Low-Overhead Profiling based on Stationary and Ergodic Assumptions

Stoyan Garbatov and João Cachopo
Software Engineering Group
INESC-ID Lisboa / Instituto Superior Técnico, Universidade Técnica de Lisboa
Lisbon, Portugal
stoyangarbatov@gmail.com and joao.cachopo@ist.utl.pt

***Abstract*—In the context of feedback-directed optimization solutions, the component responsible for collecting application behaviour data cannot afford to introduce any performance overheads, otherwise it undermines any optimization that is to be carried out. This work presents a new online solution for profiling object-oriented applications. The solution collects detailed information about accesses over domain instances and their fields, while introducing approximately zero overheads. This is accomplished by making assumptions about the stationary and ergodic properties of applications' run-time behaviour. The work has been validated with the TPC-W benchmark.**

Keywords-profiling; real-time monitoring; feedback-directed optimizations; performance; ergodic; stationary.

I. INTRODUCTION

The importance of profiling tools has been steadily increasing over the last decade. Profilers are essential for understanding the dynamic behaviour of programs. In the past, one of the most common use-case scenarios was to use a profiler for obtaining information about the resource usage of a given application, to identify performance bottlenecks. While the nature of the information supplied by profiling tools has not changed much over time, the spectrum of their possible applications has observed a significant widening. These applications include dynamic slicing, program invariants detection, program correctness and security checking, predicting data locality and just-in-time compiler optimizations, among others.

The widespread adoption of programming languages designed to execute on top of virtual machines played a major role in valorising profiling tools. Virtual machine architectures offer several properties that make them more desirable, from a software engineering point of view, than environments with statically compiled binaries. Some of these features include program portability, safety assurances, automatic memory and thread management, as well as dynamic class loading. As it can be seen from the work of Cierniak et al. [1], while these properties empower the programming model offered to users, they also cause overheads and contribute to obstruct many static program optimization techniques, making it harder to achieve good performance.

To counter these difficulties, a lot of research has been carried out, focusing on online feedback-directed optimization systems. These systems make use of techniques that seek to improve the performance of target programs by monitoring their run-time behaviour and, subsequently, by using this information for identifying and applying appropriate optimization measures. It is frequently the case to employ profiling tools for obtaining the necessary program behaviour information.

The main issue of profiling tools is that they are subject to two requirements that are practically impossible to satisfy simultaneously. The first requirement is that the profiler should provide as in-depth and detailed information as possible about the behavioural patterns exhibited by the program being monitored. The second requirement is that the monitoring should be carried out in a transparent, efficient and devoid of overheads manner that does not compromise program performance (or at least not significantly).

Independently of the way that a profiler operates (event or sampling-based), it invariably ends up disrupting the execution of its target programs, while data is being collected. These interruptions translate directly into overheads that penalize application performance. Furthermore, the fact that many profilers employ code instrumentation (the injection of additional code into the target) for achieving their goals not only slows down the execution of the program but can also change the way it operates, leading to scenarios where the application displays behaviour that would be otherwise impossible to observe, were it executing normally.

All of these factors contribute to make the task of striking an acceptable balance between performance overheads and information depth hard to achieve in practice. This is particularly true for online feedback-directed optimization systems, where the profiling is expected to be carried out in real-time, while the program is executing normally, so any "noticeable" performance overheads are not acceptable. On the other hand, the profiling information has to be sufficiently detailed to guide appropriately the optimization decisions that need to be made for improving the target system's performance.

While accounting for these considerations, the solution presented with this work consists in a system capable of monitoring the access patterns of object-oriented applications. The patterns are expressed in terms of the manipulations (read and write access operations) performed over domain instances and their fields in the execution contexts of the application methods/services that define the set of functionality offered to end-users. The novelty of the approach consists in the employment of stochastic and ergodic assumptions for the behaviour of target programs within their methods/services, making it possible to collect the access pattern information in an online fashion with minimal performance overheads while providing very high accuracy and data depth.

