Inti: Tracking Performance Issue using a Compact and Effective Visualization

Milton Mamani[1], Alejandro Infante[2], Alexandre Bergel[2]

[1]Object Profile, Chile

[2]Pleiad Lab, Department of Computer Science (DCC), University of Chile

This paper makes use of colored figures. Though colors are not mandatory for full understanding, we recommend the use of a colored printout.

***Abstract*—Current tools to measure software performance commonly use a tree widget to indicate the CPU time distribution over the execution control flow. The tree representation is known to poorly scale in presence of large dataset and inadequately convey the time distribution across software components.**

We propose Inti, a sunburst-like visualization, to represent program executions. Inti uses color maps to indicate time distribution and comparison across set of software components. Visualizations produced by Inti are both compact and interactive. This paper describes the early development stage of Inti.

I. Code Profiler

Code profilers are used to identify software execution bottlenecks and understand the cause of a slowdown. Most programming environments come with powerful code execution profilers. Execution sampling is a monitoring technique commonly employed by code profilers because of its low impact on execution. Execution sampling essentially estimates (i) the time spent in each function / method and (ii) the control flow of the application.

Profiles as trees. A great effort has been made by the software performance community to make profilers more accurate (*i.e.,* reducing the gap between the actual application execution and the profiler report). Advanced techniques have been proposed such as variable sampling time [1] and proxies for time execution [2], [3]. However, much less attention has been paid to the visual output of a profile. Consider JProfiler[1] and YourKit[2], two popular code profilers for the Java programming language. Both of them output the profile of an execution using a tree widget. To illustrate that point, consider the following Java code:

```
class A {
  public void init() { ... }
  public void run() { this.utility(); ... }
  public void utility() { ... }
}
class C {
  public void run() {
    A a = new A();
    a.init();
    a.run();
  }
  public static void main(String[] argv) {
    new C().run(); } }
```

[1]http://www.ej-technologies.com/products/jprofiler/overview.html

[2]http://www.yourkit.com

Profiling the execution of the main method will result in the following tree:

```
C.main (100%)
  C.run (100%)
    A.init (60%)
    A.run (40%)
      A.utility(40%)
```

The tree represents the call graph triggered by the execution of the main. Indentation indicates the relation between the called and calling methods. The CPU share allocated to `init`, `run`, and `utility` depends on their definitions (the value 60% and 40% are for illustrative purpose).

Most code profilers output their profile in a similar fashion. Some profilers use a textual output, other use an expandable tree widget. All in all, a tree is the favorite visual representation for code profilers.

Limitation of tree representations. Representing call-graph and CPU share distribution faces many limitations. First, in presence of a large callgraph, the user has to scroll over the profile and use textual searching facilities to navigate in the tree. Extracting relevant data from large sequential textual listing is not trivial. Secondly, by representing the CPU consumption share over the call graph, high-level information are not apparent and remain difficult to extract. For example, understanding which component is the culprit of excessive CPU consumption requires the software engineer to search in the tree all the methods belonging to the component, and manually summing up their consumption. Inti uses visual cues to identify hot regions.

Need for alternative representation. Many effort have been made to provide an alternative profile representation [4], [5], [6]. In particular, it has been shown that context ring charts are effective at representing hot methods [7]. Our work is inspired by context ring chart.

II. Inti Visualization

The preliminary work described in this section combine visual attributes and cues to identify how CPU time consumption is distributed over a call graph. *Inti* is a *sunburst*-like visualization dedicated to visualize CPU time distribution.

A. Inti visualization

Inti is a visual representation of hierarchical data. The contrived example given above is visualized with Inti as shown in Figure 1.

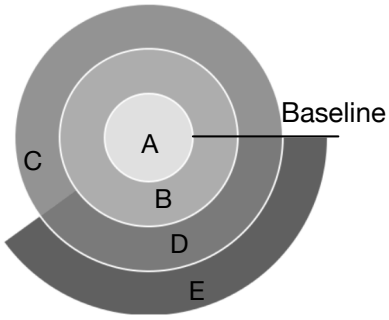

Fig. 1: Example of Inti

Each method of the Java code given above is represented by an arc in Figure 1. Method `C.main` by the disk `A`, `C.run` by arc `B`, `A.init` by `C`, `A.run` by `D` and `A.utility` by `E`. The baseline represents the starting time of the profile. The angle of each arc represents the time distribution taken by the method. In this example, `C.main` and `C.run` have an angle of 360 degrees, meaning these two methods consume 100% of the CPU time. Methods `A.init` consumes 60% and `A.run` 40%.

