Applying Control Theory in the Real World: Experience With Building a Controller for the .NET Thread Pool

Joseph L. Hellerstein[*]
Google, Inc.
650 N. 34 Street
Seattle, WA USA
jlh@google.com

Vance Morrison
Microsoft Developer Division
One Microsoft Way
Redmond, WA USA
vancem@microsoft.com

Eric Eilebrecht
Microsoft Developer Division
One Microsoft Way
Redmond, WA USA
ericeil@microsoft.com

ABSTRACT

There has been considerable interest in using control theory to build web servers, database managers, and other systems. We claim that the potential value of using control theory cannot be realized in practice without a methodology that addresses controller design, testing, and tuning. Based on our experience with building a controller for the .NET thread pool, we develop a methodology that: (a) designs for extensibility to integrate diverse control techniques, (b) scales the test infrastructure to enable running a large number of test cases, (c) constructs test cases for which the ideal controller performance is known a priori so that the outcomes of test cases can be readily assessed, and (d) tunes controller parameters to achieve good results for multiple performance metrics. We conclude by discussing how our methodology can be extended, especially to designing controllers for distributed systems.

1. INTRODUCTION

Over the last decade, many researchers have advocated the benefits of using control theory to build systems. Examples include controlling quality of service in web servers [10], regulating administrative utilities in database servers [6], controlling utilizations in real time systems [9], and optimizing TCP/IP [5]. Despite these advances, control theory is rarely used by software practitioners. We claim that this is because the successful application of control theory to systems requires addressing many methodological considerations that are largely ignored in existing research.

We demonstrate our thesis by discussing issues we encountered in developing a controller for the .NET thread pool [7]. The thread pool exposes an interface called `QueueUserWorkItem()` through which programmers place work items into a queue for asynchronous execution. The thread pool assigns work items to threads. The thread pool controller determines the number of threads or concurrency level that maximizes throughput by on-line estimation of the relationship between concurrency level and throughput. For example, Figure 1 displays the concurrency-throughput curve for a .NET application. In this case, the thread pool controller seeks a concurrency level that is approximately 18.

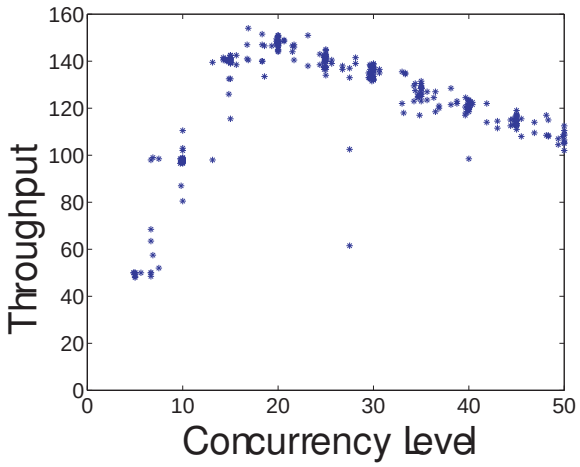

Figure 1: *Concurrency-throughput curve for a .NET application. Throughput degrades if the concurrency level exceeds 20 due to the overhead of context switching.*

Although we readily identified a set of control techniques to employ in managing the .NET thread pool, our progress was thwarted by several methodology considerations in controller design, testing, and tuning. Unfortunately, the current research literature offers little help. The ControlWare framework [11] describes middleware for building controllers, but it addresses only a limited aspect of controller design, and it does not consider testing and tuning. There are a few reports of applying control theory to commercial products such as IBM's DB2 for throttling administrative utilities [6] and optimizing memory pools [4] as well as Hewlett Packard's Global Workload Manager [1]. Beyond this, there have been a plethora of experiments in which control theory is applied to software systems in testbeds (e.g., see [2] and the references therein). Unfortunately, these papers focus almost exclusively on control laws. In summary, none of this research adequately addresses controller design, testing, and tuning.

