Performance Engineering Using Performance Antipatterns in Distributed Systems

Chia-En Lin and Krishna Kavi
Department of Computer Science and Engineering
Unitersity of North Texas
Denton, TX USA
chiaen@unt.edu, Krishna.Kavi@unt.edu

***Abstract*—Performance analysis of software systems is complex due to the number of components and the interactions among them. Without the knowledge of experienced experts, it is futile to diagnose the performance anomaly and attempt to pinpoint the root causes in the system. Design patterns are a formal way of documenting best practice approaches in software development and system architecture design. Software performance antipatterns are similar to design patterns in that they indicate what to avoid and how to fix performance problems when they appear. Although the idea of applying antipatterns is promising, there are gaps in matching the symptoms and generating feedback solutions for redesign. In this work, we analyze performance antipatterns to extract detectable features, influential factors, and resource involvements so that we can lay the foundation to detect their presence. We propose a system abstraction layering model and suggestive profiling methods as the infrastructure in building the framework for performance antipattern detection with solution suggestions. It is used in the refactoring phase of the performance modeling process, and is synchronized with the software development life cycles. Proposed tools and utilities are implemented and have been used on real production servers with RUBiS benchmark.**

Keywords–Performance Engineering; Anomaly Detection; Performance Antipattern; Profiling.

I. INTRODUCTION

Developing a software system that meets its specifications demands continuous verification and validation efforts in iterative development cycles running from analysis and design, to implementation and deployment. During these processes, engineers build the system by creating design plans, and maintaining expected functional and non-functional properties of the specifications. Testing and debugging activities take place alongside the development. Similar to conventional functional debugging, non-functional properties must also be tested and appropriate fixes be made to meet the requirements. The complexity of modern software systems makes it difficult for the designer to assure compliance of non-functional requirements.

Design patterns are a formal way of documenting best practices in software development and system architecture design. The documented solutions are represented in a pattern language, which addresses a description of the solution to the problem, and the benefit gained from applying the process [1].

Since the usability of design patterns is still fairly abstract in terms of pattern matching, they are not easily adapted and applied in practice. Computing environment or context can be thought of as a multidimensional attribute set which has great impact on the execution of applications and systems. To better match the problem description, a design pattern has to provide specific context information for which the design pattern is intended. In most cases, the context provides detailed descriptions in a natural language to identify the scenario where the pattern is applicable and not applicable. Designers can look up the patterns and see if the scenario matches. Once found, they can apply the solution as a best practice to assure the result of the design is in fact the best possible.

Although patterns are promising and of great help in system development, some gaps between practice and application still exist. One of the obstacles results from the process of identifying the exact context and matching the scenario to the system under design. The context description of a design pattern is usually described informally in natural language; it is usually the responsibility of experienced domain experts to decide if the match is effective. For a relatively large-scale system, the complexity increases quickly making design patterns unusable.

Software design patterns can be thorough in treating functional design problems, but they do not address other aspects of the design. This leads to another gap in applying design patterns. Although the solutions to the design problems optimize the components of the system while building, they do not give clues to the quality of the design. In other words, only functional enhancements are ensured, whereas non-functional properties such as availability and reliability are not fully covered.

Smith et el. [2] are among the early proponents of performance design patterns. Principles of performance-oriented design are used as strategies in the development life cycle. They are embedded during the fundamental design practice which is later documented as performance patterns. Although performance patterns proposed are to address the performance issues, they are presented at higher levels, while the context can only be determined after the implementation of the pattern has been chosen.

Instead of following the same format of design patterns, the performance patterns are published from a different perspective, documenting potential bad practices that lead to poor performance. They tell us what not to do and how to fix a problem when it appears. Such patterns are called antipatterns. Performance antipatterns are similar to design patterns in that they document recurring problems, but state the scenarios from the opposite side of best practices. If the scenarios match with a performance antipattern, the predictive outcome of performance can be poor. The solutions of how to avoid the pitfalls are documented as solution descriptions analogous to best practices. The advantage of adapting performance antipatterns over performance patterns in practice is that they are easier to apply and are clearly guided due

to the explicit coverage of the scenario description space. However, antipatterns inherit one obstacle that is common with design patterns. It is still not straightforward to apply and gain the benefits from the solution. In this work, the focus is on targeting performance antipatterns in software development, and proposing approaches and tools to make the pattern application process more performance aware.

A novel framework that assists in performance debugging of distributed software systems is described in this work. To alleviate the obstacles of applying performance antipatterns during the software development life cycles, real performance indices are made available in our framework. Real performance baselines can be established so that the performance of the designed system can be compared to discover performance deficiencies. With the established facilities, contexts of performance antipatterns can be documented with practical metrics. It will assist practitioners to match, detect, and apply performance antipatterns quantitatively. For each system or sub-component being evaluated, the framework creates profiles in what is called suggestive profiling. When used during the development life cycle, it provides a realistic means both for antipattern detection and suggested solutions during the refactoring phase of a performance debugging process. Information regarding the root causes of the detected performance problem can be used to assist the redesign efforts. An effective solution can be devised and used to eliminate the identified performance anomaly.

The main contributions of our work are (a) an analysis of performance antipattern for detectable features, influential factors, and resource involvements (b) the proposition of a system abstraction layering model and suggestive profiling methods as foundations for performance antipattern detections, root cause analysis, and redesign suggestions, and (c) a performance antipattern detection and solution suggestion framework to be used in the refactoring phase of a performance modeling process, synchronized with the software development life cycles.

The structure of the remainder of this article is as follows. Analysis of performance antipatterns and how they are used in the design processes are introduced in Section II. The framework to adapt performance antipatterns in system and software development is presented in Section III, with the description of innovative fundamental architectures and suggestive profiling tools. Section IV describes the proposed process of performance antipattern detection and solution refactoring using the framework. Section V illustrates implementations and setup of the framework with examples. Section VI describes works that are closely related to ours. Finally, Section VII provides conclusions about this work and future extensions.

