Software Streams

Big Data Challenges in Dynamic Program Analysis

Irene Finocchi

Computer Science Department, *Sapienza* University of Rome
Via Salaria, 113 - 00198 Rome, Italy
`finocchi@di.uniroma1.it`

Abstract. Dynamic program analysis encompasses the development of techniques and tools for analyzing computer software by exploiting information gathered from a program at runtime. The impressive amounts of data collected by dynamic analysis tools require efficient indexing and compression schemes, as well as on-line algorithmic techniques for mining relevant information on-the-fly in order to identify frequent events, hidden software patterns, or undesirable behaviors corresponding to bugs, malware, or intrusions. The paper explores how recent results in algorithmic theory for data-intensive scenarios can be applied to the design and implementation of dynamic program analysis tools, focusing on two important techniques: sampling and streaming.

1 Introduction

In our modern society, software has become ubiquitous in many branches of human activities and has gained an unprecedented level of complexity. This poses many challenges regarding reliability, performance, and scalability on contemporary computing platforms, thus calling for a much deeper understanding of what happens inside a software program than the conventional visibility offered by the program's output. Dynamic program analysis, defined in [1] as "*the analysis of the properties of a running software system*", encompasses the development of techniques and tools for analyzing computer software by exploiting information gathered at runtime. It can be used for a variety of tasks [2], including optimization (profiling, tracing, self-configuration), error detection (testing, assertion checking, type checking, memory safety, leak detection), error correction (runtime data structure repair, protections against security attacks), and program understanding (coverage, call graph construction, invariant detection, software visualization).

Over the past few years, dynamic analysis has emerged as a focused subject aimed at bridging the gap between the complexity-haunted field of formal verification and the ad-hoc field of testing. Being run-time information precise and sensitive to the input data, dynamic analysis can complement and reinforce traditional static analysis techniques, which might be inaccurate in modern object-oriented software systems: since software is often deployed as a collection of dynamically linked libraries or as Java bytecode that is delivered dynamically and on demand, compilers and other programming tools know less and less of the finally executing program. The use of static analysis in such programming tools requires conservative assumptions, which yield analysis results that may be too imprecise to be useful for either program optimization or program understanding tasks. In these contexts, dynamic analysis can enable new powerful techniques that would be impossible to achieve otherwise.

The development of dynamic analysis tools that can successfully assist programmers and software engineers raises issues in a variety of areas, including operating systems, algorithm design, software engineering, and programming languages. In particular, optimizing the performance of dynamic analysis tools is of crucial importance for their effective deployment and usability. For instance, tools that analyze the patterns of memory accesses of a running program, such as memory profilers, debuggers, or invariant checkers, must be able to deal with huge streams of data generated on-the-fly by monitoring traffic on the address bus and data bus at typical rates of several megabytes per second. Two main problems arise in this context. Firstly, since dynamic program analysis routines are inlined with program execution, they can substantially impact system performance, greatly reducing their practical applicability: the cost of collecting runtime information must be therefore appropriately lowered by means of available hardware/software support. Secondly, the sheer size of data collected by a dynamic analysis tool requires on-line techniques for mining relevant information on-the-fly, as well as efficient indexing and compression schemes for storing the data for post-mortem examination.

While optimizing the costs of instrumentation and analysis can largely boost the performance of dynamic analysis tools, it is a very difficult task: modern computer systems must deal with billions of events per second, such as instruction executions, accesses to main memory and caches, or packet forward operations. Hence, execution traces generated from real applications, even from very short runs, can be overwhelmingly large and processing them is very time-consuming. Exploiting advanced algorithmic techniques to cope with the sheer size of data collected throughout execution is thus regarded as a key challenge in this field [3, 4]. In the last few years the design of algorithms and data structures for handling massive data sets has sparked a lot interest in the algorithmic community, but this wealth of novel algorithmic techniques has been explored only to a very little extent in dynamic program analysis. In this paper we will discuss a few relevant examples where big-data algorithmics has provided valuable insights in the design and implementation of dynamic program analysis tools, addressing two important techniques: sampling (Section 3) and streaming (Section 4).

