New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot

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

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

Performance antipatterns document common software performance problems as well as their solutions. These problems are often introduced during the architectural or design phases of software development, but not detected until later in testing or deployment. Solutions usually require software changes as opposed to system tuning changes. This paper presents three new performance antipatterns and gives examples to illustrate them. These antipatterns will help developers and performance engineers avoid common performance problems.

1.0 INTRODUCTION

Architectural and design patterns document common solutions to frequently-occurring software development problems [Schmidt, et al. 2000], [Buschmann, et al. 1996], [Gamma, et al. 1995]. These patterns capture expert knowledge about "best practices" in software design by documenting general solutions that may be customized for a particular context. They make it possible to reuse that knowledge in the development of many different types of software.

Antipatterns [Brown, et al. 1998] are conceptually similar to patterns 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 as well as their solutions. Thus, antipatterns tell you what to avoid and how to fix the problem when you find it. Like patterns, antipatterns address both software architecture and design issues. Antipatterns may apply to the software development process itself, as well.

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 a data structure may be revised to improve the efficiency of the retrieval processing. The transformation does not alter the semantics of the application but it would improve performance. Refactoring may be used to enhance many different quality attributes of software, including: reusability, maintainability, and, of course, performance. Refactoring is discussed in detail in [Fowler 1999].

Recently, we introduced software performance patterns and antipatterns [Smith and Williams 2002]. Performance patterns describe "best practices" for developing responsive, scalable software. Architectural and design patterns focus on quality attributes such as reusability or modifiability rather than performance. Performance patterns extend the notion of patterns to explicitly include performance considerations.

Performance antipatterns document common performance mistakes made in software architectures or designs. They may also have negative impacts on other quality attributes, such as reusability or modifiability, but they are not addressed here. Our experience is that developers find antipatterns to be especially useful because they illustrate how to identify a bad situation *and* provide a way to rectify the problem. This is particularly important for performance because good performance is the *absence* of problems. Thus, by illustrating performance problems and their causes, performance antipatterns help build performance intuition in developers.

In [Smith and Williams 2002], [Smith and Williams 2001], and [Smith and Williams 2000] we introduced several software performance antipatterns. This paper describes three new performance antipatterns that were identified through our experienced in performing performance assessments of software architectures [Williams and Smith 2002]. Each of the antipatterns is described in the following sections using this standard template:

- Name: the section title
- Problem: What is the recurrent situation that causes negative consequences?
- Solution: How do we avoid, minimize or refactor the antipattern?

This paper also includes a summary of known performance antipatterns as a reference.

2.0 Related Work

Antipatterns are derived from work on patterns. As noted in the introduction, this work is aimed at capturing expert software design knowledge. There is a large body of published work on patterns including [Gamma, et al. 1995], [Buschmann, et al. 1996], and [Schmidt, et al. 2000]. While there is occasional mention of performance considerations in the work on patterns, the principal focus is on other quality attributes, such as modifiability and maintainability.

Meszaros [Meszaros 1996] presents a set of patterns that address capacity and reliability in reactive systems such as telephony switches. Petriu and Somadder [Petriu and Somadder 1997] extend these patterns for use in identifying and correcting performance problems in distributed layered client-server systems with multi-threaded servers.

As noted above, several performance patterns and antipatterns were presented in [Smith and Williams 2002], [Smith and Williams 2001], and [Smith and Williams 2000]. This paper presents new antipatterns identified since that work was published.

Dugan and co-workers describe the Sisyphus Database Retrieval Performance Antipattern [Dugan, et al. 2002]. That antipattern is a special case of The Ramp antipattern, as discussed later in this paper.

3.0 Unbalanced Processing

Concurrent processing has the potential to improve the scalability of software systems. The potential scalability is not realized, however, unless the concurrent steps do not have to wait for other processing to complete. Imagine waiting in an airline check-in line. Multiple agents can speed-up the process but, if a customer needs to change an entire itinerary, the agent serving him or her is tied-up for a long time making those changes. With this agent (processor) effectively removed from the pool for the time required to service this request, the entire line moves more slowly and, as more customers arrive, the line becomes longer.

3.1 Problem

The Unbalanced Processing Antipattern has several manifestations:

3.1.1 Concurrent processing systems

Unbalanced Processing occurs when processes cannot make effective use of available processors either because processors are dedicated to other tasks or because of single-threaded code. This manifestation has available processors and we need to ensure that the software is able to use them.

