspots by sizing ring segments proportionally to a chosen metric.

Execution traces may be used for analyzing dynamic program behavior. Execution traces are logged events, such as method entry and exit, or object allocation. However, the resulting amount of data can be excessive. In [15] execution traces are visualized with nodes representing classes and edges representing methods calls. Node size and edge thickness are mapped to properties (e.g., number of method invocations). A time range can be selected in order to limit the data to be visualized. Another approach to visualizing execution traces has been introduced in [22]. It uses the concept of hierarchical edge bundles [21], where similar edges are put together to improve the visualization of larger traces. Execution traces allow us to keep calls in sequences and to select a precise time interval to be visualized, which helps us to understand a particular phase in the execution of a program. In our approach, the profiling data is stored in a CCT, a more compact structure that does not keep calls in sequences, but gives a view of the overall program execution, which is more suitable for locating hotspots.

Several profilers, such as JProfiler [36] or NetBeans Profiler [33], generate calling context profiles, but are usually visualized as expandable trees. In such a representation, calling contexts are sorted by their contribution to an aggregated metric (usually execution time) and can be expanded or collapsed in order to show or hide callees. While such a view is simple and straightforward to use, CCRCs convey more profiling information in a single view and give a better understanding of caller/callee relationships. Thanks to subtree selection and depth limitation, CCRCs efficiently support navigation in any interesting part of the CCT.

## 9. Conclusion

Calling context profiling is an important technique for exploring the dynamic behavior of programs. However, the resulting profiles, usually represented as CCTs, can be huge, comprising up to millions of calling contexts and typically reaching tree depths of 50–400 layers. Such large profiles can hardly be analyzed in a textual representation, and prevailing visualizations, such as expandable trees used in several profiling tools, are too verbose and do not make good use of space.

To ease handling large calling context profiles, this article presented Calling Context Ring Charts (CCRCs). A CCRC visualizes a CCT in a compact way while representing all caller/callee relationships. Each calling context is displayed as a ring segment, surrounded by the ring segments representing the callees. Ring segments can be sized according to a chosen dynamic metric in order to ease the localization of hotspots. Several CCT tree manipulation operations, such as selecting subtrees or limiting the tree depth, complement CCRC. We presented two case studies where CCRCs were successfully applied to locate and fix performance problems in applications. A performance evaluation confirms that our CCRC implementation easily handles very large calling context profiles, and that the rendering time is quite fast.

Regarding ongoing research, we are working on representing two different metrics within the same CCRC using a coloring scheme. We are also conducting a study on the productivity increase thanks to CCRC visualizations when profiling complex applications. To this end, collaboration partners have started using our tools. Furthermore, we are working on an advanced set of tree manipulation and metrics aggregation operations that are exposed to the user through a domain-specific, declarative language.

### Acknowledgment

We thank Ricardo Padilha and Fernando Pedone for using our platform-independent calling context profiler and CCRC visualization tool, in order to optimize their Java implementation of Shamir's Secret Sharing algorithm, and for providing us the results of their analysis and optimizations.

### REFERENCES

1. G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96. ACM Press, 1997.
2. K. Andrews and H. Heidegger. Information slices: Visualising and exploring large hierarchies using cascading, semi-circular discs. In *InfoVis98: IEEE Symposium on Information Visualization*. IEEE Computer Society, October 1998.
3. M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. Research report, IBM, July 2000.
4. S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making Software Failures Reproducible by Preserving Object States. In J. Vitek, editor, *ECOOP '08: Proceedings of the 22th European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 542–565, Paphos, Cyprus, 2008. Springer-Verlag.
5. M. Balzer, O. Deussen, and C. Lewerentz. Voronoi treemaps for the visualization of software metrics. In *SofiVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 165–172. ACM, 2005.
6. T. Barlow and P. Neville. A comparison of 2-D visualizations of hierarchies. In *INFOVIS '01: Proceedings of the IEEE Symposium on Information Visualization 2001*, page 131, Washington, DC, USA, 2001. IEEE Computer Society.
7. W. Binder, J. Hulaas, P. Moret, and A. Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 39(1):47–79, 2009.
8. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press.
9. M. D. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming, systems and applications*, pages 97–112, New York, NY, USA, 2007. ACM.
10. A. Chakrabarti and P. Godefroid. Software partitioning for effective automated unit testing. In *EMSOFT '06: Proceedings of the 6th ACM IEEE International conference on Embedded software*, pages 262–271, New York, NY, USA, 2006. ACM.
11. P. P. Chang, S. A. Mahlke, and W. mei W. Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 21(12):1301–1321, 1991.
12. B. F. Cooper, H. B. Lee, and B. G. Zorn. ProfBuilder: A package for rapidly building Java execution profilers. Technical Report CU-CS-853-98, University of Colorado at Boulder, Department of Computer Science, Apr. 1998.
13. J. R. Dabrowski and E. V. Munson. Is 100 milliseconds too fast? In *CHI '01: CHI '01 Extended Abstracts on Human Factors in Computing Systems*, pages 317–318, New York, NY, USA, 2001. ACM.
14. R. Dachsel and J. Ebert. Collapsible cylindrical trees: A fast hierarchical navigation technique. In *IEEE Symposium on Information Visualization*, page 79, Los Alamitos, CA, USA, 2001. IEEE Computer Society.
15. P. Deelen, F. van Ham, C. Huizing, and H. van de Watering. Visualization of dynamic program aspects. In *VISSOFT 2007: 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 39–46, June 2007.