2 Execution Traces

Information collected by dynamic analysis tools is typically expressed in the form of execution traces. Traces can be recorded via different instrumentation techniques at the source code, binary code, or execution environment level [5,

Table 1. Data obtained from execution traces of routine invocations. The number of nodes in the call tree is proportional to the trace length.

Application	|Call graph|	|Call sites|	|Call tree|
amarok	13 754	113 362	991 112 563
ark	9 933	76 547	216 881 324
audacity	6 895	79 656	924 534 168
firefox	6 756	145 883	625 133 218
gedit	5 063	57 774	407 906 721
gimp	5 146	93 372	805 947 134
sudoku	5 340	49 885	325 944 813
inkscape	6 454	89 590	675 915 815
ooimpress	16 980	256 848	730 115 446
oowriter	17 012	253 713	563 763 684
pidgin	7 195	80 028	404 787 763
quanta	13 263	113 850	602 409 403

6], and can contain a variety of information related to, e.g., routine invocations, executions of program statements or basic blocks, memory accesses, and thread operations. The information recorded in a trace clearly depends on software properties that need to be analyzed. With respect to static analysis, execution traces are incomplete, since they capture only a small fraction of all possible execution paths. However, they have the advantage of being extremely precise and sensitive to the input data.

Even traces obtained from short runs of real applications can be extremely large and complex, affecting not only performance and storage of dynamic analysis tools, but also the cognitive load humans can deal with. Consider, as an example, traces of routine invocations, which are especially useful for performance profiling. These traces can be naturally regarded as a stream of tuples containing routine name, call site, event type (i.e., routine enter or exit), and possibly timing information. Table 1 is excerpted from [7] and analyzes a variety of prominent Linux applications, for which traces were obtained from short running sessions of just a few minutes. The table reports the number of distinct routines in a trace (i.e., the number of nodes of the call graph), the number of distinct call sites (i.e., the number of code lines which call a routine), and the number of nodes in the call tree. The call graph and the call tree are fundamental data structures that maintain information about interprocedural control flow: in a call graph, nodes represent routines and arcs caller-callee relationships, while each node of a call tree represents a different routine invocation. The number of call tree nodes is thus proportional to the stream length. Table 1 shows that the number of call tree nodes can be very large, even when compared with call graph nodes and call sites: execution traces obtained from short runs of real applications produce a few hundred millions of events, which result in a few GigaBytes of memory under the optimistic assumption that each stream tuple can be stored using only ten bytes. To mitigate this size explosion issue, many trace simplification and abstraction techniques have been proposed in the literature, aimed at extracting high-level views and relevant data from long raw traces: execution traces can indeed contain several repetitions, either contiguous or not, and a very large number of patterns. Each pattern, in turn, can have thousands of occurrences, which makes data mining and pattern detection techniques quite useful to understand the characteristics of program traces [4]. Redundancies can be also reduced by compression techniques as proposed, e.g., in [8–10]. In the rest of this paper we will describe some relevant trace analysis techniques based on sampling and data stream algorithmics.

3 Sampling

Sampling is used in statistics to estimate the characteristics of a large population by analyzing only a small portion of its members. A sample typically represents a subset of the population of manageable size, thus allowing faster data collection and smaller analysis costs. Many previous works, such as [11–16], have explored the use of sampling to reduce the size of execution traces and/or the runtime overhead of dynamic analysis tools. Overall, sampling appears to be a valuable tool in dynamic analysis, although sampled traces are not always representative of the original ones, and the results often heavily depend on manual tuning of a variety of parameters. Furthermore, it has been observed that the same sampling parameters might work well for one trace, while being inappropriate for a different trace [12].

Fixed rate sampling. A widespread approach consists of selecting sample points at fixed intervals, e.g., one point out of n trace items or every t milliseconds. This technique might be easily biased when the original trace exhibits regular patterns. Consider, for instance, the following scenario: the execution trace stores memory accesses, the sampling distance n is set to 10, and a memory location ℓ is accessed exactly every 10 memory operations. Then, depending on where the sampling starts, the traced sample might never contain ℓ or might contain exclusively operations on ℓ. Though this is a worst-case example, such regularities in execution traces are far from being rare.

