Exploring Large Profiles with Calling Context Ring Charts

Philippe Moret
Faculty of Informatics
University of Lugano
Switzerland
philippe.moret@usi.ch

Walter Binder
Faculty of Informatics
University of Lugano
Switzerland
walter.binder@usi.ch

Alex Villazón
Faculty of Informatics
University of Lugano
Switzerland
alex.villazon@usi.ch

Danilo Ansaloni
Faculty of Informatics
University of Lugano
Switzerland
danilo.ansaloni@usi.ch

ABSTRACT

Calling context profiling is an important technique for analyzing the performance of object-oriented software with complex inter-procedural control flow. A common data structure is the Calling Context Tree (CCT), which stores dynamic metrics, such as CPU time, separately for each calling context. As CCTs may comprise millions of nodes, there is need for a condensed visualization that eases the location of performance bottlenecks. In this paper, 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 a case study where CCRCs help detect and fix performance problems in an application. An evaluation confirms that our implementation efficiently handles large CCTs with millions of nodes.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement Techniques*; I.3.8 [**Computing Methodologies**]: Computer Graphics—*Applications*

General Terms

Measurement, Performance

Keywords

Performance analysis, visualization, calling context profiles, Calling Context Tree (CCT), dynamic metrics

1. INTRODUCTION

Calling context profiling is a common technique to explore the dynamic behavior of programs and to find the reasons for performance problems. Calling context profiling 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 helps analyze the dynamic inter-procedural control flow of applications. This technique is particularly important for understanding and optimizing object-oriented software, where polymorphism and dynamic binding often hinder static analyzes. Typically, object-oriented applications make use of many short methods such that the inter-procedural control flow can become very complex.

The Calling Context Tree (CCT) [1] is a prevailing datastructure 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. There is a large body of work dealing with different techniques to generate CCTs [1, 7, 2, 4], highlighting the importance of the CCT for calling context profiling.

CCTs are often huge trees, sometimes comprising millions of nodes. Furthermore, the depth of CCTs can be high; CCTs with 50–400 levels are common in practice. Hence, there is need for a compact representation of CCTs helping the developer analyze dynamic program behavior. Prevailing tools supporting calling context profiling typically present the CCT as an expandable tree. However, exploring large and deep CCTs, such as for locating invocations of a particular method in various calling contexts, is cumbersome with an expandable tree representation.

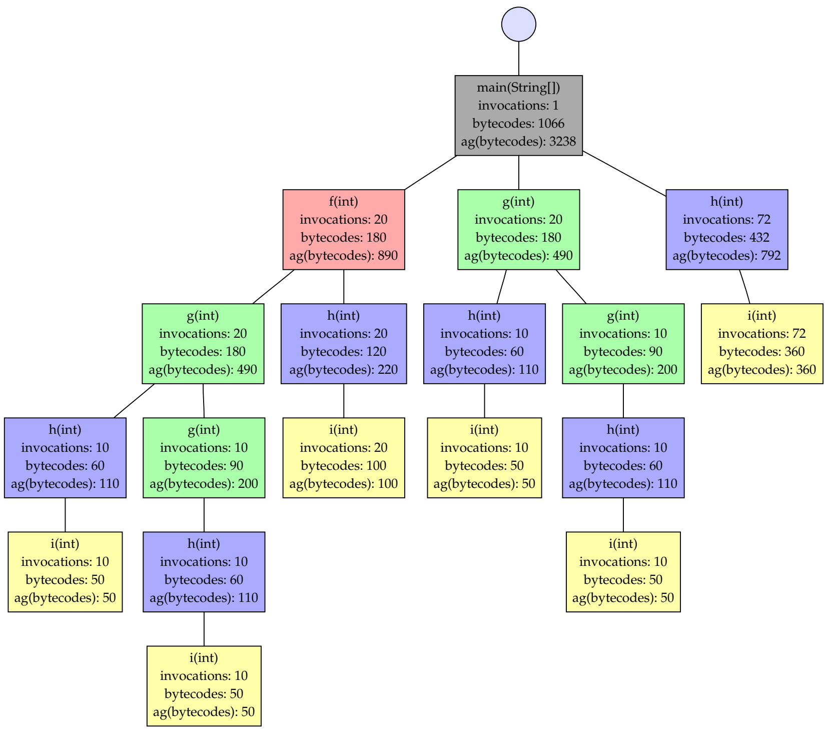
In this paper, we discuss *Calling Context Ring Charts (CCRCs)* [3], a way of visualizing and analyzing CCTs. In a CCRC, the CCT root is represented as a circle in the center. Callee methods are represented by ring segments surrounding the caller's ring segment. With CCRCs it is possible to display all 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 [3].

