# Application Performance Management: State of the Art and Challenges for The Future

Sri Sharanabasappa Raikoti, Assistant Professor, Department of Computer Science, Government Degree College Yadgir, Karnataka (India), Email: sr.raikoti@gmail.com

## Abstract

In the last decade, several research efforts have been directed to integrating performance analysis in the software development process. Traditional software development methods focus on software correctness, introducing performance issues later in the development process. This approach is not adequate since performance problems may be so severe that they may require considerable changes in the design, for example at the software architecture level, or even worse in the requirements analysis. Several approaches have been proposed to address early software performance analysis. Although some of them have been successfully applied, there is still a gap to be filled in order to see performance analysis integrated in ordinary software development. We present a comprehensive review of the recent developments of software performance research and point out the most promising research directions in the field.

**Keywords: SOFTWARE, PERFORMANCE FEEDBACK, STATE OF THE ART**

## Introduction:

With the ultimate goal of replacing proprietary hardware appliances with Virtual Network Functions (VNFs) implemented in software, Network Function Virtualization (NFV) has been gaining popularity in the past few years. Software switches route traffic between VNFs and physical Network Interface Cards (NICs). It is of paramount importance to compare the performance of different switch designs and architectures. In this paper, we propose a methodology to compare fairly and comprehensively the performance of software switches. We first explore the design spaces of seven state-of-the-art software switches and then compare their performance under four representative test scenarios. Each scenario corresponds to a specific case of routing NFV traffic between NICs and/or VNFs. In our experiments, we evaluate the throughput and latency between VNFs in two of the most popular virtualization environments, namely virtual machines (VMs) and containers. Our experimental results show that no single software switch prevails in all scenarios. It is, therefore, crucial to choose the most suitable solution for the given use case. At the same time, the presented results and analysis provide a deeper insight into the design tradeoffs and identifies potential performance bottlenecks that could inspire new designs.

Williams et al. in [10] introduced the PASA (Performance Assessment of Software Architectures) approach. It aims at achieving good performance results [8] through a deep understanding of the architectural features. This is the approach that firstly introduces the concept of antipatterns as support to the identification of performance problems in software architectural models as well as in the formulation of architectural alternatives. However, this approach is based on the interactions between software architects and performance experts, therefore its level of automation is still low. Cortellessa et al. in [3] introduced a first proposal of automated generation of feedback from the software performance analysis, where performance antipatterns play a key role in the detection of performance flaws. However, this approach considers a restricted set of antipatterns, and it uses informal interpretation matrices as support. Performance scenarios are described (e.g. the throughput is lower than the user requirement, and the response time is greater than the user requirement) and, if needed, some actions to improve such scenarios are outlined. The main limitation of this approach is that the interpretation of performance results is only demanded to the analysis of Layered Queue Networks (LQN) [1], i.e. a performance model. Such knowledge is not enriched with the features coming from the software architectural models, thus to hide feasible refactoring actions. Enterprise technologies and EJB performance antipatterns are analyzed by Parsons et al. in [8]: antipatterns are represented as sets of rules loaded into a JESS [2] engine, and written in a Lisp-like syntax [10]. A rule-based performance diagnosis tool, named Performance Antipattern Detection (PAD), is presented. However, it deals with ComponentBased Enterprise Systems, targeting only Enterprise Java Bean (EJB)

applications. It is based on the monitoring of the data from running systems, it extracts the run-time system design and detects EJB antipatterns by applying rules to it. Hence, the scope of [8] is restricted to such domain, and performance problems can neither be detected in other technology contexts nor in the early development stages. By taking a wider look out of the performance domain, the management of antipatterns is a quite recent research topic, whereas there has already been a significant effort in the area of software design patterns. It is out of scope to address such wide area, but it is worth to mention some approaches dealing with patterns. Elaasar et al. in [5] introduced a metamodeling approach to pattern specification. In the context of the OMGs 4-layer metamodeling architecture, the authors propose a pattern specification language (i.e. Epattern, at the M3 level) used to specify patterns in any MOF-compliant modeling language at the M2 layer. France et al. in [9] introduced a UML-based pattern specification technique. Design patterns are defined as models in terms of UML metamodel concepts: a pattern model describes the participants of a pattern and the relations between them in a graphical notation by means of roles, i.e. the properties that a UML model element must have to match the corresponding pattern occurrence.

