Optimizing Concurrency Levels in the .NET ThreadPool: A Case Study of Controller Design and Implementation

Joseph L. Hellerstein
Microsoft Developer Division
One Microsoft Way
Redmond, WA USA
joehe@microsoft.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

This paper presents a case study of developing a hill climbing concurrency controller (HC^3) for the .NET ThreadPool. The intent of the case study is to provide insight into software considerations for controller design, testing, and implementation. The case study is structured as a series of issues encountered and approaches taken to their resolution. Examples of issues and approaches include: (a) addressing the need to combine a hill climbing control law with rule-based techniques by the use of hybrid control; (b) increasing the efficiency and reducing the variability of the test environment by using resource emulation; and (c) effectively assessing design choices by using test scenarios for which the optimal concurrency level can be computed analytically and hence desired test results are known a priori. We believe that these issues and approaches have broad application to controllers for resource management of software systems.

1. INTRODUCTION

Over the last decade, many researchers have demonstrated the benefits of using control theory to engineer resource management solutions. Such benefits have been demonstrated for controlling quality of service in web servers [15], regulating administrative utilities in database servers [11], controlling utilizations in real time systems [14], and optimizing TCP/IP [7]. Despite these results and the availability of introductory control theory texts for computing practitioners (e.g., [6]), control theory is rarely used by software practitioners. We believe that one reason for this is that deploying closed loop systems for software products has a number of challenges related to software design, testing, and implementation that are not considered in existing research publications. This paper provides insights into these considerations through a case study of the development of a controller for optimizing concurrency levels in the Microsoft .NET Common Language Runtime (CLR) ThreadPool.

The problem of concurrency management occurs frequently in software systems. Examples include: determining the set of active jobs in virtual memory systems, selecting the set of active transactions in optimistic protocols for database locking, and determining the number of nodes enabled for transmission on a shared communications medium. Concurrency management deals with the trade-off between (a) increasing performance by having more activities happening concurrently and (b) reducing performance because of interference between concurrent activities.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FeBID Workshop 2008 Annapolis, MD USA

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

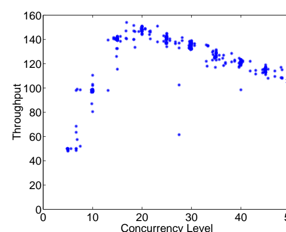


Figure 1: *Concurrency-throughput curve for a synthetic workload. Throughput degrades if the concurrency level exceeds 20 due to the overhead of context switching.*

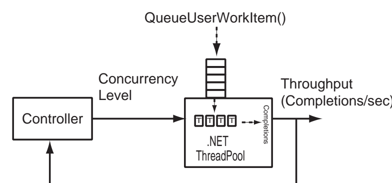
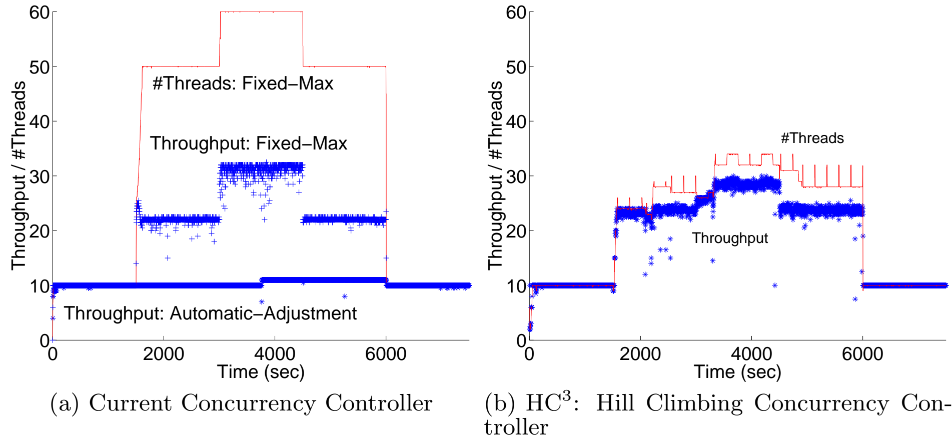


Figure 2: *Block diagram for controlling concurrency levels in the .NET ThreadPool.*

We use the term **active set** to refer to the collection of activities that take place concurrently, and we use the term **concurrency level** to refer to the size of the active set that is specified by the concurrency controller. To illustrate the trade-offs in concurrency management, consider the effect on throughput as we increase the concurrency level of executing threads for work that is 10ms of CPU time and 90ms of wait time on a dual processor computer. As shown in the concurrency-throughput curve in Figure 1, throughput initially increases with concurrency level since some threads in the active set use the CPU while others in the active set are waiting. However, when the concurrency level is too high, throughput decreases because threads in the active set interrupt one another causing context switching overheads. We use the term **thrashing** to refer to situations in which such interference occurs.



