Configuring Resource Managers Using Model Fuzzing: A Case Study of the .NET Thread Pool

Joseph L. Hellerstein
Microsoft Developer Division, Redmond, Washington USA
email: joehe@microsoft.com

***Abstract*—Resource managers (RMs) often expose configuration parameters that have a significant impact on the performance of the systems they manage. Configuring RMs is challenging because it requires accurate estimates of performance for a large number of configuration settings and many workloads, which scales poorly if configuration assessment requires running performance benchmarks. We propose an approach to evaluating RM configurations called model fuzzing that combines measurement and simple models to provide accurate and scalable configuration evaluation. Based on model fuzzing, we develop a methodology for configuring RMs that considers multiple evaluation criteria (e.g., high throughput, low number of threads). Applying this methodology to the .NET thread pool, we find a configuration that increases throughput by 240% compared with the throughput of a poorly chosen configuration. Using model fuzzing reduces the computational requirements to configure the .NET thread pool from machine-years to machine-hours.**

I. Introduction

Resource managers (RMs) abound in computing solutions. Load balancers allocate requests to servers in clusters. Database memory managers control memory allocations in database servers. And, operating systems have resource managers for CPU, memory, and user threads.

In almost all cases, RMs expose configuration parameters that significantly impact performance. Some parameters are externally visible, such as the size of memory pools for database servers or the Keep-Alive timeout in the Apache Web Server. Other parameters are internal to the resource manager and are configured "at the factory" to deliver good performance for a large number of customers. For example, a recent version of the Microsoft .NET thread pool contains about a dozen internal configuration parameters that allow Microsoft developers to adapt the behavior of the thread pool.

Configuring RMs is complex and time consuming. In large part, these difficulties arise because of the scale of configuration evaluation. For example, in the studies of the .NET thread pool presented in this paper, 1,800 configurations are evaluated for 64 system variants, which is a total of $1,800 \times 64 = 115,200$ different studies. Clearly, it is impossible to collect this volume of data using performance benchmarks that run for minutes or hours. On the other hand, the complexity of and interdependencies between configuration parameters make it difficult to construct accurate models that relate configurations directly to performance of the managed system.

Our approach to scaling configuration evaluation decomposes the problem into two parts: (1) relate RM configurations to resource allocations and (2) relate resource allocations to performance. Although (1) typically involves substantial complexities that can only be captured through measurement, (2) can sometimes be estimated accurately using a simple model of the managed system (referred to as a **system model**). For example, in the .NET thread pool, resources are threads that determine the concurrency level, and performance is measured in terms of throughput (work item completions per second). So, the system model can be expressed using the concurrency-throughput curve, a unimodal function that is readily expressed by simple equations.

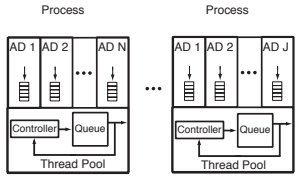


Figure 1. *Thread Pool in .NET System. AD is Application Domain, a .NET mechanism for in-process isolation.*

We introduce model fuzzing, a technique that finds good performing configurations in a computationally efficient way by combining measurements of RM resource allocations with a model of the managed system. The term "fuzzing" originates from software testing [14] in which test inputs are modified to assess the robustness of components. Model fuzzing means that the parameters of the system model are changed to characterize many variants of the managed system. Applying model fuzzing to the .NET thread pool allows us to find a configuration that increases throughput by 240% compared to a poorly chosen configuration, and reduces the time to do configuration from machine-years to machine-hours.

The remainder of the paper is organized as follows. Section 2 describes the .NET thread pool. Section 3 details model fuzzing. Section 4 presents our configuration methodology, and Section 5 applies the methodology to the .NET thread pool. Section 6 discusses related work. Our conclusions are contained in Section 7.

II. .NET Thread Pool

Microsoft .NET consists of an object-oriented framework of re-usable types and a Common Language Runtime (CLR) that provides runtime services such as in-process application isolation, automatic memory management, and managing application concurrency levels through the .NET thread pool.

