An interactive visualization framework for performance analysis

Emilio Coppa
Dept. of Computer Science
Sapienza University of Rome
coppa@di.uniroma1.it

ABSTRACT
Input-sensitive profiling is a recent methodology for analyzing how the performance of a routine scales as a function of the workload size. As increasingly more detailed profiles are collected by an input-sensitive profiler, the information conveyed to a user can quickly become overwhelming. In this paper, we present an interactive graphical tool called aprof-plot for visualizing performance profiles. Exploiting curve fitting techniques, aprof-plot can estimate the asymptotic complexity of each routine, pointing the attention of the programmer to the most critical routines of an application. Several examples based on real-world applications are discussed, showing how to conduct an effective performance investigation using aprof-plot.

Categories and Subject Descriptors
C.4 [Performance of Systems]: Measurement Techniques; H.5.2 [User Interfaces]: Graphical user interfaces (GUI)

General Terms
Performance, Measurement, Visualization.

1. INTRODUCTION
Performance optimization is a critical step during software development. Programmers use tools to understand the application runtime behavior and to spot performance bottlenecks. Traditional profilers help developers find out how specific portions of code are responsible for resource consumption, such as memory space and CPU time. Some recent works have made a step further by addressing the problem of designing and implementing performance profilers that return, instead of a single number representing the cost of a portion of code, a function that relates the cost to the input size (see, e.g., [4, 9, 21]). These approaches have been inspired by traditional asymptotic analysis of algorithms, and make it possible to analyze – and sometimes predict – the behavior of actual software implementations run on deployed systems and realistic workloads.

This paper is based on the input-sensitive profiling methodology described in [4]: this approach is able to automatically measure how the performance of individual routines scales as a function of the input size, yielding clues to their growth rate. From one or more runs of a program, an input-sensitive profiler collects several performance measurements related to the runtime behavior of an application and of its routines. These profile data are stored as text-based files and can quickly become very large as increasingly more detailed profiles are collected, making it really hard for a developer to take benefit of this valuable information. To overcome this problem, in this paper we present an interactive graphical tool, built on top of the NetBeans IDE platform, which is able to automatically estimate the asymptotic complexity of each routine, pointing the attention of the programmer to the most critical routines of an application. Noise reduction techniques and other useful features support the user towards a more effective performance investigation.

The tool is available at http://code.google.com/p/aprof/.

Related work. Performance profiling has been the subject of extensive research since the early 70’s. For many years, analysis of profile data has been performed using command line tools [10], which provides limited user interaction. Today, the importance of visualization tools for evaluating program performance has been widely recognized and the majority of integrated development environments (IDEs) offer several interactive visualizers for inspecting the runtime behavior of an application. VisualVM [19] is a widespread graphical tool, built on top of the NetBeans IDE platform, for profiling the running time and memory usage of Java applications. Several recent works specifically help the user to understand, find, and eventually fix memory-related problems in their programs. AllocRay [17] is an animated interactive viewer for memory allocation events which shows the changes of the memory over time using 2D memory map plots. dymem [16] builds a directed acyclic graph depicting group-based object ownership: a compact tree-based representation of this graph can help a developer to identify common memory problems.

source profilers. Using KCachegrind, besides analyzing how
the running time has been spent during the execution of a
program, a developer can inspect the call graph and iden-
tify performance bugs due to poor cache utilization. To the
best of our knowledge, the lack of information about the
workload of a routine does not allow these tools to evaluate
how the performance of routines scale as a function of the
workload size.

Paper organization. The remainder of this paper is orga-
nized as follows. In Section 2 we summarize the main ideas
behind the input-sensitive profiling methodology. The
design and the main features of our graphical tool are covered
in Section 3: after discussing the motivation and goals of
aprof-plot, several examples show how our tool can be ef-
effectively leveraged by programmers for performance analysis
purposes. Section 4 concludes the paper, outlining directions
for future work.

2. PROFILING METHODOLOGY

The main idea behind input-sensitive profiling is to aggre-
gate performance measurements for individual routine calls
by the size of the input on which each call operates. Dif-
fently from the classical analysis of algorithms based on
theoretical cost models, where the input size of a procedure
is a parameter known a priori, a key challenge of an auto-
mated approach is the ability to automatically infer the size
of the data given as input to a function. This can be done
using the read memory size metric introduced in [4]:

**Definition 1.** The read memory size (rms) of the exec-
ution of a routine $f$ is the number of distinct memory cells
first accessed by $f$, or by a descendant of $f$ in the call tree,
with a read operation.