3.1.2 "Pipe and Filter" Architectures

The throughput of the overall system is determined by the slowest filter. For example, in the travel analogy, passengers must go through several stages (or filters): first check in at the ticket counter, then pass through security, then go through the boarding process. Recent events have caused each stage to go more slowly. The security stage tends to be the slowest filter these days. Note that in this type of system we are primarily concerned with throughput.

3.1.3 Extensive processing

This situation is analogous to the itinerary-change example. It occurs when a long running process monopolizes a processor. The processor is removed from the pool, but unlike the pipe and filter example, other work does not have to pass through this stage before proceeding. This is particularly problematic if the extensive processing is on the processing path that is executed for the most frequent workload. In this type of system we are primarily interested in the residence time, however we are also interested in throughput. That is, customer satisfaction improves with shorter waits.

3.2 Solution

There are three possible solutions to problems introduced by Unbalanced Processing depending on the manifestation of the problem. Because both the problem and the solutions involve contention effects, models are required to evaluate the overall effect of the solution in each case.

3.2.1 Concurrent processing

If the problem is due to single-threading work, use performance models to find alternatives that either create multi-threaded tasks or use multiple copies of the process that can execute concurrently. When the problem is due to dedicating processors to specific tasks, use system execution models to quantify the net effect of the assignment. Note that this is not so much a software design issue as a system tuning issue.

Figure 1 illustrates a system with a routing algorithm based on static properties that results in more work going to one queue than others. The solution is to use a dynamic algorithm that routes work to queues based on the work requirements and the system congestion.

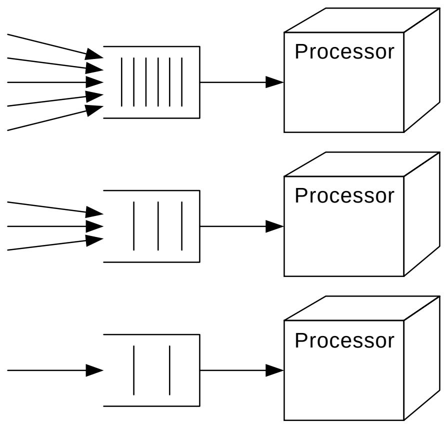

Figure 1: Unbalanced Processing Due To Routing Algorithm

3.2.2 *"Pipe and Filter" Architectures*

Use models to determine the processing requirements of each stage:

- divide long processing steps into multiple, smaller stages
- combine short processing steps to minimize context switching overhead and other delays for shared resources

Figure 2 illustrates an execution graph with Unbalanced Processing due to stages or filters of unmatched sizes. The shading for each processing step shows its relative processing time—light shading reflects small processing time whereas the dark shading shows large processing requirements. The solution is to divide long filters into multiple steps that can execute in parallel and combine short steps into one.

3.2.3 *Extensive processing*

Identify processing steps that may cause slow downs and delegate those steps to processes that will not impede the Fast Path. This is analogous to setting up special lines for checking baggage (the Fast Path) versus issuing tickets or changing itineraries.

4.0 UNNECESSARY PROCESSING

Unnecessary processing is like any other unnecessary work—it keeps you from doing what is really important.

4.1 Problem

Unnecessary processing is processing that is executed but is either not needed, or not needed at that time.

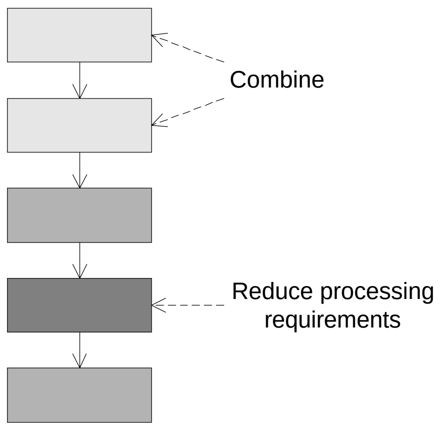

Figure 2: Unbalanced Processing in a Pipe and Filter architecture.

This is also particularly problematic if the unnecessary processing is on the processing path that is executed for the most frequent workloads.

In one system a step at the beginning of the processing wrote a copy of the inbound message to a log. The last step in the processing wrote a copy of the transformed message to a log just before forwarding it to a downstream system. The logging of the outbound message was unnecessary because it could be re-created from the inbound message if necessary. Figure 3 illustrates this unnecessary processing.