In this paper we present two CCRC visualizations; for more details see [3]. 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 the 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 cho-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WOSP/SIPEW'10, January 28–30, 2010, San Jose, California, USA.
Copyright 2009 ACM 978-1-60558-563-5/10/01 ...$10.00.

```
 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



main(String[]) invocations: 1 bytecodes: 1066 ag(bytecodes): 3238
f(int) invocations: 20 bytecodes: 180 ag(bytecodes): 890
g(int) invocations: 20 bytecodes: 180 ag(bytecodes): 490
h(int) invocations: 72 bytecodes: 432 ag(bytecodes): 792
g(int) invocations: 20 bytecodes: 180 ag(bytecodes): 490
h(int) invocations: 20 bytecodes: 120 ag(bytecodes): 220
h(int) invocations: 10 bytecodes: 60 ag(bytecodes): 110
g(int) invocations: 10 bytecodes: 90 ag(bytecodes): 200
i(int) invocations: 72 bytecodes: 360 ag(bytecodes): 360
h(int) invocations: 10 bytecodes: 60 ag(bytecodes): 110
g(int) invocations: 10 bytecodes: 90 ag(bytecodes): 200
i(int) invocations: 20 bytecodes: 100 ag(bytecodes): 100
i(int) invocations: 10 bytecodes: 50 ag(bytecodes): 50
h(int) invocations: 10 bytecodes: 60 ag(bytecodes): 110
i(int) invocations: 10 bytecodes: 50 ag(bytecodes): 50
h(int) invocations: 10 bytecodes: 60 ag(bytecodes): 110
i(int) invocations: 10 bytecodes: 50 ag(bytecodes): 50
i(int) invocations: 10 bytecodes: 50 ag(bytecodes): 50

(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 [3].

sen 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.

The original, scientific contributions of this paper are twofold.

1. We present one case study where CCRCs are successfully applied to locate performance problems in an application, which is optimized afterwards. The chosen application is a cryptography library.
2. We present our implementation of CCRC and evaluate its performance on visualizing the CCTs resulting from the execution of the DaCapo benchmarks for Java.

2. BACKGROUND: THE CALLING CONTEXT TREE (CCT)

The CCT was first introduced by Ammons et al. [1] as runtime data structure for calling context profiling.

Each node in the CCT represents 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.

The CCT does not restrict which 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. Our CCRC implementation does not constraint which and how many different metrics are stored in a CCT node. All stored metrics must be represented by numeric values.

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 (i.e., including the costs incurred by all direct and indirect callees).

For each CCT node n, $ag(M_n)$ is computed as the sum of the metric values M_x of all nodes x in the CCT subtree rooted at node n.

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 [2, 4]. 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.

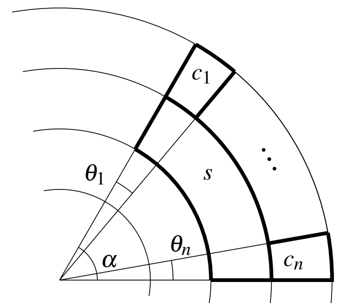


Figure 2: Representation of a CCT node as a ring segment *s*

3. CCT VISUALIZATION

In this section we describe two ways of visualizing CCTs as Calling Context Ring Charts, as introduced in [3]. Both visualizations use an *onion-like* structure with circular layers, each layer corresponding to a level in the CCT. Nodes are 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 $c_1, \ldots, c_n$. In this first visualization, given α the angle covered by s, the angle θ_i covered by its ith child c_i is computed as follow:

$$\theta_i = \alpha \frac{1}{n} \tag{1}$$

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.

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

In order to ease locating performance problems, we support a second visualization, where each ring segment is sized proportionally to an aggregated metric $ag(M)$ chosen by the user. In this visualization, θ_i is computed with the following equation:

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

Using equation (2), the sum of all θ_i can be less than α; 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(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 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.

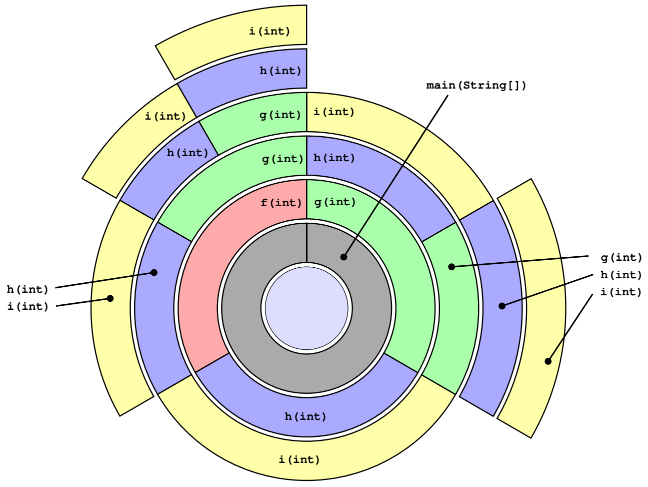


Figure 3: First CCT visualization: Callees are represented by ring segments of equal length

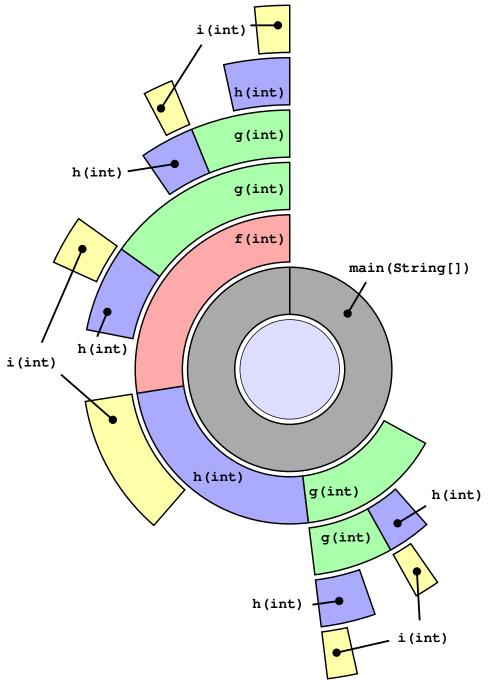


Figure 4: Second CCT visualization: Ring segment *length* is proportional to the aggregated bytecode metric

4. 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)[1]

format, whereas the second version is a Java application using the Swing toolkit. The third version is also implemented in Java using Eclipse's Standard Widget Toolkit (SWT)[2].

[1]`http://www.w3.org/Graphics/SVG/`
[2]`http://www.eclipse.org/swt/`

Our first implementation uses JavaScript to create and dynamically transform the CCT, which is represented as an XML structure. This implementation allows visualizing CCRCs directly in any standard web browser. 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 pointed 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 the one hand, this implementation is well suited for integration in 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 comprising millions of nodes. Additional features were added, such as an advanced search to find and 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 based on 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 [5], 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, an estimation of total memory consumption, etc. 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, respectively showing the method source code of a selected calling context in the CCRC.

5. CASE STUDY

In this section we present a study where CCRCs have been successfully applied to locate performance problem in an application, which was optimized afterwards.

In our case study, the calling context profiles are collected with the profiler described in [2, 4], 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 to profile overall application execution, including the execution of methods in the Java class library.

For CCRC visualization, we use our Java-based Swing implementation. For the CCRCs in this case study, we use our second visualization, 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 [3] 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 levels [3].

The primary contribution of this case study is to show that CCRCs help quickly locate hotspots in calling context profiles. As