(a) Current Concurrency Controller

(b) HC^3: Hill Climbing Concurrency Controller

Figure 3: *Performance of Concurrency Controllers for the CLR ThreadPool for a Dynamic Workload*

Our focus is the CLR ThreadPool [12], a feature that is widely used in servers running the Windows Operating System. The ThreadPool exposes an interface called `QueueUserWorkItem()` whereby programmers place work into a queue for asynchronous execution. The ThreadPool assigns work items to threads up to the concurrency level specified by its control logic. Figure 2 depicts the closed loop system used by the ThreadPool to dynamically adjust the concurrency level to maximize throughput (measured in work item completions per second) with the secondary objective of minimizing the number of threads executing so to reduce overall resource consumption.

The .NET 3.5 ThreadPool concurrency controller (hereafter, **current ThreadPool controller**) is very effective for short-running, CPU-intensive workloads. This is because the current controller makes use of CPU utilizations in its decision logic. Unfortunately, this information is less useful (and maybe even counter-productive) for workloads that are not CPU-intensive. For example, Figure 3(a) plots the throughput of the current ThreadPool controller for a dynamically changing, non-CPU intensive workload. Two control policies are considered: automatic-adjustment, where the controller tries to maximize throughput, and fixed-max, where the controller maximizes the number of executing threads up to a fixed maximum (and only injecting new threads if the ThreadPool queue is non-empty). We see that automatic-adjustment has very low throughput. Fixed-max achieves much higher throughput, but also greatly increases the number of threads and hence increases memory contention.

These measurements motivate us to develop a new approach to concurrency management. This approach, the **hill climbing concurrency controller (HC^3)**, uses hill climbing to maximize throughputs to exploit the concave structure of the concurrency-throughput curve as illustrated in Figure 1. HC^3 differs from the current ThreadPool controller in another way as well—it does not make use of CPU utilizations. The rationale for this is that CPU is only one of many resources consumed by work items. Further, the relationships between resource utilizations, controller actions, and work item throughputs are complex since resources may be shared with non-ThreadPool threads.

There are two areas of work related to this paper. The first is software considerations for controller design, test, and implementation in software products. Unfortunately, there are few reports of software products built using control engineering. IBM's DB2 v8.2 uses regulatory control to manage background work [11]; IBM's DB2 v9.1 employs a control optimization technique to dynamically size buffer pools [5]; and Hewlett Packard's Global Workload Manager uses supervisory control to optimize performance for multi-tier applications [1]. But these papers focus almost exclusively on control laws and their assessments, not on software considerations for building closed loop systems. The ControlWare framework [16] describes middleware for building controllers, but it does not address controller design, testing, and implementation.

A second area of related work is control engineering for optimizing concurrency levels. An early example is [3], who uses dynamic programming to minimize thrashing in a virtual memory system based on information about virtual memory and database resources. More recently, [4] uses fuzzy control to optimize concurrency levels in a web server. Unfortunately, neither approach addresses our requirements in that the first uses knowledge of resource utilizations and the second converges slowly. Beyond this, there are well understood mathematical techniques for optimizing convex functions with stochastics [13], although these techniques are not prescriptive in that many engineering constants must be determined.

This paper presents a case study of developing a hill climbing concurrency controller (HC^3) for the .NET ThreadPool. Our purpose is to provide insight into controller design, testing, and implementation. While HC^3 contains many innovations, controller assessment is not the focus of this paper. Rather, this paper presents a series of issues encountered and approaches taken to their resolution. Figure 8 summarizes the case study. Examples of issues and approaches include: (a) addressing the need to combine a hill climbing control law with rule-based techniques by the use of hybrid control; (b) increasing the efficiency and reducing the variability of the test environment by using resource emulation; and (c) effectively assessing design choices by using test scenarios for which the optimal concurrency level can be computed analytically and hence desired test results are known a priori. We believe that these issues and approaches have broad ap-

plication to controllers for resource management of software systems.