II. Performance AntiPattern

A. Performance Antipattern Analysis

Performance antipatterns were originally described by Smith and Williams [3][4][5]. Similar to the format of design patterns, documentation of an antipattern consists of the name of the pattern, the problem it addresses, and the best solution to solve the problem.

The first step in applying antipatterns in the design process is to extract the problem description and the feasibility for detecting the pattern in real systems. Since the research community frequently refers to these as fundamental antipatterns, there are additional attributes that highlight their usage features. For example, to be able to detect the existence of performance antipatterns, values of performance indicators have to be acquired to decide whether a specific symptom exists. Some of these can be determined by just a single value, while others require multiple samples over time. The former can be categorized as Single Value (SV), and the latter as Multiple Value (MV) antipatterns. These annotated attributes of a performance antipattern are summarized as Detectable Features (DF). A detectable feature is the extraction derived from the problem description statements, which serve as the essential indicators of existence of the pattern.

To apply solutions to overcome performance antipattens, the problem description is interpreted to extract the forces seen in the pattern description. Associated forces are defined as Influential Factors (IF) extracted from each antipattern will be used as the clues to the root causes. Forces can be extended when new forces are discovered from new archives. In a general system and software development context, the factors include:

Design Design factor is concerned with software objects and how well they are established in the design and implementation. It often relates to the policy in the design of resource sharing and recycling, as well as the arrangements of processing steps. Different design approaches result in different computing behaviors, and performance outcomes.

Algorithm Algorithm factor is distinct from Design in the way that software components can apply different strategies to achieve the same computation goal. The designer can adapt a strategic approach for computing and use different structures to manage data. Different complexities of the algorithms lead to different execution times.

Configuration Software development usually leaves options for configurations to let the user fine tune the behavior of the application to fit the usage expectation. While systems are ready to run, different management policies with corresponding configuration options can lead to different performance behaviors.

Threading Multitasking has been one of the frequent models used in software systems to cope with the complexity of parallel and distributed environments. Thread, as an abstract execution unit, plays a key role in carrying out a task along with other peer threads. Individual thread behavior, thread coordination, and management policies play a significant role in the overall system performance.

B. Design Processes with Performance Antipatterns

In most systems, the debugging activities are continuous along with an iterative software development life cycle. Analogous to general debugging activities, the performance debugging activity should also be embedded in the development process and run concurrently with the development processes to ensure the expected performance is on the right track. Taken from a generic modeling process, life cycle phases are put in order from requirement analysis and design to implementation and deployment testing. The life cycle is always iterative to make incremental improvements for each round. During the development process, engineers extract the required information to create models that assist in analyzing the design and planning for further verification and testing. These models are related to functionality of the system, and are used for validation and verification debugging purposes. Performance

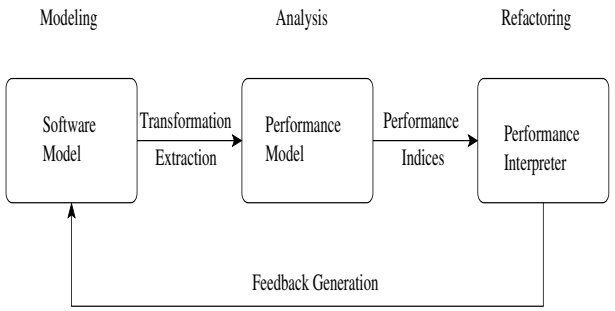

Figure 1. Performance Modeling Life Cycle with Performance Antipattern Refactoring

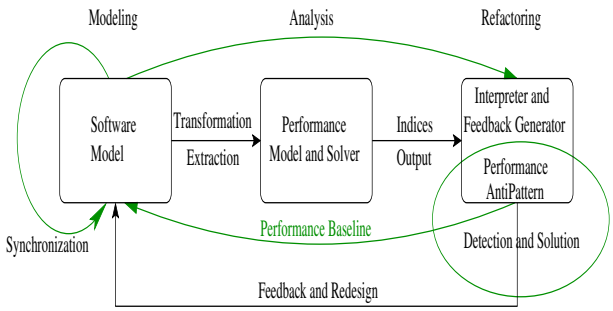

Figure 2. Refined Performance Modeling Life Cycle with Performance Antipattern Refactoring

debugging, on the other hand, requires further information to facilitate performance analysis and evaluation. In software performance engineering [6], a model-based performance analysis approach is adopted to generate performance data. They are created with the information about the architecture of the system, the capacity of its components, and the expected behavior of the system. Additional estimations such as request types and potential workloads are also needed for the modeling. The derived performance models are then used to produce quantitative numbers such as time duration, system utilization, and throughput. These values serve as indicators formally known as performance indices. The combination of these indices is used to predict the performance of the system.

The performance modeling process is depicted in Figure 1. The process is split into three phases. The modeling phase is the main stage of the system and software life span. Regular software models are built by following the life cycle phases. This software modeling phase is overlapped with performance modeling, because the updated performance attributes are gathered from the modeling activity as soon as the latest design revision is available. The second phase is analysis, and its goal is to create corresponding models for performance analysis and prediction. In this phase, model-to-model transformation is taking place. System and software models are transformed into performance models with information such as designated architecture, its topological layout, and available resources. In the analysis process, performance indices are obtained by solving the performance models using queueing network tools. These indices are used as indicators to forecast performance. Performance indices are interpreted in the third phase called refactoring. The goal of refactoring is to reflect the latest performance attributes and determine the satisfaction of the design in terms of performance qualification. If the performance does not meet the requirement, feedback can be generated to initiate design changes according to the interpreted results to resolve the performance issues. Engineers are obliged to check the predictive performance indices, and respond accordingly with changes to ensure the performance is acceptable. Analogous to software life cycles, the performance modeling process should proceed iteratively in an incremental order to synchronize with the original software model, create and analyze performance models to generate up-to-date performance indices, and give feedback with design changes for better performance.

