PASA^{SM}: A Method for the Performance Assessment of Software Architectures

Lloyd G. Williams
Software Engineering Research
264 Ridgeview Lane
Boulder, Colorado 80302
(303) 938-9847
boulderlgw@aol.com

Connie U. Smith
Performance Engineering Services
PO Box 2640
Santa Fe, New Mexico, 87504-2640
(505) 988-3811
http://www.perfeng.com/

Abstract

Architectural decisions are among the earliest made in a software development project. They are also the most costly to fix if, when the software is completed, the architecture is found to be inappropriate for meeting quality objectives. Thus, it is important to be able to assess the impact of architectural decisions on quality objectives such as performance and reliability at the time that they are made.

This paper describes PASA, a method for performance assessment of software architectures. It was developed from our experience in conducting performance assessments of software architectures in a variety of application domains including web-based systems, financial applications, and real-time systems. PASA uses the principles and techniques of software performance engineering (SPE) to determine whether an architecture is capable of supporting its performance objectives. The method may be applied to new development to uncover potential problems when they are easier and less expensive to fix. It may also be used when upgrading legacy systems to decide whether to continue to commit resources to the current architecture or migrate to a new one. The method is illustrated with an example drawn from an actual assessment.

1. Introduction

While the functionality delivered by a software application is obviously important, it is not the only concern. Over its lifetime, the cost of a software product is determined more by how well it achieves its objectives for quality attributes such as performance, reliability/availability or maintainability than by its functionality.

Recent interest in software architectures has underscored the importance of architecture in determining software quality. While decisions made at every phase of the development process are important, architectural decisions have the greatest impact on quality attributes such as modifiability, reusability, reliability, and performance. As Clements and Northrop note [Clements and Northrup 1996]:

> "Whether or not a system will be able to exhibit its desired (or required) quality attributes is largely determined by the time the architecture is chosen."

While a good architecture cannot guarantee attainment of quality goals, a poor architecture can prevent their achievement.

Architectural decisions are among the earliest made in a software development project. They are also the most costly to fix if, when the software is completed, the architecture is found to be inappropriate for meeting quality objectives. Thus, it is important to be able to assess the impact of architectural decisions on quality objectives such as performance and reliability at the time that they are made.

Performance, both responsiveness and scalability, is critical to the success of today's software systems. Many software products fail to meet their performance objectives when they are initially constructed. Fixing these problems is costly and causes schedule delays, cost overruns, lost productivity, damaged customer relations, missed market windows, lost revenues, and a host of other difficulties. In extreme cases, it may not be possible to fix performance problems without extensive redesign and re-implementation. In those cases, the project either becomes an infinite sink for time and money, or it is, mercifully, canceled.

Performance cannot be retrofitted; it must be designed into software from the beginning. The "make it run, make it run right, make it run fast" approach is danger-

SM PASA and Performance Assessment of Software Architectures Method are service marks of Performance Solutions, LLP.

ous. Our experience is that performance problems are most often due to inappropriate architectural choices rather than inefficient coding. By the time the architecture is fixed, it may be too late to achieve adequate performance by tuning.

This paper describes PASA, a method for performance assessment of software architectures. It was developed from our experience in conducting performance assessments of software architectures in a variety of application domains including web-based systems, financial applications, and real-time systems. PASA uses the principles and techniques of software performance engineering (SPE) to determine whether an architecture is capable of supporting its performance objectives [Smith and Williams 2002]. The method may be applied to new development to uncover potential problems when they are easier and less expensive to fix. It may also be used when upgrading legacy systems to decide whether to continue to commit resources to the current architecture or migrate to a new one.

2. Related Work

Kazman and co-workers describe two related approaches to the evaluation of software architectures. The Software Architecture Analysis Method (SAAM) [Kazman, et al. 1996] uses scenarios to derive information about an architecture's ability to meet certain quality objectives such as performance, reliability, or modifiability. The Architecture Tradeoff Analysis Method (ATAM) [Kazman, et al. 1998] extends SAAM to consider interactions among quality objectives and identify architectural features that are sensitive to more than one quality attribute. Once these sensitivities have been identified, tradeoffs between quality objectives can be evaluated.

Both SAAM and ATAM have similarities to this work. Like PASA, SAAM and ATAM are scenario-based. Analysis of software architectures is based on the use of scenarios to provide insight into how the architecture satisfies quality objectives. In SAAM and ATAM, scenarios are informal narratives of uses of the software. In PASA, performance scenarios are expressed formally, as described below.

SAAM and ATAM consider a variety of software quality attributes in their analysis including reliability, modifiability, and performance. ATAM makes use of Attribute-Based Architectural Styles (ABASs) as an assessment tool. An ABAS [Klein and Kazman 1999] extends the concept of an architectural style by adding a framework for reasoning about architectural decisions with respect to a particular quality attribute (e.g., performance, reliability). PASA also uses architectural styles for analysis with a focus on general characteristics of the architecture together with design guidelines [Williams and Smith in preparation].

ATAM and PASA differ in their approach to performance modeling. ATAM uses analytical models of certain architectural features while PASA uses more general software execution and system execution models that may be solved analytically or via simulation [Smith and Williams 2002]. ATAM is also concerned with interactions between quality attributes and focuses on architectural features where tradeoffs may be required. PASA's primary focus is on performance. However, other quality attributes and tradeoffs between them are considered as well, as discussed below.

Williams and Smith [Williams and Smith 1998] discuss the performance evaluation of software architectures. This paper extends that work with the inclusion of architectural styles and performance antipatterns as analysis tools. It also formalizes the architecture assessment process based on the general software performance engineering process described in [Smith and Williams 2002].