The remainder of this paper is organized as follows. Section 2 discusses controller design, Section 3 addresses testing, and Section 4 details implementation considerations. Our conclusions and summary of the case study are contained in Section 5.

2. CONTROLLER DESIGN

The primary objective of the ThreadPool controller is to adjust the concurrency level (number of executing threads) to maximize throughput as measured by work item completions per second. However, there are a number of secondary objectives. First, if there are two concurrency levels that produce the maximum throughput, we prefer a smaller concurrency level to reduce memory contention. In addition the controller should have: (a) short settling times so that cumulative throughput is maximized, (b) minimal oscillations since changing control settings incurs overheads that reduce throughput, and (c) fast adaptation to changes in workloads and resource characteristics.

We assume that the concurrency-throughput curve is concave, as in Figure 1, and so our approach is based on hill climbing. We assume that time is discrete, and is indexed by k and m. k indexes the setting of concurrency level, and m indexes the throughput value collected at the same concurrency level. Let u_k be the concurrency level, and y_{km} be the measured throughput. Then, our system model is:

$$y_{km} = f(u_k, u_{k-1}, \cdots, y_{k,m-1}, \cdots, y_{k-1,m-j}, \cdots) + \epsilon_{km} \quad (1)$$

where f is concave and the ϵ_{km} are i.i.d. with mean 0 and variance σ_ϵ^2. (The i.i.d. assumption is reasonable within a modest range of concurrency levels.) We seek the optimal concurrency level u^* such that $\partial f / \partial u|_{u^*} = 0$. Unfortunately, f is unknown, f changes over time, and f cannot be measured directly because of the ϵ_k.

Stochastic gradient approximation using finite differences [13] provides a way to find u^* in Equation (1). The control law is

$$u_{k+1} = u_k + a_k g_k,$$

where $a_k = \frac{a}{(1+k+A)^\alpha}$, and

$$g_k = \frac{y(u_k + c_k) - y(u_k - c_k)}{2c_k}.$$

This control law requires choosing values for several engineering constants: a, α, A, and c_k. Even more problematic is that the concurrency level must be changed twice (i.e., $u_k - c_k$, $u_k + c_k$) before selecting a new concurrency level. Doing so, adds to variability, and slows settling times.

Our approach is to adapt the above control law in several ways: (a) reduce the number of changes in concurrency levels, (b) address the fact that concurrency level is a discrete actuator, and (c) exploit the convex nature of the concurrency-throughput curve. We use the control law:

$$u_{k+1} = u_k + sign(\Delta_{km}) \lceil a_{km} |\Delta_{km}| \rceil, \quad (2)$$

where $|x|$ is the absolute value of x and $\lceil x \rceil$ is the ceiling function (with $\lceil 0 \rceil = 1$). Note that the concurrency level changes by at least 1 from u_k to u_{k+1}. We estimate the derivative of f using:

$$\Delta_{km} = \frac{\bar{y}_{km} - \bar{y}_{k-1}}{u_k - u_{k-1}}, \quad (3)$$

where $\bar{y}_{km}$ is average throughput at concurrency level u_k after m measurements. (No second index is used for $\bar{y}_{k-1}$ since no measurements are being collected at previous concurrency levels.) Equation (3) avoids the problems of computing throughputs at two additional concurrency levels as in Spall's approach since the curve tangent is approximated by the line through throughputs at u_k and u_{k-1}. Further, observe that if we do not collect throughputs during the transient introduced by changing concurrency levels, then

$$E(\Delta_{km}) = E(\Delta_k) = \frac{f(u_k) - f(u_{k-1})}{u_k - u_{k-1}}.$$

Equation (2) contains the term a_{km}, which has the same form as Spall's a_k. We use $a = ge^{-s_{km}}$, $\alpha = 0.5$, and $A = 0$. s_{km} is the standard deviation of the sample mean of the m throughput values collected at u_k, and $g > 0$ is the control gain. We include a term that decreases with the standard deviation of throughput so that the controller moves more slowly when throughput variance is large. Thus,

$$a_{km} = e^{-s_{km}} \frac{g}{\sqrt{k+1}}, \quad (4)$$

Observe that a_{km} converges to 0 as m becomes large since s_{km} converges to 0. Further, $a_{km} \to 0$ as k becomes large.