A case study in performance profiling. Fixed rate sampling can be naturally implemented using periodic timer interrupts, ranging from process level interrupts to processor hardware performance counters. Hence, it has become very popular in the design of performance profilers. In accordance with the well-known Pareto principle (also known as the $80 - 20$ rule), more than 80% of running time is indeed spent in only 20% of routines: “hot” routines must appear frequently in the trace, and therefore are likely to be sampled. Unfortunately, even for such skewed distributions, fixed rate sampling might not work properly if the sampling parameter (n or t) is not appropriately tuned. As a concrete example, we report the outcome of an experiment discussed in [17], aimed at comparing the set of hot routines returned by a sampling-based profiler with the set of hot routines obtained by full instrumentation (i.e., computed on the full execution

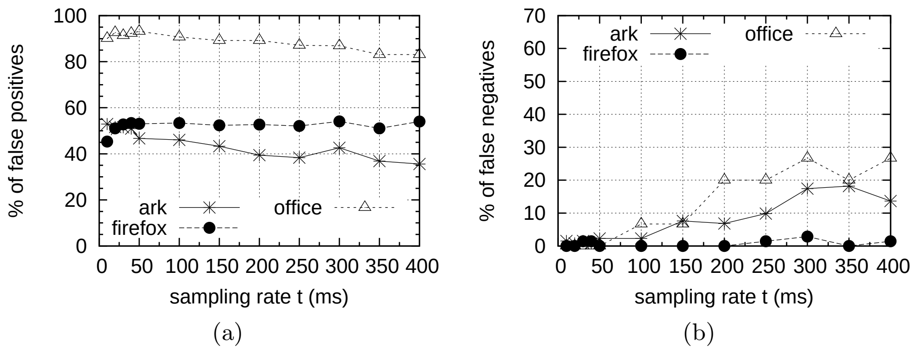

Fig. 1. False positives and false negatives generated by fixed rate sampling. (a) A routine is a false positive if it appears to be hot with respect to a sampled trace, but is not hot with respect to the full trace; (b) a routine is a false negative if is not reported as hot when using sampling, but is hot with respect to the full trace. Graphs are excerpted from [17].

trace). Let τ be a trace of routine invocations and $T(\tau)$ be the total time required by all the routines appearing in τ. We define the set $H(\tau)$ of routines that are hot with respect to trace τ as follows: all routines appearing in τ are sorted by decreasing running time and are then progressively added to $H(\tau)$, according to the precomputed order, until the total time required by routines in $H(\tau)$ becomes larger than $0.9\,T(\tau)$. Intuitively, $H(\tau)$ is the minimal set of the most costly routines that account for at least 90% of the overall running time. When τ is a sampled trace, some of the routines that are hot with respect to τ could not be hot with respect to the full trace, yielding false positives. Symmetrically, it may happen that routines that are hot with respect to the full trace are not hot with respect to the sampled trace τ, yielding false negatives. Figure 1 reports the percentage of false positives and false negatives as a function of the sampling rate t (larger values of t imply less frequent sampling and thus smaller sampled traces). The outcome of the experiment is exemplified on some of the benchmarks listed in Table 1. Both quantities are considerably large: in some applications, false positives account for up to 90% of the total number of reported hot routines. The quantity of false negatives is smaller, but remains non-negligible and is badly affected by larger sampling rates. Further details are given in [17].

Random sampling. Overall, the experiments reported in [17] confirm that fixed rate sampling can yield unrealistic results, even when data distributions are very skewed. A valuable tool to mitigate these issues is random sampling, which consists of selecting sample points with a fixed probability. Mytkowicz *et al.* [18] observe that collecting samples randomly is a fundamental requirement to obtain accurate profiles, but is often violated by commonly-used Java profilers. Unfortunately, random sampling might be difficult to implement on-line (i.e., during trace generation) and, if not done properly, might result in samples of unbounded size for long running applications. To overcome these issues, Coppa *et al.* [17] advocate the use of reservoir sampling [19]: the experimental evaluation of reservoir-based profiling shows that, while maintaining uniform sampling probability, this technique yields much better and more stable profiling results than fixed rate sampling, even when the stored sample is very small.