The article is organized as follows. Section II discusses related work. Section III describes the motivation behind our new proposal, followed by some assumptions that were necessary to be made in Section IV. Section V covers the implementation of the solution presented here. Section VI presents the benchmark that we used to evaluate the new proposal and discusses the results obtained. Finally, Section VII presents some concluding remarks.

II. RELATED WORK

The current trends in application development, software engineering, and hardware technology (such as the wide acceptance of programming languages operating in automatically managed virtual machine environments and the expansion of cloud computing, among many others) have contributed to create a great demand for solutions capable of continuously tracking the behaviour of dynamic systems and of applying performance optimization measures based on the gathered information. Adaptive optimization solutions such as these have been designated in the literature as online feedback-directed optimization (FDO) systems. As can be seen from the work of Arnold et al. [2], where an analysis of over 150 references related to online feedback-directed optimization solutions has been performed, there is a wealth of work done in this research area. Unfortunately, the great majority of these solutions have been developed to act as support for compilation and dynamic code generation optimizations, while non-compiler related literature is somehow scarce. Without intending to perform an in-depth and exhaustive analysis of the existing literature, some works on the topic of continuously tracking the behaviour of software systems shall be discussed next.

Smith [3] discussed the motivation and history of FDO techniques. The author presented three factors responsible for the importance of FDO, namely:

- FDO bypasses the restrictions imposed on static optimization approaches by making use of dynamic run-time behaviour information that is impossible to obtain statically;
- FDO makes it possible to adapt the optimization measures continuously, according to the observed changes in target application behaviour.
- Software systems can be made more flexible and easier to change through run-time binding.

From the analysis and discussion in [3], Smith argues in favour of performing optimizations based on run-time monitoring as well as accepting the notion of executables as mutable objects. Smith [3] and Arnold et al. [4] pointed out that the obstacles that need to be overcome to achieve an effective FDO are:

- Minimize or otherwise deal with the overhead introduced in the process of collecting behaviour information as well as when applying the necessary transformations over the target application for optimizing its performance;
- Being able to make informed decisions even when there is incomplete profiling data or that same information is subject to constant evolution.

In the context of virtual machine environments, it is possible to group the profiling-data collection mechanisms for FDO purposes into the following categories: run-time service monitoring, hardware performance monitors, sampling, and program instrumentation. The solution developed here belongs to the program instrumentation class of approaches. As such, the other categories shall be only referred to briefly.

Run-time service monitoring approaches track the state of the run-time services offered by the subjacent virtual machine. This is usually done for identifying temporal locality usage patterns that can be exploited for optimizations. Several applications of these techniques include dynamic dispatching, hash-codes, and synchronization. It is noteworthy that the memory management systems are particularly rich sources of data for FDO, covering information about allocation trends, heap usage and garbage collection.

Hardware performance monitoring collects data provided by specialized microprocessor hardware for guiding optimizations. There has been a multitude of FDO approaches developed to use such information but their integration into production-ready VMs has been limited.

With sampling approaches, the profiler seeks to collect a representative (as opposed to exhaustive) sub-set of observations for a given category of events. By varying the portion of events that get observed, sampling approaches can control the amount of overhead being introduced into the application that is being monitored. Nevertheless, as in all monitoring approaches, sampling techniques need to strike a balance between low overhead and collecting enough behaviour information to be considered useful.

The injection of extra instructions for collecting behaviour information into the target system is the basis for program instrumentation profiling approaches. These techniques are very flexible in their usage and can provide a wide range of behaviour data. Their main issue consists in the performance overhead caused by the need to execute the instrumented code. As such, most existing solutions attempt to minimize the overheads without compromising too much the depth of the profiled information.