While the approach described above resolves several issues with using stochastic gradient approximation, this approach deviates from Spall's assumptions and hence his convergence results no longer apply. We address this by combining Equation (2) with a set of rules. But this raises the following issue and motivates the approach taken:

Issue 1: How do we combine hill climbing with rules?

Approach 1: Use hybrid control.

Hybrid control [8] allows us to combine the control law in Equation (2) with rules to ensure convergence to u^* and adaptations to changes in f. We refer to this as HC^3, the **hill climbing concurrency controller**. Figure 4 displays the states used in HC^3, and Figure 5 describes the transitions between these states. HC^3 estimates the slope of the concurrency-throughput curve based on two points. State 1

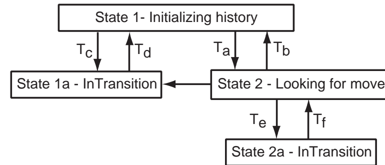


Figure 4: *HC^3 state diagram.*

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

Figure 5: *Description of HC^3 state transitions in Figure 4.*

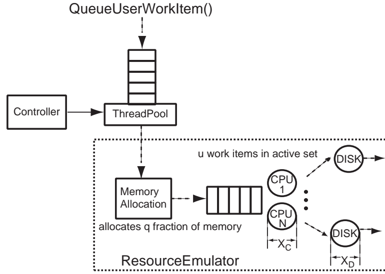


Figure 6: *Test scenario used in controller evaluation studies.*

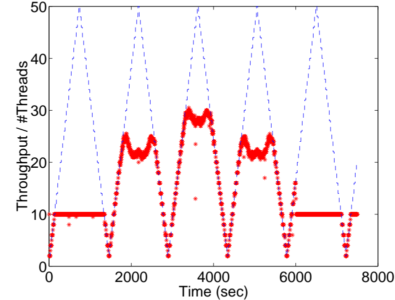


Figure 7: *Throughput (circles) at control settings specified by a cyclic ramp (line).*

collects data to estimate $f(u_{k-1})$, and state 2 does the same for $f(u_k)$. In addition, states 1a and 2a are used to address the dynamics of changing concurrency levels. Considerations for changes in the currency-throughput curve are addressed in part by transition T_b, which is described in more detail in Section 4.

The core of HC^3 is the set of rules associated with transition T_e:

- $R_{e,1}$: If $\bar{y}_{k-1}$ is significantly less than $\bar{y}_{km}$, then apply Equation (2).
- $R_{e,2}$: If $\bar{y}_{k-1}$ is significantly greater than $\bar{y}_{km}$, then $u_{k+1} = u_{k-1}$.
- $R_{e,3}$: If $\bar{y}_{k-1}$ is statistically identical to $\bar{y}_{km}$, sufficient data have been collected, and $u_{k-1} < u_k$, then $u_{k+1} = u_{k-1}$ (to minimize the number of threads).
- $R_{e,4}$: If the controller is "stuck in State 2", then make an exploratory move.

Note that the term "significantly" refers to the use of a statistical test to detect differences in population means as in [9].

We mention in passing that hybrid control lends itself to proving various properties such as convergence (see [8]). Unfortunately, space limitations preclude providing proof details for HC^3.

We address one further issue:

Issue 2: How do we obtain values of the engineering constants?

These constants are: the control gain g in Equation (4), the statistical significance level (which is used in the statistical tests transitions T_b and T_e), and the constant used in $R_{e,4}$ (to detect "stuck in state"). We resolve this by:

Approach 2: Use a test environment to evaluate engineering constants for a large number of scenarios.

Based on experiments conducted using the test environment described in Section 3, we determined the following: $g = 5$, a significance level of 0.01, and a threshold of 20 for "stuck in state." Figure 3(b) plots HC^3 performance for the same dynamic workload applied to the current ThreadPool controller in Figure 3(a). We see that HC^3 produces much larger throughputs than the current algorithm using the automatic-adjustment policy. Further, HC^3 achieves throughputs comparable to (and sometimes greater than) the current algorithm with the fixed-max policy, and HC^3 uses many fewer threads.

3. TEST ENVIRONMENT

This section describes the test environment used to address a number of considerations for which theory cannot be applied. These considerations include: determining the engineering constants identified in Section 2, debugging the controller implementation, and doing comparisons of controller designs and implementations.

Our focus is on *unit testing*. Unit tests are conducted on specific components or features early in the development cycle to eliminate logical errors and to assess performance characteristics of the component under test. We use unit tests to focus on controller robustness to "corner cases" that occur infrequently but can lead to very poor performance or even instabilities. In contrast, *system test*, which we do not address, is conducted on a complete product (e.g., the Windows Operating System) late in the development cycle, and emphasizes representative customer scenarios.