Grahn and Bosch [Grahn and Bosch 1998] report some preliminary results on characterizing three architectural styles: pipe-and-filter, layered, and blackboard. They used a simulation technique to determine the effects of varying the number of components in each style. Their work focused on general performance characteristics of each style rather than techniques for assessing individual architectures.

Balsamo and co-workers [Balsamo, et al. 1998] discuss an approach to performance evaluation of software architectures based on use of the Chemical Abstract Machine (CHAM) formalism. Their method automatically derives a Queueing Network Model (QNM) from a CHAM description of the architecture. Their work and other similar approaches such as [Cortellesa and Mirandola 2000] and [Pooley and King 1999] focus on connecting design notations to performance models.

Other, earlier publications such as [Smith and Williams 1998] and [Smith and Williams 1997] focus on the modeling of a system once it is understood. Much of the performance assessment work is based on the SPE techniques for modeling and analyzing software performance early in the life cycle [Smith and Williams 2002], [Smith 1990]. In contrast this work focuses specifically on the assessment of a software architecture, and addresses the method for gathering information, interacting with clients, and applying SPE principles and techniques to arrive at the results of the assessment.

3. The PASA Method

PASA is a method for the performance assessment of software architectures. It uses the principles and techniques of SPE [Williams and Smith 1998], [Smith and Williams 2002] to identify potential areas of risk within the architecture with respect to performance and other quality objectives. If a problem is found, PASA also identifies strategies for reducing or eliminating those risks.

Our approach is scenario-based. Scenarios for important workloads are identified and documented. These scenarios then provide a means of reasoning about the performance of the software as well as other qualities. They also serve as a starting point for constructing performance models of the architecture if more detailed studies are needed.

The PASA process consists of the nine steps summarized below. The steps are typically performed in the order given. In some cases, however, the order may be varied for some reason, such as to take advantage of the availability of someone with expertise in a particular area. For example, someone with expertise about a particular component may only be available on a particular day. Also, discovery of new information in one step often requires revisiting a previous one, so iteration is common.

1. *Process Overview*—The assessment process begins with a presentation designed to familiarize both managers and developers with the reasons for an architectural assessment, the assessment process, and the outcomes.
2. *Architecture Overview*—In this step, the development team presents the current or planned architecture.
3. *Identification of Critical Use Cases* —The externally visible behaviors of the software that are important to responsiveness or scalability are identified.
4. *Selection of Key Performance Scenarios*—For each critical use case, the scenarios that are important to performance are identified.
5. *Identification of Performance Objectives*—Precise, quantitative, measurable performance objectives are identified for each key scenario.
6. *Architecture clarification and discussion*—Participants conduct a more detailed discussion of the architecture and the specific features that support the key performance scenarios. Problem areas are explored in more depth.
7. *Architectural Analysis*—The architecture is analyzed to determine whether it will support the performance objectives.
8. *Identification of Alternatives*—If a problem is found, alternatives for meeting performance objectives are identified.
9. *Presentation of Results*—Results and recommendations are presented to managers and developers.

The following sections describe each of these steps in more detail.

In some cases, it is possible to conduct a complete assessment in one intensive week. In most others, however, it is likely that the initial assessment will identify potential problems that require performance measurements and modeling before their impact can be quantified. When measurements and modeling are needed, the process typically spans several, less-intensive weeks as data is gathered and evaluated.

3.1 Process Overview

It is important that everyone involved understand the purpose of the architecture assessment, the process that will be used, the architecture and processing information that is required, and the potential outcomes. Thus, the assessment begins with a presentation that describes:

- the rationale for performing an architecture assessment
- overview of SPE goals, model-based approach, data required, and results produced
- the steps in the PASA process
- the architecture information needed to perform the assessment
- tradeoffs between performance and other quality attributes

There is also an opportunity for managers and developers to ask questions and express their concerns.

3.2 Architecture Overview

The goal of this step is for the assessment team to gain a high-level understanding of the architecture before delving into its details. It starts with a presentation of the current or planned architecture by one or more members of the development team.

Typically, the assessment team has already reviewed the available architecture documentation. Thus, this session typically begins with a brief walkthrough of the architecture. This is followed by a question-and-answer session that focuses on missing details and validating the assessors' understanding of the architecture.

This step may involve a significant discovery phase. In many cases, the architecture is simply undocumented. With legacy systems, even if there is architecture documentation, it is likely that there were changes made during implementation that are not reflected in the documentation or that the system has evolved so that the documentation is no longer an accurate reflection of the current state.

We have also found that most architecture documentation is informal. Much of what we receive as architecture documentation consists of box-and-line diagrams that illustrate the infrastructure or "technical architecture." These diagrams may indicate that the system uses WebSphere and NTServer but they do little to reveal the nature of the components that make up the system or the relationships between them. There is also little information on the dynamic aspects of performing common functions.

To overcome these problems, it is often necessary to deduce the architecture from developer interviews, code, and other artifacts. We have found that eliciting scenarios for the important uses of the system is a good way to extract this information. Thus, this step and the next two are often iterated. In many cases, the information provided by precisely characterizing the key scenarios is a major revelation for the development team. Often, this is one of the most valuable deliverables of the assessment.

3.3 Identification of Critical Use Cases

Use cases describe externally visible behaviors of the software. Critical use cases are those that are important to the operation of the system, or that are important to responsiveness as seen by the user. Critical use cases may also include those for which there is a significant performance risk, i.e., those for which there is a risk that, if performance objectives are not met, the system will fail or be less than successful. Typically, the critical use cases are only a subset of the total number of uses of the system.