Adjusting sampling probabilities. A variety of works propose different strategies for controlling sampling probabilities. For instance, Marino, Musuvathi, and Narayanasamy apply sampling to the problem of detecting data races in concurrent applications [20]. Apparently, a sampling-based data-race detector may seem unlikely to succeed, because most memory accesses do not participate in data races and sampling approaches, in general, are not well suited at capturing rare events. Hence, [20] proposes an adaptive approach, adjusting sampling probabilities during the execution so that infrequently accessed regions of code progressively become more likely to be sampled. Experimental results show that this adaptive strategy achieves both a high detection rate and small slowdown on the running time. In [15], Pirzadeh *et al.* use stratified sampling to create samples representative of different characteristics of the entire execution. Stratified sampling turns out to be useful on heterogeneous populations that can be divided into homogeneous sub-populations, known as strata: this is often the case in execution traces, where the sequence of trace events can be typically partitioned into subsequences representing specific tasks performed by the software system. In this scenario, strata correspond to execution phases, which can be automatically identified using k-means clustering algorithms, and different sampling parameters can be then used in each stratum.

4 Streaming

The data streaming model has gained increasing popularity in the last few years as an effective paradigm for processing massive data sets. Streaming algorithms are well suited in application domains where input data come at a very high rate, cannot be stored entirely due to their huge (possibly unbounded) size, and need to be continuously monitored in order to support exploratory analyses and to detect correlations, frequent or rare events, fraud, intrusion, and anomalous activities. Relevant examples include monitoring network traffic, online auctions, transaction logs, telephone call records, automated bank machine operations, atmospheric and astronomical events. For instance, in IP traffic analysis we may want to monitor the packet log over a given link in order to estimate how many distinct IP addresses used that link in a given period of time: since the number of packets may be very large and stream items (source-destination IP address pairs) are drawn from a large universe, a space-efficient data streaming algorithm maintains a compact data structure supporting both dynamic updates upon arrival of new packets and distinct items queries. Approximate answers are allowed when it is impossible to obtain an exact solution using only one sequential pass over the data and limited space.

One-pass streaming algorithms are typically designed to optimize four main performance measures: space required to store the data structure, update time (i.e., per-item processing time), query time, and guaranteed solution quality. Starting from early papers appeared in the late 1970s (see, e.g., [21, 22]), a wide range of results have been obtained in the last decade, mainly for statistics and data sketching problems such as the computation of frequency moments [23], histograms and quantiles [24], norm estimation [25], wavelet decomposition [24], most frequent items [26], and clustering [27]. In this section we discuss two applications of streaming algorithms to dynamic program analysis, focusing on the problem of performance profiling.

Range adaptive profiling. In [28], Mysore *et al.* address a problem called profiling with adaptive precision: they devise a profiling methodology capable of hierarchically classifying items in an execution trace into increasingly precise categories based on the frequency with which they occur. Differently from traditional profiles, which produce a flat list of items together with their performance metrics, adaptive profiling outputs profile data into a hierarchical fashion, striving for higher precision on most frequent events. Assume, as an example, that items of interest are lines of code: if 90% of the running time is spent on the top half of the code, according to Amdahl's law fine-grained profile data on the bottom half would not be very useful. Hence, it makes sense to summarize the behavior of the bottom half using a single performance counter, exploring in more detail possible optimization targets in the top half: the top half could be divided in turn into a top and a bottom quarter, refining data on the quarter that consumes most of the time. In summary, in [28] profile data is grouped into ranges: the most frequently occurring ranges are broken down into more precise subranges, while the least frequently occurring events are kept as larger ranges.