The approach taken for unit test of the ThreadPool is to generate synthetic work items according to a workload profile. The profile describes the type and amount of resources to consume, such as CPU, memory, web services. Resources such as CPU and memory are of particular interest since excessive utilizations of these resources leads to thrashing, and optimizing concurrency levels in the presence of thrashing is a particular challenge for controllers. We use the term **workload** to refer to a set of work items with the same work profile. In our controller assessments, we vary the workloads dynamically to see how quickly the controller finds the optimal concurrency level.

There are two parts to our test environment—the test harness and the test scenarios. The **test harness** is the infrastructure for running tests, which encompasses generating synthetic work and reporting measurement results. The test harness should produce efficient, low variance throughput measurement to facilitate comparisons between a large number of design alternatives.

In our initial design, the test harness executed synthetic workloads on physical resources on the test machine and so consumed CPU, memory, and other resources on these machines. This resulted in long execution times and highly variable test results, both of which limited our ability to explore a large number of scenarios.

Issue 3: How can we do efficient, low variance testing?

Approach 3: Use resource emulation.

By resource emulation, we mean that threads sleep for the time that they would have consumed the resource. The con-

troller does not know that resource consumption is emulated, and so its control logic is unchanged. However, resource emulation greatly reduces the load on the test machine, which allows us to scale greatly the test scenarios.

Resource emulation is provided by the `ResourceEmulator` that exposes an interface to create resource types (e.g., CPU, memory, and disk) and to consume instances of resource types. A resource may be active or passive. Active resources perform work, like CPU executions and network transfers. Passive resources, such as memory and database locks, are required in order to use an active resource. The emulation includes thrashing resulting resulting from contention for passive resources. Thrashing is incorporated as expansion factors for active resource execution times.

Using resource emulation improved the efficiency of the test harness by a factor of twenty, and it almost eliminated measurement variability. To see the latter, Figure 7 displays results of an open loop test in which concurrency level is varied 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 using resource emulation meant that we could run a large number of scenarios in our evaluations of engineering constants and candidate algorithms.

The second part of the test environment is the scenarios executed on the test harness. These scenarios are described in terms of when workloads enter and leave and the synthetic resources that they consume. An important consideration is the issue below:

Issue 4: How do we know the optimal concurrency level for a scenario?

Approach 4: Construct scenarios for which the optimal concurrency level can be computed analytically so that expected controller performance is known a priori.

One such scenario is the widely-used central server model (e.g., [10]) that describes work flows in operating systems. Figure 6 depicts the specifics for our problem. Let M_i be the number of profile i work items that enter the ThreadPool, and so $M = \sum_i M_i$ is the total number of work items. Once the concurrency level permits, work items from profile i enter the active set and acquire a fraction q_i of available memory. In the sequel, the concurrency level u is equal to the size of the active set since actuator dead time is irrelevant for this optimization problem. There are N synthetic CPUs. Let $X_{S,i}$ be the nominal CPU execution time of a work item from workload i. Its actual execution time is expanded by the overcommitment of memory. That is, if there are I workloads and u_i is the number of profile i work items in the active set, then the expansion factor $e = Max\{1, q_1 u_1/u + \cdots + q_I u_I/u\}$, where $u = u_1 + \cdots + u_I$. So, the actual CPU execution time of a profile i work item is $eX_{S,i}$. Work items consume a synthetic disk for $X_{P,i}$ seconds. We use synthetic disks to model requirements for external resources whose service times do not depend on local resources (e.g., web service accesses). As such, memory contention does not affect execution times for synthetic disks.

The simplicity of the central server model makes it easy to to develop an analytic solution for the optimal concurrency level, under certain simplifying assumptions. Consider a single workload and deterministic execution times. There are two cases. If $Mq \leq 1$, then $e = 1$ and so $u^* = M$. Now, consider $uq > 1$ and for all concurrency levels under consideration and so $e > 1$. Clearly, we want u large enough so that we obtain the benefits of concurrent execution of CPU and disk resources, but we do not want u so large that work items wait for CPU since this increases execution times by a factor of e without an increase in disk throughput. Since execution times are deterministic, no queue forms at the CPUs as long as the flow out from the disks equals the flow out from the CPUs. That is, optimal throughput is achieved when $\frac{N}{u^* q X_S} = \frac{u^*-1}{X_P} \approx \frac{u^*}{X_P}$ Solving, we have:

$$u^* \approx \sqrt{\frac{rN}{q}}, \tag{5}$$

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 7, 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 (5) produces the estimate $u^* = 33$, which corresponds closely to concurrency level at which peak throughput occurs in Figure 7.

