Input-Sensitive Profiling

Emilio Coppa
Dept. of Computer and System Sciences
Sapienza University of Rome
ercoppa@gmail.com

Camil Demetrescu
Dept. of Computer and System Sciences
Sapienza University of Rome
demetres@dis.uniroma1.it

Irene Finocchi
Dept. of Computer Science
Sapienza University of Rome
finocchi@di.uniroma1.it

Abstract

In this paper we present a profiling methodology and toolkit for helping developers discover hidden asymptotic inefficiencies in the code. From one or more runs of a program, our profiler automatically measures how the performance of individual routines scales as a function of the input size, yielding clues to their growth rate. The output of the profiler is, for each executed routine of the program, a set of tuples that aggregate performance costs by input size. The collected profiles can be used to produce performance plots and derive trend functions by statistical curve fitting or bounding techniques. A key feature of our method is the ability to automatically measure the size of the input given to a generic code fragment: to this aim, we propose an effective metric for estimating the input size of a routine and show how to compute it efficiently. We discuss several case studies, showing that our approach can reveal asymptotic bottlenecks that other profilers may fail to detect and characterize the workload and behavior of individual routines in the context of real applications. To prove the feasibility of our techniques, we implemented a Valgrind tool called `aprof` and performed an extensive experimental evaluation on the SPEC CPU2006 benchmarks. Our experiments show that `aprof` delivers comparable performance to other prominent Valgrind tools, and can generate informative plots even from single runs on typical workloads for most algorithmically-critical routines.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics—performance measures

General Terms Algorithms, Measurement, Performance.

Keywords Performance profiling, asymptotic analysis, dynamic program analysis, instrumentation.

1. Introduction

Performance profiling plays a crucial role in software development, allowing programmers to test the efficiency of an application and discover possible performance bottlenecks. Traditional profilers associate performance metrics to nodes or paths of the control flow or call graph by collecting runtime information on specific workloads [2, 19, 27, 39]. These approaches provide valuable information for studying the dynamic behavior of a program and guiding optimizations to portions of the code that take most resources on the considered inputs. However, they may fail to characterize how the performance of a program scales as a function of the input size, which is crucial for the efficiency and reliability of software. Seemingly benign fragments of code may be fast on some testing workloads, passing unnoticed in traditional profilers, while all of a sudden they can become major performance bottlenecks when deployed on larger inputs. As an anecdotal example, we report a story [8] related to the COSMOS circuit simulator, originally developed by Randal E. Bryant and his colleagues at CMU [10]. When the project was adopted by a major semiconductor manufacturer, it underwent a major performance tuning phase, including a modification to a function in charge of mapping signal names to electrical nodes, which appeared to be especially time-consuming: by just hashing on bounded-length name prefixes rather than on entire names, the simulator became faster on all benchmarks. However, when circuits later grew larger and adopted hierarchical naming schemes, many signal names ended up sharing long common prefixes, and thus hashed to the same buckets. As a result, the simulator startup time became intolerable, taking hours for what should have required a few minutes. Identifying the problem - introduced several years before in an effort to optimize the program - required several days of analysis. There are many other examples of large software projects where this sort of problems occurred [9].

The problem of empirically studying the asymptotic behavior of a program has been the target of extensive research in experimental algorithmics [14, 25, 30, 33]. Individual portions of algorithmic code are typically analyzed on ad-hoc test harnesses, which reproduce real-world scenarios by performing multiple runs with different and determinable input parameters, collecting experimental data for comparing the actual relative performance of algorithms and studying their amenability for use in specific applications. While this approach provides valuable information for complementing theoretical analyses, it has several drawbacks as a performance evaluation method in actual software development. Firstly, it is typically difficult and time-consuming to manually extract portions of code from an application and analyze them separately on different input sizes to determine their performance growth rate. Furthermore, asymptotic inefficiencies may be introduced by programming errors in unexpected locations, whose detection requires automated and systematic profiling methodologies, especially on large-scale systems. As another point, by studying performance-critical routines out of their context, we miss how they interact with the overall application in which they are deployed, including cache effects and branch mispredictions. We also notice that it may be hard to collect real data about typical usage scenarios to be reproduced in experiments, in particular when the workloads of interest are generated by intermediate steps of a computation.

Our contributions. Motivated by the observation that critical algorithmic routines should be analyzed within the actual context of the software projects in which they are deployed, this paper makes

PLDI'12, June 11–16, 2012, Beijing, China.

a first step towards bridging the gap between the profiling practice and the methodologies used in experimental algorithmics. As a main contribution, we devise a new automated profiling technique for helping developers discover hidden asymptotic inefficiencies in the code. To prove the effectiveness of our approach, we developed a Valgrind [34] tool called `aprof`: from one or more runs of a program, `aprof` automatically measures how the performance of individual routines scales as a function of the input size. Using statistical curve fitting [11] and curve bounding [31] techniques on the collected profiles, developers can derive closed form expressions that describe mathematical cost functions for program routines, yielding clues to their growth rate. We discuss several case studies, showing that our approach can reveal asymptotic bottlenecks that other profilers may fail to detect and characterize the workload and behavior of individual routines in the context of real applications. In particular, we focus on applications included in prominent Linux distributions as well as in the SPEC CPU2006 suite. A distinguishing feature of our approach is the ability to automatically measure the size of the input given to a generic routine. This allows us to explore a novel dimension in interprocedural context-sensitive profiling where costs are associated to distinct input sizes of a routine, rather than to paths in the call graph as in traditional contextual profiling [2, 16, 46]. To support this approach, which we call *input-sensitive profiling*, we propose an effective metric for estimating the input size of a routine by counting the number of distinct memory cells first accessed by the routine with a read operation, and show how to compute it efficiently. An extensive experimental evaluation on the SPEC CPU2006 benchmarks reveals that `aprof` is as efficient as other prominent Valgrind tools, and can generate informative plots even from single runs on typical workloads for most algorithmically-critical routines. The tool is available at `http://code.google.com/p/aprof/`.

The remainder of this paper is organized as follows. In Section 2 we introduce our profiling methodology, discussing relevant properties of our approach. In Section 3 we present case studies that demonstrate the utility of input-sensitive profiling. Section 4 proposes an efficient profiling algorithm, Section 5 describes the most relevant aspect of the implementation of `aprof`, and Section 6 presents our experimental results. Related work is discussed in Section 7 and concluding remarks are given in Section 8.

2. Input-Sensitive Profiling

In this section we introduce our input-sensitive profiling methodology. Differently from the classical analysis of algorithms based on theoretical cost models, where the input size of a procedure is a parameter known a priori and clear from the abstract description of the underlying algorithm, a key challenge of an automated approach is the ability to automatically infer the size of the input data on which a function operates. We first propose a solution to this basic problem.

2.1 Read Memory Size

We introduce a metric, which we call *read memory size*, for estimating the input size of a routine invocation:

Definition 1. *The* read memory size *(*RMS*) of the execution 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 main idea is that cells that are accessed by a function for the first time with a read operation contain the input values of the routine. Conversely, if a cell is first written and then read by the routine, the read value is not part of the input as it was determined by the routine itself. We 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.