Arnold and Ryder [5] developed a framework for low overhead instrumentation sampling supporting multiple types of profiling. The framework employs code duplication and compiler-inserted counter-based sampling to enable changes between the instrumented and original version of the target code at run-time. The amount of overhead to be introduced is adjustable at run-time, by varying the ratio of execution between instrumented and non-instrumented versions of the code. The authors achieve this by keeping in memory two versions of all methods that have been modified for profiling purposes. One of them is the instrumented version that performs the monitoring measurements while the other contains the original code version with a small preamble that determines if the fast or slow version of the method should be executed, depending on the current conditions.

Another software instrumentation system, called Pin, is the one developed by Luk et al. [6]. The tools offered by the system are written in C/C++ and support portable, transparent and efficient instrumentation. While the tools are mostly architecture independent, they can provide architecture specific information when required. The systems uses dynamic compilation for instrumenting applications at run-time. Pin employs several techniques for achieving high operation efficiency while collecting data. These include register re-allocation, inlining, liveness analysis and instruction scheduling. The authors evaluate their systems against other similar purpose solutions, such as Valgrind [7] and DynamoRIO [8] and demonstrate it is capable of offering better performance while providing similarly detailed levels of information.

III. MOTIVATION

The solution presented here was developed for the purpose of supplying the input data necessary for the access pattern analysis and prediction techniques developed in [9-12]. These techniques use stochastic models for analysing and predicting, with a high degree of confidence, the domain data access patterns performed by target object-oriented applications. The models employed for this purpose are Bayesian Inference, Criticality Analysis and Markov Chains. The issue that was detected with these techniques resides in the performance overheads introduced by the process of collecting the input data necessary for the stochastic models. While the overhead is not very significant when evaluated in a single-threaded environment, where the average performance loss is in the order of 3% to 8%, when considered in a multi threaded environment, the performance loss observed is about 50%, which is absolutely unacceptable for any sort of real-time continuous application behaviour monitoring. Since the above techniques had been developed with the intent of supplying with information online feedback-directed performance optimization solutions, it is mandatory that they do not incur any noticeable performance overhead, otherwise their usefulness is compromised.

The new monitoring solution presented here was designed to deal with this issue. The reasoning behind the solution is as follows. It is very hard, if not impossible, to model and predict accurately the behaviour of an application as a whole, over long periods of time. The workload of (most) applications evolves continuously, as a function of external stimuli (such as client requests) and, apart from very specific scenarios or relatively short periods of time, it is impossible to know, a priori, the sequence of inputs/client-requests that will be issued at a given moment. This is what makes programs behave as non-stationary processes.

The stationarity of a process can be pictured intuitively as the absence of any drift in the set of realisations that defines its behaviour as time proceeds. From a mathematical point of view, this means that the probability distribution and density functions that describe such a process are unchanged by a shift in the time scale. They are applicable now and will remain so for all time.

The constantly evolving workload of applications makes it necessary to monitor them continuously, if a precise view of their behaviour is to be had. Furthermore, the monitoring has to be performed in a lightweight manner, otherwise it will introduce inadmissible performance overheads. The combination of these two factors demands an access-pattern analysis solution capable of delivering detailed information about application behaviour, which is expressed in terms of domain data manipulations being performed, without compromising program performance.

IV. ASSUMPTIONS

Several assumptions had to be made to reach a viable solution that achieves these goals. The first assumption is that the workload of an application can be described entirely by the ratio of the invocation frequencies of the methods/services that define the functionality offered by that particular application to end-users.

The second assumption is that programs behave as stationary processes [13] within the execution contexts of their methods/services. The nature of the behaviour displayed when executing a particular method should not change significantly over time, as long as its implementation remains the same. There are several factors that make this reasonable to assume. The encapsulation and modularity properties observed in (well-designed and implemented) object-oriented applications allow their methods to display functionality that is well defined and contained. This makes it highly unlikely to observe a broad range of different access-pattern behaviours when executing a particular method, independently of small shifts and variations that can occur when operating over different arguments.