Use cases are most often described from an end-user point of view. For example, with an automated teller machine (ATM) we might investigate customer use cases that describe deposits, withdrawals, etc. For architecture assessments, however, it is important to also consider other stakeholders. For example, a maintenance upgrade may require downloading large amounts of code to client machines over a local or wide-area network. Maintainers will want to know that this can be accomplished in a reasonable amount of time.

3.4 Selection of Key Performance Scenarios

Each use case consists of a set of scenarios that describe the sequence of actions required to execute the use case. Not all of the scenarios belonging to a critical use case will be important from a performance perspective, however. For example, variants are likely to be executed infrequently and, thus, will not contribute significantly to overall performance

For each critical use case, we focus on the scenarios that are executed frequently and on those that are critical to the user's perception of performance. For some systems, it may also be necessary to include scenarios that are not executed frequently, but whose performance is critical when they are executed. For example, crash recovery or maintenance upgrades may not occur frequently, but it may be important that they are done quickly.

In many cases, particularly with legacy systems, use cases and scenarios are not documented. In those cases, the assessment team must work with the development team to identify the important uses of the software and detail the processing steps that are executed for the key usage scenarios. The process used for eliciting this information is similar to that used for performance walkthroughs, as discussed in [Smith and Williams 2002].

Scenarios are documented using augmented UML sequence diagrams [Booch, et al. 1999], [Smith and Williams 2002]. In an object-oriented system, a sequence diagram describes the objects (individual objects, components, or subsystems) that cooperate to perform a function and the sequence of interactions between them. For non-object-oriented systems (as most of the architectures that we encounter in fact are), a sequence diagram documents the major software units that perform a function and their interactions. The use of sequence diagrams provides two advantages:

- The sequence diagram notation facilitates validation of the processing steps in the scenarios and makes derivation of performance models straightforward.
- When the software architecture is unclear, constructing sequence diagrams helps the assessment team understand the components and their interactions. They also help the assessment team validate their understanding of the architecture, and often inform maintainers of legacy systems of the actual behavior of their system.

3.5 Identification of Performance Objectives

As Kazman and co-workers note [Kazman, et al. 1996]:

"Software architectures are neither intrinsically good nor intrinsically bad; they can only be evaluated with respect to the needs and goals of the organizations that use them."

In order for the assessment to be meaningful, those needs and goals must be clearly defined. Each key scenario should have at least one associated performance objective. Typically, these will be end-to-end requirements. In some cases, however, it may be desirable to break an end-to-end performance objective into sub-objectives that are assigned as performance budgets to each part of the processing

Performance objectives may be expressed in several different ways, including response time, throughput, or constraints on resource usage. In each case, the objective should be quantitative and measurable. Vague statements such as "the system shall be as fast as possible" are not useful. There is no way that you can ever be sure that you have met an objective like this. An objective such as "the end-to-end time to process a typical user request should be less than 2 seconds" is much more useful.

It is also important to specify the conditions under which the required performance is to be achieved for each combination of scenario and objective. The conditions include the workload mix and intensity.

3.6 Architecture Discussions

Because the architecture descriptions provided seldom provide the information required for the assessment, we usually schedule meetings with architects and designers of key portions of the system, once we have identified them, to learn more about component interactions. When appropriate, we also meet with staff who were involved in previous tuning efforts and those who may have performance measurement data to learn as much as we can about problem areas and current performance metrics such as response time, utilizations, and resource requirements of the system.

3.7 Architecture Analysis

Several techniques are brought to bear in analyzing the performance of a software architecture. They include:

3.7.1 Identification of the underlying architectural style(s)

Software architectural styles or patterns ([Shaw and Garlan 1996], [Buschmann, et al. 1996], [Schmidt, et al. 2000]) describe the structural organization of a family of systems that share common architectural features. Architectural styles are similar to design patterns [Gamma, et al. 1995] in that they capture, at the level of overall system organization, recurring solutions to common problems in structuring software systems.

If the architecture is representative of one of the common architectural styles, we can use the general performance characteristics of the style to reason about the performance of that instance. For example, in a layered architecture there is a great deal of overhead as requests are passed from layer to layer. Thus, this style would not be appropriate for situations where high throughput is desired.

If the overall architectural style is appropriate but there are deviations from the archetype in some details, these deviations are explored to determine if they have a negative impact on performance. This is discussed in more detail below.

3.7.2 Identification of performance antipatterns

Antipatterns [Brown, et al. 1998] are conceptually similar to patterns [Gamma, et al. 1995] in that they document recurring solutions to common design problems. They are known as *anti*patterns because their use (or misuse) produces negative consequences. Antipatterns document common mistakes made during software development. They also document solutions for these mistakes. Thus, antipatterns tell you what to avoid and how to fix a problem when you find it.

Performance antipatterns document common performance problems and how to fix them [Smith and Williams 2000], [Smith and Williams 2002]. They capture the knowledge and experience of performance experts by providing a conceptual framework that helps analysts to identify performance problems and suggesting ways of solving them.

Antipatterns are refactored (restructured or reorganized) to overcome their negative consequences. A *refactoring* is a correctness-preserving transformation that improves the quality of the software. For example, the interaction between two components might be refactored to improve performance by sending fewer messages with more data per message. This transformation does not alter the semantics of the application, but it may improve overall performance. Refactoring may also be used to enhance other quality attributes including reusability, modifiability, or reliability.

3.7.3 Performance modeling and analysis

Portions of the architecture may require more quantitative analysis. Initially, a simple analysis of performance bounds is sufficient to identify problem areas. For example, if your performance objective is to process 100 transactions per second then each transaction must take less than 0.01 seconds to complete. Other performance bounds are covered in [Lüthi, et al. 1997], [Majumdar, et al. 1991], [Hsieh and Lam 1987], [Stephens and Dowdy 1984], [Dowdy, et al. 1984], [Eager and Sevcik 1983].