This paper describes a methodology for controller design, testing, and tuning based on our experience with applying control theory to the .NET thread pool. A central concern in controller design is providing extensibility, especially to integrate diverse

control techniques. Our methodology addresses this by structuring controllers as finite state machines. One concern in testing is providing a test infrastructure that scales well with the number of test cases. Our methodology addresses this by using resource emulation. Also in testing, we must construct test cases whose ideal outcomes are known a priori so that observed outcomes can be assessed. Our methodology addresses this by using a test case framework for which ideal test outcomes can be computed analytically. Last, tuning controller parameters requires selecting parameter settings that provide good results for multiple performance metrics. Our methodology addresses this by selecting tuning parameter settings that lie on the optimal frontier in the space of performance metrics.

The remainder of this paper is organized as follows. Section 2 discusses controller design, Section 3 addresses testing, and Section 4 considers tuning considerations. Our conclusions are contained in Section 5.

2. DESIGN

The objective of the .NET thread pool controller is to find a concurrency level that maximizes throughput, where throughput is measured in completed work items per second. In addition, the controller should minimize the concurrency level so that memory demands are reduced, and minimize changes in concurrency level to reduce context switching overheads. In practice, there are trade-offs between these objectives.

Our starting point for the control design is the concurrency-throughput curve, such as Figure 1. While the curve may change over time, we assume that it has a unimodal shape. This assumption suggests that hill climbing should be used to optimize the concurrency level. However, many factors make hill climbing non-trivial to implement for the thread pool controller.

- DR-1: The controller must consider the variability of throughput observations.
- DR-2: The controller must adapt to changes in the concurrency-throughput curve (e.g., due to changes in workloads).
- DR-3: The controller needs to consider the transient effects of control actions on throughput due to delays in starting new threads and terminating existing threads.

DR-1 can be addressed by Stochastic Gradient Approximation (SGA), a technique that does hill climbing on unimodal curves that have randomness [8]. We use the SGA variant based on finite differences, which has the control law:

$$u_{k+1} = u_k + gd_k, \quad (1)$$

where k indexes changes in the concurrency level, u_k is the k-th setting for the concurrency level, g is a tuning constant called the control gain, and d_k is the discrete derivative at time k.

For DR-2, we use change point detection, a statistical technique for detecting changes in the distributions of stochastic data [3]. The concurrency-throughput curve changes under several conditions: (a) new workloads arrives; (b) the existing workloads change their profiles (e.g. move from a CPU intensive phase to an I/O intensive phase); and (c) there is competition with threads in other processes that reduces the effective bandwidth of resources. Transition T_b in Figure 2 detects these situations by using change point detection [3]. Change point detection is an on-line statistical test that is widely used in manufacturing to detect process changes. For example, change point detection is used in wafer fabrication to detect anomalous changes in width widths. We use change point detection in two ways. First, we prune older throughputs in the measurement history if they differ greatly from later measurements since the older measurements may be due to transitions between concurrency levels. Second, we look for change points evident in recently observed throughputs at the same concurrency level.

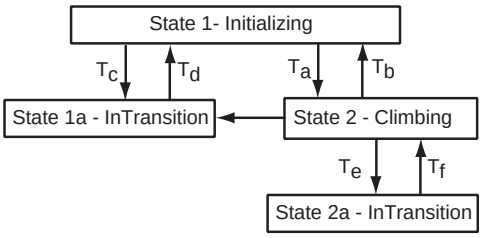

Figure 2: *State diagram for thread pool controller.*

For DR-3, we use dead-time detection, a technique that deals with delays in effecting changes in concurrency level. To elaborate, one source of throughput variability within a concurrency level arises if a controller-requested change in concurrency level is not immediately reflected in the number of active threads. Such delays, which are a kind of controller dead-time, are a consequence of the time required to create new threads or to reduce the number of active threads. We manage dead-time by including states 1a and 2a in Figure 2. The controller enters an "'InTransition"' state when it changes the concurrency level, and it leaves an "InTransition" state under either of two conditions: (1) the observed number of threads equals the controller specified concurrency level; or (2) the number of threads is less than the controller specified concurrency level, and there is no waiting work item.