Example 1. Consider the following trace of operations. Function `g` performs three first-read operations (lines 5 – 7) and its RMS is thus 3. Function `f` performs five read operations, three of which through its subroutine `g`. However, its RMS is only 2: the `read` operations at line 5, line 6, and line 10 are not first-reads with respect to `f`. Indeed, `x` has been already read at line 2 and `y` and `w` are written at lines 3 and 8, respectively, before being read. Hence, only the read operations at lines 2 and 7 contribute to the computation of the read memory size of function `f`.

```
1.  call f
2.    read x
3.    write y
4.    call g
5.      read x
6.      read y
7.      read z
8.      write w
9.      return
10.   read w
11.   return
```

In Section 4 we will propose an efficient algorithm for computing the RMS of each routine invocation from a program execution trace.

2.2 Profiling Methodology

For each routine f, we determine the set $N_f = \{n_1, n_2, \ldots\}$ of distinct RMS values on which f is called during the execution of a program. For each $n_i \in N_f$, which is an estimate of an input size, we collect a tuple $\langle n_i, c_i, max_i, min_i, sum_i, sq_i \rangle$, 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, respectively;
- 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.

We use the generic term *cost* to refer to any performance metric, e.g., time, number of executed basic blocks, etc. The cost of a routine execution is intended as a *cumulative cost*, i.e., it includes the costs of all the routine's descendants in the call tree.

Analysis metrics. The value $avg_i = sum_i/c_i$ is the average cost per invocation on input size n_i, while sq_i is used for computing the cost variance var_i. The sets of points $\langle n_i, max_i \rangle$, $\langle n_i, min_i \rangle$, and $\langle n_i, avg_i \rangle$ estimate how the worst-case, best-case, and average-case costs of a routine grow as a function of the input size (see Figure 1a for an example generated with our technique). The total cost per input size, the variance between execution costs, and the frequency distribution of the input sizes are given by points $\langle n_i, sum_i \rangle$, $\langle n_i, var_i \rangle$, and $\langle n_i, c_i \rangle$, respectively (see Figures 1b, 1c, and 1d).

Example 2. To exemplify our profiling methodology, we analyze the simple C code fragment on the left. Any execution of `get` has RMS 3. This accounts for the addresses of lvalues `v`, `i`, and `v[i]`, which contain the input values of the function. Routine `mid` reads 3 distinct addresses as well: `v` and `n` directly at line 5, and `v[n/2]` indirectly via a call to `get`. Variable `m` is not counted in the RMS of `mid` as it is first accessed by a write operation at line 5. In this example, an input-sensitive profiler would just collect one performance tuple for `get` and one for `mid`, i.e., $|N_{\texttt{get}}| = |N_{\texttt{mid}}| = 1$.

```
1: int get(int v[], int i) {
2:     return v[i];
3: }

4: int mid(int v[], int n) {
5:     int m = get(v, n/2);
6:     return m;
7: }
```

Example 3. Our next example illustrates the case where the size of the input may vary at each call:

```
1: int count_zero(int v[], int n) {
2:     int i, count = 0;
3:     for (i=0; i<n; i++) count += (v[i]==0 ? 1:0);
4:     return count;
5: }
```

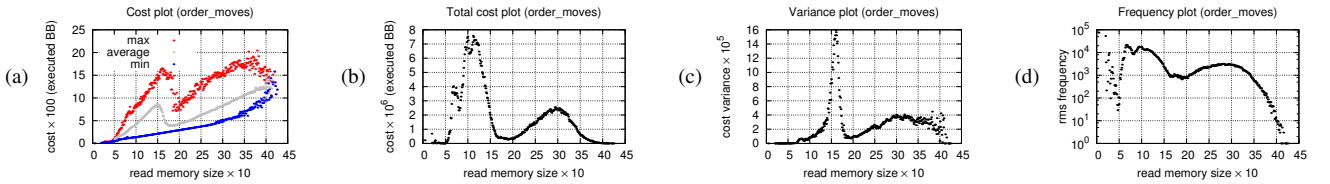


Figure 1. Input-sensitive plots automatically computed by `aprof`. All plots are related to function `order_moves()` of the `sjeng` chess playing program included in the SPEC CPU2006 benchmarks [23], executed on a single reference workload.

The function, which counts the number of 0's in the input array `v`, has RMS $\mathtt{n}+2$: it reads variables `v` and `n`, and the first `n` cells of the array pointed to by `v`. An input-sensitive profiler would collect one performance tuple for each distinct value of `n` on which `count_zero` is called during the execution of the program. In this case, the number of tuples $|N_{\texttt{count\_zero}}|$ cannot exceed the number of distinct calls of the function.

Example 4. We now show that even a *single* invocation of a function made by a program can yield several performance tuples. Consider a recursive variant of the `count_zero` function:

```
1: int count_zero(int v[], int n) {
2:     if (n<1) return 0;
3:     return (v[n-1]==0 ? 1:0) + count_zero(v, n-1);
4: }
```

Observe that, just like the iterative version, the first invocation of function `count_zero` has RMS $n_1 = \mathtt{n}+2$. However, calling `count_zero` on parameter `n` also results in `n` additional recursive activations of the function for all size values ranging from $\mathtt{n}-1$ down to 0. Therefore, we collect $\mathtt{n}+1$ performance tuples from just one starting activation, i.e., $|N_{\texttt{count\_zero}}| = \mathtt{n}+1$. The read memory size corresponding to the i-th invocation is $n_i = \mathtt{n}-i+3$, for each $i \in [1, \mathtt{n}+1]$.

2.3 Characterizing Asymptotic Behavior

The examples presented in Section 2.2 show that our profiling approach allows a tool to automatically collect performance tuples from an execution run, relating cost measures to the input size of a routine. As exemplified in Figure 1, collected profiles can be used to produce performance plots, providing effective visualizations of the behavior of a program. A natural question we address in this section is to which extent this methodology can help developers assess the asymptotic cost of a piece of code.

Asymptotics vs. finite experiments. The ultimate goal of the complexity analysis of algorithms is to find closed form expressions for the running time or other measures of resource consumption (e.g., space usage, cache misses, or number of transmitted bits). Since this may be too difficult, it is common to estimate the theoretical complexity in an asymptotic sense, i.e., for arbitrarily large inputs. To get insights on the asymptotic behavior from finite experiments, it would be necessary to extrapolate trend data beyond the range of experimentation. Unfortunately, no data analysis method for inferring asymptotic trends can be guaranteed to be correct for all data sets: as observed in [31], for any finite vector of problem sizes there are functions of arbitrarily high degree that are indistinguishable from the constant function at those sizes. Throughout this paper we will exploit two main techniques: curve fitting and curve bounding. Statistical curve fitting [11] is the process of constructing a mathematical function that has the best fit to a series of data points. Regression analysis techniques are widely used for this kind of predictions: in our examples, we used the curve fitting tools provided by `gnuplot` [45]. Given a set of data points (X_i, Y_i) obtained from an experiment such that $Y_i = f(X_i)$ for some unknown function f, curve bounding [31] makes it possible to estimate complexity classes $O(g_1)$ and $\Omega(g_2)$ such that $f \in O(g_1)$ and $f \in \Omega(g_2)$. In this paper we will perform curve bounding by "guess ratio", which works by considering a guess function h and analyzing the trend of ratio $f(n)/h(n)$: the ratio stabilizes to a nonnegative constant if $f \in O(h)$, while it (eventually) increases if $f \notin O(h)$.