If the analysis of performance bounds indicates the need for more detailed modeling, this is done in a second phase of the assessment process. The use of models makes it possible to quantitatively assess the detailed performance of the software. The models also allow analysts to quickly and easily explore architectural alternatives if problems are discovered.

The models used are deliberately simple so that feedback on the performance characteristics of the architecture can be obtained quickly and inexpensively. The goal is to use the simplest possible model that identifies problems with the proposed architecture. These models can also be carried over into the development phase and elaborated to more closely represent the performance of the emerging software.

The precision of the model results depends on the quality of the estimates of resource requirements. Because these are difficult to estimate for software architectures, SPE uses adaptive strategies, such as upper- and lower-bounds estimates and best- and worst-case analysis to manage uncertainty. For example, when there is high uncertainty about resource requirements, analysts use estimates of the upper and lower bounds of these quantities. Using these estimates, analysts produce predictions of the best-case and worst-case performance. If the predicted best-case performance is unsatisfactory, they seek feasible alternatives. If the worst case prediction is satisfactory, they proceed to the next step of the development process. If the results are somewhere in-between, analyses identify critical components whose resource estimates have the greatest effect and focus on obtaining more precise data for them. A variety of techniques can provide more precision, including: further refining the architecture and constructing more detailed models or constructing performance prototypes and measuring resource requirements for key components.

Two types of models provide information for architecture assessment: the *software execution model* and the *system execution model*. The software execution model represents key aspects of the software execution behavior. Details of the construction and evaluation of these models may be found in [Smith and Williams 2002]

Software execution models are generally sufficient to identify performance problems due to poor architectural decisions [Williams and Smith 1998]. However, in some cases, there may be questions about effects due to contention for resources. When these questions arise, it is necessary to use a system execution model.

The system execution model is a dynamic model that characterizes software performance in the presence of factors, such as multiple users or other workloads, that could cause contention for resources. The results obtained by solving the software execution model provide input parameters for the system execution model. Solving the system execution model provides the following additional information:

- more precise metrics that account for resource contention
- sensitivity of performance metrics to variations in workload composition
- effect of new software on service level objectives of other systems
- identification of bottleneck resources
- comparative data on options for improving performance via: workload changes, software changes, hardware upgrades, and various combinations of each

Details of the creation and evaluation of system execution models are also in [Smith and Williams 2002].

3.8 Identification of Alternatives

If performance problems are found, it is often possible to identify alternatives that may make it possible to meet performance objectives.

The following sections illustrate ways in which architectural alternatives may be identified.

3.8.1 Deviations from architectural style

In some cases, the architecture may resemble one of the common architectural styles in many respects but deviate from the archetype in one or more details. While a deviation from the classic style does not necessarily mean that there is a problem, it does indicate an issue that should be explored.

For example, an architecture may deviate from the classic style in a way that obviously negates one or more of the recognized performance advantages of that architectural style. In those cases, bringing the architecture into conformance with the style will produce performance gains. For example, in one assessment, we discovered that the development team had started with a classic pipe-and-filter architecture but then compromised that style during prototyping. The result was a monolithic implementation in which all of the filters ran within a single process. This limited the scalability of the application which was a primary performance goal. Implementing the software so that each filter can run independently (as in the classic pipe-and-filter style) improves scalability.

3.8.2 Alternative interactions between components

Sometimes, the interaction between two components may be a source of performance problems. In these cases, it may be possible to change the interaction to improve responsiveness or throughput. For example, using the Coupling Pattern [Smith and Williams 2002] to match an interface to its most frequent use will often improve performance.

3.8.3 Refactoring to remove an antipattern

If a performance antipattern is found during the analysis step, refactoring the architecture to remove that antipattern will improve performance.

For example, one of the antipatterns that we encounter most frequently is the One-Lane Bridge [Smith and Williams 2002]. This antipattern arises whenever only one (or a few) process(es) may proceed because of the need to wait for a resource (e.g., a database lock or synchronous call to a single-threaded process). The One-Lane Bridge can cause large backlogs that cause wide variability in response times. The general solution to this problem is to refactor the software to spread the load either spatially (e.g., by accessing different portions of the database) or temporally (e.g., by performing work at different times). The specific solution will depend on the characteristics of the application.

3.9 Presentation of Results

It is important that the PASA client receive a document containing the mission, findings, specific steps to take, the priority of the steps, and their relative importance. The document may be prose or a copy of presentation slides. This increases the likelihood that they will be able to use the results of the assessment and will be able to follow-up to quantify the benefit of the activity.

As noted above, in many cases, modeling is needed to quantify problems and their improvements. Since rapid feedback is important, in these cases preliminary results, along with a modeling and measurement plan are presented at the end of the first week. Then, when the modeling is complete, a final presentation summarizes all of the findings.

4. Example Assessment

This example is drawn from an actual architecture assessment. The details have been modified to preserve confidentiality. In some cases, they have also been simplified for presentation.

The system under consideration is a data acquisition system that receives data from multiple sources, formats and translates incoming messages, applies business rules to interpret and process messages, updates a data store with the data that was received, and prepares data for additional downstream processing. The case study is presented here as a generic data acquisition system. It is representative of many of the applications that we have reviewed, including order-processing (e.g., e-commerce), stock market data processing, call-detail record processing, payment posting, and ECM data acquisition.