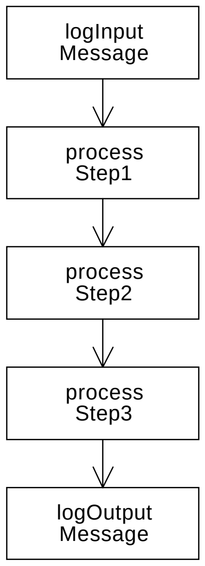

Figure 3: Unnecessary Processing to Log Output Messages

The logging step was implemented by issuing an update to a database. It was also Unnecessary Processing to wait for the update to complete before proceeding because the detection and correction of update problems could be done on a different execution path. The results of the update were not required for processing other input messages.

In the same system, it was possible for the inbound messages to experience backlogs periodically when the time to process the message exceeded the time between arrivals of new work. Unnecessary Processing was executed because each queued message was still processed even though it was later discarded due to the staleness of the data.

4.2 Solution

Sometimes Unnecessary Processing can be corrected by simply deleting the associated processing steps. In the case of the redundant logging of the output message, that step could just be deleted. If it is essential to be able to recreate the output message, however, new code may be needed to perform that function. Because the function would be needed infrequently, there is a net savings in processing time.

Another solution is to re-order processing steps. For example, we should detect earlier in the processing that results are not needed due to stale input data, before Unnecessary Processing has executed.

A third solution is to restructure the processes and delegate Unnecessary Processing to a background task rather than execute it on the most frequent execution path. For example, we could generate an asynchronous call to update the data base, and route exception handling to a different process.

5.0 THE RAMP

(Note: This antipattern was first brought to our attention by a CMG 2001 attendee.)

Have you ever used a system that exhibited good response time early in the day but, as time went on, became slower and slower? If so, you've probably experienced The Ramp. When this antipattern is present, processing time increases as the system is used. Figure 4 illustrates the effect of The Ramp on processing time. Note that, as the processing time increases, response time increases exponentially.

5.1 Problem

The Ramp can arise in several different ways. Any situation in which the amount of processing required to satisfy a request increases over time will produce the behavior illustrated in Figure 4.

For example, consider a list of daily customer requests. Each time a request arrives, the list is searched to make sure that the request is not a duplicate. At the beginning of the day, the list is small and a sequential search provides adequate response times. As the day goes on, however, the list grows and the time required to perform the search becomes unacceptable.

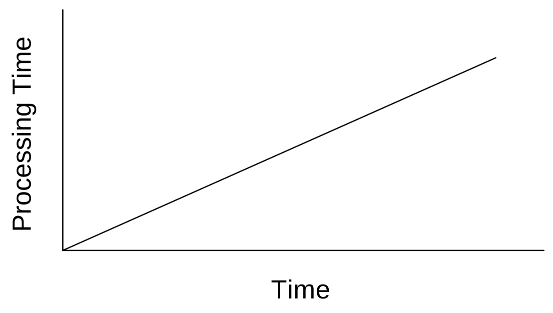

Figure 4: The Ramp

Another common example of The Ramp appears in Web searches. When you perform a search on the Web, only a subset of the results (typically 20) is presented. To see the next subset you select an option to view more. The Ramp occurs because each time you request a new subset, the search is performed again, the result is searched to find the required subset (e.g., results 41–60) and the rest are discarded. As you request more results, the search for the correct subset takes longer, reducing response time. This example of The Ramp is the one addressed in the Sisyphus Database Retrieval Performance Antipattern [Dugan, et al. 2002].

The Ramp presents a scalability problem that is often not detected during testing since test data often does not contain enough items to reveal the phenomenon.

5.2 Solution

The solution to The Ramp is to keep the processing time from increasing dramatically as the data set grows. Possible solutions include:

- Select a search algorithm that is appropriate for the larger amount of data—it may be suboptimal for small sizes, but it shouldn't hurt to do extra work then.
- Automatically invoke self-adapting algorithms based on size
- When the size increases more gradually, use instrumentation to monitor the size and upgrade the algorithm at predetermined points

As with other performance antipatterns, The Ramp is particularly problematic when it occurs on the Fast Path. If the code is not executed frequently then it may not have a significant affect on performance. Therefore, in addition to monitoring the number of items you should also track the number of times the code executes to determine when corrective action is needed.

Dugan, et al. [Dugan, et al. 2002] present quantitative analyses for several alternative search algorithms for the special case illustrated by the Web search example. They are: using additional indexing to restrict the search, upper and lower bounds on the search, sequence numbers for items in the list, and caching.

In general, the response time improvement depends on the size of the data set, the amount of time required to process an individual item, and the arrival rate for querits. The relationship is:

$$RT = \frac{n\frac{ds}{dt}s}{1 - X\left(n\frac{ds}{dt}s\right)}$$

where RT is the response time, n is the number of items in the data set, s is the amount of service time required to process a single item, $\frac{ds}{dt}$ is the slope of the ramp, and X is the arrival rate for queries.

