OPTIMISING DESIGN FOR PERFORMANCE

Literature Review

King, Simon, University College Dublin, Ireland, simon.king@ucdconnect.ie

Contents

Contents **1**

1 Introduction **1**

2 Software Performance Engineering **2**
2.1 Modelling 2
2.2 Application of SPE 3

3 Performance Patterns and Principles **4**
3.1 A brief history of patterns 4
3.2 Performance Control Principles 4
3.3 Independent Principles 5
3.4 Synergistic Principles 5
3.5 Performance patterns 6
3.6 Performance Anti-patterns 7

4 Conclusions **8**

1 Introduction

The software development process can be a hugely complex model and it is essential for organisations to determine the factors that most influence the success of a software project. Thus, gathering the knowledge to evaluate the development process is needed for proper management. Not only for visibility of the current state of the project, but also where issues will arise in the future. One of the key determining factors in project success is performance. Failing to integrate performance into the design of an application from the beginning can lead to systems that do not meet the needs of the users, or even catastrophic failure. This paper discusses the importance of integrating performance and design. As with all aspects of software engineering, certain patterns, and anti-patterns, in optimising performance have emerged that have proven to help design systems that perform to an acceptable level, while keeping costs manageable. In this paper we will examine some of the general principles and techniques used to analyse system performance, we will identify some of the common performance patterns and anti-patterns found in software and how they can be applied.

2 Software Performance Engineering

Software engineering projects are complex entities. Every project faces many pitfalls that can lead to projects running over schedule, over budget or being abandoned completely. One of the key issues in designing successful solutions is ensuring the system performs to an acceptable level. Underestimating the workload and scalability requirements can lead to catastrophic failures. Software Performance Engineering (SPE) is a method for constructing software systems that meet performance goals (Smith, 1990), and deliver software that meets performance objectives and is on-time and within budget (Smith and Williams, 2003). By using measurement and modelling techniques, a system's limitations can be identified and can be improved upon with alternative designs. This is a crucial aspect of software design, and should be considered from the very beginning of the design process. Performance is a pervasive quality of software systems (Woodside, Franks, and Petriu, 2007) and everything from hardware and distribution of the system, to the architecture and design of the software, as well as the software itself, affects it.
Like other software engineering activities, SPE is constrained by many political and technical issues such as tight project schedules, poorly defined requirements, perceived high costs, and misunderstood processes (Woodside, Franks, and Petriu, 2007). However, experience indicates that SPE can help reduce the need to perform tuning after software has been implemented, which reduces overall costs and improves maintainability (Smith and Williams, 1993). It also identifies issues within the architecture which can prevent the achievement of the performance objectives set out (Smith and Williams, 2003).
By defining the key performance requirements and objectives, and identifying the key workloads and workflows expected in the system, performance models can be generated. These models identify performance problems and can provide alternatives for correcting them (Smith and Williams, 2003). It may be the case that the alternatives that arise are not achievable, which will lead to the objectives being revised to reflect expected results (Smith, 1997).

2.1 Modelling

When assessing the architecture and design to see if a system can meet performance objectives, two models can be used: the software execution model and the system execution model.

2.1.1 Software Execution Model

The software execution model is an optimistic view of the system. It deals only with the software, and does not account for other workloads, contention, or multiple users. Problems meeting the performance goals that arise in this model must be resolved before more dynamic modelling can be performed. Meeting the goals in this model does not guarantee the real system will achieve the performance goals, however, failure to do so does imply the real system will not. It is considered to be an early model to ensure the software architecture can make it possible to meet performance goals. The results from this model can be used as inputs for the system execution model.