Business success is directly influenced by the performance of the enterprise application systems that support it. Any performance issue that may arise during the production use of such applications may bring losses in revenue, and even cause customers to turn away. Examples of these losses and their impact are well documented. Google loses 20% traffic if their web sites respond 500 ms slower [9]. Amazon loses 1% of revenue for every 100 ms in latency [8]. Mozilla's study showed that if the page is not loaded within one to five seconds, users will leave the web site [4]. Application performance management (APM), as a core IT operations discipline, aims to achieve an adequate level of performance during operations. To achieve this, APM comprises methods, techniques, and tools for i) continuously monitoring the state of an applications system and its usage, as well as for ii) detecting, diagnosing, and resolving performance-related problems using the monitored data. In this paper, we provide a state-of-the-art overview of the common APM activities (Section 2) and tools (Section 3), and highlight selected challenges and future directions (Section 4).

**ANTIPATTERN-BASED APPROACHES**

The term Antipattern appeared for the first time in [3] in contrast to the trend of focus on positive and constructive solutions. Differently from patterns, antipatterns look at the negative features of a software system and describe commonly occurring solutions to problems that generate negative consequences. Antipatterns have been applied in different domains. For example, in [8] data-flow antipatterns help to discover errors in workflows and are formalized through the CTL* temporal logic. As another example, in [1] antipatterns help to discover multi threading problems of Java applications and are specified through the LTL temporal logic. Performance Antipatterns, as the name suggests, deal with performance issues of the software systems. They have been previously documented and discussed in different works: technology-independent performance antipatterns have been defined in [3]; technologyspecific antipatterns have been defined in [5] and [7].

**RULE-BASED APPROACHES**

Barber et al. in [21] introduced heuristic algorithms that in presence of detected system bottlenecks provide alternative solutions to remove them. The heuristics are based on architectural metrics that help to compare different solutions. In a Domain Reference Architecture (DRA) the modification of functions and data allocation can affect non-functional properties (for example, performance-related properties such as component utilization). The tool RARE guides the derivation process by suggesting allocations based on heuristics driven by static architectural properties. The tool ARCADE extends the RARE scope by providing dynamic property measures. ARCADE evaluation results subsequently fed back to RARE can guide additional heuristics that further refine the architecture. However, it basically identifies and solve only software bottlenecks, more complex problems are not recognized. Dobrzanski et al. in [1] tackled the problem of refactoring UML models. In particular, bad smells are defined as structures that suggest possible problems in the system in terms of functional and non-functional aspects. Refactoring operations are

suggested in the presence of bad smells. Rules for refactoring are formally defined, and they take into account the following features: (i) cross integration of structure and behavior; (ii) support for component-based development via composite structures; and (iii) integration of action semantics with behavioral constructs. However, no specific performance issue is analyzed, and refactoring is not driven by unfulfilled requirements. McGregor et al. in [1] proposed a framework (ArchE) to support the software designers in creating architectures that meet quality requirements. It embodies knowledge of quality attributes and the relation between the achievement of quality requirements and architectural design. It helps to create architectural models by collecting requirements (in form of scenarios) and the information needed to analyze the quality criteria for the requirements. It additionally provides the evaluation tools for modifiability or performance analysis. However, the suggestions (or tactics) are not well explained, and it is not clear at which extent the approach can be applied. Kavimandan et al. in [8] presented an approach to optimize deployment and configuration decisions in the context of distributed, realtime, and embedded (DRE) componentbased systems. Bin packing algorithms have been enhanced, and schedulability analysis have been used to make fine-grained assignments that indicate how components are allocated to different middleware containers, since they are known to impact on the system performance and resource consumption. However, the scope of this approach is limited to deployment and configuration features. Xu in [8] presented an approach to software performance diagnosis that identifies performance flaws before the software system implementation. It defines a set of rules (specified with the Jess rule engine [2]) aimed at detecting patterns of interaction between resources. The method is applied to UML [2] that employ standard profiles, i.e. the SPT or Schedulability, Performance and Time profile [4] and its successor MARTE [3].
The software architectural models are translated in a performance model, i.e. Layered Queueing Networks (LQNs) [9], and then analyzed. The approach limits the detection to bottlenecks and long execution paths identified and removed at the level of the LQN performance model. The actions to solve the performance issues are: change the configuration, i.e. increase the size of a buffer pool or the amount of existing processors; and change the design, i.e. increase parallelism and splitting the execution of task in synchronous and asynchronous parts. The overall approach applies only to LQN models, hence its portability to other notations is yet to be proven and it may be quite complex