978-1-4244-3487-9/09/$25.00 © 2009 IEEE

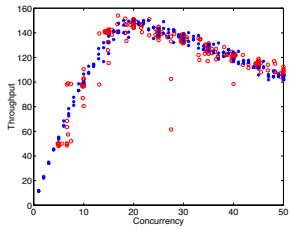


Figure 2. *Concurrency-throughput curve for a CPU intensive workload (red, open circles) and a model fitting for this curve (blue asterisks). The system model uses Equation (1) with* $\mu_1 = 20,\ \mu_2 = 150,\ \mu_3 = 1.6,\ \mu_4 = 0.55,\ \mu_5 = 80,\ \mu_6 = 0.03$.

This section briefly describes a version of the .NET thread pool that is used as a case study in this paper.

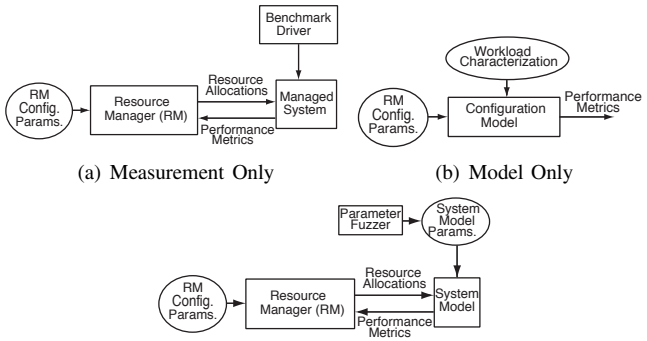
Figure 1 depicts the environment in which the .NET thread pool operates. A process may have multiple Application Domains (ADs), an architectural construct that provides isolation of services between threads within an address space. One service provided to Application Domains is **`ThreadPool.QueueUserWorkItem()`** whereby work items (specified by method name) are submitted for asynchronous execution. Arriving work items are placed in the thread pool queue. A work item begins execution when a thread is available. The number of threads or **concurrency level** is determined by the **thread pool controller,** which attempts to choose a concurrency level that maximizes throughput. Throughput is measured as the number of work item completions per second. Each process has its own thread pool with its own Application Domains. Processes compete for CPU and memory resources. Thus, the runtime behavior of the thread pool is much like a distributed system, especially if there are multiple processors with non-uniform memory access times.

We provide some details of the thread pool controller used as a case study in this paper. The controller exploits a common characteristic of the relationship between concurrency level and throughput (hereafter, **concurrency-throughput curve**). This characteristic is evident in Figure 2 in which the open circles plot concurrency levels and throughputs obtained from experiments with a CPU-intensive workload on a dual core, 2GHz machine. Note that the concurrency-throughput curve has a single peak, or is **unimodal**. The unimodal shape results from two competing effects. The first effect occurs as we move from very low to moderate concurrency levels—throughput increases as a result of increased parallelism. The second effect occurs as concurrency becomes too large—the curve flattens or decreases due to resource contention. The thread pool controller exploits the unimodal shape by using hill climbing to find concurrency levels that maximize throughput. However, hill climbing cannot proceed in a deterministic manner because of variability in the data.

These considerations motivate a thread pool controller that uses stochastic gradient approximation (SGA), a well known technique for optimizing unimodal functions with stochastics. Let x_k be the concurrency level at time k. SGA uses the control law $x_{k+1} = x_k + g_k D_k$. $D_k = \frac{v(x_k+c_k)-v(x_k-c_k)}{2c_k}$ is the discrete derivative at the k-th iteration of the controller. $v(x_k+c_k)$ is the throughput for a concurrency level of x_k+c_k, and c_k is a small increment. g_k is the control gain, which determines how aggressively the controller changes concurrency levels. The control law is designed so that eventually $D_k = 0$, at which point the concurrency level does not change. Further details on SGA can be found in [16].