Number of collected tuples. The number of collected tuples plays a fundamental role in characterizing the behavior of a routine. This number depends on several factors, including the structure of the routine, how it is used in the context of the program, and ultimately the workload on which the program is tested. We have seen that recursive functions allow it to collect several tuples with just one initial call. In Section 6 we will provide a quantitative evaluation of the number of distinct tuples that can be collected for each routine in a variety of applications on typical workloads. As we will discuss in Section 3, variety of input patterns in test workloads is more important than their sheer size: small workloads can yield large amounts of information, while large repetitive inputs may be of little help. We also observe that profiles of the same program collected from different runs can be naturally merged together to generate profiles with a larger number of tuples.

Accuracy of the* RMS *metric. For functions that read their input entirely at least once, the read memory size approximates within constant factors the actual input size n: hence, if RMS $= \Omega(n)$, the input-sensitive profile of a function f correctly estimates the theoretical cost $T_f(n)$ of the algorithm implemented by f. This is not the case for sublinear functions. For instance, consider a binary search over a sorted array, which reads only $O(\log n)$ cells of the input array: in this case RMS $= O(\log n)$, so our approach estimates $T(\log n)$ rather than $T(n)$. Sublinear functions, however, are unlikely to represent performance bottlenecks and are therefore the less interesting from a profiling perspective.

We remark that the RMS is a measure of distinct accessed memory cells. Hence, it fails to characterize computations whose running time is determined by the value of some variable: e.g., the RMS of the naive algorithm for computing the factorial of a number n would be constant, regardless of the value of n. Aggregating performance measurements by distinct values of function arguments is an alternative approach that is explored, e.g., in [28].

Parameters passed in registers, which should be regarded as part of a function's input, are also not counted in the RMS. However, they only account for small additive constants, and are therefore negligible for estimating growth rates for sufficiently large inputs. For instance, if parameters `v` and `n` of function `count_zero` discussed in Section 2.2 are held by the compiler in registers rather than on stack, the RMS would be exactly `n` rather than `n`+2.

3. Case Studies

In this section we describe scenarios where input-sensitive profiling can provide valuable information to the programmer. Our examples are based on the `aprof` tool described in Section 5 and use basic block (BB) counts as performance metric.

	index	% time	self	children	name
	[1]	99.9	2.00	16.23	main [1]
(a)	[2]	52.2	5.82	3.70	addword [2]
	[3]	31.3	2.81	2.90	str_tolower [3]
	[4]	20.3	3.70	0.00	hash [4]
	[1]	100.0	0.21	4.61	main [1]
(b)	[2]	61.8	0.98	2.00	str_tolower [2]
	[3]	41.4	2.00	0.00	wf_tolower [3]
	[4]	32.6	0.07	1.50	addword [4]

Figure 2. Output of `gprof` for application `wf` on two different texts: (a) Anna Karenina; (b) protein sequences.

Discovering hidden inefficiencies. As a first example, we show that input-sensitive profiling makes it possible to discover performance bottlenecks that may not be revealed by standard profiling methods. Our discussion is based on `wf-0.41` [15], a simple word frequency counter included in the current development head of Linux Fedora (Fedora 17–Beefy Miracle): `wf` scans a text file and counts the frequency of words in the text. To efficiently count repetitions, it adds words to a hash table using a function called `addword`, and reimplements a variety of ad-hoc string utility functions to handle the ISO Latin-1 representation, including a function (`str_tolower`) to convert all characters to lower case.

We analyzed `wf` with both `gprof` [19] and `aprof` on a text corpus taken from the *Gutenberg Project* [22], considering several books from classical literature. On all test sets, `gprof` ranked `addword` as the hottest function. For instance, on Tolstoj's "Anna Karenina", `addword` took roughly 52% of the total time, as shown in Figure 2a. This was not surprising, as it appears to be algorithmically more complex than all other basic text management functions. We also profiled `wf` with `gprof` on larger and larger input texts from [22] to study how the performance of individual routines scales as a function of the size of the program's input. Collected profiles confirmed that `addword` is consistently more expensive than all other functions. Figure 3–left reports the performance trends of `addword` and `str_tolower`, obtained by extracting performance figures from 11 different `gprof` reports for input files of different size derived from texts in [22]. Both curves seem to grow linearly as a function of `wf`'s input size. The tests we made with `gprof` did not provide any evidence of asymptotic bottlenecks in `wf`.

In contrast, `aprof` discovered a rather different scenario already on the smaller workload, pinpointing a serious inefficiency in a seemingly benign function: surprisingly, the cost of `str_tolower` turned out to grow quadratically with the length of the input string (see Figure 3–middle). Conversely, `aprof` showed that `addword` scales linearly (see Figure 3–right). By analyzing the source code of `str_tolower`:

```
1: void str_tolower(char *str) {
2:     int i;
3:     for (i=0; i < strlen(str); i++)
4:         str[i] = wf_tolower(str[i]);
5: }
```

we noticed that `strlen` is redundantly called at each iteration, resulting in a quadratic running time. Once we discovered the inefficiency using `aprof`, we made a second set of experiments on carefully selected inputs with very long words, where the inefficiency in `str_tolower` is most likely to impact significantly the overall running time. In particular, we considered a data set containing long protein sequences taken from the *Genomics-96* data set of the Genome Informatics Research Lab [17]. Indeed, on this data set `gprof` showed that `str_tolower` accounts for 61.8% of the total time, while `addword` drops to 32.6% (Figure 2b).

The inefficiency found by `aprof` in `wf` is rather common [9] and was very easy to fix by loop-invariant code motion. To complete our investigation, we compared the wall-clock time required by the original `wf-0.41` code included in the Fedora RPM, and a new version where `strlen(str)` was moved out of the loop: fixing the code improved the total running time of `wf` by 6% on Anna Karenina and by 30% on protein sequences.

Discussion. The `wf` example shows that repeating classic `gprof`-style profiling at scale to assess the computational complexity of individual routines can yield misleading results: `gprof` failed to reveal the quadratic trend of `str_tolower`, even making several tests on workloads of different size, because the input of `str_tolower` are *single* words of the input text, not the *entire* document given as input to the application. In general, the input of a routine is often produced by other routines (e.g., `wf`'s text tokenizer feeds strings to `str_tolower`) and may be unpredictably related to the input of the overall application. Hence, setting up a collection of test workloads for an application to expose the asymptotic behavior of individual routines can be rather difficult with traditional profilers. Also, different input families may be needed to characterize different routines, making this approach largely impractical. Since a typical text contains words of different lengths and `aprof` collects separate performance measurements for each distinct input size, it could correctly pinpoint the quadratic trend of `str_tolower`'s cost function even on a single run on a small input.

Workload characterization. The most realistic program executions are on deployed systems. A benefit of our profiling methodology is that it can give insights on the typical workloads on which a function is called in the context of real applications. This information might be very useful not only for code optimization, but also for algorithmic improvements, even theoretical, in specific scenarios. For instance, routing algorithms for GPS navigation systems are specifically designed to take advantage of the sparsity of the input road networks on which they are deployed. Similarly, 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 $4n \log n$.

Figure 4 provides a concrete example where the workload of a function can be rather different depending on the specific application scenario and data set, showing that our profiler can highlight such differences. In particular, we ran the word frequency counter `wf` discussed above on two different inputs (Anna Karenina and protein sequences), and analyzed the `strlen` function used by `wf`. Notice that the read memory size of `strlen` is an indirect measure of the length of the strings on which it is invoked: the maximum RMS is about 20 for Anna Karenina and larger than 900 for protein sequences, highlighting the structural difference between these two data sets. Even more significant is the fact that the frequency curves shown on the left of Figure 4 have opposite trends: decreasing on Anna Karenina, as we would expect for a natural language text, and increasing on protein sequences. Since long protein sequences tend to be very frequent, on this data set `strlen` will likely have a tangible impact on the execution cost.