Management requested an architecture assessment because they were about to commit to a system upgrade whose goal was to increase throughput by a factor of ten. While an increase in hardware capacity was considered, a ten-fold increase in hardware would not be cost-effective. So, the goal of the assessment was to determine whether the existing architecture was adequate to support the increased throughput or a new architecture was needed. If the current architecture was deemed adequate, then the development team requested that the assessment team identify opportunities, both strategic and tactical, for improving performance.

4.1 Process Overview

The first PASA step was a briefing for everyone involved to explain the what we would be doing, what they needed to provide, what we would do with it, and what they could expect as a deliverable. The actual presentation is omitted here.

4.2 The Architecture

The architecture description we received consisted of users manuals for the system administration features, design documents for several of the key components, and some class diagrams. None of the documents focused on the most important use case, they all mixed the various functions thus making it difficult to determine exactly what interactions occurred to process messages received from the data feeds. When asked specifically what processing occurred, participants drew a diagram similar to that in Figure 1 and said that the data is grabbed from the feed, deblocked into individual messages, passed to the message handler to update state and act on the data received, then an output message is formatted and written for the downstream processes.

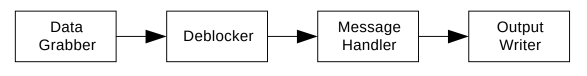


Figure 1: Architecture Diagram

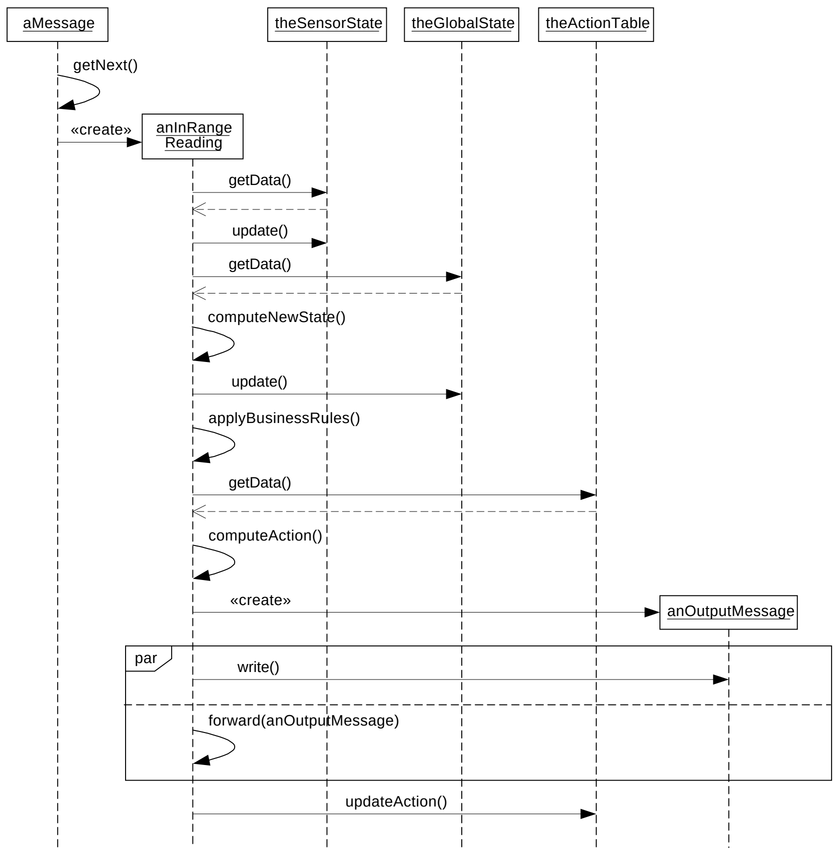


Figure 2: Sequence diagram for processing in-range data.

4.3 Use Cases
Use cases for this application include the data feeds for the acquisition system, the downstream processes that use the data, a switching feature that activates redundant processing systems in case of failure, and system administration features. After reviewing the documentation, we focused on the use case that takes messages from the feed, formats them, applies business logic, updates the data store, and sends them on for downstream processing. Different use cases deal with different types of data. The dominant use case is the one that processes an in-range data reading since these make-up the bulk of the data processed.

4.4 Key Performance Scenarios
The key performance scenario deals with processing an error-free in-range data reading. Figure 2 shows the sequence diagram for this scenario.

4.5 Performance Objectives
The system currently processes 2,000 messages per second. Management anticipates that the upgraded system must handle 20,000 messages per second. The end-to-end time to process a message was not specified, however team members felt that it should take no more than 30 seconds between the time the message arrives and when it is transmitted to downstream processes.

4.6 Architecture Discussion
This step involved several lengthy meetings with members of the development team who could explain particular details of the current processing. This information allowed us to map the processing steps in Figure 2onto the processes and threads identified in the initial documentation.

Developers felt that, in order to cost-effectively achieve a ten-fold increase in throughput, it would be necessary to run more concurrent streams, speed up the current streams to process more messages, or use a combination of these two approaches. The team felt that the middleware for passing messages between processes would be a barrier to scalability, so several discussions focused on the nature of the interactions with the middleware, whether it was essential to maintain the current collection of shared versus non-shared objects, etc.

We also reviewed all available performance measurements for the system. Most of them, however, were gathered during various focused tuning efforts and it was not possible to determine the current processing time for the steps in the scenario, or the portion of the time spent in the middleware versus the Message operations.

4.7 Architecture Analysis

It became clear from the discussions that the system as implemented would need some performance improvements in order to achieve the desired throughput. Nevertheless, we were able to conclude that the architecture itself was viable for the application, to identify some clear successes that had been achieved, to identify some performance antipatterns that should be the focus of future efforts, and to specify the steps in a more detailed performance benchmarking, measurement, and modeling study that would quantify the scalability of the system. These are covered in the following sections.

