Software Performance Antipatterns in Cyber-Physical Systems

Connie U. Smith
Performance Engineering Services
L&S Computer Technology, Inc.
Austin, TX, USA
www.spe-ed.com

ABSTRACT

Software performance antipatterns (SPAs) document common performance problems in software architecture and design and how to fix them. They differ from software antipatterns in their focus on the performance of the software. This paper addresses performance antipatterns that are common in today's Cyber-Physical Systems (CPS). We describe the characteristics of today's CPS that cause performance problems that have been uncommon in real-time embedded systems of the past. Three new performance antipatterns are defined and their impact on CPS is described. Six previously defined performance antipatterns are described that are particularly relevant to today's CPS. The paper concludes with some observations on how this work is useful in the design, implementation, and operation of CPS.

CCS CONCEPTS

• Software and its engineering~Embedded software • Software and its engineering~Software performance • Software and its engineering~Software design engineering

KEYWORDS

Performance antipatterns; Software Performance Engineering (SPE); Cyber-physical systems design; Software architecture

ACM Reference format:

Connie U. Smith. 2020. Software Performance Antipatterns in Cyber-Physical Systems. In *Proceedings of 2020 International Conference on Performance Engineering (ICPE'20), April 22-24 2020, Edmonton Alberta.* ACM, NY, NY, USA, 8 pages. https://doi.org/10.1145/3358960.3379138

1 Introduction

Software Performance Antipatterns (SPAs) document common performance problems in software architecture and design and how to fix them. They were first introduced in [20]; much work has built on the original introduction and is described in Section 2.

The demand for developers with domain expertise as well as expertise with the new Cyber-Physical Systems (CPS) technology exceeds the talent pool. This combination of new technology and lack of expertise dramatically increases the risk of performance (and other) failures. Table 1 contrasts characteristics of CPS of the past with today's CPS to illustrate why CPS performance problems are now occurring much more frequently. These CPS performance antipatterns aim to solve these performance challenges in today's CPS.

This work contributes three new SPAs in Section 3 that we have found in performance engineering of CPS. The second contribution, in Section 4, identifies other - previously defined - SPAs that we have found to be common in SPE studies of CPS.

These SPAs are not revolutionary new ideas. In fact the concepts *should* be familiar to experienced performance engineers: to be classified as an antipattern, the problem must occur frequently. Rare or one-off problems, by definition, are not antipatterns. These SPAs are not the only ones found in CPS; others are possible but in our experience not as common. Likewise, CPS are not the only type of systems where these antipatterns may be found.

This paper's contribution is in identifying three new, previously undefined SPAs and identifying the common SPAs found in CPS. This facilitates more rapid identification and correction of CPS performance problems by performance engineers. More importantly it contributes enabling technology for future automation of the detection and correction of CPS software performance problems.

2 Related Work

Patterns capture expert software design knowledge [5, 13]. Antipatterns extend the notion of patterns to capture common design errors and their solution [4]. *Performance* patterns and antipatterns explicitly address the *performance* of software architecture and design. While patterns have proven to be more useful for software design, antipatterns have proven to be more useful for software performance engineering.

Table 1: Evolution of CPS

Past	Today's CPS
Small scope	Dramatic increase in control variables and automation of tasks
Limited functions	Complex and ambitious functions
Monolithic software	Large numbers of processes that require communication and coordination
User interface limited to a few hardware buttons	Touch screens enable more commands and custom UIs
Constrained platform resources	Right-sizing is difficult and may be dictated by cost and/or availability rather than performance
Expert developers	Demand for developers exceeds availability
Low-level programming languages	Automatically generated code in high-level languages
Little or no middleware	Increase in middleware for common functions
Actions constrained by RTOS features	OS for embedded systems provide broader, built-in functionality
Predetermined schedule of tasks	Embedded OS allow dynamic scheduling.

SPAs were first introduced in [20] followed by two additional sets of antipattern definitions in [21, 23]. Other authors have contributed additional and/or variations of SPAs in [6, 11, 15, 24]. That set of antipatterns has been relatively stable for years.

Performance *requirements* antipatterns were introduced by Bondi in [2]. Six antipatterns identify performance requirement specifications that are ambiguous or even misleading. While these are not run-time SPAs they are nevertheless important to discover at design time because they can lead to performance problems later.

Recently, Microsoft reported new performance antipatterns found in Cloud applications [16]. Five of them are new (Busy Database, Busy Frontend, Monolithic Persistence, No Caching, Synchronous I/O). Three (Chatty I/O, Extraneous Fetching, Improper Instantiation) are special cases of the already known set; including them in the set of SPAs is useful for detection and correction. As new types of systems become popular, we expect new additions to the antipattern collection.