Although the foregoing provides a sound theoretical foundation, many practical considerations motivate the need for several controller configuration parameters. First, SGA expresses control gain in terms of three other constants. In contrast, the thread pool controller simplifies matters by having a single **control gain** configuration parameter. A second consideration is that updating the concurrency level x_k at each control interval creates transients that complicate control. We address this by changing x_k only if its associated throughputs are significantly different than those obtained at the last setting of concurrency level. This decision is based on a statistical test that has one parameter—**significance level** [12], and so significance level is another configuration parameter. Third, there are several situations in which the controller needs to explore different concurrency levels, such as when workloads change. The magnitude of the change in concurrency level used in these explorations is determined by the configuration parameter **magnitude of the random move,** which has units of standard deviation of throughput (with a constant that converts from units of throughput to concurrency level). A fourth configuration parameter relates to workload characteristics, specifically the ratio of work item service times to the time between controller iterations. If this ratio is much greater than 1, then work items execute for many controller invocations, during which time the controller has no information about the impact of concurrency level on throughput. Thus, the controller includes a parameter that specifies the **minimum number of completions** of work items that must occur before a control action is taken. We use the term **configuration** to refer to a binding of values to configuration parameters. In total, there are a dozen configuration parameters.

How do we evaluate the quality of a configuration of the thread pool controller? First, the controller should produce high throughputs for performance benchmarks. Second, the controller should minimize the number of threads since threads consume memory and other resources. However, there is a subtlety here since a larger number of threads may be needed to achieve higher throughputs. Thus, the evaluation criteria is not the *absolute* number of threads, but the number of *excess threads*—those threads that do not increase throughput. Last, a configuration should minimize the number of times that there is a change in concurrency level due to the CPU and memory overheads incurred by making these changes. Unfortunately, in general it is difficult to simultaneously optimize all three evaluation criteria, and so trade-offs between evaluation criteria must be made.



(a) Measurement Only (b) Model Only

(c) Measurement + Model Fuzzing

Figure 3. *Approaches to configuration evaluation.*

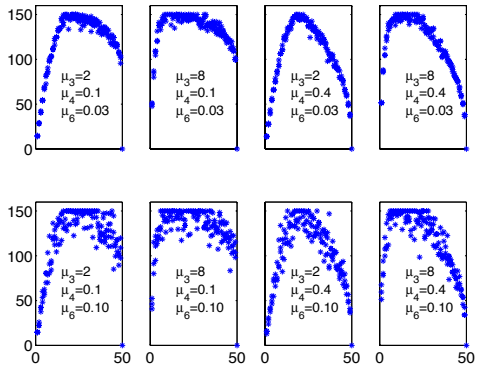


Figure 4. *Concurrency-throughput curves for fuzzed models of the system in Figure 2*

III. Model Fuzzing

Configuring RMs requires understanding the performance characteristics of a large number of interdependent configuration parameters. This section introduces model fuzzing, a technique that combines measurement and modeling to achieve computationally efficient and accurate estimates of the performance of RM configurations.

Figure 3(a) displays a measurement-only approach to evaluating RM configurations. In this approach, the RM and Managed System are installed on one or more computers, and then performance benchmarks are run for many RM configurations. During the benchmark, the Managed System provides the RM with performance metrics such as throughputs, and the RM returns resource allocations such as concurrency level.

This approach scales poorly. Typically, it takes minutes or hours to run a performance benchmark, and the number of configurations to evaluate is quite large. For example, if each benchmark executes for one minute, it takes 80 days to evaluate the 1,800 thread pool configurations and 64 variants of the managed system considered in Section 4. If each benchmark executes for an hour (which is more realistic than one minute), it takes 10 years! While considerable speedup can be achieved by parallelizing these benchmarks, the 1,800 configurations only address four of the twelve configuration parameters of the thread pool controller.

Figure 3(b) displays a more efficient approach to evaluating RM configurations. Here, a model is constructed that relates RM configuration and workload characteristics to performance of the Managed System. Such models typically consist of simple equations, and so the approach scales well.