In the performance engineering process, the goal is to detect a performance anomaly in the design and resolve the issue effectively and precisely. However, performance indices can only provide the location of the problematic components anomaly. To be able to come up with a change of plan, the practitioner must look into the design of the system to find the cause and estimate the performance penalties accordingly. It is difficult for performance experts to reason using only performance indices if the system being built is relatively new. This is where performance antipattern can help; especially in the refactoring phase. Running parallel with interpretation steps, an antipattern detection engine can be installed to assist performance antipattern identification. Once detected, the known solutions can be provided as feedback suggestions to remodel the system.

Antipattern detection mechanisms largely depend on the problem description to discover instances of a performance anomaly in the system, while feedback adjustments depend on the solution description. Ideally, the performance antipattern mechanism should be easy to adapt and build an engine for detection and find a solution. Figure 2 depicts the integrated process of the software development modeling and performance modeling processes. Both of the processes are synchronized in the modeling phase. In the refactoring phase, mechanisms of antipattern detection are integrated to assist in identifying performance problems and generate solutions accordingly as feedback for redesign. The integrated process is synchronized incrementally and iteratively with the software modeling.

Although the promises of performance antipatterns, or design patterns in general, are great, the intricate nature of documenting a scenario and its environment in a computing system makes direct application of antipatterns difficult. If the goal is to put it into automatic practice, there are many gaps and challenges. One example of deficiency is the difficulty in recognizing the context where the performance antipattern exists. For instance, in *unbalanced processing-extensive processing* antipattern, the problem description states "the extensive processing impedes overall response time." It is left to the discretion of engineers to realize what exactly the response time change is and how it should be modeled in the specific application. Another example in the *ramp* antipattern, the statement like "processing time increases" in the problem description, has to be determined by the engineers as to what is the significance of time indices in modeling so as to detect the symptom in the specific application.

Another noticeable hurdle in applying performance antipatterns is getting the solutions as feedback. Inherited from generic design patterns, solution descriptions are essential parts in pattern documentation that carry key expert knowledge. The secret to completing the performance modeling process

TABLE I. SYSTEM ABSTRACTION LAYERS

Division	Layer	Design Abstraction
Software	System	
	Sub-System	Integration
	Component	Composition
	Task	Configuration
	Thread	Execution
Platform	Layer	Abstraction
	Middleware	Resource Management
	Operating System	Scheduling Policy
Hardware	Layer	Abstraction
	Execution Unit	Processing Element

largely depends on the precision of solutions, which enables debugging processes to tweak the design to overcome the performance pitfalls mentioned in the antipattern. The problem with context ambiguity, similar to the counterparts of performance antipatterns in detection the mechanism mentioned above, appears in the solution description as well. For example, in *more is less* antipattern, to determine whether the architecture can meet its performance goals by staying below the thresholds, one has to decide the appropriate value for the threshold. The threshold value is not only affected by the underlying architecture attributes; it can also be affected by the degree of discretion which in turn depends on the context of the application. Another example, in *Traffic Jam* antipattern, one of the solutions is to provide sufficient processing power to handle the worst-case load. The processing power adjustment is an open issue to be determined to remedy the bottleneck. These examples show that applying performance antipatterns in the refactoring phase would need reasoning tools to assist the feedback generation. In other words, once an instance of antipattern has been identified, applying the solution description to generate feedback for system improvement is not straightforward. Tools that can reason about the context for the specific system should be available to enable the reasoning process. This is where profiling approaches can be useful, which are described in the next sections.

III. PERFORMANCE EVALUATION FRAMEWORK

A. System Abstraction Layers

Since performance results derive from the integration and cooperation of software and system architecture, an abstract structure is proposed to help us identify essential elements of the performance forces and express how they organize and play different roles in computing. A system and all its entities is modeled in a structure called System Abstraction Layers (SAL). The modeled layers consist of three divisions from top to bottom: software, platform, and hardware. Table I depicts the contents of each layer and their design abstraction. Each design abstraction represents orthogonal forces from a system development activity that affect the behavior of the elements in the layer.

In the software layer, systems are realized by the integration of sub-systems, each of which is responsible for a specific functionality. For each subsystem at the software layer, entities of software in terms of components and libraries are composed to create the sub-system. Each component in this setting is executing the tasks designated. The abstraction of task control can be related to configuration if the tuning mechanism is available for the software entity. The real execution in the software layer to carry out the tasks of a component is given to the elementary execution entity known as the thread. The abstraction is also compatible with the implementation using only the processes, where each process is treated as a special main thread.

Many software systems need to take advantage of using services from middleware to ease the complexity of developing and deploying applications. Middleware is used to manage the communication resources and hide the interaction details from the users, especially for distributed systems. In a broader sense, it also manages the server resources by regulating how control and information flow is distributed under the designated architecture topology. Below the middleware, it is the operating system that provides services for resource management and process scheduling. The platform layer is about resource management where the system and its software entities reside and access the computing resources.

The bottom layer of the server under the platform layer is related to hardware component organization. It is where the actual performance is measured. Performance revealed from the hardware layer depends on the grade of components installed. Utilization of hardware components can be acquired from this layer which includes processors, memory, network, and disks among others.