Each method frame is presented as an arc. Distance between an arc and the center of the visualization (where the `A` label is located) indicates the depth of the frame in the method call stack. A nested method call is represented as a stacked arc.

Benefices of Inti are numerous. Inti is very compact. A typical execution, like the one presented in Figure 2, have 1916 nodes. To represent this tree, the classical tools would use 1916 lines, one for each node. This number of lines at a reasonable font size would need to be split into several pages and is largely outperformed by Inti. Inti shows larger part of the control flow and CPU time distribution in less space.

Details about each stack frame is accessible via tooltip. Hovering the mouse over an arc triggers a popup window that indicates the CPU time consumption and the method source code.

B. Navigation and coloring map

A "visual cue" is a visual signal that is self-explanatory and pre-attentive. A visual cue typically brings to the mind a common knowledge or experience. Inti exploit visual cues in many different ways. The default Inti visualization use the red color to indicate relatively high CPU consumption.

Figure 2 is an example of a complex application. Each arc has a color ranging from gray to red. The center of the Inti visualization is red, indicating a 100% CPU consumption. This is no surprise since the center corresponds to the `main` method, the starting point of the program execution.

The very irregular border of the visualization indicates depth of the method call stack, which greatly varies.

Because of the limited space, we decided to require the user to interact with the visualization to obtain some relevant information, like the method name. For example, to get the name of a method represented by an arc, the user must locate the mouse over that arc. There are other valuable interactions such as method highlighting or browse the source code of a method by clicking on the arcs.

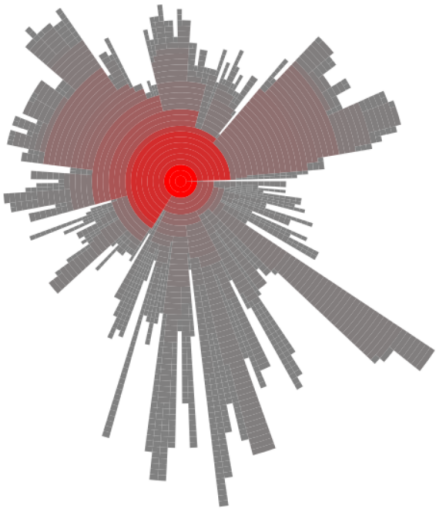

Fig. 2: Inti default visualization

Also a number of navigation options use colors to indicates the CPU time consumption across methods, classes, and packages. For a given arc, other arc may be painted according to if they belong to the same class, same package or correspond to different implementations.

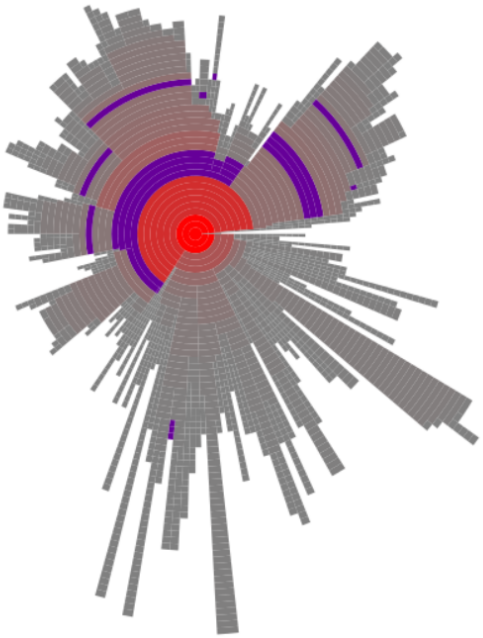

Fig. 3: Distribution of consumption across different methods

Inti provides an easy-to-use interaction for painting the elements of the visualization. It is possible to use this technique to paint all the methods related to a component, and therefore, providing a visual help to identify the performance problems of related pieces of code. An example of this feature is Figure 3.

It is frequent to measure software performance after having some modification. Supporting comparison of profile is therefore important and deserve to be carefully handled [8]. Inti addresses this by supporting a visualization to quickly