Ranges can be naturally stored in a tree, together with their associated counters. The tree can be easily updated by incrementing the appropriate counter to keep track of stream events. However, since relative range frequencies can dynamically change over time, the tree structure must be also adaptively changed to resemble the hottest ranges. The solution proposed in [28] exploits a streaming algorithm for adaptive spatial partitioning of multidimensional data streams described in [29]. When a range becomes sufficiently hot, the corresponding tree node is split into subranges. Symmetrically, ranges that get colder are merged together, pruning the tree in order to maintain the least number of relevant counters. Tree update, split, and merge operations can be performed on-line and are designed so as to guarantee worst case bounds on precision and space usage.

Let $[0, R]$ be the largest range to be considered (in the example above, R might denote the number of code lines), let $\varepsilon \in (0, 1)$ be a user defined constant, and let n be the number of stream items at any time during the execution. The streaming data structure of [29] splits nodes whenever their counter becomes larger than the threshold $\varepsilon \cdot n / \log(R)$. It can be proved that this guarantees $O(\log R/\varepsilon)$ size, independently of the length of the stream, and maximum error upper bounded by $\varepsilon \cdot n$. We notice that the error for any given range is relative to the entire input stream, and not to the actual counter of that range.

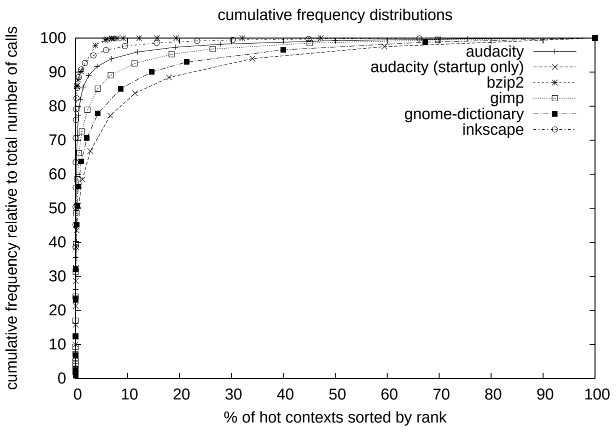

Fig. 2. Skewness of calling context distribution (graphs excerpted from [7]).

In [28], the authors show that this streaming approach can be efficiently implemented via specialized hardware. Experimental results indicate that, using just a few kilobytes of memory, it is possible to maintain range profiles with an average accuracy of 98%. We remark that range adaptive profiling is not a fully general technique, but can be nevertheless applied to a variety of scenarios, such as profiling segments of code, blocks of data and IP addresses, or ranges of memory addresses in order to quantify cache locality issues.

Mining hot calling contexts. In [7], D'Elia *et al.* show an efficient and accurate solution for context sensitive profiling based on the computation of frequent items in the data stream model. Calling contexts are typically stored in a data structure called *calling context tree*, which is substantially smaller than standard call trees, but may still be very large and difficult to analyze in real applications. However, only the most frequent contexts are of interest, since they represent the hot spots to which optimizations must be directed. Context frequency distribution satisfies the well-known Pareto principle: Figure 2, excerpted from [7], shows on a variety of real applications that only a small fraction of contexts are hot, and typically more than 90% of routine calls take place in only 10% of calling contexts. This skewness suggests that space could be greatly reduced by keeping information about hot contexts only, discarding on the fly contexts that are likely to be cold. This is the approach taken in [7], where the problem of identifying the most frequent contexts on the fly is cast into a data streaming setting, exploiting fast and space-efficient algorithms for mining frequent items.