2.1.2 System Execution Model

The system execution model is a more dynamic model that describes the software performance while considering factors that may cause contention for resources. These resources are represented as queues and servers. Sources of contention that arise in this model could be multiple concurrent users, or concurrent applications on the same hardware, tasks waiting on components such as CPU, I/O or network transactions. The system execution model gives a more realistic view of the performance of the system. It can be used to predict how the system performs with average and peak loads, to highlight the most sensitive components, where performance bottlenecks might occur and how the system will scale. When creating models, it is important to capture all the critical components and measure their performance throughout the life cycle of the development process to ensure the changes in the system do not invalidate the model (Smith and

Williams, 2003). This especially helps in preventing problems surfacing late in development. Properly instrumenting the software provides the insight necessary to understand the system, plan and implement performance enhancements (Chauhan et al., 2014).

2.2 Application of SPE

SPE should be applied early in projects to build in performance as it requires significant effort and cost to rework and retrofit performance into architecture and software; the more refactoring required, the greater the cost in terms of time and resources (Smith and Williams, 2003). However, performance problems discovered after implementation can be address by tuning. This process can introduce errors and ruin carefully constructed design (Williams and Smith, 1995). Performance maintenance is a process the looks to improve the performance of a software system after delivery (Bezemer and Zaidman, 2014). Sometimes it is not possible to properly test the performance, for example it may be too expensive to create a testing environment that is equivalent to the production environment of a large system. A well constructed model should overcome these issues. Software design and SPE can be closely integrated, allowing designers to explore design alternatives and select a design that provides the best overall combination of understandability, reusability, modifiability and performance (Williams and Smith, 1995). Education is vital to maximizing the benefits of performance modeling (Smith, 2002). Everyone should understand how their part of the system contributes to the overall performance. If certain areas are not performing, other areas may be able to compensate so the system as a whole meets the requirements.

3 Performance Patterns and Principles

3.1 A brief history of patterns

A pattern is typically described as a common solution to a problem in a context. Patterns as we refer to them here originated from the work of Christopher Alexander, a famous architect, in the late seventies:

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" - (Alexander et al., 1977)

The idea of patterns in software was first put forward by Beck and Cunningham (Beck, 1999) in the late 1980s. The foremost resource in the area of design patterns, however, is the work of the Gang of Four in their book “Design Patterns: Elements of Reusable Object Oriented Design” (Gamma et al., 1993). This identified patterns as abstractions at a higher level than classes and objects and is hugely influential in the area of object-oriented design theory. Patterns give programmers a common language when discussing issues and allow better understanding of problems and solutions across all platforms. While patterns have proven very useful, they are not always effective when not implemented properly. Anti-patterns are used to identify and resolve common recurring mistakes by identifying their causes and consequences. They also offer a set of solutions to help resolve to these problems. The idea was first proposed by Koenig (Koenig, 1998), but the work of Brown et al. (Brown et al., 1998) gives a comprehensive guide to anti-patterns, their symptoms and consequences when they arise, and offer solutions to refactor the issues. “When patterns are applied in the wrong context, the solution can lead to overwhelmingly negative consequences”(Brown et al., 1998). Patterns and anti-patterns are found in all aspects of software engineering. When we consider performance, the work of Connie Smith and Lloyd Williams tends to be the foundation on which all things are built. In their book “Performance Solutions”(Smith and Williams, 2001), they outline a number of performance principles that should be adhered to to design performance into software. These principles form the basis for the patterns we see later on. They are categorised into three areas: Performance Control Principles, which help in controlling a system and ensuring it meets the performance objectives; Independent Principles, which aim to reduce resource consumption; and Synergistic Principles, which aim to improve performance across the whole system by having processes competing for resources cooperate. Performance patterns could be considered fulfilment of one or more of these principles, while a design pattern could be said to be an implementation of these performance patterns. Performance patterns are best practices for producing responsive, scalable software (Smith and Williams, 2001).

3.2 Performance Control Principles

The following section will describe the performance principles as outlined mainly in (Smith and Williams, 2001), (Smith, 1988). We will then look at 7 of the most common patterns used in performance engineering that concern responsiveness and scalability.