4.7.1 Architecture Classification

After reviewing the initial documentation and architecture discussions, it was clear that the overall architecture is a classic pipe-and-filter style [Shaw and Garlan 1996] in which each stage in the pipeline applies an incremental transformation to an incoming message before passing it to the next stage or sending it on for downstream processing. The current implementation ran 20 streams (pipelines) concurrently with each stream processing approximately 100 messages per second to achieve a throughput of 2000 messages per second.

The fundamental conclusion was that, while some performance improvements were needed, the current architecture would be able to support the goal of a ten-fold increase in throughput.

4.7.2 Performance Antipatterns

We found several performance antipatterns in the existing implementation [Smith and Williams 2002], [Smith and Williams in preparation]. The presence of these antipatterns presented significant limits to scalability.

- *Excessive Dynamic Allocation*—New message objects were created every time a message was received. For example, Figure 2 shows the creation of new InRangeReading and OutputMessage objects. Figure 3 shows the class hierarchy for messages. This is a deep hierarchy that is likely to result in considerable expense for creation of objects at the bottom of the lattice.
- *god Class*—The MessageHandler in Figure 2 behaves like a god class. It gets data from the other objects (i.e., theSensorState, theGlobalState, theActionTable), uses the data to determine processing requirements, then sends the updated data back to the container object. This results in extra message traffic and potentially limits the concurrency in the system because the Message Handler performs most of the work.

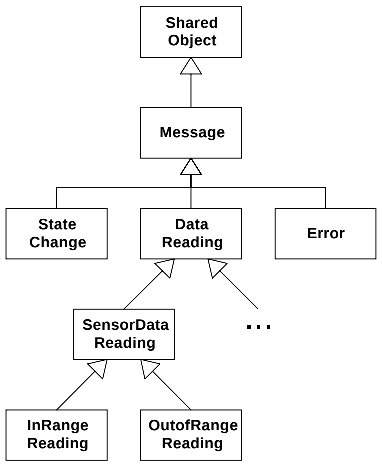


Figure 3: Message Class Hierarchy

- *Unbalanced Processing*—The algorithm used to route messages from the data feed to the appropriate parallel stream caused some of the parallel streams to be much busier than others. Throughput is maximized if all streams execute at their maximum rate.
- *Unnecessary Processing*—There were several processing steps that could potentially be eliminated. Both an Input Message and an Output Message were logged, but only one was necessary. When a (temporary) backlog developed, old messages were still processed by the system, but they should have been discarded. Many messages that were not needed by the system were received and processed only to be discarded late in the processing.

4.7.3 Modeling

Several of the issues that were identified required modeling to quantify their impact and as well as the improvements to be realized from design alternatives. In this case study, it was necessary to quantify the scalability of the system to precisely determine the hardware cost and software changes that would be necessary.

We constructed a software performance model from the sequence diagram in Figure 2. A performance benchmarking and measurement study was undertaken to determine the resource requirements for the processing steps in the scenario.

The first goal was to determine the performance budget for the stages in the pipe-and-filter architecture. Table 1 shows that average amount of time for each stage is a

Table 1: Performance Objectives

	Machines	Streams	Stream Throughput	Performance Objective	Total Throughput
1	1	20	100	0.01	2,000
2	10	20	100	0.01	20,000
3	4	10	500	0.002	20,000
4	4	20	250	0.004	20,000

function of the number of machines, the number of parallel pipeline streams on each machine, and the throughput of each stream. For example, the first row shows that with 20 streams running on one machine and a throughput of 100 messages per second, each stage must complete in 0.01 seconds to achieve 2,000 messages per second. Several options are shown for achieving the desired throughput of 20,000 messages per second. Option 2 simply solves the problem by adding more hardware (10 machines). Option 3 uses 4 machines, reduces the number of pipelines to 10, and increases the throughput of each stream, and so on. We will construct a model to determine the viability of each alternative for achieving the desired scalability.

We begin by constructing a model of the existing system for validation. This model focuses on the `Message Handler` stage in the pipe and filter because the measurements confirmed that it is the step that limits the overall throughput and scalability. The results of this model are shown in Figure 4.[†] The overall time for the `Message Handler` is under 0.01 seconds as required, and the first step takes the majority of this time. The utilization statistics (not shown) matched those measured on the system. Several other models were run under varying workload intensities to confirm that the model results matched the system measurements.

The next step modeled the case in row 4 of Table 1 to see if the current implementation of the `Message Handler` could meet the performance goal of 0.004 seconds. The results in Figure 4 show that the total time was 0.015 seconds—far greater than the 0.004 seconds required. The time required to create the `inRangeReading` and `anOutputMessage` (Excessive Dynamic Allocation) are significant problems in meeting this performance objective. Furthermore, because the `Message Handler` is a god class and performs most of the work of the system, we cannot easily break it into multiple stages in a pipe and filter to increase throughput. If it were redesigned, each processing step in the redesigned scenario would have 0.004 seconds to complete rather than requiring the entire scenario to complete in that time.

†. The models were constructed and solved using the *SPE•ED* SPE tool. www.perfeng.com.

The models showed that the primary problem was not with the messaging middleware as suspected, but with the excessive processing in one stage of the pipeline (`Message Handler`) and with the Excessive Dynamic Allocation.