Given a frequency threshold $\phi \in [0,1]$ and a stream of lenght N, the frequent items problem is to find all items that appear in the stream at least $\lfloor \phi N \rfloor$ times. Since computing an exact solution requires $\Omega(N)$ bits, even using randomization [30], research focused on solving an approximate version of the problem, called (ϕ, ε)-*heavy hitters*: given two parameters $\phi, \varepsilon \in [0,1]$, with $\varepsilon < \phi$, return all items with frequency $\geq \lfloor \phi N \rfloor$ and no item with frequency $\leq \lfloor (\phi - \varepsilon)N \rfloor$. In the approximate solution, false negatives cannot exist, i.e., all frequent items must be returned. Instead, false positives are allowed, but their real frequency must be guaranteed to be at most εN-far from the threshold $\lfloor \phi N \rfloor$. Different algorithms for computing (ϕ, ε)-heavy hitters are known in the literature. Among them, *Space Saving* [31] and *Sticky Sampling* [26] are counter-based algorithms that track a subset of the input items, monitoring counts associated with them. For each new arrival, they decide whether to store the item or not, and, if so, what counts to associate with it. Sticky Sampling is probabilistic: it fails to produce the correct answer with a minuscule probability, say δ, and uses at most $\frac{2}{\varepsilon} \log(\phi^{-1}\delta^{-1})$ entries in its data structure [26]. Space Saving [31] is instead deterministic and uses $\frac{1}{\varepsilon}$ entries.

Frequent items algorithms can be naturally adapted to context sensitive profiling: during the computation the profiler maintains a subtree of the full calling context tree, called *Hot Calling Context Tree (HCCT)*, storing only hot contexts and their ancestors. More formally, the HCCT is defined as the (unique) subtree of the calling context tree obtained by pruning all cold nodes that are not ancestors of a hot node: by definition all hot nodes are included in the HCCT, whose leaves are necessarily hot (the converse, however, is not true). The frequent items algorithms decide which hot nodes must be monitored, and the profiler updates the HCCT accordingly so as to maintain also their ancestors. The space used by the HCCT includes both monitored hot contexts, and their (possibly cold) ancestors. The former quantity can be analyzed theoretically, as in [26, 31], while the latter depends on properties of the execution trace and on the structure of the calling context tree. In practice, experiments show that this amount is negligible with respect to the number of hot nodes. Hence, the HCCT represents the hot portions of the full calling context tree very well using only an extremely small percentage of space: even when the peak memory usage of the stream-based profiler of [7] is only 1% of standard context-sensitive profilers, all the hottest calling contexts are always identified correctly (no false negatives), the number of false positives (cold contexts that are considered as hot) is very small, and frequency counters are very close to the true values.

5 Concluding Remarks

In this paper we have discussed how recent results in algorithmic theory for data-intensive scenarios can be applied to the design and implementation of dynamic program analysis tools. We have focused on sampling and data stream algorithmics. The examples illustrated in this work should be considered as a non-exhaustive starting point: many other techniques (e.g., data mining or compression) may prove to be valuable in dynamic program analysis. This is indeed a fresh area, and we believe that it represents a novel fertile ground for fundamental algorithmic research: not only dynamic program analysis can inspire many novel, interesting algorithmic questions (or genuinely new variants of well understood problems), but the transfer of algorithmic knowledge in the implementation of program analysis tools can also have a significant practical impact. Furthermore, algorithm engineering techniques for developing fast, robust, and scalable implementations can play a major role in this scenario, where architectural aspects, such as the presence of memory hierarchies and of multiple cores, can be successfully exploited in order to leverage the runtime impact of dynamic analysis tools.