Another important body of work builds on the definition of SPAs seeking to automate the detection and correction of performance problems caused by SPAs. A representative sample and overview of this evolving work is in the following:

- A rule-based Performance AntiPattern Detection (PAD) tool diagnoses component-based enterprise Java Bean (EJB) applications [18].
- Formal logic-based specifications of SPAs are combined with queueing model (QN) results derived from automatic transformation of an architectural model to QN [9].
- Model-driven specification of SPAs (PAML) were combined with Palladio (PCM) performance results derived from UML/MARTE for automatic detection of SPAs [9] – automatic correction was deferred.
- A measurement-based approach explores understanding and formalizing the iterative process of measuring performance, identifying performance antipatterns, (manually) correcting them, then repeating until performance requirements are met in [17]. It focuses on runtime behavior of the platform rather than the software design.
- SPA based detection and refactoring with PADRE combined with Traceability links to automate detection and refactoring from runtime data, ties designs to runtime behavior [1].

These steps - detection, classification, refactoring - are currently design language and system-dependent. They have been demonstrated for UML [7], ADL [10], Palladio [25] and others.

Formal specification and detection of antipatterns is a requisite first step, but the antipattern may or may not cause performance problems. For example, there may be a "One Lane Bridge" in the software, but the usage may be low enough that it does not cause a performance problem. Cortellessa, et.al. classify the problems and identify the "guilty" antipatterns in [8]. The techniques are implemented in Palladio [26].

Some SPAs are easier to automate than others. The above work focuses on the easier ones for proof of concept. This paper focuses on SPAs in the CPS domain; this focus may make automation easier for some SPAs such as, "More is Less" or "The Ramp."

Other work improves performance of CPS using performance model results [14, 27]. That work is not explicitly based on SPAs and is not included here.

The definition of new SPAs and the identification of likely SPAs for a domain such as CPS are important topics because they are necessary enabling technologies for future automation. The overall goal is to achieve the same level of automatic optimization for software architecture/design as optimizing compilers have done for code improvement. The first necessary step is this identification of common performance problems.

3 New Performance Antipatterns

Three new software performance antipatterns are defined in the following sections using this standard template:

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

To be considered an SPA, a problem must be found in many CPS. Each section provides an example of the SPA. We have not found one single case study that perfectly illustrates all of them; instead we describe the best example of each one.

3.1 Are we there yet?

This antipattern refers to repeatedly checking to see if some event has occurred, such as a child on a trip "constantly" asking if they have arrived. The problem is the frequency and overhead of the checking relative to the time it takes for the event to occur. Parents become annoyed when a child asks "are we there yet" every minute or so of a journey that takes several hours. A related problem is (too) frequently reporting status information or logging data. This would be analogous to a parent "constantly" providing updates on current position.

3.1.1 Problem

In CPS, this antipattern is often due to "polling" for information, such as state information, that changes much less often than the polling interval. Polling may also detect whether a new request has arrived or whether an event has occurred. Figure 1 shows an example of polling for the arrival of a request.

The performance problem results when the resources for checking are high and the polling interval is too short. Polling requires overhead processing to be awakened and scheduled, often greater than the actual state checking. It may also require dynamic thread creation/destruction.

Polling is easily recognized in CPS. The analogous problem of overly-frequently reporting or logging status information is less often recognized as a performance problem. The sequence diagram in Figure 1 illustrates both polling and logging.

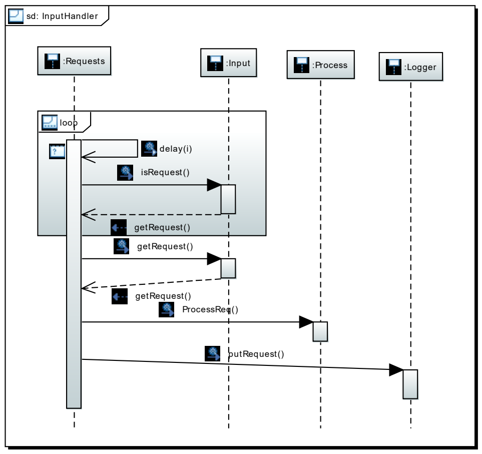


Figure 1. Polling for New Request

3.1.2 Solution

If the polling frequency is extremely high for the circumstances, the solution may be to increase the polling interval delay. In this case the polling savings s will be

$$s = (n - m)\, p$$

where n is the original number of polls or logs, m is the number of new polls or logs, and p is the processing time.