```
PUT: input: key value pair $< k, v >$
  • calculate $T$ unique participant share keys $pk_j$ by hashing $k$ with the participant's $id$
  • split $v$ by calculating $T$ points $p_j$ from $f(x)$
  • return $T$ key/value pairs $< pk_j, p_j >$
GET: input: key $k$
  • calculate $T$ unique participant share keys $pk_j$ by hashing $k$ with the participant's $id$
  • join $T$ shares from participants using the $pk_j$ keys into $v$
  • return $v$
```

Figure 5: Algorithms used to split the secret into T shares and to reconstruct the secret

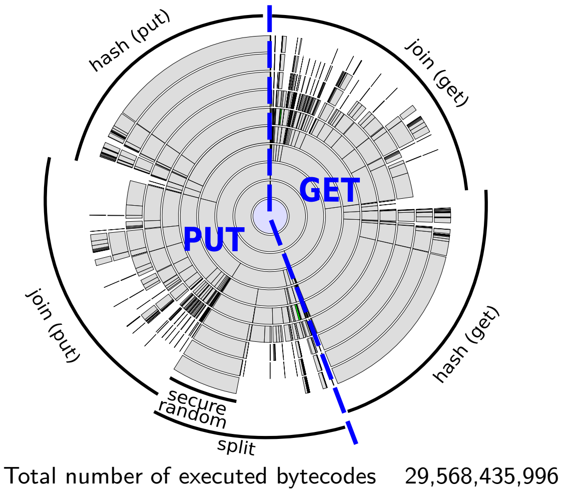


Figure 6: CCRC for the initial implementation of SSS

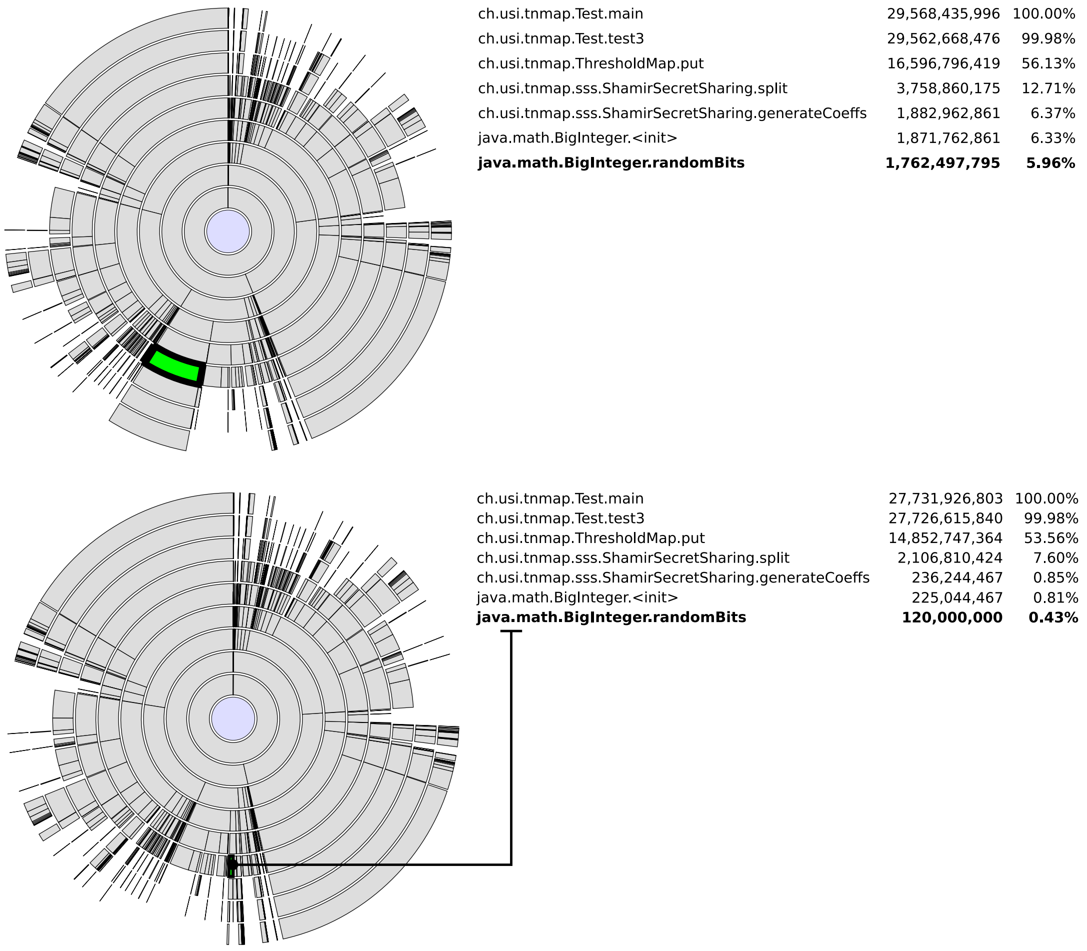
secondary contribution, the case study also demonstrate that profiling using a platform-independent metric (i.e., the number of executed bytecodes) – instead of the common CPU time metric – is a useful technique for detecting hotspots and for optimizing applications. The resulting optimized application not only executes less bytecodes, but it also performs significantly better in terms of execution time.

Shamir's Secret Sharing (SSS) [6] is a cryptography algorithm, where a secret is divided into parts. Each participant receives its own unique part, where some of the parts or all of them are needed in order to reconstruct the secret.

SSS is based on the idea that a polynomial's coefficient and points generated by the polynomial are interchangeable. A polynomial function f is constructed (see equation (3)), such that the constant term S is the secret to share, and T is the minimum number of *shares* required to reassemble the secret. K_i are randomly generated numbers. The shares given to the participants are tuples $< x, f(x) >$, that is, points in the polynomial curve.

$$f(x) = S + \sum_{i=1}^{T-1} K_i x^i \qquad (3)$$

In our case study, we analyze an SSS implementation developed by Ricardo Padilha at the University of Lugano, which handles S of arbitrary size (whereas most available implementations limit S



ch.usi.tnmap.Test.main	29,568,435,996	100.00%
ch.usi.tnmap.Test.test3	29,562,668,476	99.98%
ch.usi.tnmap.ThresholdMap.put	16,596,796,419	56.13%
ch.usi.tnmap.sss.ShamirSecretSharing.split	3,758,860,175	12.71%
ch.usi.tnmap.sss.ShamirSecretSharing.generateCoeffs	1,882,962,861	6.37%
java.math.BigInteger.<init>	1,871,762,861	6.33%
java.math.BigInteger.randomBits	**1,762,497,795**	**5.96%**

ch.usi.tnmap.Test.main	27,731,926,803	100.00%
ch.usi.tnmap.Test.test3	27,726,615,840	99.98%
ch.usi.tnmap.ThresholdMap.put	14,852,747,364	53.56%
ch.usi.tnmap.sss.ShamirSecretSharing.split	2,106,810,424	7.60%
ch.usi.tnmap.sss.ShamirSecretSharing.generateCoeffs	236,244,467	0.85%
java.math.BigInteger.<init>	225,044,467	0.81%
java.math.BigInteger.randomBits	**120,000,000**	**0.43%**

Figure 7: SSS using `java.security.SecureRandom` (top) versus SSS using `java.util.Random` (bottom)

to 128 bit). We let the developer analyze his implementation with CCRCs, in order to validate the soundness of his design choices and to optimize his code.

