

A Framework for Detecting Performance Design and Deployment Antipatterns in Component Based Enterprise Systems

Trevor Parsons
Performance Engineering Laboratory
University College Dublin
Ireland
trevparsons@gmail.com

Abstract

In this paper a framework to automatically detect design and deployment antipatterns in component based enterprise systems is presented. The approach taken monitors a running system and makes use of statistical analysis and techniques from the field of data mining to summarise the performance data collected. Performance antipatterns are detected from the summarised data using a rule-engine approach and are assessed in terms of their performance impact using performance models. Any antipatterns found are presented to the user in a diagrammatic format.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques - *Computer-aided software engineering (CASE)*

General Terms

Measurement, Performance, Design.

Keywords

Enterprise Java Beans (EJB), Antipatterns, Data Mining.

1 DESCRIPTION OF PURPOSE

Poor performance is a major obstacle that has to be overcome when delivering large internet-enabled enterprise systems. Often projects fail or face serious delays when they do not meet their performance goals. As shown in the literature [11] the design of enterprise systems plays a major role in how they meet their performance requirements. However, bad design from performance perspective is very common in such systems. This is quite evident from the number of common performance design mistakes which have been well documented [6] [5] [7]. Design mistakes generally results from the fact that these multi-tier distributed systems are extremely complex

and often developers do not have a complete understanding of the entire application. As a result developers can be oblivious to the performance implications of their design decisions. Fixing serious design-level performance problems late in development is very expensive and can not be achieved through "code optimizations". While the approach of "developing first, optimize later" may work at a class level, re-engineering large parts of an enterprise architecture may prove extremely expensive.

Furthermore, while good design decisions are imperative for attaining performance goals, there are also important decisions that need to be taken during deployment of enterprise systems which affect performance significantly. This stems from the fact that in the case of enterprise frameworks (such as EJB), many decisions, that were inherent in the design and coding of systems in the past, have been abstracted out of the application source code into the deployment settings of the system. In EJB for example the granularity and type of transactions can be specified in the XML deployment descriptors of the application. Many such decisions can now be made at the deployment level. As a result, when configuring deployment time settings, different design trade-offs that can significantly impact performance must be considered. However, a real problem for system deployers is that component based frameworks can require a complex deployment strategy in order to make use of the many services provided. There are numerous different settings that need to be tuned correctly in order to have an efficiently running system. Examples include caching policies, object pools, database settings etc. Again a lack of understanding of enterprise applications (due to their complex nature) means that developers are often unsure as to how to go about configuring them correctly.

The question remains: *how do the current performance tools address the situation?* Performance tools to date have in most cases focused on profiling running systems. However, there is generally a vast amount of performance data produced by monitoring even the smallest of applications. When profiling large-scale enterprise applications with high user loads, this information can become unmanageable. The main problem is that current tools merely present this data to the user and the onus is on the tool user to figure out what this data means. Developers and performance engineers attempting to identify performance problems, are required to sift through large volumes of information which can be an onerous task. Such tools do not greatly help developers to find performance design problems or to gain an understanding of the system under test. Furthermore, even if developers manage to find performance problems in their applications using such tools, often they are unsure how to go about rectifying the situation.

There is clearly a need to produce more sophisticated performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
2nd International Doctoral Symposium on Middleware '05 November 28- December 2, 2005 Grenoble, France
Copyright 2005 ACM 1-59593-267-4/05/11 ...\$5.00

tools that do not merely monitor, systems but that also perform analysis on the data collected, in order to make sense of this data, and present it to the user in a meaningful format. Tools are especially required to help developers reason about their design decisions, to identify where poor choices have been made, and to suggest alternative courses of action that might be taken. Such tools should fit well with modern development processes such that performance engineering can be easily integrated into the development process, thus helping to avoid late and expensive discovery of performance issues.

2 GOAL STATEMENT

Recently there has been an effort to automatically analyse run-time data in order to alleviate the burden of the tool user [12], [15]. Bottlenecks [12] is a tool that can be used to find performance hotspots in (possibly extremely large) profiles. It uses a search tool to find the hotspots. The search tool manages the bookkeeping of the search and provides heuristics that automatically suggest likely bottlenecks. Our previous work [15] shows how data mining algorithms can be used to detect instances of performance antipatterns by analysing run-time data. This effort of automatically analysing performance data needs to be continued in order to improve performance tools, so that they can better inform users of where the performance problems lie. Problems detected should ideally be annotated with descriptions of the issue detected as well as a solution that can be applied to alleviate the problem. For example the Smart Analysis Error Reduction Tool (SABER) [10] used for programming error detection provides supporting information which explains why the code is defective. It also provides contextual path and data flow information which can explain how the defect occurred. The research presented in this paper focuses on the need to continue this trend and to produce performance tools that do not merely monitor systems but that also perform analysis on the data and advise developers based on expert knowledge. In particular this work focuses on analysing the design of enterprise systems, since good performance design is a crucial factor that is required to have an efficient system. Because many design decisions are now taken at the deployment stage, deployment settings is also analysed. In this paper I introduce a framework that automatically detects common, well known performance design and deployment bad practices (i.e. antipatterns [6]) in component based systems by analysing run-time data.