Exposing empirical asymptotics. Input-sensitive profiles can characterize the empirical asymptotic behavior of program routines in realistic execution scenarios, often yielding more precise results than theoretical cost models. For instance, the `gg_sort` routine included in SPEC CPU2006 component `gobmk` [23] implements the combsort algorithm [1], a variant of bubblesort that compares items at distance larger than 1. Combsort is known to have at least quadratic running time in the worst case, but can rival fast algorithms like quicksort in many practical cases. This is confirmed by the guess ratio plot of `gg_sort`, which stabilizes to a constant value when divided by the guess function $n \log n$, as

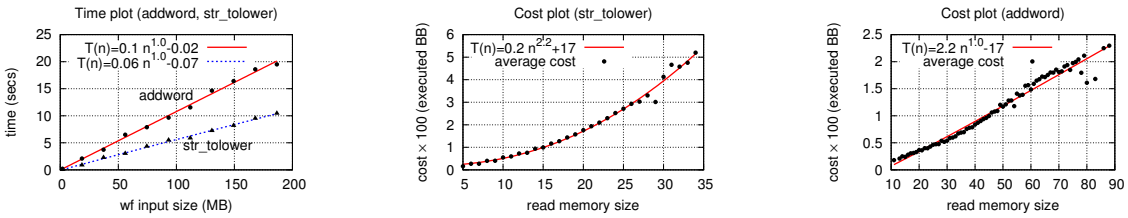


Figure 3. Profiling of `wf`'s functions `addword` and `str_tolower` on texts from classical literature: cost trends derived from the profiles produced by 11 runs of `gprof` on input files of different size (left); cost trends derived by a single run of `aprof` on the smallest workload (middle and right). Regression curves are obtained by least-squares fitting.

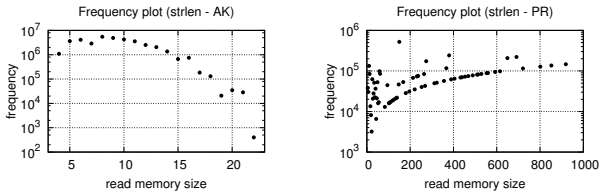


Figure 4. Workload characterization of function `strlen` used by application `wf` on two different texts: Anna Karenina (AK) and protein sequences (PR).

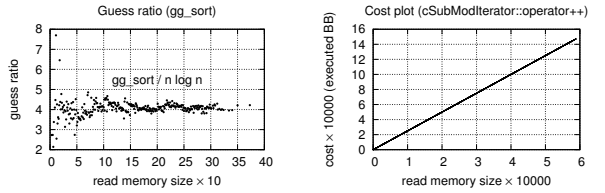


Figure 5. Input-sensitive profiles excerpted from two SPEC CPU2006 benchmarks: (a) routine `gg_sort` from the SPEC component `gobmk` (curve bounding is based on guess function $n \log n$); (b) routine `cSubModIterator::operator++` from the SPEC component `omnetpp`.

shown in Figure 5(a). In other cases, the asymptotic trends empirically observed by `aprof` can direct the programmer's attention to critical routines that may otherwise pass unnoticed. As an example, the chart in Figure 5(b) shows that the cost trend of routine `cSubModIterator::operator++` defined in SPEC benchmark `omnetpp` [23] is linear, differently from what one might expect from a `++` operator. Actually, after profiling `omnetpp` with `aprof`, we found in `cSubModIterator`'s source code the comment: "this should be replaced with something faster".

4. Algorithms

In this section we describe an efficient algorithm for computing the read memory size and the input-sensitive profile of a routine. An input-sensitive profiler is given as input a trace of program operations, including routine activations (`call`), routine completions (`return`), and `read`/`write` memory accesses (see Example 1 in Section 2.1). Additional operations might be traced, depending on the specific performance metrics to be computed. For each operation, the profiler must update RMS and cost information. Before describing our algorithm, we discuss a simple-minded approach as a warm-up for the reader.

4.1 A Simple-Minded Approach

Each activation of a routine f has its own read memory size, which is part of the set N_f of input sizes on which f is invoked during the execution of the program (see Section 2.2). Computing the RMS of an activation of f requires to monitor the set of locations that are read during that activation (not only by f itself, but also by its descendants in the call tree) before being written. A simple-minded approach is to maintain a list A_f of all memory locations accessed (both read and written) during the activation of f. Immediately after entering f, this list is empty and the RMS is equal to 0. When f accesses a location w for the first time, then w is added to A_f and, if the access is a `read` operation, the RMS is increased by 1. The same check and update must be performed for all pending routine activations in the call stack, which are implicitly accessing location w through their descendant f. If we start from the most recent activations, it is possible to stop stack-walking when the first routine h is encountered such that w already belongs to A_h. However, h may be the program root in the worst case, making updates based on stack-walking prohibitively time-consuming. The space usage of this approach is also quite demanding: in the worst case, each distinct memory location could be stored in all lists A_h of pending routine activations, and in that case the space would be proportional to the memory size times the maximum stack depth.

4.2 The Latest-Access Algorithm

We now describe a more space- and time-efficient algorithm, sketched in Figure 6. The main idea is to avoid maintaining explicitly the RMS and the lists A_f of accessed locations, but to store only partial information that can be updated quickly during the computation and from which the RMS can be easily derived upon the completion of a routine. In more detail, for each pending activation of a routine f, we store the set L_f of locations whose *latest* access was done by f (either directly or by its completed subroutines). It is not difficult to see that each memory location is stored in exactly one set, i.e., at any time during the execution of the program the sets L_f partition the locations accessed so far. The sets L_f of latest accesses are stored implicitly by associating timestamps to routines and memory locations, as described below.

Data structures. The algorithm uses a shadow run-time stack S, whose top is indexed by variable top. For each $i \in [1, top]$, the i-th stack entry $S[i]$ stores:

- The id $S[i].rtn$ of the i-th pending routine activation.
- The timestamp $S[i].ts$ of the i-th pending activation, i.e., the time at which the routine was entered.
- The cumulative cost $S[i].cost$ of the activation.

procedure `call`(r):
1: $count$++
2: top++
3: $S[top].rtn \leftarrow r$
4: $S[top].ts \leftarrow count$
5: $S[top].rms \leftarrow 0$
6: $S[top].cost \leftarrow$ `get_cost()`

procedure `return`():
1: `collect`($S[top].rtn$, $S[top].rms$, `get_cost()` $- S[top].cost$)
2: $S[top-1].rms$ += $S[top].rms$
3: $top--$

procedure `read`(w):
1: **if** $ts[w] < S[top].ts$ **then**
2: $S[top].rms$++
3: **if** $ts[w] \neq 0$ **then**
4: let i be the max index in S such that $S[i].ts \leq ts[w]$
5: $S[i].rms--$
6: **end if**
7: **end if**
8: $ts[w] \leftarrow count$

procedure `write`(w):
1: $ts[w] \leftarrow count$

Figure 6. Algorithm for RMS computation and input-sensitive profiling: procedures for processing execution trace events.

- The *partial read memory size* $S[i].rms$ of the activation, defined so that the following invariant property holds throughout the execution:

$$\forall i,\ 1 \leq i \leq top: \qquad \text{RMS}(i) = \sum_{j=i}^{top} S[j].rms \qquad (1)$$

where $\text{RMS}(i)$ denotes the current RMS value of the i-th pending activation on the portion of the execution trace already seen.