There is considerable complexity in designing a controller that integrates SGA, change-point detection, and dead-time detection. Further, we want it to be easy to extend the thread pool controller to integrate additional control techniques. This led to the following considerations:

Methodology challenge 1: Provide an extensible controller design that integrates diverse control techniques.

Approach 1: Structure the controller as a finite state machine in which states encapsulate different control techniques.

Structuring the controller as a finite state machine allows us to integrate diverse control techniques. Figure 2 displays such a structure for our thread pool controller. SGA is implemented by a combination of the logic in State 1, which computes throughput at the initial concurrency level, and State 2, which implements Equation (1). Change-point detection is handled by the

Transition	Description
T_a	Completed initialization
T_b	Change point while looking for a move
T_c	Changed concurrency level
T_d	End of initialization transient
T_e	Changed concurrency level
T_f	End of climbing transient

Figure 3: *Description of state transitions in Figure 2.*

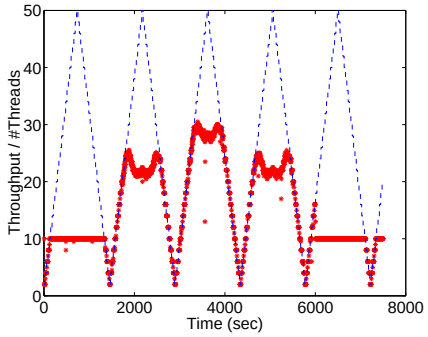

Figure 4: *Throughput (circles) at control settings specified by a cyclic ramp (line).*

transition T_b. Dead-time detection is addressed by including States 1a and 2a and their associated transitions.

The controller is implemented in C#, an object-oriented language similar to JAVA^{TM}. An object-oriented design helps us address certain implementation requirements. For example, we want to experiment with multiple controller implementations, many of which have features in common (e.g., logging). We use inheritance so that features common to several controllers are implemented in classes from which other controllers inherit. The controller code is structured into three parts: implementation of the state machine in Figure 2, implementation of the conditions for the transition in Figure 3, and implementation of the action part of transitions.

3. TESTING

The wide-spread and diverse use of the .NET thread pool mandates that there be extensive testing for both correctness and performance. Some of the testing is done with performance benchmarks such as those from the Transaction Processing Council (e.g., TPC-W). However, to cover the diversity of .NET applications, we also use a set of synthetic applications. This section focuses on the latter.

A synthetic work item is described in terms of its resource profile, such as the CPU, memory, and web services it consumes. CPU and memory are of particular interest since excessive utilizations of these resources leads to thrashing, which is a specific area of concern for the thread pool controller. We use the term **workload** to refer to a set of work items with the same resource profile.

In our controller assessments, we vary the workloads dynamically to see how well the controller adjusts. There are two requirements here:

- TR-1: The test infrastructure must scale well since a large number of tests must be run to provide adequate coverage of the diverse operating environments of the thread pool.
- TR-2: There must be a priori knowledge of the ideal controller performance in order to assess observed outcomes of test cases.

We begin with TR-1. In our initial design, tests executed on physical resources consuming real CPU, memory, and other resources. This resulted in long execution times and highly variable test results, both of which limited our ability to explore a large number of test cases.

Methodology challenge 2: Provide a test infrastructure that can efficiently execute a large number of test cases.

Approach 2: Use resource emulation.

By resource emulation, we mean that threads sleep for the time that they would have consumed the resource. This works well for active resources such as CPU, and it can be generalized to incorporate thrashing for memory by expanding nominal execution times based on memory over-commitment. In terms of controller assessments, it does not matter that resource consumption is emulated; the controller's logic is unchanged. However, resource emulation greatly reduces the load on test machines.