The solution may not be as simple as changing the polling frequency. For example, if polling checks for the arrival of a new request, increasing the polling interval could cause the system to be unresponsive if a request arrives just after a polling cycle completes because there will be a delay before the next cycle begins. A better mechanism would be to notify the Request handler when a new request arrives. An implementation that uses a notification when a new request arrives is shown in Figure 2. The logging is revised to use some application logic to determine when to log requests. It also uses asynchronous logging so the request processing is not delayed.

3.2 Is Everything OK?

This antipattern refers to repeatedly checking the CPS platform status, such as the remaining battery life, storage space, etc. This antipattern is similar to "Are we there yet" in that it also has overly-frequent processing; but we distinguish this one based on the purpose of the processing. By distinguishing it, we draw attention to pro-active design to prevent the problem and to detection when performance problems occur. Otherwise it is an easy problem to miss.

3.2.1 Problem

In today's CPS, status checks are often performed in separate processes/threads that run at designated periodic intervals. They are activated, make a "quick" check of the status of the target resource (eg. battery), report the result, and deactivate. Thus, they are often assumed to be simple overhead tasks and they are seldom considered in the design of the system. The problem occurs when the designated activation interval is very short relative to the occurrence of a status problem and when the accumulated overhead of activation/process/deactivation slows down the main CPS processing.

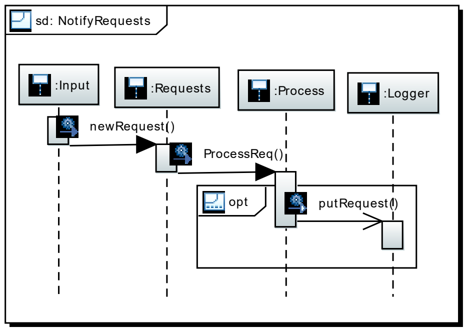


Figure 2. Refactored Request Handling

The simplified SD in Figure 3 shows 3 instances of "Is Everything OK" in one process/thread for convenience. In one case study, there were more checks and they were in separate threads thus increasing the overhead for performing the checks. Each check was invoked once per second. The performance problem was that the start-up time for the system was unacceptably long and because the status checks ran on time intervals there were nearly 1,000 total checks performed during the start-up scenario alone contributing approximately 5 seconds to the overly long startup time! The overhead was substantial because more than 20 threads were created for the checks then destroyed again each second. The processing time required for each check was also longer than one would expect ranging from 1.5 to 4 milliseconds per check/status update.

The status checks contributed to reduced availability of the platform because they reduced the battery life, storage space, etc. to do the checks. In addition, if storage was too low at startup, for example, this design issued frequent warnings then; but the user is generally starting the system to do some work and does not want to have to stop and clean up the disk before proceeding. If the problem occurs while performing a user function it is likely to occur at a random time while in the middle of a task that requires high concentration, and those warnings were frequent and annoying. So, in this case it is better to check storage at the completion of user tasks to make sure there is enough for the next session.

During a scenario of interest (eg., the time for start-up) with total execution time *t; i* is the time interval between checks (e.g., 1 second), each status check *c* runs n_c times (1). Note that "Is Everything OK" applies when $t > i$ so status checks occur far too frequently. The total processing time t_c for status check *c* is its processing time p_c plus the overhead time o_c times the number of checks (2). The total time for all status checks is (3).

$$n_c = t/i_c \quad (1)$$

$$t_c = n_c\,(p_c + o_c) \quad (2)$$

$$T = \Sigma_c\ t_c \quad (3)$$

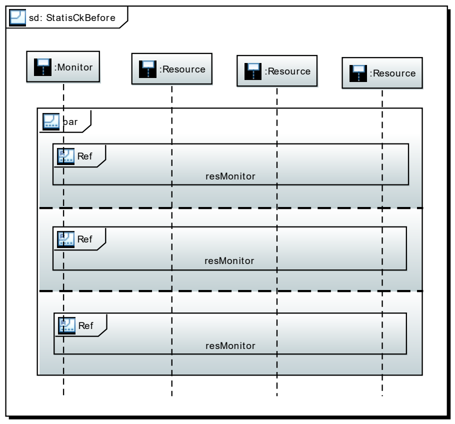
3.2.2 Solution

The solution is to make the platform status check part of the design. Checks can be a combination of any or all of the following:

- Event-based – make checks when a specific event occurs or at a specific point in processing. For example, at a convenient point in startup, check the storage status once; then check status upon completion of each user task when it is convenient to "clean up" before the next use.
- State-based – make checks based on the state of the resource. For example, if the battery state is 90% the next check may be minutes later, but if it is 20% the next check may be seconds later.
- Time-based – make checks at predetermined intervals appropriate for each resource and its depletion rate. For example, check for stale data once per 15 minutes rather than once per second with frequency varying with the type of data and the time it takes to become stale.
- Event-triggered – notification is sent from the resource monitor to subscribers when a specified state occurs, such as battery 20%.