It can be proved that Invariant 1 is equivalent to the following equality:

$$S[i].rms = \text{RMS}(i) - \text{RMS}(i+1)$$

i.e., that the partial read memory size maintained by our algorithm is the difference between the RMS of an activation and the RMS of its pending child (if any). Moreover, it follows from Invariant 1 that the RMS and the partial RMS of the topmost routine coincide:

$$\text{RMS}(top) = S[top].rms \qquad (2)$$

In particular, this is true upon completion of a routine, and guarantees that any algorithm able to maintain Invariant 1 will correctly compute the RMS of any routine activation.

Besides the shadow stack S, our algorithm also maintains, for each memory location w, a timestamp $ts[w]$ containing the time of the latest access (read or write) to w. Memory timestamps are initialized to 0 and time is measured by a global counter $count$, that maintains the total number of routine activations.

Algorithm and analysis. The partial read memory size can be maintained more efficiently than the RMS, as shown in Figure 6. To prove the correctness of our approach, in the following we show that all procedures correctly preserve Invariant 1.

When a routine r is invoked (see procedure `call`(r)), the time counter is incremented by 1 and a new shadow stack entry associated with r is appropriately initialized. Upon completion of the routine, its partial RMS is added to the partial RMS of its parent (see line 2 of procedure `return`). It is not difficult to see that this operation preserves Invariant 1. Performance metrics of the completed routine are also collected (line 1): these metrics are associated with the RMS value $S[top].rms$, which at this point coincides with the true RMS (see Equation 2).

Memory access operations on a location w update the timestamp $ts[w]$ with the current counter value (line 1 of procedure `write` and line 8 of procedure `read`). The read operation might also update the partial memory sizes of two distinct routine activations (lines 2 and 5). Namely, if location w has never been accessed by the topmost routine or by any of its completed descendants (condition tested in line 1), then the current access is a first read to w with respect to the topmost routine, whose partial RMS is increased by 1. We have now two cases:

- If the timestamp of location w is still 0 (test at line 3), this is the first access to w in the entire execution of the program, and no other counter needs to be changed. Notice that Invariant 1 is maintained by the execution of line 2, since in this case the RMS of all pending routines increases by 1.
- If the timestamp of location w is not 0, location w has been accessed before during the execution and, in view of the inequality in line 1, the last access happened in some ancestor v of the topmost routine (or in one of v's completed descendants). In this case, the algorithm finds the deepest ancestor that has accessed w (line 4) and decreases its partial RMS by 1: this restores Invariant 1 for all pending activations j such that $1 \leq j \leq i$, whose RMS must not be affected by the current read operation.

The running time of all operations is constant, except for line 4 of procedure `read`. Since the timestamps $S[i].ts$ of routine activations on any call path are increasing, line 4 can be implemented with a binary search and requires $O(\log d)$ time in the worst case, where d is the maximum stack depth during the execution of the program.

We also designed and implemented an asymptotically faster algorithm based on disjoint-sets data structures [40], but in our experiments it was slower and required more space than the solution we presented in this section.

5. Implementation

In this section we discuss our implementation of `aprof` based on the Valgrind [34] framework. Valgrind provides a dynamic instrumentation infrastructure that translates the binary code into an architecture-neutral intermediate representation (VEX). Analysis tools provide callbacks for events generated by the stream of VEX executed instructions.

Instrumentation. While tracing memory accesses is relatively simple in Valgrind, reliably instrumenting function calls and returns is instead rather complex. `aprof` uses a similar approach as the `callgrind` profiling tool [42]: we maintain a shadow runtime stack and cover a wide range of exotic cases to detect function entry/exit events, including jumps to different ELF sections/objects, special handling for dynamically linked functions via `_dl_runtime_resolve`, etc. Our tool takes advantage of the flexible infrastructure of Valgrind and provides full support for multithreaded applications by generating separate profiles for each thread.

Shadow memory. To maintain the timestamps ts of memory cells needed for computing the RMS values, we shadow each memory location accessed by the program with a 32-bit counter. Similarly to the `memcheck` tool [38], we use a two-levels lookup table. The address space is divided in 64 K chunks of 64 KB each (on 64-bit machines, we extend address space coverage to 256 GB). So, the primary table indexes all 64 KB chunks. When an address is referenced by a program, if not already done, we allocate a new secondary table for covering that chunk and we update the primary table for future references. The secondary table contains the set of 32-bit timestamps of addresses covered by the chunk.

Memory tracing resolution: space-accuracy tradeoffs. To reduce the space needed by the lookup table, `aprof` allows users to configure the resolution of distinct observable memory objects, trading space for accuracy. This can potentially impact the number of distinct RMS values observed by `aprof`, and therefore the number of collected performance tuples. We denote by k the size in bytes of the smallest observable objects, which we assume to be aligned to addresses multiple of k. For $k = 1$, we have the finest resolution, shadowing the addresses of all accessed individual memory bytes. For $k = 2$, we trace accesses to 2-bytes words aligned at 16-bit

	TIME					SPACE				
	secs	slowdown				MB	overhead			
	native	mem check	callgrind base	callgrind cache	aprof	native	mem check	callgrind base	callgrind cache	aprof
perlbench	585	34.8	98	178.3	55.4	757	2.1	1.1	1.1	2.1
bzip2	852	11.0	32	90.1	28.7	959	1.3	1.1	1.1	2.0
gcc	523	17.5	47	97.5	33.8	488	2.7	2.1	2.3	3.9
mcf	504	5.8	15	35.1	12.2	1,785	1.3	1.0	1.0	2.0
gobmk	645	28.2	72	135.7	44.8	138	1.8	1.5	1.6	2.1
hmmer	1,153	10.8	23	90.7	27.6	137	1.9	1.4	1.5	1.9
sjeng	798	28.4	94	149.3	45.8	279	1.4	1.2	1.2	1.9
libquantum	790	8.0	31	80.2	21.3	171	1.7	1.5	1.6	2.5
h264ref	903	28.4	85	193.5	67.5	173	1.7	1.4	1.4	2.0
omnetpp	517	16.1	45	82.9	28.8	280	2.8	1.3	1.2	2.0
astar	698	11.8	26	67.1	22.8	444	1.5	1.2	1.2	2.0
xalancbmk	341	35.0	95	171.3	78.3	536	1.8	1.2	1.2	2.3
bwaves	1,239	15.7	17.2	79.1	24.3	982	1.3	1.1	1.1	2.0
gamess	1,161	28.3	42.1	147.4	40.9	748	1.1	1.1	1.1	1.1
milc	520	13.8	12.3	60.9	18.5	795	1.3	1.1	1.1	2.0
gromacs	727	21.1	14.0	76.0	21.5	153	1.5	1.2	1.3	1.5
cactusADM	1.934	14.2	6.4	52.7	17.4	1,111	1.2	1.1	1.1	1.6
leslie3d	745	16.4	12.9	83.2	24.3	231	1.5	1.3	1.3	1.9
namd	640	20.3	15.7	89.5	21.5	161	1.8	1.4	1.4	1.8
soplex	355	11.1	25.6	73.4	26.3	738	1.3	1.2	1.2	2.2
povray	298	41.3	81.0	166.0	55.2	118	2.7	1.6	1.6	1.8
calculix	1,880	24.1	22.1	104.3	27.0	232	1.6	1.9	1.9	2.5
GemsFDTD	610	20.3	10.4	86.8	26.1	937	1.3	1.1	1.1	2.0
tonto	770	31.0	48.1	136.7	42.3	147	2.0	1.5	1.6	2.0
lbm	452	25.4	10.4	85.2	26.6	517	1.3	1.1	1.1	2.0
wrf	996	24.6	27.9	108.4	29.9	809	1.6	1.2	1.2	2.1
sphinx3	800	30.8	39.6	109.5	38.7	159	2.0	1.4	1.4	2.1
geometric mean		19.1	29.4	97.3	30.6		1.6	1.3	1.3	2.0

Table 1. Performance comparison of `aprof` and other Valgrind tools on the SPEC CPU2006 benchmarks.

boundaries, halving the universe of timestamps. The larger k, the smaller the RMS accuracy for routines working on small objects (e.g., strings of characters) and the smaller the size of the shadow memory. By default, $k = 4$ in `aprof`.