Using resource emulation allows us to increase the rate of test case execution by a factor of twenty. It also provides (although does not require) the ability to produce low-variance test results, a capability that is often needed to understand the effects of a change in controller design or parameter settings. The ability of our test infrastructure to produce low variance results is evidenced in Figure 4. This figure displays the results of an open loop test in which the concurrency level changes from 5 to 50 over 7,500 seconds for a dynamic workload. Because of the low measurement variability, we can clearly see the effects of thrashing, such as the drop in throughput around time 2,000 as concurrency level is increased beyond 27. The increased efficiency and reduced variability of the test infrastructure meant that we could run a large number of test cases to obtain better coverage of controller operating environments.

TR-2 concerns our ability to assess the outcome of test cases. For the thread pool controller, this means knowing the ideal concurrency level u^*, which is the minimum concurrency level at which the maximum throughput is achieved. Clearly, u^* depends on the test case.

Methodology challenge 3: Construct test cases for which the ideal controller performance is known a priori to provide a way to assess observed controller performance.

Approach 3: Use a test case framework for which ideal

controller performance can be computed analytically.

Our approach is to provide a broadly parameterizable framework for test cases that requires the controller to resolve complex performance trade-offs (although we make no claim that such test cases are representative). We consider test cases for which a work item first waits in a FIFO queue to execute on a serially-accessed resource (which may have multiple instances). Then, the work item executes on a resource that is accessed in parallel without contention. Work items acquire a fraction of system memory before using the serial resource, and if memory is over-committed, then the nominal execution time of the serial resource is expanded by the memory over-commitment. Completed work items are immediately inserted back in the FIFO queue. An example of the serial resource is CPU, and examples of parallel resources are lightly loaded web services and disks.

Let M_i be the number of profile i work items that enter the thread pool, and so $M = \sum_i M_i$ is the total number of work items. Similarly, we use u_i to denote the concurrency level for the i-th workload, and so the total concurrency level $u = \sum_i u_i$. Let q_i denote the fraction of memory required by a work item in workload i. There are N instances of the serial resource. Let $X_{S,i}$ be the nominal execution time on the serial resource for a work item from workload i. If there are I workloads, then the expansion factor e is $Max\{1, q_1u_1 + \cdots + q_Iu_I\}$ (since q_iu_i is the fraction of memory requested by workload i work items in the active set). Thus, the actual execution time of workload i on the serial resource is $eX_{S,i}$. Work items execute in parallel for $X_{P,i}$ seconds.

It is easy to find a concurrency level u^* that maximizes throughput for a single workload. There are two cases. If $Mq \leq 1$, then $e = 1$ and so $u^* = M$. Now, consider $uq > 1$ and so $e > 1$. Clearly, we want u large enough so that we obtain the benefits of concurrent execution of serial and parallel resources, but we do not want u so large that work items wait for the serial resource since this increases execution times by a factor of e without an increase in throughput. So, we want the average flow out from the serial resources to equal the flow out from the parallel resources. This is achieved when $\frac{N}{u^*qX_S} = \frac{u^*-N}{X_P} \approx \frac{u^*}{X_P}$ (if $N << u^*$). Solving, we have:

$$u^* \approx \sqrt{\frac{rN}{q}}, \tag{2}$$

where $r = X_P/X_S$. This is easily extended to multiple workloads by having $q = \sum_i \frac{M_i}{M} q_i$, $X_S = \sum \frac{M_i}{M} X_{S,i}$, and $X_P = \sum \frac{M_i}{M} X_{P,i}$. For example, in Figure 4, there are two workloads during Region III (time 3,000 to 4,500) with $M_1 = 20$, $X_{S,1} = 0$, $X_{P,1} = 1000ms$, $M_2 = 40$, $q_2 = 0.04$, $X_{S,2} = 50ms$, $X_{P,2} = 950ms$. Equation (2) produces the estimate $u^* \approx 33$, which corresponds closely to concurrency level at which the maximum throughput occurs in Figure 4.