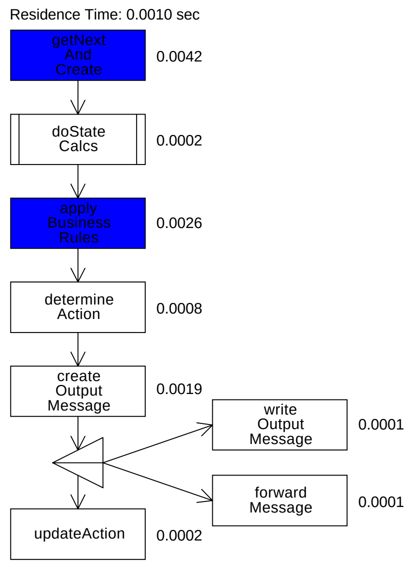
Note that it is possible to get these results from the measurements without constructing the software model. We have found it useful, however, to construct the model and use it to explain the current performance of the system, its limitations, and alternatives for improving performance. It is much easier to "see" the performance bottlenecks in the diagram than to find them in a table of numbers. (If that were easy, the developers would have already identified the problem and corrected it). The software performance model can then be used to evaluate different designs for the `Message Handler` that would enable it to operate in more stages, and to evaluate other combinations of machines, streams and throughputs to achieve the desired scalability.

4.8 Identification of Alternatives

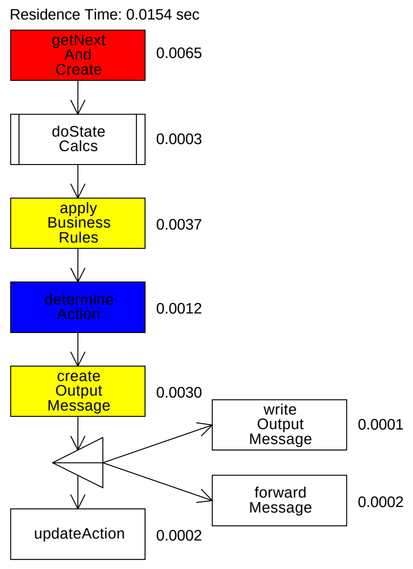
We were able to identify several alternatives for improving performance. They are categorized as either strategic (those that require a significant amount of work but have a potentially large payoff) and tactical (those that require little work but have a smaller payoff).

Strategic Improvements—In addition to improvements discussed above for removing the Excessive Dynamic Allocation and god Class antipatterns, the Unnecessary Processing and the Unbalanced Processing, there were other opportunities to significantly improve performance by applying Performance Principles:

- *Instrumentation Principle*—the software should have additional code to understand and control per-



(a) Current Implementation



(b) Option 4 Results

Figure 4: Model Results

formance. It was impossible to determine the resource requirements for critical processing steps without the special benchmarking and measurement study. It is vital to quantify the resource demand of processing steps to better understand and control performance; to identify bottlenecks and quantify proposed tactical improvements for effective priorities on implementation, and establish performance budgets for stages in the pipeline.

- *Spread-the-Load Principle*—monitor and control the scheduling of messages to parallel streams, purge aged messages, and filter unnecessary messages.

Tactical Improvements—Other Performance Patterns could also be applied to immediately improve system throughput:

- *Slender Cyclic Functions*—Remove all unnecessary processing from the critical path, and allocate processing that can be performed off the critical path to other concurrent processes
- *Batching*—Reduce processing by getting a batch of messages to process rather than one at a time

4.9 Presentation of Results

A preliminary presentation discussed the proposed improvements and outlined a plan for the measurement and modeling steps. Once the modeling phase was complete, a final presentation summarized all the findings and recomendations.

4.10 Summary

The architecture assessment was successful. It documented the overall end-to-end processing for messages in the current architecture. It determined that the current architecture was viable for achieving the desired scalability. It identified problem areas that required correction in order to achieve the desired scalability, and quantified the alternatives so that developers could select the most cost-effective solution. They ultimately implemented the changes and were able to meet their throughput goals.

5. Conflicts and Tradeoffs

Software performance is not achieved in isolation. Performance objectives must be balanced with other software quality concerns including: reliability/availability, safety and modifiability. Sometimes, these objectives conflict when architectural features have opposing effects on different quality attributes. For example, redundancy may increase availability but negatively impact performance. Identifying the areas of the architecture where conflicts occur and quantifying their effects makes it possible to find a workable compromise.

As with performance objectives, in order to evaluate the effect of architectural decisions on qualities such as modifiability or reliability, it is important that the requirements for these attributes be stated precisely. Evaluating tradeoffs also requires that quality requirements be prioritized. Obtaining precise quality requirements and prioritizing them is often the most difficult part of the process.

6. Summary and Conclusions

The architecture of a software system is the primary factor in determining whether or not a system will meet its performance and other quality goals. Architecture assessment is a vital step in the creation of new systems and the evaluation of the viability of legacy systems for controlling the performance and quality of systems.

This paper presented PASA, a method for performance assessment of software architectures. It described the method we use in a variety of application domains including web-based system, financial applications, and real-time systems. It described the nine steps in the method:

1. Process Overview
2. Architecture Overview
3. Identification of Critical Use Cases
4. Selection of Key Performance Scenarios
5. Identification of Performance Objectives
6. Architecture Clarification and Discussion
7. Architectural Analysis
8. Identification of Alternatives
9. Presentation of Results

A case study based on an actual performance assessment of a system architecture illustrated the steps in the method as well as typical findings for such an assessment.

The PASA method is evolving as we gain more experience on a variety of applications. With this experience, we are discovereing and documenting new Performance Antipatterns [Smith and Williams in preparation]. We are also currently codifying the results from multiple similar assessments into some general observations about the applicability of architectural styles to particular types of applications [Williams and Smith in preparation].

7. References

[Balsamo, et al. 1998] S. Balsamo, P. Inverardi, and C. Mangano, "An Approach to Performance Evaluation of Software Architectures," *Proceedings of the First International Workshop on Software and Performance (WOSP98)*, Santa Fe, NM, October, 1998, pp. 178-190.