Optimizations. `aprof` performs several optimizations at instrumentation time. For instance, we reduce the number of traced memory accesses by coalescing into a single event each pair of load/store events (caused by the same guest instruction) operating on the same object. We also discard all other events related to objects already referenced within the same basic block.

A performance-critical operation is the timestamp search of line 4 in the `read` procedure of Figure 6. As we observed in Section 4.2, implementing this operation with a binary search on the shadow stack guarantees a worst-case bound of $O(\log d)$, where d is the current depth of the stack. However, our experiments revealed that a sequential scan tends to be faster in practice, as on average it performs a very small number of iterations.

Performance metric. We count basic blocks as performance measure: we observed that, compared to running time measurements, this adds a light burden to the analysis time overhead, and improves accuracy in characterizing asymptotic behavior even on small workloads. The choice of counting basic blocks rather than measuring time for studying asymptotic trends has several other advantages, very well motivated in [18].

Other issues. We noticed that calls to dynamically linked libraries made via `_dl_runtime_resolve` generate spurious memory accesses that may introduce some noise in the collected performance tuples. To improve reliability of RMS computations, we handled `_dl_runtime_resolve` as a special case by disabling its cost and RMS propagation to the ancestors.

Finally, to handle overflows of the main routine activations counter ($count$) in long-running applications, `aprof` performs a periodical global renumbering of timestamps in its data structures.

6. Experimental Evaluation

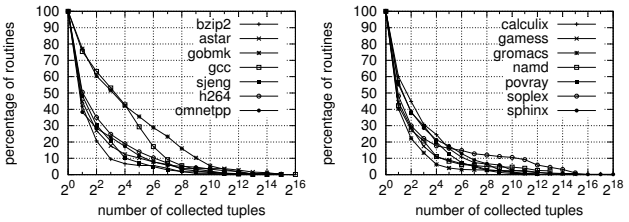
In this section we discuss the results of an extensive experimental evaluation of `aprof` on the CPU2006 benchmarks of the Standard Performance Evaluation Corporation [23]. Our experiments aim both at studying the resources required by our tool compared to other prominent heavyweight dynamic program analysis tools, and at analyzing relevant properties of the input-sensitive profiles generated for the considered benchmarks.

6.1 Experimental Setup

Benchmarks. Our experiments are mainly based on the SPEC CPU2006 v1.1 suite, considering both integer (CINT) and floating-point (CFP) components. All of them were run on the SPEC reference workloads in 64-bit mode. For CINT, we successfully tested all 12 components. In the case of CFP, we omitted `zeusmp` due to a known Valgrind issue, and `dealII` as it failed to terminate on all evaluated tools. We could successfully test all remaining 15 components. For the sake of completeness, we also included in our tests the `wf-0.41` [15] word frequency counter discussed in Section 3 on a 117 MB input text file consisting of the concatenation of several copies of Anna Karenina [22] and protein sequences [17].

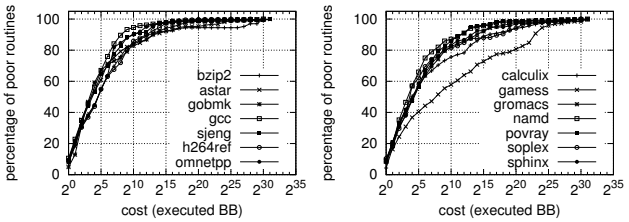
Platform. Experiments were performed on a cluster machine with two nodes, each equipped with eight 64-bit Intel Xeon CPU E5520 at 2.27 GHz, with 48 GB of RAM running Linux kernel 2.6.18 with gcc 4.1.2 and Valgrind 3.7.0 – SVN rev. 12129.

Evaluation metrics. We collected running times reported by `specrun`, the execution engine of the CPU2006 test harness, and the peak virtual memory space required over all tested workloads for each benchmark. We also collected all the profiling reports generated by `aprof` on the CPU2006 suite, measuring relevant parameters such as the number of collected tuples for each routine. Profiled functions covered both the executable binary and all dynamically linked libraries.



(a) integer benchmarks (b) floating-point benchmarks

Figure 7. Percentage of routines for which `aprof` collected at least a given number of performance tuples for a representative set of SPEC CPU2006 benchmarks.



(a) integer benchmarks (b) floating-point benchmarks

Figure 8. Percentage of routines with less than 10 collected tuples and at least a given maximum cost for a representative set of SPEC CPU2006 benchmarks.

Evaluated tools. We compared the performance of `aprof` to two prominent and widely used Valgrind tools: `memcheck` [38], a tool for detecting memory-related errors, and `callgrind` [42], a call-graph generating cache and branch prediction profiler. We considered two settings for `callgrind`: the basic call-graph generating tool and the tool extended with cache simulation, calling them `callgrind-base` and `callgrind-cache`, respectively. Although the considered tools solve different analysis problems, all of them share the same instrumentation infrastructure provided by Valgrind, which accounts for a significant fraction of the execution times: `memcheck` does not trace function calls/returns and mainly relies on memory read/write events; `callgrind-base` instruments function calls/returns, but not memory accesses, while `callgrind-cache` traces both kinds of events like `aprof`. In our experiments with `aprof`, we also considered different values of the memory tracing resolution k discussed in Section 5. Unless otherwise stated, we used `aprof` with $k = 4$.

6.2 Experimental Results

Performance results. Performance figures of our evaluated tools on the SPEC CPU2006 benchmarks are summarized in Table 1. Compared to native execution, `aprof`'s mean slowdown factor is $30.6\times$ ($31.9\times$ on CINT and $27.9\times$ on CFP), with a peak slowdown of $78.3\times$ for `xalancbmk`, an XSLT engine for XML processing. `callgrind-cache`, which is most similar to `aprof` from a tracing perspective, is up to $4.2\times$ slower than `aprof`, with a mean slowdown of $3.2\times$. `callgrind-base` is $1.5\times$ slower than `aprof` on CINT and $1.4\times$ faster on CFP: overall, the two tools deliver comparable performance, even if `callgrind-base` does not trace memory accesses. Compared to `memcheck`, which is the less heavyweight of all considered tools, `aprof` is about $1.6\times$ slower. However, `memcheck` does not trace function calls/returns, which account for a significant fraction of `aprof`'s performance.