The best option depends on the domain, risk of a status-based failure, and consequences of failure. They should be *designed* (rather than overhead running at some default interval).

Figure 4 shows an alternative in which the Application conditionally calls each resMonitor when appropriate (event-, state- or time-based); the Monitor checks the Resource status and calls handleProblem when needed.



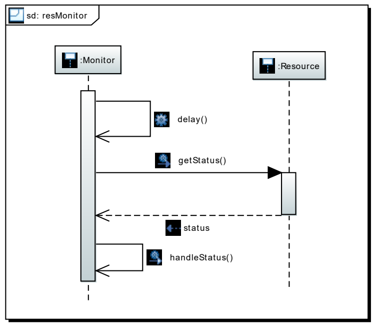


Figure 3. Status Checks

The revised service time $t_c{'}$ is

$$t_c{'} = n_c{'}\,(p_c + o_c)$$

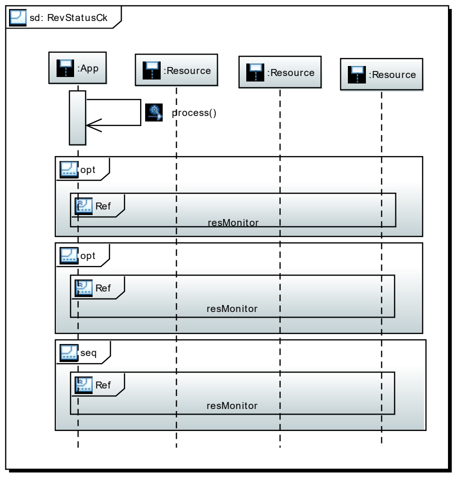
where $n_c{'}$ is the revised number of times the check executes. Event-based or state-based checks no longer depend on the execution time or interval so the savings can be significant.

3.3 Where Was I?

This antipattern refers to processes that do not remember state information and when they (re-)start they start from a predefined state that is frequently not the user's desired state. An example is with intermittent windshield wipers: a change to the setting, such as turning off, on, or changing interval, starts with an extra "wipe." In this simple example the result may be only an extra, noisy scrape of wipers across a dry windshield when turning them off. The antipattern in other applications may result in excessive overhead to recalculate state, or worse a failure as described in the following subsection.

3.3.1 Problem

It is easier to design systems that start from the same initial step rather than remembering or checking the last state to determine the desired starting point - especially in systems with multiple users that may be in different states. This antipattern not only causes excessive overhead to recalculate state, but it also affects usability.



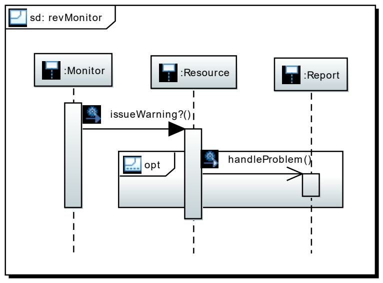


Figure 4. Revised Status Checks

Consider an avionics system that reports weather conditions along a specified route. When the device is activated to review weather predictions while in flight, "Where was I" recalculates weather predictions. This requires network connectivity to retrieve weather data. When flying in a location that does not have network connectivity, after a very long delay, the device reports a connectivity error and no displayed results; however, the desired result is a re-display of the last prediction. The sequence diagram in Figure 5 shows its typical behavior.

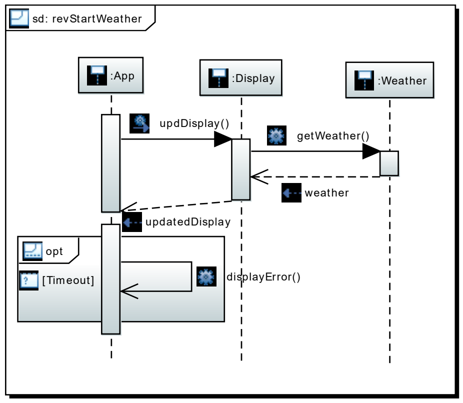


Figure 5. Recalculate Display

A second problem with this system is that the displayed results are not automatically recalculated if/when the weather data changes while viewing the results display. Fortunately the data changes relatively infrequently, but there was no "refresh" command while viewing predictions; one must close the app and restart it to view updated data.