To summarize our insights on test scenarios, using the central server model allows us to assess controller performance based on the scenario's (analytically computed) optimal concurrency levels. Further, the fact that central server scenarios have a simple parameterization (M_i, q_i, $X_{S,i}$, and $X_{P,i}$) provides a way to construct a test matrix that systematically explores design alternatives.

4. CONTROLLER IMPLEMENTATION

This section describes key implementation considerations in HC[3]. Among these considerations are the structure of the controller code and techniques for managing the variance of measured throughputs.

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 4, implementation of the "if" part of transition rules such as $R_{e,1}, \cdots, R_{e,4}$, and implementation of the "then" part of transition rules (e.g., Equation (2)).

The effectiveness of hill climbing with stochastics is largely determined by the variability of the estimates of the throughputs at each concurrency level.

Issue 5: How can the effects of throughput variance be minimized?

This issue is addressed in two ways. The first is:

Approach 5a: Do not collect throughputs during concurrency transitions.

To elaborate, one source of throughput variability is changes in the concurrency level as a result of controller actions. We manage this by including states 1a and 2a in Figure 4. 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

#	Area	Issue	Approach
1	Design	How combine hill climbing and rules?	Use hybrid control.
2	Design	How obtain values of engineering constants?	Use test environment to run many scenarios.
3	Test	How do efficient, low-variance testing?	Use resource emulation in the test harness.
4	Test	How know optimal concurrency level?	Use scenarios with analytic solutions.
5	Implement	How minimize effects of variance?	(a) Do not collect throughputs during transitions. (b) Discard dissimilar throughputs.

Figure 8: *Summary of issues and approaches to their resolution used in the development of HC*^{3}.

threads equals the controller specified concurrency level; or (2) the observed concurrency level is less than the number of threads, and the ThreadPool queue is empty.

The second approach to Issue 5 is:

Approach 5b: Discard dissimilar throughput observations.

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 4 detects these situations by using change point detection [2]. 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.

5. CONCLUSIONS

This paper aids in making control engineering more accessible to software practitioners by addressing software issues in controller design, testing, and implementation. This is done through a case study of a controller for optimizing concurrency levels in the .NET ThreadPool.

We structure the case study in terms of issues encountered and approaches taken to their resolution. Figure 8 summarizes the results. We believe that many of the issues and approaches have broad application. For example, Issue 1 concerns how to systematically combine formal control laws with engineering-based insights expressed as rules. Our approach, using hybrid control, is a very general solution that is easily implemented in software. Issue 3 also has broad application. This issue concerns the development of an efficient, low variance test environment, a common challenge in controller development for software systems. Our approach uses resource emulation, which improves efficiency by a factor of twenty in our experiments. Issue 4, knowing the desired outcome of a performance test, is a broad concern in assessing resource management solutions of software systems. Our approach, using scenarios for which analytic solutions can be obtained, improves the effectiveness of testing and provides a systematic approach to test case construction based on the parameters of the analytic model.

In terms of future work, we are exploring broader applications of HC^3, such as to load balancing and configuration optimization. These potential applications in turn motivate some new directions with HC^3—extending it to multiple input, multiple output control.

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

[3] R. Blake. Optimal control of thrashing. In *SIGMETRICS Performance Evaluation Review*, volume 11, pages 1–10, 1982.

[4] Y. Diao, J. L. Hellerstein, and S. Parekh. Optimizing quality of service using fuzzy control. In *Distributed Systems Operations and Management*, pages 42–53, 2002.

[5] 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.

[6] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[7] 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.

[8] E. A. Lee and P. Varaiya. *Signals and Systems*. Addison Wesley, 1st edition, 2003.

[9] B. W. Lindgren. *Statistical Theory*. The MacMillian Company, 4th edition, 1968.

[10] D. A. Menasce, V. A. Almeida, and L. W. Dowdy. *Capacity Planning and Performance Modeling*. Prentice Hall, 1994.

[11] 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.

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

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

[14] 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.

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

[16] 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.