Why Feedback Implementations Fail: The Importance of Systematic Testing

Joseph L. Hellerstein
Google, Inc.
jlh@google.com

Abstract

Over the last decade, there has been great progress in using formal methods from control theory to design closed loops in software systems. Despite this progress, formal methods are rarely used by software practitioners. One reason is the substantial risk of making changes to closed loops in software products, code that is typically complex and performance sensitive. We argue that broad adoption of formal methods for controller design require addressing how to reduce the risk of making changes in controller implementations. To this end, we propose a framework for testing controller implementations that focuses on scenario coverage, scenario evaluation, and runtime efficiencies. We give examples of applying this framework to the Microsoft .NET Thread Pool, the Google Cluster Manager, and a Google stream processing system.

Keywords

1. Introduction

The last decade has seen growing interest in using formal methods from control theory to design and implement closed loops in software systems. There have been a large number of research prototypes as well as several commercial implementations including: managing buffer pools in the IBM DB2 Universal Database Server, scheduling jobs in the Hewlett Packard Global Workload Manager, and optimizing thread concurrency levels in Microsoft .NET. (See [1] and [3] and the references therein.) Despite these results, formal methods are rarely used by software practitioners to build closed loop systems. This paper asserts that one reason for the limited use of formal approaches is the lack of an effective testing methodology to mitigate the risks of making changes to controllers in software products.

Closed loops in software products typically involve some of the most complex and performance sensitive parts of the product. As such, changes to controllers, especially radical re-designs, involve a great deal of risk to the software vendor. For example, changes to the .NET Thread Pool have the potential to affect almost all of the 1B Windows systems on the planet. This means that even if the use of formal methods can provide superior results in most cases, the possibility of unanticipated "corner cases" can tip the balance against making substantial changes in controller design.

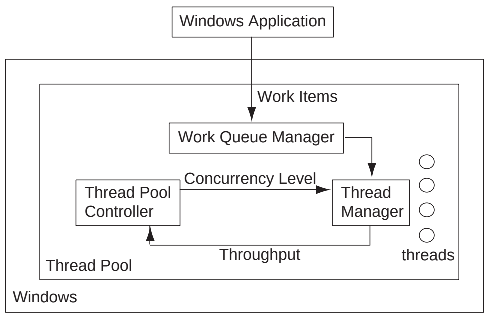

Figure 1. *.NET Thread Pool*

A key consideration in mitigating the risk of a software change is to provide testing that covers a wide spectrum of scenarios encountered by the controller. For example, in the .NET Thread Pool, this means having tests that incorporate the possible arrival patterns and execution characteristics of work items presented to the Thread Pool. Unfortunately, such considerations are not addressed by the current literature on software testing. Rather, this literature focuses on specification-driven tests, testing levels (e.g., unit, integration, system), and the degree of test transparency (e.g., black box vs. white box) [4].

Herein, we advocate a framework for unit testing of controllers for software systems. Our intent is that this framework greatly reduce the chances of corner cases causing failures or degradations when the controller is deployed widely. Our framework centers on three concepts:

- *Scenario coverage.* There must be a tractable approach to generating all scenarios of interest.
- *Scenario evaluation.* There must be a simple criteria to determine if the outcome of a scenario results in acceptable performance.
- *Runtime efficiency.* The runtime of a scenario should be sufficiently short so that a large number of scenarios can be explored. Typically, this means running a scenario in seconds or even milliseconds, not minutes or hours.

2. Case Study of the .NET Thread Pool

This section provides a brief description of testing issues encountered in the development of a new controller for the .NET Thread Pool. More details can be found in [2].

The .NET Thread Pool provides users with a simple abstraction for running work in parallel. As depicted in Figure 1, Windows applications place work items into the Thread Pool, and the Thread Pool Thread Manager initiates the asynchronous exe-

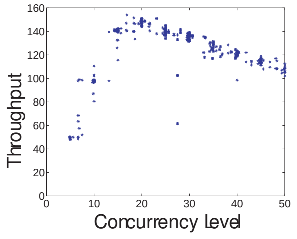

Figure 2. *Concurrency-throughput curve for a .NET application.*

cution of work items once a thread is available [5]. The Thread Pool Controller estimates the number of threads, or concurrency level, that maximizes throughput. For example, Figure 2 displays the **concurrency-throughput curve** for a .NET application. In this case, the Thread Pool Controller seeks a concurrency level that is approximately 18. Note that this curve is **unimodal** in that it rises to a peak and then eventually declines. (Unimodal is a somewhat weaker condition than concave.)

At first glance, it seems that testing a new Thread Pool Controller requires running a large number of benchmarks to ensure that there is coverage for the possible scenarios of work item arrivals and execution characteristics. Such an approach is time-consuming and expensive. It is also unnecessary.