References

1. Ball, T.: The concept of dynamic analysis. In: Software Engineering (ESEC/FSE 1999). Volume 1687 of LNCS. (1999) 216–234
2. Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R.: A systematic survey of program comprehension through dynamic analysis. IEEE Transactions on Software Engineering **35**(5) (2009) 684 –702
3. Finkbeiner, B., Havelund, K., Rosu, G., Sokolsky, O.: Runtime verification, dagstuhl sem. 07011 executive summary. Technical report (2007)
4. Hamou-Lhadj, A., Lethbridge, T.: Measuring various properties of execution traces to help build better trace analysis tools. In: 10th IEEE Int. Conference on Engineering of Complex Computer Systems. (2005) 559 – 568
5. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI 2005). (2005) 190–200
6. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI 2007). (2007) 89–100
7. D'Elia, D.C., Demetrescu, C., Finocchi, I.: Mining hot calling contexts in small space. In: Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011), ACM (2011) 516–527
8. Larus, J.R.: Whole program paths. In: ACM SIGPLAN Conference on Programming language design and implementation (PLDI'99), ACM (1999) 259–269
9. Nevill-Manning, C.G., Witten, I.H.: Compression and explanation using hierarchical grammars. The Computer Journal **40**(2/3) (1997) 103–116
10. Nevill-Manning, C.G., Witten, I.H.: Linear-time, incremental hierarchy inference for compression. In: 7th Data Compression Conference (DCC'97), IEEE Computer Society (1997) 3–11
11. Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. SIGPLAN Not. **36**(5) (2001) 168–179
12. Chan, A., Holmes, R., Murphy, G.C., Ying, A.T.T.: Scaling an object-oriented system execution visualizer through sampling. In: 11th Int. Workshop on Program Comprehension (IWPC 2003), IEEE Computer Society (2003) 237–244
13. Dugerdil, P.: Using trace sampling techniques to identify dynamic clusters of classes. In: Conference of the Center for Advanced Studies on Collaborative Research (CASCON '07), IBM Corporation (2007) 306–314
14. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003), ACM (2003) 141–154
15. Pirzadeh, H., Shanian, S., Hamou-Lhadj, A., Alawneh, L., Shafiee, A.: Stratified sampling of execution traces: Execution phases serving as strata. Science of Computer Programming (2012) In press.

16. Zhuang, X., Serrano, M.J., Cain, H.W., Choi, J.D.: Accurate, efficient, and adaptive calling context profiling. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006), ACM (2006) 263–271
17. Coppa, E., Finocchi, I., Lo Re, D.: Reservoir profiling. Unpublished manuscript (January 2013)
18. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F.: Evaluating the accuracy of Java profilers. In: Proc. 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10). (2010) 187–197
19. Vitter, J.S.: Random sampling with a reservoir. ACM Trans. Math. Softw. **11**(1) (1985) 37–57
20. Marino, D., Musuvathi, M., Narayanasamy, S.: Literace: effective sampling for lightweight data-race detection. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI'09). (2009) 134–143
21. Morris, R.: Counting large numbers of events in small registers. Comm. ACM **21**(10) (1978) 840–842
22. Munro, J., Paterson, M.: Selection and sorting with limited storage. Theoretical Computer Science **12**(3) (1980) 315 – 323
23. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. Journal of Computer and System Sciences **58**(1) (1999) 137 – 147
24. Gilbert, A.C., Guha, S., Indyk, P., Kotidis, Y., Muthukrishnan, S., Strauss, M.J.: Fast, small-space algorithms for approximate histogram maintenance. In: Proceedings of the 34th annual ACM symposium on Theory of Computing. (2002) 389–398
25. Indyk, P.: Stable distributions, pseudorandom generators, embeddings, and data stream computation. J. ACM **53**(3) (2006) 307–323
26. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Proceedings of the 28th international conference on Very Large Data Bases. (2002) 346–357
27. Charikar, M., O'Callaghan, L., Panigrahy, R.: Better streaming algorithms for clustering problems. In: Proceedings of the 35th annual ACM symposium on Theory of computing. STOC '03 (2003) 30–39
28. Mysore, S., Agrawal, B., Sherwood, T., Shrivastava, N., Suri, S.: Profiling over adaptive ranges. In: 4th IEEE/ACM Int. Symposium on Code Generation and Optimization (CGO 2006), IEEE Computer Society (2006) 147–158
29. Hershberger, J., Shrivastava, N., Suri, S., Tóth, C.D.: Adaptive spatial partitioning for multidimensional data streams. In: 15th Int. Symposium on Algorithms and Computation (ISAAC 2004). Volume 3341 of LNCS. (2004) 522–533
30. Muthukrishnan, S.: Data streams: Algorithms and applications. Foundations and Trends in Theoretical Computer Science **1**(2) (2005)
31. Metwally, A., Agrawal, D., Abbadi, A.E.: An integrated efficient solution for computing frequent and top-k elements in data streams. ACM Trans. Database Syst. **31**(3) (2006) 1095–1133