[Booch, et al. 1999] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Reading, MA, Addison-Wesley, 1999.

[Brown, et al. 1998] W. J. Brown, R. C. Malveau, H. W. McCormick III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, New York, John Wiley and Sons, Inc., 1998.

[Buschmann, et al. 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Chichester, England, John Wiley and Sons, 1996.

[Cortellesa and Mirandola 2000] V. Cortellesa and R. Mirandola, “Deriving A Queueing Network-based Performance Model from UML Diagrams,” *Proceedings of the Second International Workshop on Software and Performance (WOSP2000)*, Ottawa, Canada, September, 2000, pp. 58-70.

[Clements and Northrup 1996] P. C. Clements and L. M. Northrup, "Software Architecture: An Executive Overview," Technical Report No. CMU/SEI-96-TR-003, Carnegie Mellon University, Pittsburgh, PA, February, 1996.

[Dowdy, et al. 1984] Lawrence W. Dowdy, Derek L. Eager, Karen D. Gordon and Lawrence V. Saxton, “Throughput Concavity and Response Time Convexity,” *Information Processing Letters*, vol. 19, no. 4, pp. 209-212, 1984.

[Eager and Sevcik 1983] Derek L. Eager and Kenneth C. Sevcik, “Performance Bound Hierarchies for Queueing Networks,” *Transactions On Computer Systems* vol. 1, no. 2, pp. 99-115, 1983.

[Gamma, et al. 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA, Addison-Wesley, 1995.

[Grahn and Bosch 1998] H. Grahn and J. Bosch, "Some Initial Performance Characteristics of Three Architectural Styles," *Proceedings of the First International Workshop on Software and Performance (WOSP98)*, Santa Fe, NM, October, 1998, pp. 197-198.

[Hsieh and Lam 1987] Ching-Tarng Hsieh and Simon S. Lam, “Two Classes of Performance Bounds for Closed Queueing Networks,” *Performance Evaluation*, vol. 7, no. 1, pp. 3-30, 1987.

[Kazman, et al. 1998] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The Architecture Tradeoff Analysis Method," *Proceedings of the Fourth International Conference on Engineering of Complex Computer Systems (ICECCS98),* August, 1998.

[Kazman, et al. 1996] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-Based Analysis of Software Architecture," *IEEE Software*, vol. 13, no. 6, pp. 47-55, 1996.

[Klein and Kazman 1999] M. Klein and R. Kazman, "Attribute-Based Architectural Styles," Technical Report No. CMU/SEI-99-TR-022, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, October, 1999.

[Lüthi, et al. 1997] Johannes Lüthi, Shikharesh Majumdar, Gabriele Kotsis, and Günter Haring, "Performance Bounds for Distributed Systems with Workload Variabilities and Uncertainties," *Parallel Computing*, vol. 22, no. 13, pp. 1789-1806, 1997.

[Majumdar, et al. 1991] Shikharesh Majumdar, C. Murray Woodside, J. E. Neilson and Dorina C. Petriu, "Performance Bounds for Concurrent Software with Rendezvous, *Performance Evaluation*, vol. 13, no. 4, pp. 207-236, 1991.

[Pooley and King 1999] R. Pooley and P. King, "The Unified Modeling Language and Performance Engineering," IEE Proceedings-Software, vol. 146, no. 1, pp. 2-10, 1999.

[Schmidt, et al. 2000] D. Schmidt, M. Stal, H. Ronert, and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*, Chichester, England, John Wiley and Sons, 2000.

[Shaw and Garlan 1996] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Upper Saddle River, NJ, Prentice Hall, 1996.

[Smith and Williams in preparation] C. U. Smith and L. G. Williams, "New Software Performance Antipatterns," manuscript in preparation.

[Smith and Williams 2002] C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Reading, MA, Addison-Wesley, 2002.

[Smith and Williams 2000] C. U. Smith and L. G. Williams, "Software Performance Antipatterns," *Proceedings of the Second International Workshop on Software and Performance (WOSP2000)*, Ottawa, Canada, September, 2000, pp. 127-136.

[Smith and Williams 1998] C. U. Smith and L. G. Williams, "Performance Engineering Evaluation of CORBA-based Distributed Systems with *SPEED*," in *Computer Performance Evaluation, Lecture Notes in Computer Science*, vol. 1469, R. Puigjaner, N. N. Savino and B. Serra, ed., Berlin, Springer-Verlag, 1998, pp. 321-335.

[Smith and Williams 1997] C. U. Smith and L. G. Williams, "Performance Engineering Evaluation of Object-Oriented Systems with SPEED," in *Computer Performance Evaluation: Modelling Techniques and Tools, Lecture Notes in Computer Science*, vol. 1245, R. Marie, B. Plateau, M. Calzarossa and G. Rubino, ed., Berlin, Springer-Verlag, 1997, pp. 135-154.

[Smith 1990] C. U. Smith, *Performance Engineering of Software Systems*, Reading, MA, Addison-Wesley, 1990.

[Stephens and Dowdy 1984] Lindsey E. Stephens and Lawrence W. Dowdy, "Convolutional Bound Hierarchies," SIGMETRICS, pp. 120-133, 1984

[Williams and Smith 1998] L. G. Williams and C. U. Smith, "Performance Evaluation of Software Architectures," *Proceedings of the Workshop on Software and Performance (WOSP98),* Santa Fe, NM, October, 1998.

[Williams and Smith in preparation] L. G. Williams and C. U. Smith, "Performance Characteristics of Common Architectural Styles: Pipe-and-Filter and Client-Server," manuscript in preparation.