Another example is when one wants to change one setting in an application, but "Where Was I" starts over from the beginning and requires specifications for all settings. This is problematic when the change is needed immediately but the multiple steps required cause the function to be unavailable for a period of time. For example, changing one setting for an instrument approach while flying in instrument weather conditions needs to be done very quickly and having to specify all settings may be an unacceptable, even dangerous situation.

In another IoT application, "Where was I" first tries to connect to the all of its last known devices, but there has been a change to the environment and all devices are not available to re-connect. Its timeout interval is much too long (over 1 minute) so the user experiences a frustrating delay before she/he is able to interact with the app and reach the desired state.

3.3.2 Solution

The solution usually depends on the CPS and its use. In the avionics weather example, it is easy to first check for connectivity then either display previous results or calculate new results as in the sequence diagram in Figure 6.

In other situations it may require saving state, offering context-dependent actions, specifying a reasonable timeout interval, or a custom-designed solution based on specific CPS domain/usage.

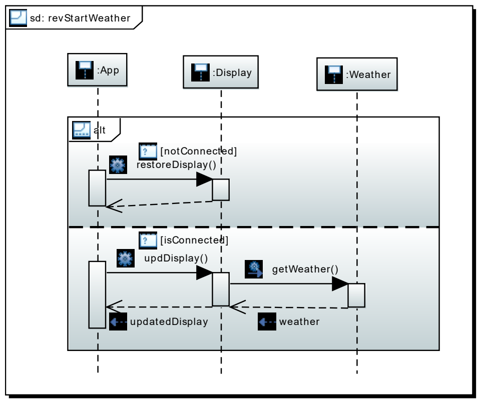


Figure 6. Check Connectivity

We could quantify the savings in resource consumption for re-calculating state versus saving state. The more important savings is likely to be in the end-to-end response time for the user to execute the desired task. Resolving an emergency situation, and being able to do so in less time, may be priceless!

4 Other Common CPS Antipatterns

The following SPAs have been defined previously. In the following subsections we focus on how they typically apply to CPS. The earlier publications have quantified the improvements that can be achieved by refactoring. For brevity, we do not include previous sequence diagrams nor quantify improvements here.

4.1 Unnecessary Processing

This performance antipattern addresses processing that is executed in a critical scenario that is either not needed, or not needed at that time [21]. While it is difficult to judge whether or not processing is needed at the architectural level [9], it is possible to determine that the processing *results* are not used in a critical scenario and can be deferred.

Startup is a critical scenario in most CPS. One common example of "Unnecessary Processing" is creating many platform and application screens during startup. This is often an expensive process requiring many constructors for widgets on the screen. In one CPS approximately 30 screens were created during startup requiring between 100 ms for simple ones, 0.5 sec for average screens, and up to 9 seconds for the most expensive screen (which was rarely used but created during startup nonetheless).

When there are substantial periods of "think time" that occur after startup, during which the processor is idle or less busy, those periods can be used to create screens that are unlikely to be needed until later. Screens can be ranked by when and how often they are likely to be needed: important/frequently needed screens can be created during startup, Screens unlikely to be needed can be created only when they are requested, and others can be created in the background during less busy periods. This can be explicitly designed into the application, or it can be accomplished in a background process that executes at a lower priority.

Note that the total processing time required is the same if you only change the point in time that the code executes, but it will improve overall responsiveness.

4.2 How Many Times Do I Have to Tell You?

Originally documented in [6], this antipattern occurs when a common method is called many times by other methods, but it is only needed once. This happens when multiple paths in a call tree repeatedly call the same commonMethod. Because the implementation of methods is (deliberately) hidden, it is not obvious that these methods are called so many times.

Figure 7 shows a sample call tree: the red arrows show the multiple paths to the commonMethod (bottom-left of figure), the black arrows do not lead to commonMethod. The revised scenario (not shown) removed the redundant calls from the method implementation and explicitly called the commonMethod once. An 80% reduction in processing time for this scenario was achieved.

This antipattern is usually detected in performance improvement projects to identify and eliminate redundant calls rather than at design time. It is detected by measuring the number of times each method is called during performance tests then analyzing why the processing-intensive methods are called and who calls them.