3.2.1 Performance Objectives

As a system is evolving, specific quantitative metrics should be defined. This involves outlining a number of performance scenarios and setting quantitative measurable performance objectives. These should not be vague or qualitative and where possible, more than one objective should be outlined for each scenario. Over the product’s lifetime, these objectives may change, but whenever possible, the performance objectives should consider future use. A range of intensities can be set, which can help account for varying workloads or complex scenarios

3.2.2 Instrumentation

This involves adding instruments, or probes, to collect data and enable proper measurement and analysis of the system. While this will not improve performance directly, it is an essential step in improving performance.

3.3 Independent Principles

3.3.1 Fixing Point Principle

The Fixing Point refers to a point in time. Fixing connects actions to the functions used to complete those actions. It also connects desired results to the data used to produce them.. These connections should be done at the earliest feasible point in time such that retaining the connection is cost effective. A common example is a banker requesting summary information for accounts in his branch. The latest fixing point would be summarizing all the account data when the request is made. An earlier fixing point would be updating the summary data as account information arrives, which would improve the responsiveness of the summary request.

3.3.2 Locality Design Principle

This involves having a close mapping of actions and results to the physical resources used to produce them. There are different types of locality, depending of the definition of the term close. Locality can be spatial, temporal, effectual or degree. Spatial locality refers to task being close the physical resources that will perform them, such as having information in a processor's local memory, rather than a remote drive. Temporal Locality refers to when in time actions should be performed. Effectual locality refers actions that are similar in purpose or intent. Degree locality refers to matching the intensity or size of the resources to the actions that need to be performed.

3.3.3 Processing vs Frequency Trade-Off Principle

This principle concerns finding a balance between the amount of work needed in processing a request, and how often these requests are made. For example it may be possible to reduce the number of requests if more work is done per request. However, having a request take too long to process will lead to a queue at the server that is handling the request if these requests are made more often. The goal is to minimise the product of processing and frequency.

3.3.4 Centering Principle

This principle identifies the dominant workloads in the system and aims to minimize their processing times. It is based in part on the idea that 80% of the requests are handled by 20% of the code. Identifying the functions of the system that will be used most often and improving their performance will have a significant impact on the performance of the system as a whole. This principle also considers optimising requests that have the largest demand on resources.

3.4 Synergistic Principles

3.4.1 Shared Resource Principle

Sharing a resource helps to improve performance by allowing multiple requests be processed at once. This reduces the overhead associated with scheduling the resource and the possible wait times for requests waiting to gain access to the resource. If exclusive access is required the scheduling and holding times should be minimised.

3.4.2 Parallel processing Principle

Processing can be reduced by separating requests to run concurrently. This can be achieved by multiplexing on a single processor, or using separate processors. Problems can arise with parallel processing, such as resource contention or overhead associated with communication and coordinating the processes. Parallel processing should only be used when the speedup offsets these overheads.

3.4.3 Spread the load principle

Processes that, for example, require the same resource are said to be conflicting, in that if one has use of the resource, the other must wait before it can continue. Spreading the load aims to reduce the number of processes needing a resource at the same time by processing at different times or in different places.

3.5 Performance patterns

3.5.1 Fast Path

In a system, some processes may be used much more often than others. Fast Path aims to optimise the most commonly used aspects of the system, i.e. reducing the processing required for the dominant workload. The solution is based around the centering principle. A short path is created for the most common processes that are used and the processing for these are reduced. The most commonly used data could also be cached to reduce processing time. The benefits of using Fast Path are response times for the most common processes are reduced, which can lead to the overall load on the system being reduced. The usage must be monitored over time, and if necessary the system should be adapted to suit the most common usages should they change.

3.5.2 First Things First

In busy systems which may be subject to temporary overloads, it is necessary to ensure the most important tasks are performed. The First Things First pattern introduces a priority to the tasks that need to be executed, insuring the most important ones are handled first. In times where the system is overloaded temporarily and not all tasks can be completed, the least important tasks are the ones omitted. The solution is again based around centering. This can increase the quality of service and scalability of the system. However, it should only be applied to systems that experience temporary overloads. Otherwise, the amount of processing required should be reduced or the resources should be improved to handle the greater load.