These analytic models allow us to assess choices for controller design and parameter settings by comparing observed controller performance of a test case with the controller's ideal performance for the test case.

4. TUNING

Our thread pool controller has approximately ten tuning parameters, many of which have significant impact on performance. Examples of tuning parameters are: (a) the control gain parameter g in Equation (1); (b) the significance level (p value) used in statistical tests to detect changes in throughputs; (c) the minimum number of observations that must be collected at a concurrency level before a statistical test is conducted; and (d) the size of the "random move" concurrency level made when exiting State 1 (and for exploratory moves). These tuning parameters interact in complex ways. For example, both the control gain and the significance level impact the trade-off between moving quickly in response to a change in throughputs and being robust to noise in throughput observations.

The goal of tuning is to find a few settings (values) of tuning parameters that result in good controller performance for many workloads. Clearly, tuning requires running a large number of tests cases, which in turn demands a scalable test infrastructure as addressed in Section 3. There is a second challenge as well:

Methodology challenge 4: Select tuning parameter settings that optimize multiple performance metrics.

Approach 4: Only consider tuning parameter settings on the optimal frontier of the space of performance metrics.

We have three performance metrics: throughput, number of threads, and standard deviation of number of threads. Using the test cases described in Section 3, we can normalize the throughput measured in a test case by dividing by the optimal throughput achievable for the test case (obtained from the analytic model developed in Section 3). And, we can compute the excess number of threads, which is the number of threads used in the test case that exceed u^*. We define the **optimal frontier** of tuning parameter settings in the three dimensional space of performance metrics to be the parameter settings for which no other setting has better values of all performance metrics. Figure 5 visualizes the results of performance tests for approximately 2,000 tuning parameter settings. Each circle indicates the values of performance metrics for a single setting of tuning parameters averaged over 64 resource profiles. The optimal frontier is indicated by the blue plus signs. Note that A is a setting of tuning parameters that does not lie on the optimal frontier. Settings B-E are on the optimal frontier, and represent different trade-offs between the performance metrics. For example, E has very few excess threads and low throughput, while C has high throughput and a large number of excess threads.

5. CONCLUSIONS

Control theory has the potential to provide substantial benefits in system design. However, using control theory in the real world requires a methodology for controller design, testing, and tuning.

Our experience with building a controller for the .NET thread pool motivated the development of a methodology for applying control theory to a broad range of systems. A central concern in controller design is providing extensibility, especially to in-

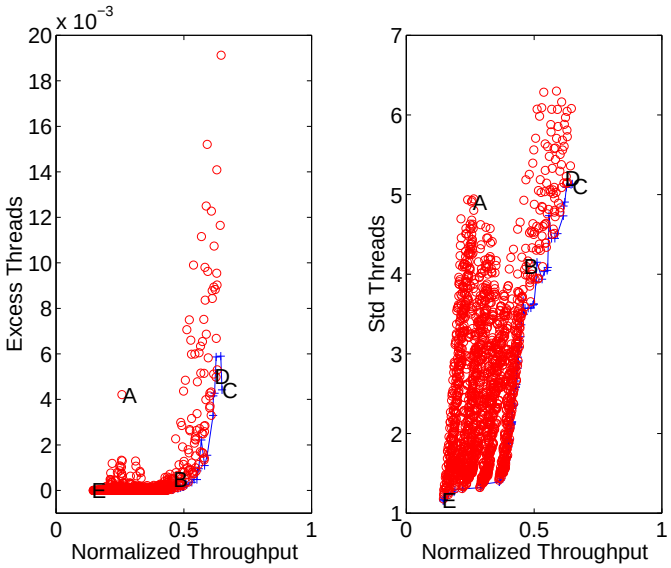

Figure 5: *Performance of settings of tuning parameters for three performance metrics. The blue plus signs indicate the optimal frontier.*

tegrate diverse control techniques. Our methodology addresses this by designing controllers as finite state machines. One concern in testing is providing a test infrastructure that scales well with the number of test cases. Our methodology addresses this by using resource emulation. Also in testing, we must construct test cases whose ideal outcomes are known a priori so that observed outcomes can be assessed. Our methodology addresses this by using a test case framework for which ideal test outcomes can be computed analytically. Last, tuning controller parameters requires selecting parameter settings that provide good results for multiple performance metrics. Our methodology addresses this by selecting tuning parameter settings that lie on the optimal frontier in the space of performance metrics.