It is important to identify these scalability problems as early as possible. Models are an important tool for identifying when you have The Ramp since, as noted above, test data sets are typically not large enough to reveal its presence. Also, by the time you've found The Ramp with test data, it may be too late or too expensive to do anything about it. A model will allow you to quantify the effect of The Ramp and determine the point at which performance problems will occur.

6.0 More is Less

.(Note: This antipattern was first brought to our attention by Rick Boyer [Boyer 2002].) We briefly review it here so that this paper can provide a compendium of all known performance antipatterns.

You've probably noticed times when the harder you worked the less you accomplished. This antipattern addresses the situation when we try to do too much on computer systems and they end up thrashing rather than accomplishing useful work.

6.1 Problem

More is Less used to be primarily a memory problem. Trying to run too many programs over time causes them to do too much paging and systems spend all their time servicing page faults rather than processing requests. Now, particularly in distributed systems, there are more causes.They include:

- Creating too many database connections
- Allowing too many internet connections
- Creating too many pooled resources
- Allowing too many concurrent streams relative to the number of available processors

6.2 Solution

Computer systems have diminishing returns due to contention for resources. Therefore, the solution is to quantify the point when resource contention exceeds an acceptable threshold. Models or measurement experiments can identify these points.

One way to reduce problems created by too many threads is to use a single thread to perform background processing [Larman and Guthrie 2000]. The thread maintains a priority queue of command objects that perform various background tasks. Each command object has a polymorphic execute() operation (see the Command pattern in [Gamma, et al. 1995]). This eliminates the need to have a separate thread for each background task.

While setting limits on concurrency can be a simple tuning solution, there are other situations when this phenomenon may dictate whether an architecture is appropriate for a software application. For example, if a system requires 200 parallel streams to achieve its message throughput, but the system can only support 100 database connections before thrashing occurs, then the system will require a different architecture that achieves throughput without so many parallel streams.

7.0 Use of Performance Antipatterns

These software Performance Antipatterns have four primary uses: identifying problems, focusing on the right level of abstraction, effectively communicating their causes to others, and prescribing solutions. Each of these is discussed in the following paragraphs.

The primary benefit of software Performance Antipatterns is to easily identify potential problems in software architectures and designs as early as possible in development when they can be easily corrected. We use them extensively as part of our Performance Assessment of Software Architectures (PASA^{SM}) approach [Williams and Smith 2002].

Because these performance antipatterns occur frequently, it is easy to find them. You still need to quantify execution characteristics, such as the arrival rate of requests or processing time requirements, to determine whether it limits scalability or if they are within scalability targets. For example, you may find The Ramp, but if you only execute it twice a day it won't pose the same problem that it will if it executes on the Fast Path millions of times a day.

The Performance Antipatterns help to focus on the right level of abstraction by identifying the fundamental problem and its solution rather than a specific “fix” that might be outdated over time. For example, long ago an optimal blocksize for files was 2048 bytes. Unfortunately, even though that has changed, developers may continue to use that block size because they remember the specific solution rather than the general concept. Antipatterns focus on the concept and a general solution rather than a specific “fix.”

It is also much easier to grasp the concept of Unbalanced Processing than to understand pages of modeling or measurement results because the concept is a much higher level of abstraction. In addition, it is much easier to understand the significance of the quantitative data *after* you have grasped the essence of the underlying problem with a performance antipattern.

The Performance Antipatterns provide an easy way to communicate what the problem is and why it is a problem. A simple analogy from electrical engineering would be using examples of series and parallel circuits (i.e., patterns) to illustrate how to build proper circuits and an example of a short circuit (i.e., an antipattern) to show what to avoid. Feedback from students in our classes indicates that both types of example are needed to instill performance intuition.

The solutions to these antipatterns embody sound, well-accepted performance principles [Smith and Williams 2002]. These performance principles are similar to architectural or design patterns [Gamma, et al. 1995], [Buschmann, et al. 1996], [Schmidt, et al. 2000] in that they capture “best practices” for creating quality software. They complement architectural and design patterns by providing guidelines for creating responsive software. The antipatterns presented here connect performance principles to solutions to commonly occurring performance problems.

8.0 Summary and Conclusions

Performance antipatterns document common performance mistakes made in software architectures or designs. The use of Software Performance Antipatterns has proven to be valuable in detecting and correcting performance problems as well as building performance intuition in developers.