"How many times" occurs in all types of systems. It is especially problematic for CPS that require a high level of responsiveness and are implemented with many independent processes that have relatively high overhead for calling/communicating with other processes. The redundant work causes a noticeable slow down.

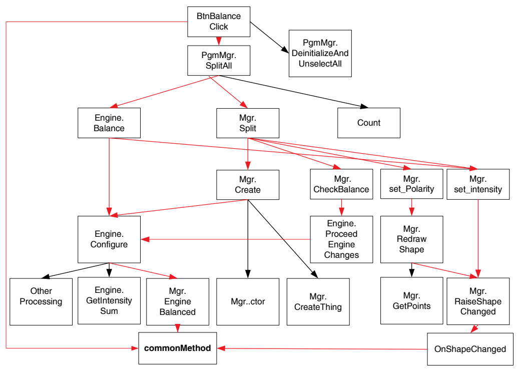


Figure 7. How Many Times.

4.3 More is Less

This antipattern occurs when having too many of some resource results in poorer overall performance [19]. Examples are too many processes or threads relative to the number of processors, and too many pooled resources.

Often CPS are created with large numbers of independent processes/threads, but in single-user CPS or IoT applications, running on small platforms with few processors, these threads do not increase the concurrency in the software, but they do introduce additional overhead for scheduling, dispatching, context switching, communication, and possibly page faults.

Profiler data shows the time spent in various threads, but it does not show how much of the time is due to operating system overhead so performance problems due to threading may not be obvious. Figure 8 shows the high CPU usage that resulted from too many threads for a CPS application (in the figure, the blue is the CPU usage, the red is the number of threads).

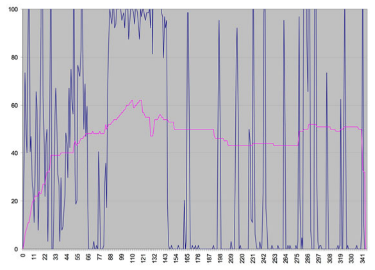


Figure 8. Cost of More is Less

4.4 The Ramp

When this antipattern is present, processing time increases as the system is used [21]. With "The Ramp" the response time increases exponentially as processing time increases linearly. It presents a scalability problem that is often not detected during testing when test data does not contain enough items to reveal the phenomenon.

Previous instances of "The Ramp" were usually associated with database applications. As CPS expand into new domains they are subject to similar pitfalls. In a medical application, for example, one screen displayed a list of all patients and the user scrolled through it to select the current patient before beginning the actual use of the device. Even when the device was first tested, the time to populate that screen was 10 seconds. It would have gotten much worse as the system was used and the number of patients increased.

In CPS other occurrences of "The Ramp" are associated with:

- Changing environments - For example, when the number of nodes in networks increases so does the communication processing overhead
- History - For example, if future results depend on past behavior an increasing amount of processing may be required as the amount of historical information increases
- Environmental influences - For example, when security cameras have more processing to do when there is an increase in activity in the frame or even when there is background movement such as trees blowing in the wind.

The solution to "The Ramp" when it involves databases is to change the search algorithm. For example, for the medical patient application, rather than present a long list of patients, let the user begin typing the patient's name to retrieve a smaller set of possibilities. Other design alternatives include downloading only the patients to be treated that day so that the list is always small.

The solution to "The Ramp" for CPS is also likely to require a different design and will be domain dependent thus making it difficult to automatically correct the antipattern.

4.5 Museum Checkroom

This antipattern, first reported in Bondi [2], "leads to a deadlock in a system in which the elements of a resource pool are acquired and released singly by processes that pass through a common FCFS queue to do one and then the other" [3]. Bondi establishes that the system *will* reach a deadlock state, and that the time to the onset of deadlock depends on the request rate for resources in the pool, the average holding time of the resources, and the size of the pool.

The solution is to use priority queuing rather than FCFS queueing and to assign a higher priority to processes queueing to release the resource.

Deadlocks cause a system to fail: it appears that the system "hangs up" and the usual fix is to reboot the system. When this antipattern is the cause of the problem, the system will run fine for a while then eventually the same failure occurs. It is often quite difficult for a performance engineer to determine the root cause of the problem, but once diagnosed the solution is straightforward.

This antipattern is much easier to prevent at design time than to detect and correct after it has become a problem. It should be rather easy to detect it from design specifications and/or performance models. The automatic detection and refactoring techniques cited in Section 2 could be a powerful tool.