The third and last assumption is that the operation of application methods displays ergodic properties. A process is ergodic [14] if it is stationary and, furthermore, if it is possible to extract its statistical descriptors from realizations that cover a single finite period of time. Intuitively, it may be said that the realisations obtained from this time period are "typical" of all the possible realisations, if the process is to be ergodic. In practice, it is not necessary for the methods to be strongly ergodic. It is enough to be able to extract the behaviour descriptors from a finite number of observations. This translates into being able to extract the typical access-pattern behaviour of a method from a limited number of invocations.

V. IMPLEMENTATION

A. Compile-time

There are two main components of the solution presented here - a code injection module and a data acquisition module. The first module employs the ASM byte-code manipulation toolkit [15] for injecting, at compile-time, code into target applications. This code invokes functionality in the data acquisition module. In particular, the injected code serves two purposes. The first consists in updating the information about changes in the execution context within which operations are taking place. By manipulating byte-code, the first instruction of all application methods or services (of interest) is defined to invoke a method responsible for updating the profiler state information that a new execution context has been initiated. Similarly, the last instruction before returning (or otherwise terminating the current execution context) calls functionality that clears up the profiler state information about the no-longer-active context.

The second task of the code injection module is to replace all accesses to domain instances (and their fields) with the invocation of a distinct method, depending on the type of access being replaced (read or write operation). This method is responsible for resolving the surrounding execution context as well as for updating the statistical information about the domain data access that is to be performed within the context that has been identified.

B. *Run-time*

The second module is responsible for collecting the behaviour data displayed by the target application, in terms of the domain data access patterns performed while it is executing. To obtain high-precision data without introducing performance overheads, the gathering is performed in two stages - a learning period and a steady-state period.

The first stage defines a period during which the monitoring system builds a detailed profile for all methods and/or services (of interest) of the target application. Each of the profiles assembled in this stage contains the typical access-patterns that are performed, per invocation of the corresponding method or service. The patterns are described in terms of the frequency of accesses performed over domain instances and their fields. Consequently, at the end of this stage, the information collected within the profiles indicates the number of read and write operations that are usually observed to be performed over domain data types when executing the associated methods/services (e.g. MethodY {DomainDataA.Field2 = 54 reads; DomainDataC.Field7 = 17 writes}, MethodX {DataD.Field3 = 7 reads}, etc.).

While the target application is executing in profile-building mode, it operates with an enriched version of its byte-code that contains the calls to the context updating functionality, as well as statistical behaviour collection. The necessity to execute all this extra code leads to significant performance overheads. That is why the learning stage proceeds only until a representative profile has been built for all the noteworthy methods. Once this has been accomplished, the application can move on to the next stage.

In the second stage, the only injected code that is kept in the target application is the one responsible for keeping track of the changes in execution contexts. Behaviour data is no longer collected about the access patterns that are effectively being practiced by the application. This allows the target system to operate with practically unperturbed performance, when compared to its original version, as shall be seen and demonstrated in the results and evaluation section. The extremely low overhead makes it possible for the application to operate normally, while the monitoring system solution keeps its profiling data up-to-date.

The question that remains is how does the profiler system update the overall access-pattern behaviour information, if the only application aspect that it keeps track of is the change in execution contexts. If the program behaviour, at method level, is stationary (and does not drift significantly over time) then the method profiles built during the learning stage continue to provide a precise view of the behaviour displayed when executing those methods, as long as their implementation does not change. As such, whenever an updated overall view of the domain-data access patterns is necessary, the profiler determines the composition of the workload, based on the observed ratios of method/service invocations (e.g. MethodX = 1045 invocations, MethodY = 703 invocations, etc). The interval for which the workload is determined corresponds to the period of time from the previous update up to the moment when the new update is requested.

Once the workload has been identified, this information is used along with the individual method/service profiles for building an application-level view of the domain access patterns performed during that period. Simply put, the individual profiles are weighted by the workload ratio that their respective methods assumed in the workload, for that particular period of time.