The mean memory requirements of `aprof` are within a factor of 2 of native execution, with a peak of $3.9\times$ for `gcc`. This is needed for shadowing accessed memory cells with 32-bit timestamps and for maintaining the performance tuples. Compared to `memcheck`, which also uses a memory shadowing approach, `aprof` requires about 20% more space. However, the current version of `aprof` does not use any shadow memory compression scheme as `memcheck` does, and the amount of generated profile data is higher. Both versions of `callgrind`, which do not use shadow values, require less space than `aprof` and `memcheck` (30% overhead compared to native execution). Performance figures on `wf-0.41` are similar, with a slowdown of $43.1\times$ and a space overhead of $1.9\times$.

Analysis of input-sensitive profiles. Our second set of experiments aims at evaluating some relevant properties of the profiling data generated by `aprof` on the SPEC CPU2006 benchmarks. A first natural question is how many performance tuples can be automatically collected for each routine from a *single* run of a program on a typical workload. Charts in Figure 7 plot the percentage of routines that have at least a given number of tuples for a representative set of programs and workloads from both the integer and floating-point suites. The experiment shows that the results can significantly vary across different benchmarks, ranging from programs where the fraction of routines drops quickly with the number of collected tuples such as `bzip2`, which does most of the work in a handful of hot routines, to larger-scale programs containing a wealth of algorithmic-intensive functions such as `gcc`, with several rich routines having many tuples. For instance, the fraction of routines with at least 10 tuples ranges from a minimum of 7.8% for `bzip2` to a maximum of 49.1% for `gcc`, with an average value of 18.1% for all benchmarks. This observation draws an interesting parallel with the well-known Pareto principle, which accounts most of a program's cost to a small fraction of the routines. Notice that for some functions `aprof` collected hundreds of thousands of tuples. To assess the potential relevance of "poor" routines, for which we may not have enough tuples to produce informative plots, we studied their cost distribution, reported in Figure 8. The chart shows that the *maximum* cost per invocation of most poor functions with less than 10 tuples is small (e.g., 70% of poor functions *never* execute more than 100 basic blocks per invocation), and therefore they are unlikely to be of any asymptotic interest in the context of the application in which they are deployed.

Space-accuracy tradeoffs. As a final experiment, we analyzed the impact of the memory tracing resolution k (see Section 5) on the performance of `aprof` and on the profiles it generated on a representative subset of CINT and CFP benchmarks. Our tests, reported in Figure 9, show that the mean running time for the considered benchmarks is barely affected by varying k, while space usage decreases with k as expected. We also studied how the percentage of routines with at least 10 tuples varies with k, showing that the loss of accuracy for lower resolutions is small compared to the space savings, hence it is reasonable to use $k > 1$.

7. Related Work

Performance profiling. Performance profiling has been the subject of extensive research since the early 70's [27]. In early profilers, performance measurements were associated to isolated syntactic units of a program, such as procedures or statements. The importance of contextual information was recognized early and pioneered by gprof's call graph profiles [19]. Since a single level of context sensitivity may be inaccurate [36, 39], Ammons, Ball, and Larus introduced the calling context tree (CCT), a compact data structure to associate performance metrics with entire paths through a program's call graph [2]. A variety of techniques have later been proposed to reduce the slowdown incurred by CCT-based profilers that work by exhaustive instrumentation, including sampling [16, 21, 43] and bursting [3, 24, 46]. Since the CCT can be

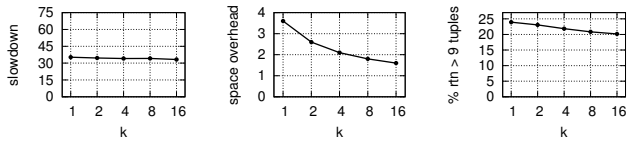


Figure 9. Impact of memory tracing resolution k on time, space and accuracy, as geometric mean computed over SPEC CPU2006 benchmarks `bzip2`, `gobmk`, `astar`, `gcc`, `sphinx3`, and `gamess`.

very large and difficult to analyze, a different set of works targets space issues in context-sensitive profiling [7, 13, 20].

At the intraprocedural level, context information is encoded by path profiles [4], that determine how many times each path in the control flow graph executes. The seminal work of Ball and Larus [4] has spawned much research on flow-sensitive profiling: see, e.g., [2, 5, 6, 26, 32, 37, 41].

All these works on context-sensitive profiling aim at associating performance metrics to distinct paths traversed either in the call graph or in the control flow graph during a program's execution, but do not explore input-sensitivity issues that are the target of this paper. Context- and input-sensitivity represent two orthogonal aspects of program profiles and can be naturally combined. The method proposed by Goldsmith, Aiken, and Wilkerson [18] allows it measure the empirical computational complexity of a program and is much closer in spirit to our work: the program is run on different workloads (possibly spanning several orders of magnitude in size), the performance of its routines is measured, and all the observations are fit to a model that predicts performance as a function of the workload size. Differently from `aprof`, however, the workload size of the program's routines is not computed automatically, but numerical features characterizing the different workloads must be explicitly specified by the user.

Performance prediction. Performance prediction provides means to estimate the running time of a program on different platforms. Profile-based prediction tools, which are most closely related to our work, first run benchmark programs once under lightweight instrumentation tools in order to generate average statistics for a program run, and then feed these statistics to analysis tools that compute an estimate of the run time on a specific machine. While the instrumentation phase runs the entire program, the analysis phase runs in a time roughly proportional to the number of static instructions, which is typically several orders of magnitude smaller than the number of instructions actually executed. It has been shown in [35] that this technique can accurately predict the performance of a detailed out-of-order issue processor model, largely improving over earlier static analysis methods. Differently from our work, the goal in [35] is to predict the performance of the same benchmark on different platform models, and not how the performance scales with the input size: repeated runs of the instrumentation phase on different inputs are necessary to pinpoint scalability issues.

The problem of understanding how an application's performance scales given different problem sizes is addressed in [29], describing a methodology for constructing semi-automatically models of an application's characteristics parameterized by problem size. In [29], data from multiple runs with different and determinable input parameters are first collected and then used to compute a curve parameterized by a parameter related to the problem size. This is quite different from our approach, where the input size of each routine activation is automatically computed by the profiler and the analysis of data extracted from a single program run may be sufficient to determine the growth rate of the routines' performance: with our methodology, we cannot predict how an entire program will scale with different problem sizes, but we can rather automatically discover hidden asymptotic inefficiencies of different program's components.

WCET (Worst-Case Execution Time) analysis tools [44] used in the development of real-time systems also address the problem of estimating execution times depending on the input data. In particular, measurement-based methods need to direct input-data generation in search for worst-case or long program execution times. This is fundamentally different from input-sensitive profiling, which does not aim at identifying worst-case instances, but rather at understanding how the execution cost scales on specific workloads as the input size grows.

Experimental algorithmics. Although worst-case asymptotic analysis provides a strong mathematical framework for the analysis of algorithms, predicting the actual behavior of an algorithmic code may still be a very difficult task [14, 30, 33], since general theoretical models and techniques do not directly fit to actual existing machines and to real-world problem instances. Experimental algorithmics complements and reinforces theoretical analyses with empirical studies aimed at discovering easy and hard instances for a problem and measuring practical indicators (implementation constant factors, locality of references, cache and communication complexity effects) that may be difficult to predict theoretically. The problem of inferring asymptotic bounds from experimental data is rather difficult and there is no sound and generally accepted solution. Some researchers have nevertheless proposed heuristics for the "empirical curve bounding" problem, showing their effectiveness for several synthetic and real datasets [31].