A number of issues will be addressed by this research:

(a) The **lack of understanding** that developers experience when building enterprise systems can be **addressed** on a number of levels. Firstly, software antipatterns document bad practices in software development as well as their corresponding solutions. Thus, detecting instances of antipatterns and highlighting them to the developer, not only allows the developer to understand what is wrong with the application, but it also allows the developer to easily rectify the situation by refactoring the application with the solution provided. Secondly, antipatterns provide a high-level language for discussing design issues. Developers can use pattern names to clearly and efficiently discuss within the development team particular implementation issues that may arise. Also, exposing developers to performance related antipatterns helps them form a sense of performance intuition since the underlying reason as to why a particular antipattern had an adverse effect on performance is underlined in the antipattern description. Consequently developers are less likely to make the same mistake in the future.

(b) The approach **reduces the large volume of data** produced during monitoring by applying data mining and statistical analysis techniques to summarise and find patterns of interest in the run-time data.

(c) Previous approaches to reverse-engineering have relied on static analysis [16]. **Dynamic analysis is however more suitable when analysing the system from a performance perspective.** Dynamic analysis allows for the collection of performance metrics and workload distribution (e.g. by profiling). Performance antipatterns can thus be evaluated in terms of their performance impact. Furthermore dynamic analysis fits well with modern agile software development processes which require a running implementation of the system at each iteration of development.

(d) While much work has been done in the area of bug detection [9] [8], this approach **analyses the system from a performance design perspective.** Performance design and deployment antipatterns differ from performance bugs (or programming errors). Performance bugs can be considered as mistakes made by developers that they are generally unaware of and that always have a negative effect on performance. Examples of performance bugs are not releasing resources (e.g. database connections), deadlocks, memory leaks etc. Performance design and deployment antipatterns on the other hand are instances of sub-optimal design and deployment choices. Often developers are aware of these decisions, but do not realize the consequences that they have on system performance (e.g. making fine-grained remote calls [15], or setting the value of the thread pool size to a sub-optimal level for the current workload). Although removing bugs can yield performance improvements for a given design, situations may arise where performance goals cannot be met unless the system design is modified.

3 APPROACH

A **prototype** tool for EJB systems is currently being developed. It is made up of five main modules: monitoring, analysis, detection, assessment and presentation (see figure 1).

3.1 Monitoring

The monitoring module is responsible for collecting performance data on the system under test. The data collected can be grouped into three categories (a) information on the running application (b) information relating to the server resources (c) and contextual data.

- Information on the running application (a) is collected using two monitoring approaches. The first approach, makes use of a Java Virtual Machine Tools Interface (JVMTI) [4] agent to monitor the resource usage of the application. The JVMTI is a standard profiling interface for all Java 1.5 compliant JVMs. It can be used to collect resource usage information (such as CPU and memory usage) for Java applications. The second approach collects run-time path information on the running application. A runtime path [14] contains the control flow (i.e. the ordered sequence of methods called required to service a user request), resources and performance characteristics associated with servicing a request. By analysing these paths one can easily see how system resources are being used, how the different components in the system interact and how user requests traverse through the different tiers that make up the system. Run-time path tracing can be used help developers and performance engineers to understand the overall system structure of complex distributed applications. Further ad-