It should be noted that if the implementation of a method does change, at some point in time, then it is necessary to revert the application back to stage one so that new and updated profiles can be built. Otherwise, there would be no guarantee as to the correctness of the access-pattern information generated by the profiler.

VI. RESULTS AND EVALUATION

The TPC-W benchmark [16] was used to evaluate the performance of the profiling solution presented here. The benchmark specifies an e-commerce workload that simulates the activities of an online retail store, where emulated clients can browse and order products from the site.

The TPC-W evaluation metric is the number of web interactions per second (WIPS) that the system can sustain. The benchmark execution is characterized by a series of input parameters. One of these defines the type of workload, which specifies the percentage of read and write operations that is to be simulated by the emulated browser (EB) clients. The workloads considered were Mix1 (95% read and 5% write); Mix2 (80% read and 20% write) and Mix3 (50% read and 50% write). The remaining configuration parameters were 10 emulated browsers; 300 seconds of ramp-up time; 1200 seconds for measurement interval, after the ramp-up time; 120 seconds of ramp-down time; 1k, 10k, and 100k book items in the database and think time of 0 seconds, ensuring that EBs do not pause in between requests.

All results were obtained as the average of 10 independent executions of the benchmark, for identical configurations. The EBs, database and the benchmark server were run on the same physical machine. The measurements were carried out with the benchmark running on a machine equipped with 2x Intel Xeon E5520 (a total of 8 physical cores with hyper-threading running at 2.26 GHz) and 24 GB of RAM. Its operating system was Ubuntu 10.04.3, and the JVM used was Java (TM) SE Runtime Environment (build 1.6.0 22-b04), Java HotSpot (TM) 64-Bit ServerVM (build

17.1-b03, mixed mode). The benchmark was run on top of Apache Tomcat 6.0.24, with the options set to "-server -Xms64m -Xmx$(heapSize)m -Xshare:off -XX: +UseConcMarkSweep GC -XX:+AggressiveOpts".

The throughput of the benchmark was evaluated for 14 different modes of monitoring. The operation mode designated as *BaseLine* corresponds to the TPC-W operating in pristine conditions, with its original implementation, without any byte-code manipulations. The newly developed profiler shall be referred to as the *Stationary* solution, while the old monitoring solution, that keeps track of domain access pattern occurrences continuously, shall be referred to as *NonStationary*.

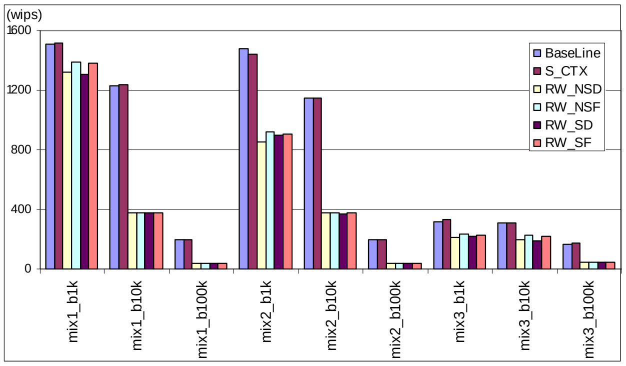
Two different approaches (orthogonal to the profiling solution in use) for storing data about changes in execution context shall be considered as well. The first of these approaches, which shall be called *Deep*, maintains information about the sequence of context changes that led to the currently active one. Such an approach makes it possible to know the exact sequence of method invocations that preceded any given point in the execution of the program. The alternative approach, called *Flat*, only keeps track of the currently active execution context, independently of the execution flow that might have been displayed by the program to reach it.