To see why we need not consider the specifics of work items, observe that the controller's behavior does not depend on work items per se. Rather, from Figure 1, the behavior of the controller is determined by the relationship between the concurrency level and throughput; that is, by the concurrency-throughput curve. Further, note that having a large number of scenarios based on work item characteristics does not necessarily mean that there is better coverage of controller corner cases if many of the work item scenarios have the same concurrency-throughput curve.

The foregoing observation means that controller scenarios should be defined in terms of concurrency-throughput curves such as Figure 2. Such curves are relatively easy to generate. Further, since these curves have a single peak, we can compare the throughout that the controller achieves with the maximum throughput possible for the scenario.

From the foregoing, we see that testing the controller requires three components: (a) the controller implementation, (b) a scenario driver that generates unimodal concurrency-throughput curves; and (c) a scenario evaluator that quantifies how close the controller is to optimal performance. Items (b) and (c) are fairly simple to implement (a few hundred lines of C++ code), and the execution of a scenario takes milliseconds. In contrast, running a single performance benchmark such as TPC-W takes minutes to an hour.

3. Testing Framework

The .NET Thread Pool provides an excellent example of the testing framework that we advocate. Specifically,

- *Scenario coverage* is addressed by considering a large set of unimodal concurrency-throughput curves (e.g., based on existing performance benchmarks).
- *Scenario evaluation* is done by comparing the throughput achieved by the controller with the maximum throughput for the scenario.
- *Runtime efficiency* is achieved by only executing those parts of Windows and .NET that are relevant to the controller evaluation. In contrast, benchmarks such as TPC-W require running Windows applications and a complete .NET instantiation.

This framework has been applied at Google as well. One application is to the Google Cluster Manager that schedules tasks on thousands of machines. This is a complex feedback system that considers task priority and class. Here, priority indicates the importance of the work to Google (production or non-production), and class indicates whether the task interacts with end-users (interactive or batch). Thus, a framework for controller evaluation can use scenarios expressed in terms of the task being scheduled and the tasks running on a candidate machine. For each task, we consider its priority (2 levels), class (2 levels), and resource requirements. Resource requirements has two dimensions: resource type (memory or CPU) and size (low or high). This means that there are 16 possibilities for each task, and so 16^{mn+1} possible scenarios if there are n tasks running on m candidate machines. Although the number of scenarios rapidly becomes unmanageable, the case of $m = 2 = n$ is tractable, and seems to be sufficient in practice. Scheduling outcomes are evaluated based on the semantics of the interaction (e.g., production priority is preferred to non-production). This approach provides a very efficient way to test new schedulers since only the affected scheduling logic needs to be invoked to assess correctness and performance.

A second example of applying the controller testing framework at Google relates to a stream processing system used to provide aggregations in real time. Here, a scenario is expressed in terms of the possible performance bottlenecks, such as: (a) persistent inefficiencies at particular stages in the stream pipeline, (b) data-dependent inefficiencies (either periodic or random), and (c) inefficiencies in parallelizing pipeline stages. Scenarios are evaluated based on how rapidly inefficiencies are corrected by the stream processing system (e.g., by adding more instances of a bottleneck stage). The efficiency of testing in this manner largely rests on the ability to create a virtual execution environment in which lower level services used by the stream processing system (e.g., file system accesses) can be emulated.

4. Conclusions

At present, there is very limited use of formal methods from control theory in the design of closed loops in commercial software products. One reason for this is that closed loops in software products are typically complex and performance sensitive; so, changing closed loops has high risk. We argue that the broader use of formal approaches to controller design requires mitigating these risks by having a systematic approach to testing controller implementations. To this end, we propose a testing framework that focuses on scenario coverage, effective approaches to scenario evaluation, and runtime efficiencies to be able to assess a large number of scenarios.

References

[1] Tarek F. Abdelzaher, John A. Stankovic, Chenyang Lu, Ronghua Zhang, and Ying Lu. Feedback performance control in software services. *IEEE Control Systems Magazine*, 23(3):74–90, 2003.

[2] Joseph L. Hellerstein, Vance Morrison, and Eric Eilebrecht. Applying control theory in the real world: experience with building a controller for the .net thread pool. *ACM SIGMETRICS Performance Evaluation Review Archive*, 37(3):38–42, 2010.

[3] Joseph L. Hellerstein, Sharad Shingal, and Qian Wang. Research challenges in control engineering of computing systems. *IEEE Transactions on Network and Service Management*, 6(4):206–211, 2010.

[4] Cem Kaner, Jack Falk, and Hung Quoc Hguyen. *Testing Computer Software*. John Wiley and Sons, 1999.

[5] Steven Pratschner. *Common Language Runtime*. Microsoft Press, 1st edition, 2005.