To generate the K_i random values, the original implementation uses the `java.security.SecureRandom` class. It associates a specific key value k to each secret S. Each share distributed to the participants is also associated with a key pk_j, calculated with the SHA-1 hashing algorithm, by hashing k with the participant's *id*. This feature aims at obfuscating the relation between the key and the value, which is useful, for instance, when storing the shares in an insecure repository. The algorithms to *"put"* respectively *"get"* the secret are outlined in Figure 5.

Figure 6 shows a CCRC corresponding to one hundred thousand put and get operations. It illustrates the bytecode distribution of the different computations. We observe a symmetry between the calculations of the hash value and the join to put, respectively to get the shares. Thanks to our visualization, this symmetry becomes immediately apparent. The reader may wonder why a join operation is also needed for the put operation. This is due to the API used to store the shares. When replacing a stored value, the old value must be returned, forcing the join to be calculated also upon the put operation.

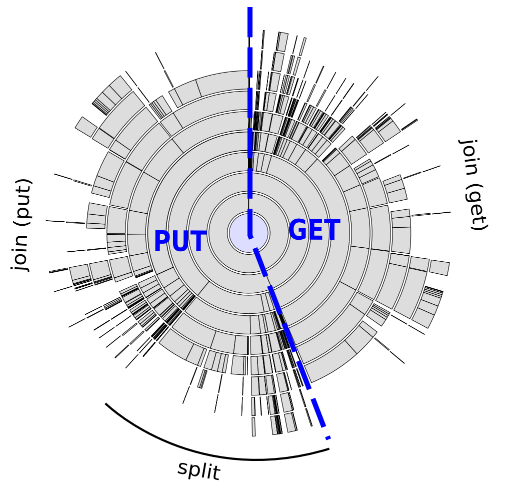
First, the developer observed that the execution of the split operation was dominated by the use of `java.security.SecureRandom`. As an optimization, the developer decided to use the non-secure class `java.util.Random` instead. Figure 7 shows the CCRCs before and after the change. The highlighted segments in Figure 7 (top and bottom) correspond to `BigInteger.randomBits()`, which is the common code executed when using `SecureRandom` respectively `Random`. Thanks to this optimization, the calculation of the random coefficients 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 for the obfuscation feature (see Figure 6). He decided to reengineer SSS in order to make this feature optional, since it was not required in common use cases. Figure 8 shows the CCRC for the final SSS version. Compared to the original implementation (see Figure 6), 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), with JDK 1.6_11 running on an Intel Core 2 Duo 2.33Ghz computer with 2GB RAM (Linux Fedora 10). 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%).

6. EVALUATION

In order to validate our CCRC implementation with large profiles, we generated and visualized the CCTs for the standard Da-



Total number of executed bytecodes 15,129,000,509

Figure 8: CCRC of the final implementation of SSS (without hashing and without secure random)

DaCapo Bench.	Method Calls	Different Methods	CCT Nodes	Max. Depth	Time [ms]
antlr	165,465,913	2,498	627,577	164	41
bloat	1,055,927,864	3,033	645,316	128	42
chart	404,140,995	4,554	92,272	65	81
eclipse	941,528,217	11,555	2,166,169	131	58
fop	43,143,714	4,013	149,492	76	111
hsqldb	214,764,027	2,507	66,391	62	61