Although our methodology was developed to address challenges in designing a thread pool on a single machine, we have found it to be valuable in designing controllers for large scale distributed systems. Consider the control challenges in Internet Data Centers such as those operated by Google, Microsoft, and Yahoo!. Typically, such systems consist of thousands of machines on which many jobs execute, where each job consists of tens to tens of thousands of tasks. One control problem is to assign tasks to machines in a way that maximizes throughput, minimizes response times, and abides by data center power constraints. This can be viewed as a multi-dimensional bin-packing problem in which the dimensions are task resource requirements (e.g., CPU, memory, network bandwidth), and the machines are multi-dimensional bins. Some parts of our methodology apply directly. For example, having a scalable test infrastructure is critical to evaluating the performance of the task assignment controller, something that we have addressed by using a variety of performance evaluation techniques. As described in Section 3, we use resource emulation to construct an efficient test infrastructure by employing models of individual machines instead of detailed simulations. The models are calibrated by measurements and/or detailed simulations. Also, as in the .NET controller, controllers for distributed systems have a number of tuning constants that must be selected. The parameter tuning techniques in Section 4 are applicable here, such as finding the optimal frontier in a multi-dimensional space of performance metrics (i.e., throughput, response time, power consumption). Other parts of are methodology have been extended. For example, a state representation of a distributed system scales poorly. We can scale better by constructing equivalence classes of machines and having the state space expressed in terms of these equivalence classes.

6. References

[1] T. Abdelzaher, Y. Diao, J. L. Hellerstein, C. Yu, and X. Zhu. Introduction to control theory and its application to computing systems. In Z. Liu and C. Xia, editors, *Performance Modeling and Engineering*, pages 185–216. Springer-Verlag, 2008.

[2] T. F. Abdelzaher, J. A. Stankovic, C. Lu, R. Zhang, and Y. Lu. Feedback performance control in software services. *IEEE Control Systems Magazine*, 23(3):74–90, 2003.

[3] M. Basseville and I. Nikiforov. *Detection of Abrupt Changes: Theory and Applications*. Prentice Hall, 1993.

[4] Y. Diao, J. L. Hellerstein, A. Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano. Using MIMO linear control for load balancing in computing systems. In *Proceedings of the American Control Conference*, pages 2045–2050, June 2004.

[5] C. V. Hollot, V. Misra, D. Towsley, and W. B. Gong. A control theoretic analysis of RED. In *Proceedings of IEEE INFOCOM*, pages 1510–1519, Anchorage, Alaska, Apr. 2001.

[6] S. Parekh, K. Rose, Y. Diao, V. Chang, J. L. Hellerstein, S. Lightstone, and M. Huras. Throttling utilities in the ibm db2 universal database server. In *Proceedings of the American Control Conference*, June 2004.

[7] S. Pratschner. *Common Language Runtime*. Microsoft Press, 1st edition, 2005.

[8] J. C. Spall. *Introduction to Stochastic Search and Optimization*. Wiley-Interscience, 1st edition, 2003.

[9] X. Wang, D. Jia, C. Lu, and X. Koutsoukos. Deucon: Decentralized end-to-end utilization control for distributed real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(7):996–1009, 2007.

[10] C.-Z. Xu and B. Liu. Model predictive feedback control for qos assurance in webservers. *IEEE Computer*, 41(3):66–72, 2008.

[11] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *Internation Conference on Distributed Computing Systems*, pages 301–310, 2002.