3.5.3 Coupling

Applications can use fine grained objects to request information from external resources. This leads to a large number of interactions, which can be expensive, especially in distributed systems. Using coarse grained objects when making information requests combines common objects to make one large request for many objects, rather than a large number of smaller requests for a single objects. This utilises the centering principle to identify interfaces, the locality principle to combine the information, and the processing versus frequency principle to minimise the total processing. This solution matches the interface of objects to their most frequent uses, which reduces the total resource requirements of the system.

3.5.4 Batching

Some requests come with high overhead for initialisation, transmission, and termination. For frequent requests of this nature, the cost of the setup overhead can be more expensive than the processing actually done for a particular request. Combining multiple requests into a single batch means the overhead is spread across several requests. Using the processing versus frequency principle, the product of processing

time and frequency of requests can be reduced. This should be applied when the overhead processing for the tasks is very large and the tasks are frequent.

3.5.5 Alternate Routes

When tasks need exclusive access to the same resources, e.g. a database to perform updates, processing can be affected. If the holding time for the resource is high and the resource is in high demand, a bottleneck can occur. Alternate Routes proposes reducing these contention delays by spreading the demand spatially, i.e. accessing different physical locations. This uses the Spread the Load principle to reduce serialization delays, and reduce variability in performance. The alternate route should spread the load effectively so that all routes are used efficiently to maximise the gains.

3.5.6 Flex Time

Processes that run at specific regular intervals using high-demand objects can lead to congestion. Similar to Alternate Routes, Flex Time aims to reduce these contention delays, by identifying these regularly occurring processes and changing when they occur to reduce blocking and improving system efficiency. This could mean moving processes to times when no other processes are running, or randomising the times these processes start. It is important to ensure that the not all processes choose to run at the same time.

3.5.7 Slender Cyclic Functions

This pattern deals with work that is done at specific, regular intervals. If these patterns occur frequently, they may use up resources that the system could otherwise use. The Centering Principle and Shared Resource principle are used to reduce the processing, making these resources are available to share, which can reduce queuing and contention issues. .

3.6 Performance Anti-patterns

3.6.1 God Class

One class may perform all of the work on a system, or contain all of the data for a system. This causes a performance problem due to excessive message traffic. The solution to this problem is to use the Locality principle to refactor the design to create a number of top-level classes that keep related data and behaviour together.

3.6.2 Excessive Dynamic Allocation

An application may create and destroy excessive amounts of objects during execution. The overhead associated with creating and destroying may have a negative impact on performance. The solution could be to either share objects and eliminate the need to create new ones, or recycle the objects using an object pool.

3.6.3 Circuitous Treasure Hunt

This occurs when requests are dependant on the results of other requests, meaning the final result object has to get information from several places. This can have a negative impact on performance, especially on distributed systems where the information is coming from remote servers, for example. The solution would be to organise the data required to be close to where is it used or provided alternative access paths that are not dependant on each other.

3.6.4 One Lane Bridge

This occurs where only one or a small amount of processes can occur concurrently. This can result in multiple parallel paths converging on one single execution path, causing a bottleneck. The solution is to provide parallel paths to reduce traffic on the bridge, or by using the Shared Resource principle to improve the execution time on the bridge.

3.6.5 Traffic Jam

This occurs when there is more traffic than resources able to handle it, perhaps as a result of one problem that causes a backlog of jobs to build up. This can cause a wide variability in response times which may take a long time to return to normal. There are many solutions to this problem, like applying the Alternate Routes or Flex Time patterns to spread or deter the load. Eliminating the issue that caused the backlog may not always be possible. The ideal solution is building a system with sufficient processing power to handle the worst case loads