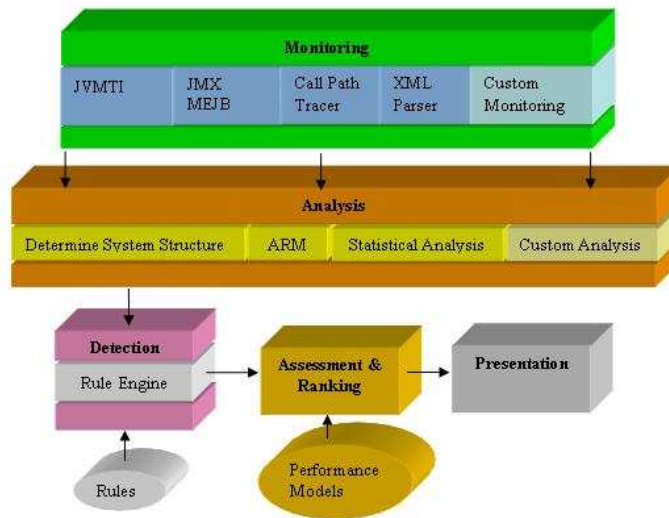


Figure 1. Tool Architecture

vanced analysis can also be performed on the run-time paths e.g. Chen et al. have shown how run-time paths can be used for fault detection and diagnosis [13]. My previous work also highlights the utility of run-time paths and shows how they can be used to detect performance antipatterns [15]. We have recently implemented an end-to-end run-time path tracer for J2EE systems [1] that monitors the path user requests take when traversing through the different tiers in the system (i.e. web, business and database tiers). The tool is non-intrusive and therefore can be applied to the system without having to modify the application or server source code.

- Server resource data (b) is collected to monitor the state of the servers' resources (e.g. thread pools, database connection pools, object pools etc). According to the J2EE Management specification [2] application servers are required to expose this data through a Java Management Extensions (JMX) interface.
- Finally contextual data (c) that describes the different components that make up the system is collected by parsing the EJB XML deployment descriptors. This can be done offline and has no impact on the performance of the running system. This data gives a context to the run-time data collected and avoids the need for static (i.e. source code) analysis. For example, from this data it can be seen what type of components (e.g. Entity/ Session/ Message-Driven) make up the running system, whether they expose remote or local interfaces etc.

3.2 Analysis

The analysis module performs two main tasks:

- Determine the overall system structure: Parsing the XML deployment descriptors (performed by the monitoring module) gives the different components that make up the system. This information can be utilised together with the system run-time paths (which show the relationships between the different components) to determine the overall system structure.
- Apply data mining and statistical analysis techniques to the

run-time data: Using these techniques the volume of data summarised and related to the different components in the system. Interesting relationships between components can be found using advanced data mining techniques (e.g. Association Rule Mining). The output of this module is both in the form of statistical information (that describes the different components) and rules (e.g. association rules) that describe patterns of interest and relationships in the data. Rules that suggest interesting patterns of method calls have been documented in [15].

This module has been implemented. However research into data mining algorithms that can be applied to run-time data is ongoing (e.g. I am currently investigating different data mining techniques that may be used to identify loops in call paths).

3.3 Detection

The detection module uses a rule-engine approach to highlight the different antipatterns that exist in the system. Antipatterns are described in terms of *rules* (i.e. rule-engine rules, as opposed to the rules produce by data mining discussed in the last section). The rules are applied to the data produced during the analysis phase which is loaded into the rule-engine in the form of *facts*. The current implementation makes use of the Jess [3] rule Engine. Figure 2 shows a sample rule.

```

Example Rule

(defrule Detect_Needless_Session_Antipattern
?comp <- (component (type stateless) (callee ?comps_called))
(test(EntityorDB ?comps_called)
(test(DoesComponentRequireServices ?comp))
=>
(assess_needless_session_antipattern))

```

Figure 2. Example of a Jess Rule

The antipattern that can be detected by the rule in figure 2 describes a situation where session beans are being overused within an appli-