4.6 Falling Dominoes

"Falling Dominoes" occurs when one failure causes performance failures in other components. An example occurred in a component that received input then broadcast it to many other components [23]. When a communication channel failure caused one of the receiving components to repeatedly request re-transmission, it slowed down the entire system. Another instance of the antipattern occurred when one receiver failed, and it caused a feeder process to quit sending to all receivers.

Today's CPS are prone to this type of performance problem because they typically have many interacting pieces. They may rely on other COTS products for some of their features, such as network connectivity, so the failure modes may not be known at design time. These are not only performance problems, they are also reliability and fault tolerance problems. Thus performance engineering needs to include scenarios of failure modes to detect these problems early.

The solution is to make sure that broken pieces are isolated until they are repaired. The broadcast component could monitor re-transmission requests and when a threshold is reached, stop sending to a receiver until it is repaired. Feeder processes should not stop when one receiver fails. The failed process should be isolated until it is repaired.

Another solution to "Falling Dominoes" is to use autonomic techniques and monitor the ratio of error processing to useful work, and, when a designated threshold has been reached, shut down failed components rather than continue to execute error processing.

5. Observations

Some of the SPAs are similar. At the extreme they all result in extensive processing that contributes to overly long response times. Some of the documented performance antipatterns are special cases of an already known set (see for example [16]). It is worthwhile to have SPAs adapted to specific domains because it is more likely that the problems will be detected and corrected – either manually as in PASA [28] or with automated techniques as in Section 2.

The solution for SPAs may correlate with OO patterns. In some cases, "Tell Don't Ask" [https://pragprog.com/articles/tell-dont-ask] could be a solution for "Are We There Yet" or "Is Everything OK?" If the resMonitor (Fig 3) is a system component rather than part of the application, pattern applicability may not be recognized.

Measurement-based approaches such as [17] have been used to identify "guilty" antipatterns. Unless the measurements are tied back to the design, though, many problems appear to be extensive processing (a subset of "Unbalanced Processing"). It is possible that connecting the measurements to design models as in [1] could identify other antipatterns that are the root cause of the problem and make automatic correction easier.

Another powerful extension could be using SPAs for identifying elements of designs that should include proactive instrumentation to detect when performance antipatterns are nearing a threshold where they may become guilty.

Some SPAs have been excluded from automation research. "Unnecessary processing" was excluded in [9] because it was deemed too difficult to judge the importance of application code at the architectural level to determine the necessity of the processing. Future work could use requirements specifications to determine guilt caused by "Unnecessary processing." Another possibility is to use measurement-based analysis to look at the components with the highest usage, and then question whether they are necessary at the time they are invoked.

"Falling Dominoes" has been excluded because it was considered a reliability/availability concern. We argue that it should be part of the quest for automation because it affects usability thus it affects end-to-end user performance and performability. Also when "Falling Dominoes" occurs in operation the cause is often difficult to detect, and thus it benefits most from automatic detection. Analysis techniques such as those of [3] could assist with automatic detection.

Lastly, "bad smells" are defined as signs of potential problems in code; they have been used to guide refactoring in [12]. Unless those "bad smells" occur in "guilty" components, though, it is not necessary to refactor them. In fact, in many of our SPE engagements, we were brought in after much time was lost correcting "bad smells" that made little or no improvement in the overall performance of the system. For this reason, the SPAs are far more useful for performance improvement.

6 Conclusions

Performance antipatterns document common performance mistakes made in software architectures or designs. The use of SPAs has proven to be valuable in detecting and correcting performance problems as well as building performance intuition in developers by explaining the problems in an easy-to-understand way. This paper introduces the new antipatterns but does not classify them, nor does it address automatic detection or automatic refactoring. We leave these extensions to the experts in the automation area.

This paper documented three new SPAs that we have found to be common in CPS. It is useful to know which types of potential problems to look for when conducting SPE studies to quickly detect and correct problems. The new antipatterns also may occur in systems other than CPS.

Other, previously defined performance antipatterns could occur in future CPS. For example, database antipatterns are likely to appear as CPS evolve and incorporate large data.

Experience with these SPAs has largely been expert-based performance engineering with extensive efforts to understand designs, collect data, construct models, analyze results and explain options for improvement. This was useful for identifying the new performance antipatterns; these SPAs provide enabling technology that is necessary for further research and development into automation. The real potential for adoption and use for software development will come with extending methods for automatic detection and correction of guilty performance antipatterns.