The intuition behind this metric is the following. Consider
the first time a memory location $\ell$ is accessed by a routine
activation $f$: if this first access is a read operation, then $\ell$
contains an input value for $f$. Conversely, if $\ell$ is first writ-
ten by $f$, then later read operations will not contribute to
increase the rms since the value stored in $\ell$ was produced
by $f$ itself.

Notice that the rms definition, which is based on tracing
low-level memory accesses made by the program, supports
memory dereferencing and pointers in a natural way. How-
ever, the rms metric ignores any communication between
threads and data received via system calls from the OS ker-
nel, failing to accurately characterize the behavior of rou-
tines executed in the context of modern concurrent and in-
teractive applications. A more recent work [6] has extended
the rms metric in order to include dynamic input sources
such as communication between threads and I/O. For the
sake of presentation, in this paper we refer to the original
metric, but any consideration can be naturally applied to
the latter extension.

**Input-sensitive profile.** Given a metric for estimating
the input size of a routine activation, an input-sensitive
profiler collects several performance measures in order to
evaluate the routine performance scalability. For each rou-
tine $f$, let $N_f = \{n_1, n_2, \ldots\}$ be the set of distinct in-
put sizes on which $f$ is called during the execution of a
program. For each $n_i \in N_f$, the profiler collects a tuple
$(n_i, c_i, max_i, min_i, sum_i, sq_i)$, where:

- $c_i$ is the number of times the routine is called on input
  size $n_i$;
- $max_i$ and $min_i$ are the maximum and minimum costs
  required by any execution of $f$ on input size $n_i$, re-
  spectively;
- $sum_i$ and $sq_i$ are the sum of the costs required by the
  executions of $f$ on input size $n_i$ and the sum of the
costs’ squares, respectively.

In principle, the term cost may refer to any performance
metric, e.g., time, number of executed basic blocks, or cache
misses. Since the focus of the input-sensitive profiling method-
ology is on modeling scalability rather than on exact running
times, the results presented in this article are based on basic
block counts, which have several advantages for studying the
asymptotic behavior of a program, as explained in [9]. After
running an application under an input-sensitive profiler, the
programmer gets a text-based profile which contains a dump
of all performance tuples collected for any executed routine.

3. VISUAL MINING OF INPUT-SENSITIVE
PROFILES

As discussed in Section 2, several performance measure-
ments are automatically collected by an input-sensitive pro-
filer: although these data can be aggregated at runtime ac-
cording to several criteria, the resulting profiles may easily
become very large due to the high number of routines typ-
cally executed by real-world applications. Table 1 shows
some profile statistics related to several applications taken
from the SPEC CPU2006, PARSEC 2.1, and SPEC OMP2012.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>no. routines</th>
<th>profile size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>403.gcc</td>
<td>2551</td>
<td>19.2</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>1780</td>
<td>36.5</td>
</tr>
<tr>
<td>454:calculix</td>
<td>777</td>
<td>9.7</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>1125</td>
<td>2.1</td>
</tr>
<tr>
<td>bodytrack</td>
<td>617</td>
<td>3.1</td>
</tr>
<tr>
<td>canneal</td>
<td>631</td>
<td>0.8</td>
</tr>
<tr>
<td>facesim</td>
<td>776</td>
<td>0.5</td>
</tr>
<tr>
<td>ferret</td>
<td>906</td>
<td>16.4</td>
</tr>
<tr>
<td>vips</td>
<td>982</td>
<td>47</td>
</tr>
<tr>
<td>352.nab</td>
<td>361</td>
<td>3.4</td>
</tr>
<tr>
<td>367.imagick</td>
<td>685</td>
<td>0.3</td>
</tr>
<tr>
<td>376.kdtree</td>
<td>223</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 1: Number of routines and profile sizes of sev-
eral benchmarks taken from SPEC CPU2006, PAR-
SEC 2.1, and SPEC OMP2012.

- $c_i$ is the number of times the routine is called on input
  size $n_i$;
- $max_i$ and $min_i$ are the maximum and minimum costs
  required by any execution of $f$ on input size $n_i$, re-
  spectively;
- $sum_i$ and $sq_i$ are the sum of the costs required by the
  executions of $f$ on input size $n_i$ and the sum of the
costs’ squares, respectively.