16. G. M. Draper, Y. Livnat, and R. F. Riesenfeld. A survey of radial methods for information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15:759–776, 2009.
17. Elliot Berk. JLex: A Lexical Analyzer Generator for Java. Web pages at <http://www.cs.princeton.edu/~appel/modern/java/JLex/>, 2003.
18. J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black-box components. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 101–111, New York, NY, USA, 2007. ACM.
19. I. Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6:24–43, 2000.
20. A. Heydarnoori, K. Czarnecki, and T. T. Bartolomei. Supporting framework use via automatically extracted concept-implementation templates. In *ECOOP '09: Proceedings of the 23rd European Conference on Object-Oriented Programming*, July 2009.
21. D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, Sept.–Oct. 2006.
22. D. Holten, B. Cornelissen, and J. van Wijk. Trace visualization using hierarchical edge bundles and massive sequence views. In *VISSOFT 2007: 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 47–54, June 2007.
23. C.-S. Jeong and A. Pang. Reconfigurable disc trees for visualizing large hierarchical information space. In *INFOVIS '98: Proceedings of the 1998 IEEE Symposium on Information Visualization*, pages 19–25, Washington, DC, USA, 1998. IEEE Computer Society.
24. B. Johnson and B. Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 284–291. IEEE Computer Society Press, 1991.
25. JSharing. Shamir's secret sharing scheme library in Java. Web pages at <http://code.google.com/p/jsharing/>.
26. E. Kleiberg, H. van de Wetering, and J. J. V. Wijk. Botanical visualization of huge hierarchies. In *INFOVIS '01: Proceedings of the IEEE Symposium on Information Visualization 2001*, page 87, Washington, DC, USA, 2001. IEEE Computer Society Press.
27. J. Lamping and R. Rao. The hyperbolic browser: A focus+context technique for visualizing large hierarchies. *Journal of Visual Languages and Computing*, 7(1):33–55, 1996.
28. Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering Through Context-Aware Monitored Execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.
29. P. Moret, W. Binder, D. Ansaloni, and A. Villazón. Visualizing Calling Context Profiles with Ring Charts. In *VISSOFT 2009: 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 33–36, Edmonton, Alberta, Canada, Sept. 2009. IEEE Computer Society.
30. P. Moret, W. Binder, D. Ansaloni, and A. Villazón. Exploring Large Profiles with Calling Context Ring Charts. In *WOSP/SIPEW 2010: Proceedings of the First Joint International Conference on Performance Engineering*, pages 63–68. ACM Press, Jan. 2010.
31. P. Moret, W. Binder, and A. Villazón. CCCP: Complete calling context profiling in virtual execution environments. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 151–160, Savannah, GA, USA, 2009. ACM.
32. T. Munzner. Drawing large graphs with h3viewer and site manager. In *GD '98: Proceedings of the 6th International Symposium on Graph Drawing*, pages 384–393, London, UK, 1998. Springer-Verlag.
33. NetBeans. The NetBeans Profiler Project. Web pages at <http://profiler.netbeans.org/>.
34. N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI 2007: Proceedings of the ACM SIGPLAN 2007 conference on Programming Language Design and Implementation*, pages 89–100. ACM Press, 2007.
35. B. Poettering. SSSS: Shamir's Secret Sharing Scheme. Web pages at <http://point-at-infinity.org/ssss/>.
36. ProSyst. JProfiler. Web pages at [http://www.prosyst.com/products/tools\\_jprofiler.html](http://www.prosyst.com/products/tools_jprofiler.html).
37. G. Robertson, J. Mackinlay, and S. Card. Cone trees: Animated 3D visualization of hierarchical information. In *CHI '91: Computer-Human Interaction*, pages 189–194, 1991.
38. D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting Static Source Views in IDEs with Dynamic Metrics. In *ICSM '09: Proceedings of the 2009 IEEE International Conference on Software Maintenance*, pages 253–262, Edmonton, Alberta, Canada, 2009. IEEE Computer Society.
39. A. Rountev, S. Kagan, and J. Sawin. Coverage criteria for testing of object interactions in sequence diagrams. In *FASE '05: Fundamental Approaches to Software Engineering*, volume 3442 of *Lecture Notes in Computer Science (LNCS)*, pages 282–297, April 2005.

40. A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
41. B. Shneiderman and M. Wattenberg. Ordered treemap layouts. In *INFOVIS 2001: IEEE Symposium on Information Visualization*, pages 73–78, 2001.
42. J. M. Spivey. Fast, accurate call graph profiling. *Software Practice and Experience*, 34(3):249–264, 2004.
43. J. Stasko, R. Catrambone, M. Guzdial, and K. McDonald. An evaluation of space-filling information visualizations for depicting hierarchical structures. *International Journal of Human-Computer Studies*, 53(5):663–694, 2000.
44. J. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *INFOVIS '00: Proceedings of the IEEE Symposium on Information Visualization 2000*, page 57, Washington, DC, USA, 2000. IEEE Computer Society.
45. M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using SHriMP views. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 275, Washington, DC, USA, 1995. IEEE Computer Society.
46. I. G. Tollis, G. D. Battista, P. Eades, and R. Tamassia. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, July 1998.
47. R. Vliegen, J. J. van Wijk, and E.-J. van der Linden. Visualizing business data with generalized treemaps. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):789–796, 2006.
48. J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. In *ICCS 2004: 4th International Conference on Computational Science*, volume 3038 of *LNCS*, pages 440–447. Springer, 2004.
49. 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, June 2000.
50. X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 263–271, New York, NY, USA, 2006. ACM.