To get a better grasp of how the *Stationary* and *NonStationary* solutions behave, three further variants are taken into account, based on the types of accesses over domain data that are tracked. These are read-write, write-only and read-only modes. The list of the 14 different modes of monitoring evaluated here is:

- BaseLine - vanilla version of TPC-W
- S_CTX - *Stationary* solution in context-only mode;
- RW/WO/RO_NSD - *NonStationary* solution with *Deep* context tracking in Read-Write, Write-Only and Read-Only modes;
- RW/WO/RO_NSF - *NonStationary* solution with *Flat* context tracking in Read-Write, Write-Only and Read-Only modes;
- RW/WO/RO_SD - *Stationary* solution under profile-building mode, with *Deep* context tracking in Read-Write, Write-Only and Read-Only modes;
- RW/WO/RO_SF - *Stationary* solution under profile-building mode, with *Flat* context tracking in Read-Write, Write-Only and Read-Only modes.

A total of 126 distinct benchmark configurations were evaluated (14 operation modes, 3 workload types and 3 database sizes). Additionally, every configuration was executed 10 times, independently of previous runs, to provide a more comprehensive view of the behaviour displayed by the system. Taking this into account, along with the fact that a single execution of the benchmark takes approximately 15min (14min benchmark execution and 1min for Tomcat reboot, benchmark redeploy and database refresh), the results presented in this section took a total of 315h to generate.

The WIPS achieved by the *BaseLine*, *Stationary* in context-only mode and the Read-Write of the *Stationary* and *NonStationary* can be seen in Figure 1 (top), while the Write-Only and Read-Only variants Figure 1 (bottom). Every group of bars corresponds to a particular benchmark configuration in terms of workload (mix1, mix2 and mix3) and database size (1k, 10k and 100k book instances).



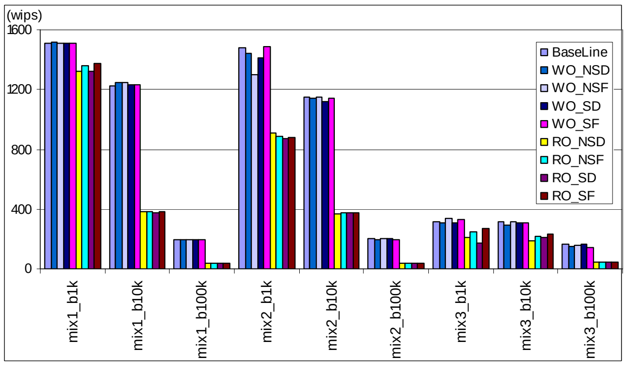


Figure 1. WIPS - baseline, stationary and non-stationary monitoring

From the analysis of these results, several remarks can be made. The throughput displayed when TPC-W is operating with *Stationary* in context-only mode appears to be very similar to the one when the benchmark is operating in its original version. When the benchmark is operating with *Stationary* (ReadWrite and ReadOnly) in profile-building mode as well as with *NonStationary* (ReadWrite and ReadOnly), the throughput observed is significantly lower than the one of *BaseLine*. The throughput displayed by the benchmark when the monitoring solutions are only tracking write-access operations over domain data is very similar to the BaseLine. The last two observations confirm what was already to be expected, namely that the factor that contributes most for the performance overheads observed when profiling a program is the tracking of read-access operations. Adding a fixed overhead to many short duration operations is bound to cause a greater impact than adding the same overhead to few long-duration operations.

It is interesting to note that for both *Stationary* and *NonStationary* approaches, whenever they are operating with *Flat* context-tracking mode, the WIPS achieved are slightly but consistently better than their counterparts with *Deep* context-tracking mode.

A thorough comparison of the relative throughput difference between all 14 monitoring modes and the BaseLine can be seen in Table I and II. The relative throughput difference has been calculated as

$\left(\left(T_A - T_{BaseLine}\right)/T_{BaseLine}\right)\times 100$, in %, where T_A indicates the throughput achieved when TPC-W is operating with monitoring approach A. The average throughput difference, per approach, is displayed with a shaded background.