This paper has presented three previously unreported performance antipatterns and reviewed two others that have appeared recently. The table below summarizes all these antipatterns along with those described in [Smith and Williams 2002] for reference.

Antipattern	Problem	Solution
Unbalanced Processing	Occurs when processing cannot make use of available processors, the slowest filter in a “pipe and filter” architecture causes the system to have unacceptable throughput, or when extensive processing in general impedes overall response time.	1) Restructure software or change scheduling algorithms to enable concurrent execution. 2) Break large filters into more stages and combine very small ones to reduce overhead. 3) Move extensive processing so that it doesn’t impede high traffic or more important work.
Unnecessary Processing	Occurs when processing is not needed or not needed at that time.	Delete the extra processing steps, reorder steps to detect unnecessary steps earlier, or restructure to delegate those steps to a background task.
The Ramp	Occurs when processing time increases as the system is used.	Select algorithms or data structures based on maximum size or use algorithms that adapt to the size.
Sisyphus Database Retrieval Performance Antipattern [Dugan, et al. 2002]	Special case of The Ramp. Occurs when performing repeated queries that need only a subset of the results.	Use advanced search techniques that only return the needed subset.
More is Less [Boyer 2002]	Occurs when a system spends more time “thrashing” than accomplishing real work because there are too many processes relative to available resources.	Quantify the thresholds where thrashing occurs (using models or measurements) and determine if the architecture can meet its performance goals while staying below the thresholds.

Antipattern	Problem	Solution
“god” Class [Smith and Williams 2002]	Occurs when a single class either 1) performs all of the work of an application or 2) holds all of the application’s data. Either manifestation results in excessive message traffic that can degrade performance.	Refactor the design to distribute intelligence uniformly over the application’s top-level classes, and to keep related data and behavior together.
Excessive Dynamic Allocation [Smith and Williams 2002]	Occurs when an application unnecessarily creates and destroys large numbers of objects during its execution. The overhead required to create and destroy these objects has a negative impact on performance.	1) “Recycle” objects (via an object “pool”) rather than creating new ones each time they are needed. 2) Use the Flyweight pattern to eliminate the need to create new objects.
Circuitous Treasure Hunt [Smith and Williams 2002]	Occurs when an object must look in several places to find the information that it needs. If a large amount of processing is required for each “look,” performance will suffer.	Refactor the design to provide alternative access paths that do not require a Circuitous Treasure Hunt (or to reduce the cost of each “look”).
One-Lane Bridge [Smith and Williams 2002]	Occurs at a point in execution where only one, or a few, processes may continue to execute concurrently (e.g., when accessing a database). Other processes are delayed while they wait for their turn.	To alleviate the congestion, use the Shared Resources Principle to minimize conflicts.
Traffic Jam [Smith and Williams 2002]	Occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared.	Begin by eliminating the original cause of the backlog. If this is not possible, provide sufficient processing power to handle the worst-case load.

9.0 References

[Boyer 2002] R. Boyer and G. Rogers, “The More is Less Antipattern” private communication. Paper in development.

[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.

[Dugan, et al. 2002] R. F. Dugan Jr., E. P. Glinert, A. Shokoufandeh, “The Sisyphus Database Retrieval Performance Antipattern,” *Proceedings of the Workshop on Software and Performance* (WOSP 2002), Rome, July 2002.

[Fowler 1999] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Reading, MA, Addison-Wesley Longman, 1999.

[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.

[Larman and Guthrie 2000] C. Larman and R. Guthrie, *Java 2 Performance and Idiom Guide*, Upper Saddle River, NJ, Prentice Hall, 2000.

[Meszaros 1996] G. Meszaros, “A Pattern Language for Improving the Capacity of Reactive Systems,” in *Pattern Languages of Program Design 2*, J. M. Vlissides, J. O. Coplein and N. L. Kerth, ed., Reading, MA, Addison-Wesley, 1996, pp. 575-591.

[Petriu and Somadder 1997] D. Petriu and G. Somadder, “A Pattern Language For Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers,” *Proceedings of EuroPLoP'97*, Kloster Irsee, Germany, July, 1997.

[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.

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

[Smith and Williams 2001] C. U. Smith and L. G. Williams, “Software Performance AntiPatterns: Common Performance Problems and Their Solutions,” *Proc. CMG*, Anaheim, December 2001.

[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.

[Williams and Smith 2002] “L. G. Williams and C. U. Smith, “PASA^{SM}: A Method for the Performance Assessment of Software Architectures,” *Proceedings of the Workshop on Software and Performance* (WOSP 2002), Rome, July 2002.