**SEARCH-BASED APPROACHES**

A wide range of different optimization and search techniques have been introduced in the field of Search-Based Software Engineering (SBSE) [7], i.e. a software engineering discipline in which search-based optimization algorithms are used to address problems where a suitable balance between competing and potentially conflicting goals has to be found. Two key ingredients are required: (i) the representation of the problem; (ii) the definition of a fitness function. In fact, SBSE usually applies to problems in which there are numerous candidate solutions and where there is a fitness function that can guide the search process to locate reasonably good solutions. A suitable representation of the problem allows to automatically explore the search space for the solutions that best fit the fitness function [2] that drives towards the sequence of the refactoring steps to apply to this system (i.e. altering its architectural structure without altering its semantics). In the software performance domain both the suitable representation of the problem and the formulation of the fitness function are not trivial tasks, since the performance analysis results are derived from many uncertainties like the workload, the operational profile, etc. that might completely modify the perception of considering candidate solutions as good ones. Some assumptions can be introduced to simplify the problem and some design options can be explicitly defined in advance to constitute the population [2] on which search based optimization algorithms apply. However, we believe that in the performance domain it is of crucial relevance to find a synergy between the search techniques that involve the definition of a fitness function to automatically capture what is required from the system, and the antipatterns that might support such function with the knowledge of bad practices and suggest common solutions, in order to quickly converge towards performance improvements. In fact, as recently outlined in [1], there is a mutually

beneficial relationship between SBSE and predictive models. In particular eleven broad areas of open problems (e.g. balancing functional, nonfunctional properties of predictive models) in SBSE for predictive modeling are discussed, explaining how techniques emerging from the SBSE community may find potentially innovative applications in predictive modeling.

**DESIGN SPACE EXPLORATION APPROACHES**

Zheng et al. in [4] described an approach to find optimal deployment and scheduling priorities for tasks in a class of distributed real-time systems. In particular, it is intended to evaluate the deployment of such tasks by applying a heuristic search strategy to LQN models. However, its scope is restricted to adjust the priorities of tasks competing for a processor, and the only refactoring action is to change the allocation of tasks to processors. Bondarev et al. in [5] proposed a design space exploration methodology, i.e. DeSiX (DEsign, SImulate, eXplore), for software component-based systems. It adopts multidimensional quality attribute analysis and it is based on (i) various types of models for software components, processing nodes, memories and bus links, (ii) scenarios of system critical execution, allowing the designer to focus only on relevant static and dynamic system configurations, (iii) simulation of tasks automatically reconstructed for each scenario, and (iv) Pareto curves [4] for identification of optimal architecture alternatives. An evolution of [3] can be found in [2], where a design space exploration framework for component-based software systems is presented. It allows an architect to get insight into a space of possible design alternatives with further evaluation and comparison of these alternatives. However, it requires a manual definition of design alternatives of software and hardware architectures, and it is meant to only identify bottlenecks. Ipek et al. in [8] described an approach to automatically explore the design space for hardware architectures, such as multiprocessors or memory hierarchies. The multiple design space points are simulated and the results are used to train a neural network. Such network can be solved quickly for different architecture candidates and delivers accurate results with a prediction error of less than 5%. However, the approach is limited to hardware properties, whereas software architectures are more complex, because architectural models spread on a wide rage of features.