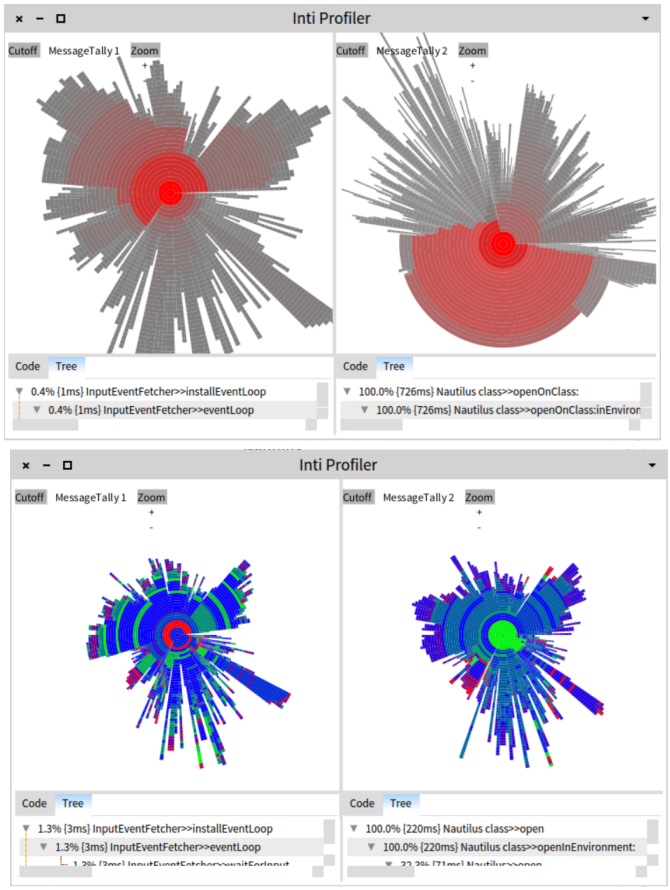

Fig. 4: Differences of profiles

identify application performance variations, while preserving a representation of the overall results. This has been done to avoid the user to invest time on minor performance changes that are insignificant compared to the application as a whole, as shown in Figure 4.

III. Conclusion and Future work

Properly addressing performance is known to be complex and poorly supported by traditional programming environments. Inti, as described in this paper, is an initial effort to representation of execution data that allow the users to extract the relevant information faster and more accurately. In order to do this, Inti makes use of a sophisticated visual representation of a program execution.

Our hypothesis is that the visual representation conveyed by Inti scales better than the classical tree representation of classical code profilers. We also expect the color maps offered by Inti to significantly play a role in efficiently identifying execution bottlenecks. We will verify this hypothesis as future work.

Acknowledgments. We deeply grateful to Lam Research for their encouragements and financial support. This work is partially funded by FONDECYT Project 1120094.

References

[1] T. Mytkowicz, A. Diwan, M. Hauswirth, P. F. Sweeney, Evaluating the accuracy of java profilers, in: Proceedings of the 31st conference on Programming language design and implementation, PLDI '10, ACM, New York, NY, USA, 2010, pp. 187–197. doi:10.1145/1806596.1806618.
URL http://doi.acm.org/10.1145/1806596.1806618

[2] A. Camesi, J. Hulaas, W. Binder, Continuous bytecode instruction counting for cpu consumption estimation, in: Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems, IEEE Computer Society, Washington, DC, USA, 2006, pp. 19–30. doi:10.1109/QEST.2006.12.
URL http://portal.acm.org/citation.cfm?id=1173695.1173954

[3] A. Bergel, Counting messages as a proxy for average execution time in pharo, in: Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11), LNCS, Springer-Verlag, 2011, pp. 533–557.
URL http://bergel.eu/download/papers/Berg11c-compteur.pdf

[4] D. Holten, B. Cornelissen, J. J. van Wijk, Trace visualization using hierarchical edge bundles and massive sequence views, in: Proceedings of Visualizing Software for Understanding and Analysis, 2007 (VISSOFT'07), IEEE Computer Society, 2007, pp. 47 – 54. doi:10.1109/VISSOF.2007.4290699.

[5] D. J. Jerding, J. T. Stasko, T. Ball, Visualizing interactions in program executions, in: Proceedings of International Conference on Software Engineering (ICSE'97), 1997, pp. 360–370.

[6] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, J. J. van Wijk, Execution trace analysis through massive sequence and circular bundle views, J. Syst. Softw. 81 (2008) 2252–2268. doi:10.1016/j.jss.2008.02.068.
URL http://dl.acm.org/citation.cfm?id=1454787.1454981

[7] P. Moret, W. Binder, A. Villazón, D. Ansaloni, A. Heydarnoori, Visualizing and exploring profiles with calling context ring charts, Softw. Pract. Exper. 40 (9) (2010) 825–847. doi:10.1002/spe.v40:9.
URL http://dx.doi.org/10.1002/spe.v40:9

[8] J. P. S. Alcocer, A. Bergel, S. Ducasse, M. Denker, Performance evolution blueprint: Understanding the impact of software evolution on performance, in: A. Telea, A. Kerren, A. Marcus (Eds.), VISSOFT, IEEE, 2013, pp. 1–9.