With the defined conceptual layers in SAL, each of them relating to a specific design abstraction, we can describe a performance scenario flexibly both at higher and lower layers. A high level scenario expression can be refined and mapped to its corresponding lower level counterparts. Through the process of mapping, we can identify the related elements in each layer and reason about the forces associated with them. This lays out the foundation to detect the case of a performance anomaly, when the root cause elements can be identified in the hierarchical approach. To accommodate the context information, the structure of layers can be represented with a description language. The performance context of a system or application can likewise be expressed.

B. Performance Suggestive Profiling

The purpose of profiling in our framework is to identify the performance anomaly and to locate the root causes. Once performance antipatterns appear in the performance modeling, the practitioner should be able to detect and get the suggestive solutions depending on the current context of the system to remedy the problem. To gain the causality reasoning capability, the proposed profiling mechanism is set to conform to the system abstract layers. The profiling mechanism can also serve as the toolkit to access the performance baselines in the SAL structure. For the assistance role, the profiling mechanism should be compatible to both software development and the performance modeling process, making it easy to adapt in all phases in the process. The profiling mechanism should be easy to setup for performance testing and evaluation. Since profiling is also applied to the baseline, profiling can aid in detecting antipatterns and suggest solutions. With these design requirements in mind, the following discussion provides the design rationale and discusses the components of the profiling mechanism, the context where they can be applied, and how they can be utilized in the system development.

Conventional software profilers usually focus on the source code or its corresponding executable binaries to get statistical measurements of the software package or library. The infor-

TABLE II. SUGGESTIVE PROFILING METHOD IN THE SYSTEM ABSTRACT LAYER CONTEXT

SAL Context	Profiling Method	Suggestive Profiling Method
Subsystem, component	Path-Oriented	Alternative Path options
Thread	Thread Behavior	Thread Behavior comparisons
Middleware	Networking Profiling	Request traces and communication protocol verification
Hardware	System Resource Profiling	Physical and Abstract resource summarization

mation includes frequency and duration of routines such as function calls. The goal of a conventional profiler is program optimization. System profilers focus on resource usages of the server. They monitor the state of hardware resources such as processors, and report consumption summaries. All of these profiling mechanisms are essential to our purpose. However, more precision and reasoning structures are needed to achieve our goal. We need the following:

- Specific timeline information that can identify not only spatial hot spots but also the temporal features.
- Profiling information from one layered aspect that can relate to another, such that a reasonable mapping can be inferred.
- Profiling information that can be summarized and compared with the associated computing context.
- Inter and intra communication should be integrated in the profiling mechanism.

To this end, we put together the profiling mechanism needed to fulfill the requirements of our purpose. In particular, our goal is to assist in performance antipattern detection, as well as feedback generation. In addition, software systems often run in networked environments; the profiling mechanism needs to flexibly accommodate and adapt to distributed settings as well. In Table II, the profiling mechanisms are categorized into contexts that are matched to SALs. Each of the profiling methods is given a brief description followed by its suggestive profiling method. The purpose of the method is to explore other available options in the same context level of the system to give leeway in enhancing the performance result or avoiding bad practices. The practitioner can take advantage of the suggestive profiling approach to explore design options to achieve better performance. In the performance refactoring phase for antipattern detection, exploring the suggestive solution methods may provide a clue to a final solution.

1) Path Profiling: The framework adapted the terminology of Path Profiling from data flow analysis [7], and the pathwise decomposition concept from path-oriented analysis [8]. The concept of path-oriented profiling is based on the measurement of different execution paths. If we can make the most common path execute faster, the response time may be shorter. Path profiling also provides insights on improving performance by revising the chances of executing certain paths, or improving the efficiency of the path. If the frequency of execution of a path is relatively high, the savings in execution time can become significant. The dependency of paths and associated components can be identified.

Path profiling can be seen mostly at the level of software components and libraries, and in the programs. In the software layer, execution path is the lowest unit of refinement for the software system. For each thread, path profiling is also essential to discover performance problems. Software elements can be explored along the execution path.

The concept of execution path can be extended to accommodate information flows. Information flow tracking in the program is done to understand the pattern of execution paths as certain requests are being executed. It can also be extended to include communication routes that connect the execution path between server nodes. The high level view of path profiling can be observed at the subsystem level where interactions between clients and servers use different routes. Alternative routes between them may be the result of dispatching policy or adapting flexible algorithms to react to traffic congestions.

2) Thread Behavior Profiling: A thread is a sequence of instructions and the representation of a logical computational unit, which can be scheduled to run by the operating system. A pool of working threads can be initialized before the real workload picks up and be ready to respond without delay. Adapting this thread model also has the benefit of executing true concurrency in a multicore environment. Thread Behavior profiling is about the observation of thread creation, execution, destruction, and management of threads. At the system architecture level, processor affinity can be monitored as multithreaded programming specifies the arrangements. The combination of resource distribution and the management policy such as the number of threads and their running priorities affect the overall performance. Threads can also be viewed as another form of dynamic path, because every thread runs on its own copy of instructions. The observation and summary of individual threads can be performance indicators of how well they coordinate and cooperate.

The context of thread profiling is at the task level where the system adapts a multithreaded programming model to carry out designated services. Depending on the features of the application, a threading system usually provides facilities to adjust the behavior of threads to improve their performance. A thread is the lowest logical task unit that we can monitor in the profiling. It provides the flexibility of measurement in both higher and lower levels of the system. At the higher level, an end-to-end performance scenario can be profiled by integrating thread behavior profiling in each subsystem with the information flows. At the lower level, each task and its resource usage by a specific thread can be analyzed. The flexibility of thread monitoring facilitates the whole system profiling at a fine-grained level.