**METAHEURISTIC APPROACHES**

Canfora et al. in [5] used genetic algorithms for Quality of Service (QoS)-aware service composition, i.e. to determine a set of concrete services to be bound to the abstract ones in the workflow of a composite service. However, each basic service is considered as a black-box element, where performance metrics are fixed to a certain unit (e.g. cost=5, resp.time=10), and the genetic algorithms search the best solutions by evaluating the composition options. Hence, no real feedback (in terms of refactoring actions in the software architectural model such as split a component) is given to the designer, with the exception of pre-defined basic services. Aleti et al. in [6] presented a framework for the optimization of embedded system architectures. In particular, it uses the AADL (Architecture Analysis and Description Language) [7] as the underlying architecture description language and provides plug-in mechanisms to replace the optimization engine, the quality evaluation algorithms and the constraints checking. Architectural models are optimized with evolutionary algorithms considering multiple arbitrary quality criteria. However, the only refactoring action the framework currently allows is the component re-deployment. Martens et al. in [8] presented an approach for a performance-oriented design space exploration of component-based software architectures. An evolution of this work can be found in [9] where meta-heuristic search techniques are used for improving performance, reliability, and costs of of component-based software systems. In particular, evolutionary algorithms search the architectural design space for optimal trade-offs by means of Pareto curves. However, this approach is quite time-consuming, because it uses random changes (spanning on all feasible solutions) of the architecture, and the optimality is not guaranteed.

**Data Storage and Processing**

In order to have a central view on the collected data, it is usually transferred by the agents to a data storage for further processing and analysis. Proprietary or standard technologies (e.g., database management systems) can be and are being used. APM usually results in very large

data sets that need to be handled efficiently [14]. Two data representations are commonly used: time series and execution traces. While time series represent summary statistics (e.g., counts, percentile, etc.) over time, execution traces [3] provide a detailed representation of the application-internal control flow that results from individual system requests. From this data, architectural information, including logical and physical deployments and interactions (topology), can be extracted.

**References**

1. Franco-Santos, M., Kennerley, M., Micheli, P., Martinez, V., Mason, S., Marr, B., et al. (2007). Towards a definition of a business performance measurement system. International Journal of Operations and Production Management, 27, 784–801.
2. Galbraith, J. R. (1973). Designing complex organizations. Massachusetts: AddisonWesley Publishing, Reading. Garengo, P. (2009). Performance measurement system in SMEs taking part to quality award programs. Total Quality Management and Business Excellence, 20, 91–105.
3. Garengo, P., Biazzo, S., & Bititci, U. S. (2005). Performance measurement systems in SMEs: A review for a research agenda. International Journal of Management Reviews, 7, 25–47.
4. Kroes, J. R., & Ghosh, S. (2010). Outsourcing congruence with competitive priorities: Impact on supply chain and firm performance. Journal of Operations Management, 28, 124–143.
5. Kueng, P. (2001). Performance measurement systems in the service sector – The potential of IT is not yet utilized, internal working paper no. 01–05, Department of Informatics, University of Fribourg, Rue Faucingy 2, 1700 Fribourg, Switzerland.
6. Leavitt, H. J., & Whistler, T. L. (1958). Management in the 1980s. Harvard Business Review, 41–48.
7. Marchand, M., & Raymond, L. (2008). Researching performance measurement systems – An information systems perspective. International Journal of Operations and Production Management, 28(7), 663–686.
8. Schonberger, R. J. (1982). Japanese manufacturing techniques: Nine hidden lessons in simplicity. The Free Press Publishers. Shepherd, C., & Gunter, H. (2006). Measuring supply chain performance: Current research and future directions. International Journal of Productivity and Performance Management, 55, 242–258.
9. Skinner, W. (1974). The decline, fall, and renewal of manufacturing. Industrial Engineering, 32, 38.
10. Wise, R., & Baumgartner, P. (1999). Go downstream: The new profit imperative in manufacturing. Harvard Business Review, 77, 133–141.
11. Woodruff, R. B. (1997). Customer value: The next source for competitive advantage. Journal of the Academy of Marketing Science, 25, 139–153.
12. Yamakawa, T., Ahmed, S., Kelston, A. (2009). The BRICs as drivers of global consumption, Goldman sachs global economics, commodities and strategy research (06.08.09)