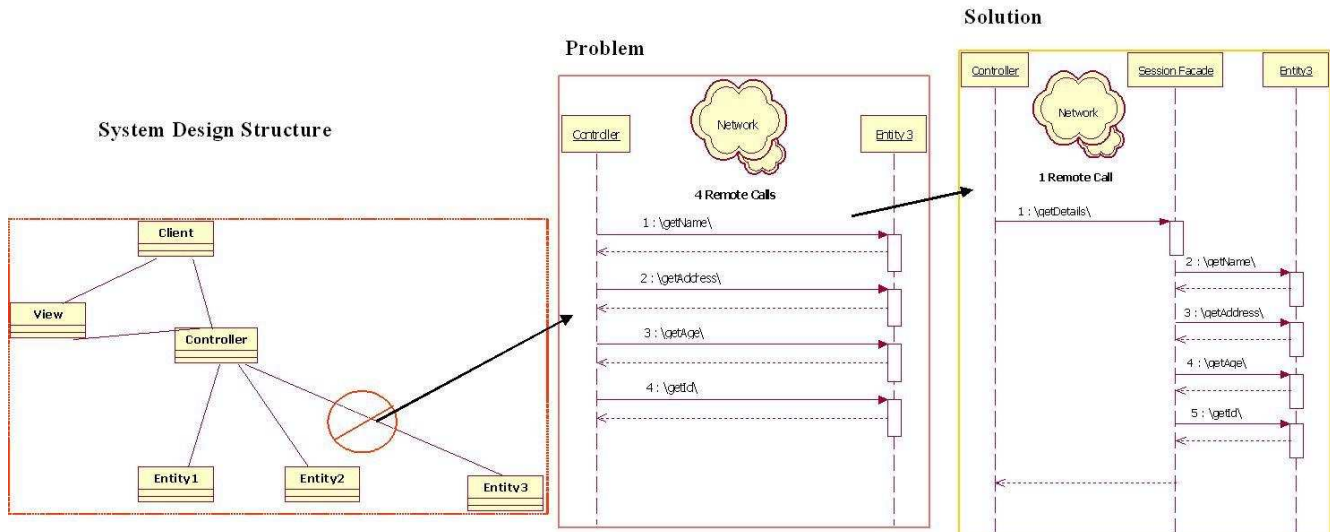


Figure 3. Overall System Design Highlighted with Antipatterns

cation. Session beans are generally used (justifiably) to implement process logic or workflow for business entities. In general a session bean should be used only if services that are supplied by the EJB container are required (e.g. security, transactional services etc.). If no services are required and no transactional context exists a plain old java object (POJO) will generally suffice. The consequences of overzealous use of sessions is typically suboptimal system functioning from a performance perspective. This results from the overhead at runtime required to create instances of session beans when lighter weight POJOs would do. This antipattern is described in detail in the literature [5].

The rule in figure 2 is written in Jess [3] a syntax similar to Lisp. The rule describes the antipattern outlined in the previous paragraph. It identifies session beans that do not require container services and that do not interact with any business entities. The first line of the rule defines the rule name. Line 2 matches a fact that represents a stateless session bean and a list of the components that it calls (i.e. callees). The fact is stored in a variable *comp*. The list of callees is stored in a variable *comps-called*. The variables can be passed as arguments to functions. The function *EntityOrDB* in line 3 is a boolean function that checks the list of callees to see if they contain either an entity bean or database (i.e. is the session bean used to process logic or workflow for business entities?). It returns true if no entities or databases are found in the list of callees. The function *DoesComponentRequireServices* in line 4 is a boolean function that checks if the component *comp* makes use of any of the EJB container services. It returns true if no services are used by the component. If all conditions of the rule are met, the rule fires and it is assessed.

An implementation of our detection module has been developed for a number of well know EJB performance antipatterns [6] [5]. The remaining two modules have yet to be implemented.

3.4 Assessment

When an antipattern is detected a rule is fired in the rule engine and the antipattern is assessed. For each individual antipattern type a corresponding *performance model* is created. The performance

models can be used in conjunction with the collected performance metrics (collected by the monitoring module) to assess the performance impact of each antipattern instance. The assessment module's ranking mechanism ranks antipattern instances of the same type in terms of their performance impact. This allows tool users to easily find antipatterns of greater significance.

3.5 Presentation

The *presentation module* displays any antipatterns found in the system using UML diagrams. The antipatterns are augmented with performance data, example scenarios, the problem description and corresponding solution. Figure 3 shows sample UML output from the presentation module. The diagram shows the overall system structure highlighted with any antipatterns found. In the diagram a situation is shown where there is an opportunity to replace fine grained remote calls with one coarse remote call and thus reduce network traffic.

The prototype outlined above requires **the creation of performance models** for antipatterns so that they can be assessed. The models will be created separately by performing performance tests on the antipatterns. The results (performance models and example scenarios) will be documented as part of the antipattern description, thus adding valuable information that might help developers to understand when antipatterns can really impact on the system.

4 Evaluation

To validate the work, it will be applied to a number of real applications, as well as to a number of applications that contain known antipatterns. We will refactor any antipatterns found, as suggested by the tool. System performance measured before and after using our tool will be compared.

Also, for the approach to be useful it needs to have a certain number of *quality attributes* which will be evaluated:

- It has been claimed that current performance tools are lacking since they require a large effort on the part of developers who

use them. Therefore, it is imperative, that our tool *does not inundate developers with large numbers of antipatterns*. To address this the assessment module's ranking mechanism will rank antipatterns so that developers can focus on those that are most severe first.