3) Network Profiling: Networked systems have become the infrastructure for every computing system no matter where it resides, either in enterprise clusters, virtual hosts, or Clouds. The complexity of interaction patterns among servers increases exponentially. As the network becomes the computing platform, it isnecessary to profile and monitor network traffic. Our model classified the networking in the context of middleware and includes proxy, router, programming middleware, and other network topologies such as multi-tier and clustering. Network profiling focuses on getting information about requests and responses, and the underlying communication protocols. Measurement information can be about the number of requests at the higher level, and the number of network packets at a lower level. It can also look into the data that packets carry, and profile the characteristics of the request message. Performance of the network activities can contribute to either the network interface capacity such as queueing buffers, or the processing speed of the server.

Network profiling information can be referenced by its connected systems and software components to make more

informed design decisions. It can also contribute to the construction of analytical models such as queueing networks with more realistic estimation of queue lengths and arrival rates. Finally, for information flow analysis, networking profiling facilitates visualization of the activities in detail, and provides the overall picture of the networked components and systems.

4) Resource Profiling: Conventional resource profiling is about profiling physical resource usage. The resource here refers to the hardware components of the server architecture. The higher capacity the server has, the higher the performance. In practice, cost, overhead, and other limitations can affect the choices and ability to obtain more capacity. Resource profiling gives information about resource characterization of the server and its computational capacity. It usually hints at the potential performance outcome of the system.

One puzzle of resource profiling is in the context of the software abstraction layer, where engineers often want to know precisely the resource consumption of a specific software entity. For example, if the load a working thread contributed to a processor load can be calculated, then the total resource requirements can be estimated given the number of threads. Physical resource monitoring alone cannot fulfill the job because it is hard to get a clear view of workload of a working thread without being affected by other programs and the operating system management policies.

To estimate the resource consumption of a software entity, instead of applying measurement and estimation using a system utility, we turn to resource consumption estimation using the number of instructions executed. Resource profiling in a software entity can be abstracted to the lowest level of computation using instructions before putting them into operation. These instructions are the ones that are consuming resources. Therefore, the system resources that need to accommodate these executions can be estimated. Depending on the type of resource, a processor's workload can be approximated by the number of instructions running on it. The instructions can be further categorized by the type of memory access that can have different number of cycles. For a storage disk, the number of I/O accesses resulting from the execution can also be counted toward resource consumption. Advanced resource modeling such as register and cache behavior can be devised to extract the measurements. We name this type of resource profiling as abstract resource profiling in contrast to the physical resource profiling method discussed above. The mapping between software entity and the abstract resource can be clearly identified and the resource consumption can be reasonably estimated.

The application of resource profiling can be applied to most of the system. Software systems usually adapt monitoring mechanisms to extract the information from system hardware components. Abstract resource profiling will need the other profiling mechanisms in the suggestive profiling to supply the measurement. All the conventional resources in the antipattern domain, such as CPU, Memory, Network, and Disk, can participate in resource profiling.

5) Request and Workload Profiling: Workload plays an influential role on the performance. Although the profiling method is not categorized into any system abstract layer, it affects every part of the system. Depending on request patterns, one subsystem or component may have a larger workload then that of the others. If the workload exceeds the planned capacity of components, a bottleneck would occur. Similar considerations apply to the performance impact of threads, in that they may spend more time processing requests, and the throughput may suffer. In the context of middleware, interactions between servers may take longer under heavy workload due to waiting for responses. In the system architecture layer, the computing workload coming from the above layers has a direct impact on resources and overheads when switching between tasks to meet the services.

The request and workload profiling provides information about the impact on each context of the system. Engineers can evaluate the scenario of request pattern in each focused context independently. The profile relating only to an individual context can be used as a specific source of information to focus on that particular design improvement. On the other hand, the profiler can characterize the workload, so that the design of the system can be adjusted flexibly if it is possible to increase the performance. For example, if the profile of the request is CPU bound, engineers may consider distributing them as evenly as possible to available servers. Another alternative solution occurs when CPU bound requests prefer to be sent to a CPU with higher performance. We also note that both physical and abstract resource profiling information can be used to characterize workloads. This information can help engineers understand the impact of requests.

In practice, request types and their temporal patterns are dynamic, and the workload characteristic is not known a priori. The suggestive profiling mechanism can be used at deployment to selectively monitor requests, and create the workload profile associated with the context. The profiling can select time periods or focus on a specific component for performance monitoring.

IV. Detection and Solution Suggestion Processes

To be able to use suggestive profiling in performance antipattern detection and solution feedback, one has to understand performance baseline. Performance baseline is the summary of current performance of the system. It can be used for performance debugging to check against requirements. Preliminary performance evaluation can be obtained by establishing the baseline of the target system or components. The content structure of the baseline is compatible with the system abstraction layer, in which path-oriented, threading, networking, and resource profiles are recorded. In each context of the profiling, performance metrics such as execution time and process utilization are available for verification. Performance baseline can be created for every element in each context of suggestive profiling method including subsystem, component, thread, network, and hardware component. Depending on the needs of debugging activity, engineers can zoom in on targeted components and their interactions when high level information is not enough. In short, performance baseline is an agile performance filtering and debugging tool used in the software development process to collect targeted performance snapshots.

With performance baseline as the debugging framework used in the system development process, activities in performance modeling processes can share the data it collects. Since both of the processes are synchronized, the performance metrics collected are reflected in the latest status of the system. In the refactoring phase, performance antipattern detection and solution suggestion feedback mechanism can be executed with the help of suggestive profiling. Performance baseline

is accessed to extract the metrics of the detectable features needed with each antipattern. The value of detectable features can be checked to see if the symptom appears. If not detected then the antipattern does not exit, and no action is needed. If detected, the solution may be applied to solve the performance anomaly in the system. Antipattern solution can point to problem spots and give approaches to resolve the problem. As discussed in previous sections, refactoring dilemma exists because we need more clues to generate detailed feedback for redesign and eliminate the anomaly. To close the gap, suggestive profiling can be used to narrow down the root causes.