REFERENCES

[1] D. Arcelli, V. Cortellessa, D. DiPompeo. 2018. Performance-driven software model refactoring. Information and Software Technology 95. 366-397.
[2] André Bondi. 2015. Foundations of Software and System Performance Engineering. Addison-Wesley.
[3] André Bondi. 2018. Predicting the time to migrate into deadlock using a discrete time Markov chain. WOSP-C Workshop. ICPE'18 Companion. Berlin.
[4] W. J. Brown, R. C. Malveau, H. W. McCormick III, and T. J. Mowbray. 1998. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, Inc. New York.
[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. 1996. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley and Sons. Chichester, England.
[6] C. Cates. 2004. Where's Waldo: Uncovering hard-to-find application killers. Proc. CMG. Las Vegas.
[7] V.Cortellessa, A.DiMarco, R.Eramo, A.Pierantonio, C.Trubiani. 2010. Digging into UML models to remove performance antipatterns. ICSE Workshop. Quovadis, 9–16. ?</bib>
[8] V.Cortellessa, A.Martens, R.Reussner, C.Trubiani. 2010. A process to effectively identify "Guilty" performance antipatterns. D.S. Rosenblum, G.Taentzer (eds.). FASE 2010. LNCS vol. 6013. 368–382. Springer. Heidelberg.
[9] V.Cortellessa, A.DiMarco, C.Trubiani. 2012. Software performance antipatterns: Modeling and analysis. M.Bernardo, V.Cortellessa, A.Pierantonio (eds). Formal Methods for Model-Driven Engineering. SFM 2012. Lecture Notes in Computer Science, vol 7320. Springer. Berlin, Heidelberg.
[10] Martina De Sanctis, C. Trubiani, V. Cortellessa, A. Di Marco, M. Flamminj. 2016. A model-driven approach to catch performance antipatterns in ADL specifications, Information and Software Technology. DOI: http://dx.doi.org/10.1016/j.infsof.2016.11.008 </bib>
[11] R. F. Dugan Jr., E. P. Glinert, A. Shokoufandeh. 2002. The Sisyphus database retrieval performance antipattern," Proceedings of the Workshop on Software and Performance (WOSP 2002),Rome.
[12] M. Fowler, K. Beck. 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional. ?</bib>
[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. Reading, MA.
[14] A. Gómez, C.U. Smith, A.C. Spellmann, J.Cabot. 2018. Enabling performance modeling for the masses: Initial experiences. SAM 2018. 105-126.
[15] C. Larman. 2000. Aggregate entity pattern. Software Development, vol. 8 no. 4. 46-52.
[16] https://docs.microsoft.com/enus/azure/architecture/antipatterns/</bib>
[17] L. Pagliari, R. Mirandola, C. Trubiani. 2019. Engineering cyber-physical systems through performance-based modelling and analysis. Journal of Software Evolution and Process, Wiley, to appear.
[18] T.Parsons, J.Murphy. 2008. Detecting performance antipatterns in component based enterprise systems. Journal of Object Technology 7. 55–91.
[19] G. Rogers and R. Boyer. 2002. The More is Less antipattern. Private communication.
[20] C. U. Smith and L. G. Williams. 2000. Software performance antipatterns. Proceedings Second International Workshop on Software and Performance (WOSP2000). Ottawa, Canada. 127-136.
[21] C. U. Smith and L. G. Williams. 2002. New software performance antipatterns: More ways to shoot yourself in the foot. Proc. CMG, Reno.
[22] C. U. Smith and L. G. Williams. 2002. Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley. Boston, MA.
C.U. Smith, L.G. Williams. 2003. More new software antipatterns: Even more ways to shoot yourself in the foot. International Computer Measurement Group Conference. 717–725. ?</bib>
[24]B.A. Tate. 2002. A taste of 'Bitter Java:' The Round-tripping antipattern. http://www-106.ibm.com/developerworks/java/library/j-bitterj-java/bjsidebar1.html.
[25] C.Trubiani, A.Koziolek. 2011. Detection and solution of software performance antipatterns in Palladio architectural models. International Conference on Performance Engineering (ICPE), 19–30.
[26] C. Trubiani, A. Koziolek, V. Cortellessa, R. Reussner. 2018. Guilt-based handling of software performance antipatterns in Palladio architectural models. Journal of Systems and Software 95, 141-165.
[27] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, H. Knoche. 2018. Exploiting load testing and profiling for performance antipattern detection. Journal of Information and Software Technology. Elsevier.
[28] L.G.Williams, C.U.Smith. 1998. Performance engineering of software architectures. Proceedings Workshop on Software and Performance, Santa Fe, NM.