- It is important that false positives/negatives be kept to a minimum. The number of false positives/negatives will give a good indication of how *accurate* the detection module is.
- For the approach to be useful it should be highly *extensible*, so that developers can add in antipattern descriptions (and corresponding performance models) that may be specific to their own development process, application or domain.
- The tool prototype should *perform* reasonably well. The monitoring phase uses techniques similar to those used by current profilers and thus it is expected to be acceptable. Although, the analysis, detection, assessment and presentation phases of our framework are performed offline, the performance of these phases should not exceed an acceptable level, which might inconvenience users.
- The above quality attributes should all *scale* when the tool is applied to large systems.

5 Contributions

Its main aim of this work is to assist developers attempting to build performant enterprise systems. This work contains a number of contributions:

- It reduces and makes sense of the data collected by many of today's performance profilers. Current tools tend to produce too much data. This work makes use of statistical analysis and data mining techniques to summarise the data collected and to find patterns of interest that might suggest performance problems.
- While most of today's performance tools tend to focus on identifying hotspots and programming errors (e.g. memory leaks, deadlocks etc.), this work focuses on analysing the system from a performance design perspective. Since design has such a significant effect on performance it is essential that work continues in this area.
- Through the use of performance models our approach has the ability to identify potential antipatterns. Potential bottlenecks can be highlighted by the assessment module. Potential antipatterns can be analysed in respect of different workloads and the potential performance impact calculated.
- Unlike with many of today's performance tools problems identified are annotated with descriptions of the issue detected as well as a solution that can be applied to alleviate the problem. This approach of identifying and presenting antipatterns to developers helps them understand the mistakes that have been made and the underlying reason as to why performance was effected. It also allows developers to easily rectify the situation by applying the solution provided, as well as providing for a high level language to discuss such problems. This has been achieved by applying an expert system to the data collected during monitoring.
- Antipatterns will be further documented with performance models and example scenarios. This will give a more complete description of antipatterns allowing developers to evaluate when the different antipatterns really effect performance and when they have only a minor impact.

- The monitoring module extends current profiling tools by providing a monitoring infrastructure with the ability to collect run-time paths for J2EE systems. Today's commercial performance tools do not capture this data. The approach is completely portable and non-intrusive and thus does not require changes to application source code or the server implementation.

6 Acknowledgements

Our work is funded under the Research Innovation Fund and the Advanced Technology Research Programme from the Informatics Research Initiative of Enterprise Ireland.

7 References

- [1] COMPAS J2EE. <http://compas.sourceforge.net/>.
- [2] Java Service Request 77, J2EE Management. <http://www.jcp.org/en/jsr/detail?id=77>.
- [3] Sandia National Laboratories. Jess, the Rule Engine for the Java Platform. <http://herzberg.ca.sandia.gov/jess/>.
- [4] The Java Virtual Machine Tools Interface. <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html>.
- [5] B. Dudney et al. *J2EE Antipatterns*. Wiley, 2003.
- [6] B. Tate et al. *Bitter EJB*. Manning, 2003.
- [7] C.U. Smith and L.G. Williams. *Performance Solutions, A Practical Guide to Creating Responsive, Scalable, Software*. Addison-Wesley, 2002.
- [8] D. Evans. Static Detection of Dynamic Memory Errors. In *In Proc. of PLDI*, 1996.
- [9] D. Hovemeyer and W. Pugh. Finding bugs is easy. <http://www.cs.umd.edu/pugh/java/bugs/docs/findbugsPaper.pdf>.
- [10] D. Reimer et al. SABER: Smart Analysis Based Error Reduction. In *Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis*, 2004.
- [11] E. Cecchet et al. Performance and scalability of EJB applications. In *Proceedings of OOPSLA*, 2002.
- [12] G. Ammons et al. Finding and removing performance bottlenecks in large systems. In *Proceedings of ECOOP*, 2004.
- [13] M. Chen et. al. Pinpoint, Problem Determination in Large, Dynamic, Internet Services. In *In Proceedings of the International Conference on Dependable Systems and Networks (IPDS Track)*, 2002.
- [14] M. Chen et. al. Using Runtime Paths for Macro Analysis. In *In Proceedings of 9th Workshop on Hot Topics in Operating Systems*, May 2003.
- [15] T. Parsons and J. Murphy. A Framework for Automatically Detecting and Assessing Performance Antipatterns in Component Based Systems using Run-Time Analysis. In *The 9th International Workshop on Component Oriented Programming, part of ECOOP*, 2004.
- [16] R. Keller et al. Pattern-based reverse-engineering of design components. In *Proceedings of the International Conference on Software Engineering*, 1999.