We recall that the suggestive profiling method consists of profiling path, thread behavior, networking, and resources in the layered context defined in SAL, and each profile can be evaluated independently. In order to converge on to the root cause, we examine the suspicious context revealed by profiling. Within the root cause, performance metrics gathered by the profiling mechanism are verified against the performance antipattern symptoms to discover corresponding solutions. For example, if the root cause specifies a component is the bottleneck in the system, we can further analyze execution time profiles of paths, and make a specific solution suggestion in refactoring. Although we reason at the specific component level, levels higher and lower than the root cause context can also be inferred for potential redesign options. For example, if the root cause is a thread's performance, execution paths at a higher level or resources at a lower level can be inferred as the relevant factors. Solution suggestions can therefore provide more relevant information.

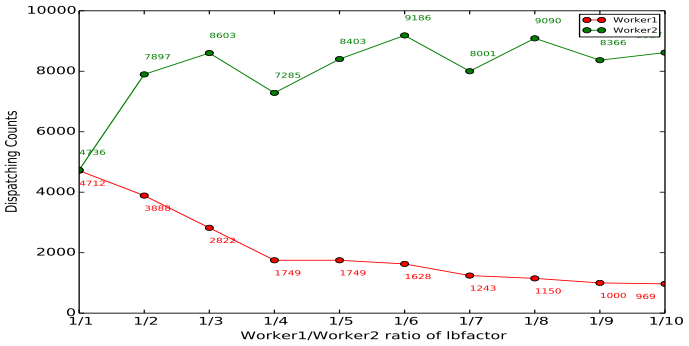

Figure 3. Number of Requests Dispatched to worker1 and worker2 of JBoss

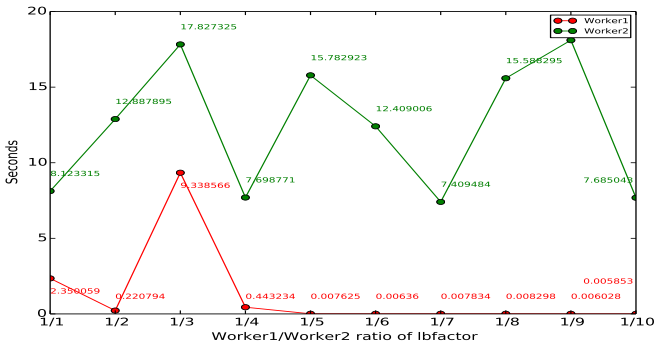

Figure 4. Average Execution Time of Requests Dispatched to worker1 and worker2 of JBoss

V. Case Study

A. Framework and Tools Implementations

The suggestive profiling was implemented using Pin tool [9]. An associated data analytic framework for performance debugging, antipattern detection, and solution suggestions were created to work with the tool. Together they can trace each executed instruction of an application for path-oriented analysis. It also provides other instrumentation points including basic blocks, routines, images, and complete application. These abstractions can be used to identify call graphs, accesses to libraries, and inter and intra component communication, which can easily fit in the system models.

In the framework, our tool includes following utilities to facilitate suggestive profiling:

- Data collection, processing, and management for different profiling methods.
- Communication between software components and systems is profiled with the help of protocol plugins. Each plugin specifies the pattern of interactions.
- System resource monitoring, logging, and analysis.

B. Experimental Setup in Production Systems

RUBiS [10] was setup on Xen 3.1.2 virtual machines hosted on Dell Optiplex 960 with 4 CPU and 4GB RAM. Each virtual machine runs on one virtual CPU and 512MB RAM. Each virtual server is connected to a virtual network interface with a unique network address. The virtual network connection is created by an ethernet bridge, and a DHCP server is setup to assign unique network address to each virtual server.

RUBiS was installed with Apache2 httpd [11], JBoss AS 4.3.2 [12], and MySQL [13] as the web server, application server, and database respectively. The suggestive profiling Pin tool was installed on the web server to generate the performance baseline for the web server, and to demonstrate performance antipattern detection and solution feedback suggestions. On each server virtual machine, we measure the load with sysstat utility to collect CPU, memory, network, and disk usage every one second. All the traces and logs generated from the suggestive profiling Pin tool, and the system utility measurement logs are collected afterward to avoid interferences with the workload of the server.

C. Performance Antipattern Detection and Solution Suggestion

The proposed framework is applied to detect root causes using the exemplified performance antipatterns in this study. Once a performance antipattern is documented and relevant context-dependent solution suggestions are recorded, they can be supplied directly as possible solutions. Practitioners have the option to choose between applying documented solutions, or creating a new antipattern instance that is specific to the scenario of the system under review. A short discussion for each antipattern studied in our case study are described below.

- Unbalanced Processing Antipattern

  Description Problem occurs when processing connot make use of available processors.

  Application Best practices of dispatching between an Apache Web Server(WS) and multiple JBoss Application Servers(ASs).

  Detection Unbalanced processor utilization or service

time duration observed between ASs.

Solution Adjust dispatching configurations by making changes to the proportion of load sent to different wrokers, using lbfactor in mod_jk.

Experiment Experiment using default transition workload with 500 users whose requests are served by two JBoss ASs and one Apache WS is presented with different ratios of lbfactors. Figure 3 depicts the number of requests dispatched from WS to either worker1 of AS1 or worker2 of AS2. The load balance ratios are marked as the ticks on x-axis. We observed that the number of requests to worker1 and worker2 are approximately proportional to the weight of the lbfactor. Figure 4 depicts the execution times of requests dispatched from WS to worker1 or worker2 with different dispatching ratios. The represenation is not liner, but it reflects the scenario in which the preferred node spends more time processing because of the improper load setting. Both the dispatching number and the time duration with different ratio are interrelated.

- More is Less Antipattern

  Description Problem occurs when a system spends more time trashing than accomplishing real work because there are too many processes relative to available resources.