jython	945,580,102	4,574	1,941,246	223	46
luindex	443,252,127	1,969	58,834	52	51
lusearch	502,521,414	1,768	33,189	46	63
pmd	462,558,862	3,663	800,071	416	44
xalan	636,834,930	3,679	211,213	79	42

Table 1: CCT statistics and CCRC rendering time

Capo benchmark suite.We computed the rendering time for the full CCTs using our Java Swing implementation of CCRC. The measurements correspond to the rendering of the CCTs generated by the execution of one run of the benchmarks with the default problem size. We use the same measurement environment as in the case study. The CCTs were generated beforehand using the same calling context profiler used in our case study [2, 4].

Table 1 presents the CCRC rendering time for the benchmark CCTs, as well as some statistics computed from the CCTs.

Column "Method Calls" shows the total number of method invocations (in all CCT nodes), which can be very high (over 1 billion for the 'bloat' benchmark). Column "Different Methods" gives the number of different methods that were invoked at least once. The number of CCT nodes (column "CCT Nodes") and the maximum depth (column "Max Depth") give an idea of the size of the CCTs to be visualized. For 'eclipse', the CCT consists of more than 2 million nodes, and for 'pmd' the maximum tree depth exceeds 400 layers.

Table 1 shows the time for rendering the CCRCs representing the CCTs (using our second visualization where the length of ring segments is proportional to the aggregated bytecode metric), which is between 41ms and 111ms. Interestingly, the rendering time is not directly proportional to the number of nodes in the CCT because of various optimizations, such as rendering very small ring segments as single lines. Therefore, rendering is perceived almost as instantaneous by the user, even for very large CCTs.

7. 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.

In order to ease handling large calling context profiles, we presented Calling Context Ring Charts (CCRCs). A CCRC visualizes a CCT in a compact way, correctly 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 location of hotspots.

We presented a case study where CCRCs were successfully applied to locate and fix performance problems in an application. An evaluation confirms that our CCRC implementation easily handles very large calling context profiles.

Acknowledgements: The work presented in this paper has been supported by the Swiss National Science Foundation.

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.

8. 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] 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. `http://dx.doi.org/10.1002/spe.890`.

[3] 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, Sep. 2009. IEEE Computer Society.

[4] 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.

[5] 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.

[6] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[7] 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.