In principle, the term cost may refer to any performance
metric, e.g., time, number of executed basic blocks, or cache
misses. Since the focus of the input-sensitive profiling method-
ology is on modeling scalability rather than on exact running
times, the results presented in this article are based on basic
block counts, which have several advantages for studying the
asymptotic behavior of a program, as explained in [9]. After
running an application under an input-sensitive profiler, the
programmer gets a text-based profile which contains a dump
of all performance tuples collected for any executed routine.
Figure 1: Main window of aprof-plot: list of profiled routines (bottom left), automatically generated charts (top) and list of performance tuples (bottom right) of a selected routine.

plot. The goal of this graphical tool is twofold. From one side, the user can inspect the scalability of a specific routine by analyzing several routine-based performance charts. On the other side, our tool attempts to focus the attention of the programmer to the most critical routines, possibly pinpointing unexpected performance trends. Before discussing how aprof-plot can support the user towards these two performance analysis’ directions, we briefly provide an overview of its design. The main interface is shown in Figure 1. After selecting an input-sensitive profile, a list of routines is presented (bottom left) to the user with several pieces of information for each routine: alongside the routine signature and its executable binary, various cumulative performance metrics summarize the impact on the overall application execution (e.g., percentage of cost spent inside the routine, number of calls, number of collected performance tuples). Whenever the user selects from this list a specific routine, on the top part of the interface several routine charts are automatically generated. Each plot can be customized by the user through different tools available in the chart action bar. Finally, the performance tuples collected for the selected routine are listed in the bottom right part of the interface: this allows programmer to carefully inspect the routine profile and to find out details of performance behaviors graphically represented by the routine charts.

3.1 Routine performance analysis

Performance as a function of workload size. Input-sensitive profiling naturally allows the programmer to investigate how the running time of a routine scales as a function of its workload size. This kind of analysis is actually critical for any software: seemingly benign fragments of code may be fast on some testing workloads, passing unnoticed with traditional profilers, while all of a sudden they can become major performance bottlenecks when deployed on larger inputs (see, e.g., examples in [5]). To this aim, aprof-plot can automatically generate worst-case, average-case, and best-case cost plots. Indeed, given the tuples \( \langle n_i, c_i, \text{max}_i, \text{min}_i, \text{sum}_i, \text{sq}_i \rangle \) collected for a routine \( f \) (see Section 2), the sets of points \( \langle n_i, \text{max}_i \rangle \) and \( \langle n_i, \text{min}_i \rangle \) can be used to estimate how the empirical worst-case and best-case costs of a routine grow as a function of the input size. The average behavior is instead given by the set \( \langle n_i, \text{avg}_i \rangle \), where the average cost per invocation on input size \( n_i \) is obtained by computing \( \text{avg}_i = \frac{\text{sum}_i}{c_i} \). An example of these charts is provided in Figure 2a and is based on routine heap_sort_pairs of SPEC OMP2012 benchmark 352.nab: the best-case, average-case, and worst-case trends appear to be relatively similar and rather smooth. To get more precise insights on asymptotic performance of a routine, our tool allows the programmer to easily apply a technique known as curve bounding. In particular, the guess ratio rule (see [13] and [14]) estimates the trend of a function \( f(n) \) by considering a guess function \( h(n) \) and analyzing the trend of ratio \( f(n)/h(n) \): the ratio stabilizes to a non negative constant if \( f \in O(h(n)) \), while it (eventually) increases if \( f \not\in O(h(n)) \). In our example we divided the worst-case trend of Figure 2a by three different guess functions: \( n, n \log n, \) and \( n^2 \). The three resulting curves are shown in Figure 2b, Figure 2c, and Figure 2d, respectively. The cost of routine heap_sort_pairs increases when divided by \( n \), decreases when divided by \( n^2 \), and stabilizes to a positive constant when divided by \( n \log n \). This confirms that the trend is \( n \log n \), as expected from any bug-free implementation of the heap sort algorithm. An interactive popup menu, available when displaying the curve bounding plot in aprof-plot, allows the programmer to easily test several user-defined guess functions.