4 Conclusions

This paper discusses the importance of designing performance into software projects, rather than considering it as a project approaches release, when it may be too late to really affect it without significant effort and cost. While it may have a seemingly high upfront cost, the benefits of considering performance at the early stages of design are significant to the health and quality of the project. Software performance engineering can be used identify designs that will not meet performance targets and suggest alternatives, using relatively simple models and measurements. It proves to be worth the initial effort. We also see the concept of patterns being applied across the area of performance. The principles of performance design offer engineers some best practices that have been proven to improve the responsiveness and scalability of projects. These principles and patterns when applied can prevent pitfalls and failures. We also see some of the commonly encountered problems with performance in software. By being aware of these principles, patterns, and anti-patterns, we can create well performing software projects. These patterns come from years of experience, appearing time and time again, and have proven track records of working across a multitude of platforms and disciplines. As engineers it is crucial to be aware of these patterns for several reasons: It gives a *lingua franca* for engineers to discuss complex topics that could be otherwise difficult to convey. To not reinvent the wheel when trying to solve problems. In all likelihood any problem has been encountered before and has a solution that is well documented and proven to work.

References

Alexander, C., S. Ishikawa, M. Silverstein, J. R. i Ramió, M. Jacobson, and I. Fiksdahl-King (1977). *A pattern language*. Gustavo Gili.

Beck, K. (1999). *Kent Beck's guide to better Smalltalk: a sorted collection*. Vol. 14. Cambridge University Press.

Bezemer, C.-P. and A. Zaidman (2014). "Performance optimization of deployed software-as-a-service applications." *Journal of Systems and Software* 87, 87–103.

Brown, W. H., R. C. Malveau, H. W. McCormick, and T. J. Mowbray (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

Chauhan, N., G. Kabra, T. Kittelmann, R. Langenberg, R. Mandrysch, A. Salzburger, R. Seuster, E. Ritsch, G. Stewart, N. van Eldik, et al. (2014). "ATLAS offline software performance monitoring and optimization." In: *Journal of Physics: Conference Series*. Vol. 513. 5. IOP Publishing, p. 052022.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1993). "Design patterns: Abstraction and reuse of object-oriented design." In: *European Conference on Object-Oriented Programming*. Springer, pp. 406–431.

Koenig, A. (1998). "Patterns and antipatterns." *The patterns handbook: techniques, strategies, and applications* 13, 383.

O'Connor, P. D. T. (1991). "The Art of Computer Systems Performance Analysis (Book)." *Quality Reliability Engineering International* 7 (5), 432. ISSN: 07488017. URL: `http://search.ebscohost.com.ucd.idm.oclc.org/login.aspx?direct=true&db=a9h&AN=12784759&site=ehost-live`.

Smith, C. U. (1988). "Applying synthesis principles to create responsive software systems." *IEEE Transactions on Software Engineering* 14 (10), 1394–1408.

Smith, C. U. (1990). *Performance engineering of software systems*. Addison-Wesley Longman Publishing Co., Inc.

— (1997). "Performance engineering for software architectures." In: *Computer Software and Applications Conference, 1997. COMPSAC'97. Proceedings., The Twenty-First Annual International*. IEEE, pp. 166–167.

— (2002). "Performance models for computer and telecommunications systems: maximizing the benefits." In: *Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. MASCOTS 2002. Proceedings. 10th IEEE International Symposium on*. IEEE, p. 225.

Smith, C. U. and L. G. Williams (1993). "Software performance engineering: A case study including performance comparison with design alternatives." *IEEE Transactions on software engineering* 19 (7), 720–741.

Smith, C. U. and L. G. Williams (1999). "A performance model interchange format." *Journal of Systems and Software* 49 (1), 63–80.

— (2001). "Performance solutions: a practical guide to creating responsive, scalable software."

Smith, C. U. and L. G. Williams (2003). "Best practices for software performance engineering." In: *Int. CMG Conference*, pp. 83–92.

Williams, L. G. and C. U. Smith (1995). "Information requirements for software performance engineering." In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, pp. 86–101.

Woodside, M., G. Franks, and D. C. Petriu (2007). "The future of software performance engineering." In: *Future of Software Engineering, 2007. FOSE'07*. IEEE, pp. 171–187.