  Application Best practices of deciding what is the appropriate number of working threads needed to serve in an Apache Web Server.

  Detection Comparison of throughputs between settings using different number of threads.

  Solution Adjust the number of threads based on peformance.

  Experiment In this experiment, the RUBiS benchmark with different sets of configurations was run, and the outcome of the performance baselines and their differences were observed. Figure 5 depicts a test run with 800 users using various sets of worker configuration shown in the legend. The numbers in the legend correspond to the order of Apache httpd server's configuration variables: StartServers, MinSpareThreads, MaxSpareThreads, ThreadsPerChild, MaxRequestWorkers, and MaxConnectionPerChild. For each request types from the benchmark, the corresponding average response time in seconds is shown.

- God Class Antipattern

  Description Problem occurs when a single class either performs all of the work or holds all of the data of the application.

  Application Checking both design and implementations for better object-oriented paradigms.

  Detection The number of control or data flow in a programming class that is higher than predefined threshold.

  Solution Refactor the design and implementations of the detected class.

  Experiment Developer documentation of httpd states that, all requests pass through ap_process_request_internal() in request.c of the web server. We want to observe the information flow and its frequencies when a real workload is used. Before checking the flow of information to the targeted function from suggestive profiling tool, symbol table of the ap_process_request_internal() function is extracted from the Executable and Linkable Format (ELF) of httpd to find the static address of the function. This information is used to acquire the structure of instructions in execution order. Request flow information collected from suggestive profiling tool is checked with the static structure of the function to produce the request flow graph at run time. It should be noted that this type of analysis is possible only when source code is available.

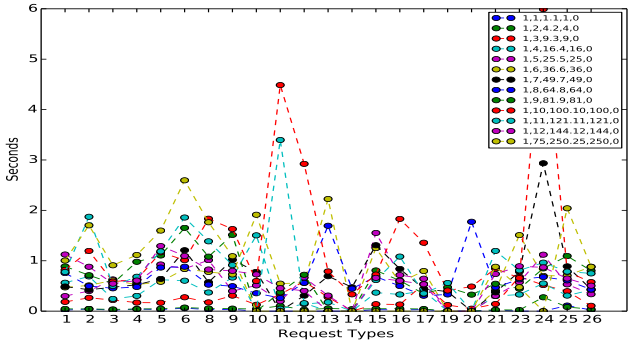

Figure 5. RUBiS Benchmark with 800 Users Using Various Sets of MPM Configuration

VI. Related Works

In the following subsections, approaches related to performance antipattern detection, diagnosis, and solutions that are closely related to our work are discussed.

A. Performance Antipattern Detection

Performance antipattern detection has been addressed in different systems and models. Performance detection in component based enterprise systems was proposed in [14], where a rule-based performance diagnostic tool is presented. The tool can work with EJB applications, in which data from runtime systems is extracted and applied with rules for antipattern detection. The method is limited to EJB systems. Another performance detection and solution approach presented in [15] discusses the performance antipattern in the context of the Palladio Component Model (PCM) [16] software architecture modeling language. A queueing model is derived from the software model in PCM, and is solved to generate performance indicators. The predictive values are matched against performance antipattern rules in PCM to determine whether an antipattern exists. Once detected, solutions can be applied. It uses iterative processes to solve antipattern one by one. A similar approach but using Architecture Description Language (ADL) can be found in [17]. In [18], performance antipatterns are presented using logical predicates. The problem description for an antipattern is interpreted and presented using first order logic equations. The approach focused on antipattern presentation and detection. In [19], Performance Problem Diagnostics (PPD) approach combines search techniques with systematic experiments for performance antipatterns detections. The search is based on a decision tree technique to locate possible root causes, while the detection strategies are based on goal-oriented experiments. All these techniques described

above are based on models and heuristics, and do not describe how to understand baseline performance or setting threshold values at which system configurations should be changed. Our framework relies on runtime data collection to understand performance bottlenecks and how to tune system parameters.

B. Automatic Diagnosis and Feedback Generation

In [20], a rule-based automatic software performance diagnosis framework is proposed for detecting performance bottlenecks. Layered queueing models are used to generate performance predictions. The generated performance indices are checked against predefined rules to detect performance bottlenecks. The rules will also suggest mitigation approaches to reduce operations or add resources. The solution feedback is largely dependent on the definition of the rules. The success of the system depends on the extensibility of the rules. The feedback solution depends on translating performance model attributes in design, which are not provided. A similar approach is presented in [21], which extracts software and system architecture and creates a queueing model for performance anomaly detection. In the feedback process, the architecture model is used for redesign considerations. Our framework does not use queueing modules, but relies on profiling. An approach is proposed to address these concerns regarding detection and solution feedback with real system thresholds so that performance antipatterns can be applied in real practice. In [22], a special detection approach for finding the most guilty performance antipattern is proposed. The process checks performance antipattern symptoms against system requirements, and filters out the ones that do not violate them. The final list of performance antipatterns are ranked using scores calculated from equations defined for specific performance criteria. In our current framework, we do not rank the antipatterns. However, the use of baseline will eliminate some antipatterns from consideration, if the performance is acceptable.

C. Solution Suggestions

There are many published approaches suggesting solutions to overcome performance bottlenecks. Our discussion here focuses on the ones that are related to performance anomaly detection and root cause analysis. In [23], the performance anomaly clustering method is used to narrow suspicious components in distributed systems. Clusters are used to chain components together when they are affected by the same root causes. The clustering is based on the similarity of the performance indices. To identify the problematic performance spots, relationships between groups of clusters are compared. Thus performance anomalies are identified at higher levels: such as the server level. Further diagnosis steps will need to rely on the practitioner's system knowledge. A framework for controlling system configuration parameters to adjust performance was proposed by Stewart et al. [24]. The coverage of the approach depends on the number of controlled configuration used. In practice, it is not feasible that every configuration and manifestation can be covered. Our approach collects data on the software and the system, and establishes the performance measurement specifically reflecting the real scenarios of the system under performance debugging. To discover the root causes, systematic processes are proposed which provide suggestive performance anomaly solutions.