TABLE I. THROUGHPUT DIFFERENCE (%), ALL VS. BASELINE (R&W)

	BaseLine	S_CTX	RW_NSD	RW_NSF	RW_SD	RW_SF
mix1_b1k	0.00	0.51	-12.36	-8.06	-13.15	-8.57
mix1_b10k	0.00	0.63	-69.01	-69.12	-69.53	-68.95
mix1_b100k	0.00	-0.01	-81.45	-80.99	-81.50	-81.26
mix2_b1k	0.00	-2.49	-42.11	-37.75	-39.28	-38.70
mix2_b10k	0.00	-0.22	-66.85	-66.91	-67.44	-67.10
mix2_b100k	0.00	-0.34	-82.10	-81.73	-82.13	-81.53
mix3_b1k	0.00	5.16	-32.85	-25.43	-30.87	-27.70
mix3_b10k	0.00	-1.46	-36.17	-28.74	-40.24	-29.25
mix3_b100k	0.00	3.45	-73.08	-72.71	-72.81	-72.04
avg	0.00	0.58	-55.11	-52.38	-55.22	-52.79

The analysis of these results indicates that the performance of the benchmark, when the Stationary approach is operating in context-only mode (S_CTX) is practically identical (0.58% difference) to the one of the original version of TPC-W (BaseLine). While it would be logical to expect that the S_CTX should display throughput that is strictly less than the one of the BaseLine, the few configurations where the opposite is observed can be (eventually) explained by the fact that the byte-code manipulations performed over the benchmark, for keeping track of the changes in execution contexts, allowed the just-in-time compiler of the JVM to perform some further optimizations that would be otherwise unable to apply. Another (possibly more likely) explanation for this phenomenon would relate to the intermediate-to-high measurement uncertainty observed for the two configurations where the S_CTX performs better than BaseLine (Mix3 with b1k and b100k).

TABLE II. THROUGHPUT DIFFERENCE (%), ALL VS. BASELINE (READ-ONLY AND WRITE-ONLY)

	WO_NSD	WO_NSF	WO_SD	WO_SF	RO_NSD	RO_NSF	RO_SD	RO_SF
mix1_b1k	0.47	0.25	0.22	-0.03	-12.33	-9.75	-12.47	-8.59
mix1_b10k	1.69	1.29	0.68	0.52	-69.06	-68.57	-69.18	-68.92
mix1_b100k	-0.08	-0.01	0.25	-0.24	-81.63	-81.49	-81.73	-81.06
mix2_b1k	-2.19	-11.88	-4.08	1.00	-38.27	-39.82	-40.94	-40.48
mix2_b10k	-0.52	0.44	-2.20	-0.10	-67.86	-67.18	-67.17	-67.07
mix2_b100k	-0.66	-0.17	-0.15	-0.55	-82.18	-81.61	-82.02	-81.90
mix3_b1k	-4.01	5.61	-3.42	4.02	-33.10	-23.13	-45.84	-14.76
mix3_b10k	-5.32	-0.26	-1.76	-1.40	-38.98	-30.35	-33.02	-24.80
mix3_b100k	-9.60	-3.25	0.53	-15.55	-73.18	-72.57	-72.70	-72.71
avg	-2.25	-0.89	-1.10	-1.37	-55.18	-52.72	-56.12	-51.14

The throughput displayed by the *Stationary* solution in profiling-mode is very similar to the *NonStationary* approach, across all evaluated configurations. This was to be expected since both approaches perform very similar tasks, in those operation modes. On the average, the throughput that the TPC-W can maintain while operating with *Stationary* profile-mode or any of the *NonStationary* variants is from 51% to 56% lower than the throughput of the unmodified benchmark.

The last performance aspect that can be appreciated, based on these results is the effect of the *Deep* and *Flat* context-tracking modes. As could be seen from Figure 1 and can now be confirmed numerically, the *Flat* context-tracking mode allows for small but consistent performance improvements. These are most noticeable for the Read-Write and Read-Only variants of the monitoring solutions and range from 3% to 5%.