Workload analysis. A natural question is which are the typical workloads of a routine. Even more interesting is in-
investigating how the actual workload is impacting the routine performance. Two orthogonal considerations can be made. From one side, aprof-plot can give insights on the typical workloads on which a routine is called during the execution of a program: as an example, Figure 3a depicts the frequency distribution of the workload sizes observed for routine heapsort_pairs: a peak around the RMS value 200 provides a rough estimation about the typical size of arrays sorted by this routine in this specific application. In general, this information might be very useful not only for code optimization, but also for algorithmic improvements, even theoretical, in specific scenarios. For instance, if an application always needs to sort arrays with less than 16 items, it may be convenient to use a non-optimal sorting algorithm with runtime \( n^2 \) instead of an asymptotically optimal one with runtime \( n \log n \). A case study of this flavour is discussed in [5].

On the other side, in many scenarios a routine may be generally cheap, but sporadically require a high cost. Consider for instance an operation that appends an item at the end of a resizable array such as std::vector’s push_back function of the C++ STL. If the array capacity is not exceeded, then the operation takes constant time. Otherwise, the array must be reallocated, typically requiring an expensive copy of all items from the current array to a new larger one. Expanding the array by a constant multiplicative factor at each reallocation (e.g., doubling the array), the expensive append calls can be guaranteed to be exponentially less frequent than constant append calls [7]. This kind of analysis, which measures the average time of a function over a sequence of invocations, is called \textit{amortized analysis} [18]. The amortized cost metric given in [5] can be be computed in terms of the profiling tuples introduced in Section 2. Due to lack of space, we omit the definition of this metric whose main idea is that the cost of expensive but infrequent calls can be amortized over frequent but cheap calls. Exploiting this intuition, more informative plots can be generated by shaving off expensive peaks, leaving just the points where most of the routine work is performed. For instance, Figure 3 reports an example based on C++ STL routine std::vector::push_back. Even if the cost plot of this routine presents a linear trend (Figure 3b), expensive calls are rather infrequent (Figure 3c). A careful analysis of the routine performance tuples reveals that less than 0.000025\% of the routine calls has executed more than 250 basic blocks. The goal of the amortized cost plot is to automatically expose this kind of consideration: as shown by Figure 3d, calls over RMS values larger than 80 can be amortized (zeroed points), leaving only inexpensive calls which actually characterize most of the routine work.

3.2 Plot customization and noise reduction

aprf-plot supports different kinds of interaction, allowing the programmer to perform a more effective performance investigation. Several useful features can be triggered by the user in order to customize a routine chart. For instance, the x and y axes of a plot can be set to logarithmic scales (see, e.g., Figure 3c). Furthermore, since the interesting behavior
of a routine may be confined to a specific area of a chart, the user can zoom in and out from a specific set of points. A common (and more important) issue when analyzing a routine is due to noisy profile data. To this aim, we have implemented two noise reduction techniques: point aggregation and smoothing. The first approach decreases the number of points, while the second one preserves the cardinality of the original set. Given the set of $N$ points $(x, y)$ of a chart sorted by the $x$ values, point aggregation partitions this set in groups of $d$ points and then computes the arithmetic mean within each group. Notice that $d$ is a positive constant value which can be customized by the user. Smoothing, instead, calculates the centered moving average [12]:

$$\forall k, \frac{w}{2} \leq k \leq N - \frac{w}{2}, \quad y_i = \frac{1}{w} \sum_{i=k-w/2}^{k+w/2} y_i$$

where $w$ is the user-customizable size of the moving window. Figure 4b provides an example based on the routine cse_basic_block of the SPEC benchmark 403.gcc: the original noisy trend (Figure 4a) has been smoothed applying a moving window $w = 256$.

Finally, we implemented in aprof-plot a source code browsing feature that allows the programmer to analyze the code while she is still looking at the routine charts. After choosing the source directory of a C/C++ application, aprof-plot can automatically open the appropriate source code file and show the relevant piece of code inside a text-editor widget. This makes it easier for programmers to find out why a certain piece of code exhibits some performance behaviors.