Performance profilers are especially useful in the experimental analysis of algorithmic code. In particular, input-sensitive profiling conjugates asymptotic algorithmic theory with profiling practice, and we believe that it may represent a useful tool in this domain, providing valuable hints to experimenters about asymptotic inefficiencies and typical usage scenarios of critical subroutines. Moreover, papers in experimental algorithmics only rarely analyze algorithms *in situ*, i.e., within the actual context of applications they are deployed in. Our approach makes *in situ* analysis viable (and easy), thus exposing possible performance effects (e.g., a larger number of cache misses) due to the interaction with the overall application.

8. Conclusions

We have proposed a novel approach to performance profiling inspired by asymptotic cost models. The running time of an algorithm is typically estimated by theoretical means as a function relating the size of the input to the number of steps of the algorithm. On the other side, profilers collect runtime information on a single run on a specific input: this kind of information, although useful, does not provide insights on code scalability with respect to the input size. Our profiler relates the measured cost required by the execution of each code fragment to the size of the processed input data. In this way, it can induce an estimated growth rate of the running time, pinpointing scalability problems more precisely than traditional code profiling. We have shown that our approach is both methodologically sound and practical.

Measuring automatically the size of the input given to a generic code fragment raises a variety of interesting questions. Namely, the RMS metric is a measure of distinct accessed memory cells, but does not consider data that are not stored in the process memory at the beginning of a function's execution. For instance, data received on-line (e.g., reads from external devices such as the network, keyboard, or timer) or non-deterministic input values read from I/Os (e.g., input items read in a loop over a file iterator) are not counted in the input size. A possible solution to these issues could hinge upon the techniques for logging non-determinism described in [12]. We regard extending our model towards this direction as an interesting open issue.

Acknowledgements

We would like to thank Randal Bryant and David O'Hallaron for their feedback on the difficulty of discovering asymptotic inefficiencies in large software applications, and to Matthew Hammer for many useful discussions. We are also indebted to Umberto Ferraro Petrillo for hosting our experiments on the Intel Xeon cluster machine described in Section 6.1 and to Bruno Aleandri for developing an earlier version of `aprof`.

This work was supported in part by the Italian Ministry of Education, University, and Research (MIUR) under PRIN 2008TFBWL4 national research project "AlgoDEEP: Algorithmic Challenges for Data-Intensive Processing on Emerging Computing Platforms".

References

[1] An efficient variation of bubble sort. *Information Processing Letters*, 11(1):5 – 6, 1980.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, 1997.

[3] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *PLDI*, pages 168–179. ACM, 2001.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, 1996.

[5] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: the showdown. In *POPL*, pages 134–148. ACM, 1998.

[6] M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizers. In *CGO*, pages 205–216. IEEE Computer Society, 2005.

[7] M. D. Bond and K. S. McKinley. Probabilistic calling context. *SIGPLAN Not. (Proc. OOPSLA 2007)*, 42(10):97–112, 2007.

[8] R. E. Bryant. Personal communication, September 2011.

[9] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson Education, 2010.

[10] R. E. Bryant, D. L. Beatty, K. S. Brace, K. Cho, and T. J. Sheffler. COSMOS: A Compiled Simulator for MOS Circuits. In *DAC*, pages 9–16, 1987.

[11] J. M. Chambers, W. S. Cleveland, B. Kleiner, and P. A. Tukey. *Graphical Methods for Data Analysis*. Chapman and Hall, New York, 1983.

[12] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference*, pages 1–14, 2008.

[13] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining hot calling contexts in small space. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 516–527. ACM, 2011.

[14] C. Demetrescu, I. Finocchi, and G. F. Italiano. Algorithm engineering. *Bulletin of the EATCS (algorithmics column)*, 79:48–63, 2003.

[15] Fedora Project. `wf`: simple word frequency counter. `http://rpmfind.net//linux/RPM/fedora/devel/rawhide/src/w/wf-0.41-6.fc17.src.html`.

[16] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proc. 19th Annual International Conf. on Supercomputing*, pages 81–90. ACM, 2005.

[17] Genome bioinformatics research laboratory. Resources and datasets. `http://genome.crg.es/main/databases.html`.

[18] S. Goldsmith, A. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *ESEC/SIGSOFT FSE*, pages 395–404, 2007.

[19] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler (with retrospective). In K. S. McKinley, editor, *Best of PLDI*, pages 49–57. ACM, 1982.

[20] R. J. Hall. Call path refinement profiles. *IEEE Trans. Softw. Eng.*, 21 (6):481–496, 1995.

[21] R. J. Hall and A. J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proc. Summer 1993 USENIX Technical Conference*, pages 1–19. USENIX Association, 1993.

[22] M. Hart. Gutenberg Project. `http://www.gutenberg.org/`.

[23] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, 2006.

[24] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Proc. 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[25] D. Johnson. A theoretician's guide to the experimental analysis of algorithms. In *Data Structures, Near Neighbor Searches, and Methodology*, pages 215–250. American Mathematical Society, 2002.

[26] R. Joshi, M. D. Bond, and C. Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *CGO*, pages 239–250. IEEE Computer Society, 2004.

[27] D. E. Knuth and F. R. Stevenson. Optimal measurement points for program frequency counts. *BIT*, 13:313–322, 1973.

[28] T. Küstner, J. Weidendorfer, and T. Weinzierl. Argument controlled profiling. In *Euro-Par'09*, pages 177–184, 2010.

[29] G. Marin and J. M. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proc. SIGMETRICS 2004*, pages 2–13, 2004.

[30] C. C. McGeoch. Experimental algorithmics. *Communications of the ACM*, 50(11):27–31, 2007.

[31] C. C. McGeoch, P. Sanders, R. Fleischer, P. R. Cohen, and D. Precup. Using finite experiments to study asymptotic performance. In *Experimental Algorithmics*, LNCS 2547, pages 93–126, 2002.

[32] D. Melski and T. W. Reps. Interprocedural path profiling. In *Proc. 8th Int. Conf. on Compiler Construction*, LNCS 1575, pages 47–62, 1999.

[33] B. M. E. Moret. Towards a discipline of experimental algorithmics. In *Data Structures, Near Neighbor Searches, and Methodology*, pages 197–250. American Mathematical Society, 2002.

[34] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[35] D. Ofelt and J. L. Hennessy. Efficient performance prediction for modern microprocessors. In *SIGMETRICS*, pages 229–239, 2000.

[36] C. Ponder and R. J. Fateman. Inaccuracies in program profilers. *Softw., Pract. Exper.*, 18(5):459–467, 1988.

[37] S. Roy and Y. N. Srikant. Profiling k-iteration paths: A generalization of the ball-larus profiling algorithm. In *CGO*, pages 70–80, 2009.

[38] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*, pages 17–30. USENIX, 2005.

[39] J. M. Spivey. Fast, accurate call graph profiling. *Softw., Pract. Exper.*, 34(3):249–264, 2004.

[40] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[41] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: compactly numbering interesting paths. In *POPL*, pages 351–362. ACM, 2007.

[42] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory access behavior. In *Int. Conf. on Computational Science*, LNCS 3038, pages 440–447, 2004.

[43] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *Proc. ACM 2000 Conf. on Java Grande*, pages 78–87, 2000.

[44] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[45] T. Williams and C. Kelley. Gnuplot: command-driven interactive function plotting program. `http://www.gnuplot.info/`.

[46] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI*, pages 263–271, 2006.