VII. CONCLUSION

This work presented a new solution for profiling the behaviour of object-oriented applications, in terms of the access-patterns performed at run-time over domain data. By making certain assumptions about the stationary and ergodic properties of the run-time behaviour of object-oriented applications, the new solution can provide detailed and continuously updated information about the effectively practiced domain-data access-patterns, by the target application, without introducing any noteworthy performance overheads. This feature allows the newly developed solution to monitor any application in real-time, while the target system is operating in steady-state.

The solution was evaluated on the TPC-W benchmark, against multiple variants of previously existing solutions. It was possible to demonstrate that the new approach reduces the performance overheads of previous alternatives from an average of 55% down to approximately zero, while providing the same degree of information.

ACKNOWLEDGMENT

This work was partially supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2013, as well as by FCT (INESC-ID multiannual funding) PIDDAC Program funds and by the Specific Targeted Research Project (STReP) Cloud-TM, which is co-financed by the European Commission through the contract no. 257784. The first author has been funded by the Portuguese FCT under contract SFRH/BD/64379/2009.

REFERENCES

[1] M. Cierniak, M. Eng, N. Glew, B. Lewis and J. Stichnoth, 2003, The Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment, Intel Technology Journal, 7, (1), pp. 5-18.

[2] M. Arnold, S. Fink, D. Grove, M. Hind and P. Sweeney, 2005, A survey of adaptive optimization in virtual machines, Proceedings of the IEEE, 93, (2), pp. 449-466.

[3] M. Smith, 2000, Overcoming the challenges to feedback-directed optimization, ACM SIGPLAN Notices, ACM, Vol. 35, pp. 1-11.

[4] M. Arnold, M. Hind and B. Ryder, 2002, Online feedback-directed optimization of Java, ACM SIGPLAN Notices, ACM, Vol. 37, pp. 111-129.

[5] M. Arnold and B. Ryder, 2001, A framework for reducing the cost of instrumented code, ACM SIGPLAN Notices, ACM, Vol. 36, pp. 168-179.

[6] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi and K. Hazelwood, 2005, Pin: building customized program analysis tools with dynamic instrumentation, ACM SIGPLAN Notices, ACM, Vol. 40, pp. 190-200.

[7] Valgrind, http://valgrind.org/ [accessed 10 May 2013].

[8] DynamoRIO, http://www.dynamorio.org/ [accessed 10 May 2013].

[9] S. Garbatov, J. Cachopo and J. Pereira, 2009, Data Access Pattern Analysis based on Bayesian Updating, Proceedings of the First Symposium of Informatics (INForum 2009), Lisbon, p. Paper 23.

[10] S. Garbatov and J. Cachopo, 2010, Importance Analysis for Predicting Data Access Behaviour in Object-Oriented Applications, Journal of Computer Science and Technologies, 14, (1), pp. 37-43.

[11] S. Garbatov and J. Cachopo, 2010, Predicting Data Access Patterns in Object-Oriented Applications Based on Markov Chains, Proceedings of the Fifth International Conference on Software Engineering Advances (ICSEA 2010), Nice, France, pp. 465-470.

[12] S. Garbatov and J. Cachopo, 2011, Data Access Pattern Analysis and Prediction for Object-Oriented Applications, INFOCOMP Journal of Computer Science, 10, (4), pp. 1-14.

[13] G. Lindgren, 2012, Stationary Stochastic Processes: Theory and Applications: Chapman and Hall/CRC.

[14] P. Walters, 1982, An Introduction to Ergodic Theory: Springer.

[15] ASM, http://asm.ow2.org/, [accessed 10 May 2013].

[16] W. Smith. TPC-W: Benchmarking An Ecommerce Solution. Intel Corporation, 2000.