3.3 Spotting critical routines

In Section 3.1, we have discussed how aprof-plot can enable a deep analysis of relevant aspects of a routine’s behavior. However, since real-world applications may be composed by thousands of routines, this kind of analysis could easily become overwhelming for a developer. For this reason, aprof-plot attempts at pointing the attention of the programmer to the most critical routines of an application. In particular, it helps developers prioritize their analysis by highlighting routines which are likely to contain performance issues. By sorting routines based on the percentage of cost, a developer can immediately understand how the running time has been spent during the execution of the program. Several filtering strategies can help to refine this list: uninteresting routines, such as library functions, can be filtered out based on the executable binary, while insignificant routines can be hidden using different metric thresholds (e.g., by filtering routines with small cost percentage or with a small number of performance tuples). Exploiting these tools, the number of routines which needs to be analyzed by a developer can be significantly reduced. However, since the ultimate goal of input-sensitive profiling is to detect the routines with high asymptotic cost, we have implemented in aprof-plot an automatic approach for estimating the asymptotic complexity of a routine. Using regression analysis techniques [3], our tool constructs for each routine a mathematical function that has the best fit to its data points. In particular, we choose as a cost model the function $b \cdot n^c + a$, which generalizes both the power law and the linear models (when $a = 0$ and $c = 1$, respectively). After estimating the fitting coefficients for any program routine, aprof-plot can sort routines based on their asymptotic complexity (i.e., the $c$ coefficient), allowing the developer to possibly detect unexpected performance trends. Since this kind of analysis crucially depends on the quality of the fitting results, aprof-plot allows the user to filter routines which have low fitting quality (e.g., $R^2 \leq 0.92$) or unrealistic coefficient values (e.g., $b < 0.001$). The validity of this approach has been empirically assessed in [5].

Figure 5a provides an excerpt taken from the routine list of the PARSEC benchmark ferret. The original 906 routines have been filtered according to the following criteria: more than 10 performance tuples ($|N| > 10$), high fitting quality ($R^2 > 0.92$), and reasonable $b$ coefficient value ($b > 0.01$). After filtering, 75 routines remain and are sorted based on the percentage of executed basic blocks (i.e., their cost): among the top four routines (shown by Figure 5a), an interesting example is provided by cass_result_merge_lists. This routine has been called 3500 times, requiring 5.06% of the total program cost, and aprof-plot has been able to automatically estimate a quadratic asymptotic complexity ($c = 2.086$). As discussed in [5], apart from specific algorithmically-critical routines usually well known to the programmer, most benign routines appear to have a sub-quadratic trend and many common programming mistakes tend to introduce quadratic inefficiencies, e.g., by invoking a

---

**Figure 5**: (a) Routine list of PARSEC 2.1 benchmark ferret sorted by percentage of cost, number of calls, and best-fit parameters. (b) Cost plot of routine cass_result_merge_lists with regression trend obtained by least-squares fitting.
subroutine in a loop under the incorrect assumption that it takes constant time. Since the fitting quality is rather high ($R^2 = 0.99$), routine \texttt{case_result_merge_lists} is a good candidate for further performance investigation. The worst-case cost plot is given in Figure 6b: the regression curve accurately predicts the actual cost trend. By inspecting the source of this routine, we were able to conclude that its algorithm is indeed $O(n^2)$ due to a doubly-nested loop used for merging two input lists. Although this quadratic trend is not given by any trivial programming mistake, we believe that some algorithmic optimizations could be implemented to improve the running time of this routine.

As shown by this example, the filtering and ranking strategies implemented by \texttt{aprof-plot} can significantly support the developer towards performance investigation of the most critical routines, possibly revealing unexpected performance issues.

4. CONCLUSIONS

In this paper we have presented an interactive visualization framework for performance analysis of input-sensitive profiles. The key benefit of \texttt{aprof-plot} is to allow the programmer to investigate how the running time of a routine scales as a function of the workload size. Exploiting techniques such as curve fitting and bounding, our tool can focus the attention of the programmer to the most critical routines of an application, possibly pinpointing unexpected scalability problems. Useful insights on the typical workload sizes can help developers optimize their programs.

As a future direction, it would be interesting to improve the support in \texttt{aprof-plot} for input-sensitive profiles with calling-contexts annotations [8]. This further level of performance characterization can help developers analyze routines with context-dependent performance trends.

Another interesting improvement would be to implement ad-hoc fitting algorithms, specifically tailored to curves related to execution costs. In particular, we notice that a routine may exhibit rather different performance trends based on several runtime conditions and workload features. Since a classical curve fitting algorithm is unable to detect multiple trends, \texttt{aplot-plot} fails to automatically estimate the asymptotic complexity of the routine, forcing the user towards manual performance analysis.

Acknowledgements

We would like to thank Camil Demetrescu and Irene Finocchi for their valuable feedback during the development of \texttt{aprof-plot} and for many useful discussions. We are also indebted to Bruno Aleandri for developing an earlier version of \texttt{aprof-plot}.

5. REFERENCES

[1] \texttt{aprof}: an input-sensitive profiler. 
https://code.google.com/p/aprof/.
http://visualvm.java.net/.