VII. Conclusion and future work

In this paper, we address a critical need in detecting peformance bottlenecks, relating them to known antipatterns and utilizing appropriate soultions. Our approach is based on suggestive profiling methods for different levels of abstractions. Common profiling include path-oriented profiling, thread behavior profiling, networking profiling, and system resource profiling. For each of these profiling methods, we include a suggestive profiling method, and we suggest alternatives for re-engineering the software system to achieve better performance. Request and workload profiles can also be generated through the suggestive profiling tool. This technique is used in the solution suggestion during refactoring phase of performance engieering, and is synchronized with software development cycles. The suggestive profiling tool and the framework utility tools have been implemented and demonstrated using RUBiS benchmark to evaluate performance bottlenecks.

There are limitations in matching some performance antipatterns with detectable features, and thus they cannot be detected directly. Most of these undetectable antipatterns are due to design decisions. Thus, an intimate knowledge of the designs can help in the detection and elimination of those performance antipatterns. If the design decisions can be systematically codified, then it will be possible to extend our framework to other performance antipatterns.

In the future, we plan to further analyze factors influencing antipatterns in different domains including high-performance computing, e-commerce or workflow data management, and extend the framework with appropriate tools. We also plan to make the framework Cloud-ready, so that general performance antipatterns in the computation of distributed systems can be categorized, detected, and resolved systematically.

Acknowledgment

This work is supported in part by the NSF Net-Centric and Cloud Software and Systems Industry/University Cooperative Research Center and award 1128344. The authors would also like to thank Dr. Shih-Kun Huang of National Chiao Tung University, Taiwan for his assistance with this project, and valuable comments and suggestions to improve the quality of the paper. We also acknowledge David Struble for his help in proofreading.

References

[1] R. Johnson, R. Helm, J. Vlissides, and E. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1995.

[2] C. U. Smith and L. G. Williams, Performance solutions: a practical guide to creating responsive, scalable software. Addison-Wesley Reading, 2002, vol. 1.

[3] C. U. Smith and L. G. Williams, "Software performance antipatterns." in Workshop on Software and Performance, 2000, pp. 127–136.

[4] C. U. Smith and L. G. Williams, "New software performance antipatterns: More ways to shoot yourself in the foot," in Int. CMG Conference, 2002, pp. 667–674.

[5] C. U. Smith and L. G. Williams, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in Computer Measurement Group Conference, 2003, pp. 717–725.

[6] C. U. Smith, "Introduction to software performance engineering: Origins and outstanding problems," in Formal Methods for Performance Evaluation. Springer, 2007, pp. 395–428.

[7] T. Ball and J. R. Larus, "Efficient path profiling," in Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture. IEEE Computer Society, 1996, pp. 46–57.

[8] J. Huang, "State constraints and pathwise decomposition of programs," Software Engineering, IEEE Transactions on, vol. 16, no. 8, 1990, pp. 880–896.

[9] C.-K. Luk et al., "Pin: building customized program analysis tools with dynamic instrumentation," in Acm Sigplan Notices, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[10] Rubis web site. [Online]. Available: http://rubis.ow2.org/ [retrieved: August 2014]

[11] Apache http server project web site. [Online]. Available: http://httpd.apache.org/ [retrieved: August 2014]

[12] Jboss application server web site. [Online]. Available: http://jbossas.jboss.org/ [retrieved: August 2014]

[13] Mysql community server web site. [Online]. Available: http://dev.mysql.com/ [retrieved: August 2014]

[14] T. Parsons, "Automatic detection of performance design and deployment antipatterns in component based enterprise systems," Ph.D. dissertation, Citeseer, 2007.

[15] C. Trubiani and A. Koziolek, "Detection and solution of software performance antipatterns in palladio architectural models." in ICPE, 2011, pp. 19–30.

[16] F. Brosig, S. Kounev, and K. Krogmann, "Automated extraction of palladio component models from running enterprise java applications," in Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, p. 10.

[17] V. Cortellessa, M. De Sanctis, A. Di Marco, and C. Trubiani, "Enabling performance antipatterns to arise from an adl-based software architecture," in Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on. IEEE, 2012, pp. 310–314.

[18] V. Cortellessa, A. Di Marco, and C. Trubiani, "Performance antipatterns as logical predicates," in Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on. IEEE, 2010, pp. 146–156.

[19] A. Wert, J. Happe, and L. Happe, "Supporting swift reaction: Automatically uncovering performance problems by systematic experiments," in Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013, pp. 552–561.

[20] J. Xu, "Rule-based automatic software performance diagnosis and improvement," Performance Evaluation, vol. 67, no. 8, 2010, pp. 585–611.

[21] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier internet services and its applications," in ACM SIGMETRICS Performance Evaluation Review, vol. 33, no. 1. ACM, 2005, pp. 291–302.

[22] V. Cortellessa, A. Martens, R. Reussner, and C. Trubiani, "A process to effectively identify guilty performance antipatterns," in Fundamental Approaches to Software Engineering. Springer, 2010, pp. 368–382.

[23] S. Iwata and K. Kono, "Clustering performance anomalies based on similarity in processing time changes," IPSJ Online Transactions, vol. 5, no. 0, 2012, pp. 1–12.

[24] C. Stewart, K. Shen, A. Iyengar, and J. Yin, "Entomomodel: Understanding and avoiding performance anomaly manifestations," in Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on. IEEE, 2010, pp. 3–13.