The challenge with the second approach is constructing a sufficiently accurate performance model. Some configuration parameters such as buffer pool sizes are readily modelled using queuing theory [11] and other techniques. However, performance models are very difficult to construct for parameters such as control gain, significance level, magnitude of the random move, and minimum number of completions.

Our observation is that even if we cannot accurately model the relationship between RM configurations and performance of the managed system, we can achieve a considerable increase in computational efficiency by making use of a **system model** that relates RM resource allocation to performance of the managed system. In particular for the .NET thread pool, the system model is a unimodal concurrency-throughput curve. This model is used to conduct numerical studies with the RM implementation to efficiently and accurately evaluate the performance of RM configurations.

Figure 3(c) depicts how we make use of the system model to provide efficient and accurate evaluation of configuration parameters. There are two requirements. First, the RM must be structured so that it can operate stand alone. By this, we mean that there is a well defined interface whereby the RM inputs performance metrics, and the RM outputs resource allocations. Note that the system model inputs resource allocations and outputs performance metrics. Thus, we can readily construct a test harness in which the RM implementation interacts with the system model to efficiently and accurately produce performance metrics that evaluate RM configurations.

The second requirement for implementing the approach depicted in Figure 3(c) is that the system model externalize parameters that can be manipulated to characterize many variants of the managed systems. We refer to this manipulation as **model fuzzing,** a phrase inspired by fuzz testing [14]. Fuzzing provides a way to assess the robustness of the performance of configurations.

We now construct a system model for the .NET thread pool. This is a parameterization of concurrency-throughput curves such as Figure 2. We parameterize the following characteristics: (a) the optimal concurrency level, denoted by μ_1, which is the concurrency level at which throughput is maximized; (b) the maximum throughput μ_2; (c) the number of work items μ_5; (d) the shape of the curve as it changes in a non-decreasing way from 1 to μ_1, which is parameterized by μ_3; (e) the shape of the curve as it changes in a non-increasing way from μ_1 to μ_5, which is parameterized by μ_4; and (f) the variability of measured throughputs, which is described by the parameter μ_6. There are several constraints on these parameters: (i) all parameters are non-negative; (ii) $\mu_2, \mu_5 > 0$; and (iii) $\mu_1 < \mu_5$. More insights into the semantics of these parameters are provided below.

We structure the system model for the .NET thread pool as

two sub-models to address deterministic and random effects. The deterministic sub-model is expressed as a function f that depends on the concurrency level and the μ_i. We use the notation $f(x_k;\mu)$ with $\mu = (\mu_1, \cdots, \mu_6)$ where k indexes time. There are two cases. If the concurrency level x_k is in the pre-peak region of the concurrency-throughput curve, then:

$$f_a(x_k;\mu) \quad = \quad \mu_2 \left(1 - \left(1 - \frac{x_k}{\mu_1}\right)^{\mu_3}\right)$$

and if x_k is in the post-peak region, then:

$$f_b(x_k;\mu) = \mu_2 \left(1 - \frac{x_k - \mu_1}{\mu_5 - \mu_1}\right)^{\mu_4}$$

Combining these, we have:

$$f(x_k;\mu) = \begin{cases} f_a(x_k;\mu) & 0 \leq x_k \leq \mu_1 \\ f_b(x_k;\mu) & \mu_1 \leq x_k \leq \mu_5 \end{cases}$$

A few observations about this model. First, f is continuous since f_a and f_b are continuous, and $f_a(\mu_1;\mu) = f_b(\mu_1;\mu)$. Also, if $\mu_3 = 0$, then $f_a(x_k;\mu) = \mu_2$. If $\mu_3 = 1$, then $f_a(x_k\mu)$ is linear. In the latter case, the slope is determined by the fact that (i) throughput is 0 if $x_k = 0$ and (ii) throughput is μ_2 if $x_k = \mu_1$. Similar observations apply to μ_4 and $f_b(x_k;\mu)$ (although the slope in the linear case also depends on μ_5). Note further that $f(x_k;\mu)$ is unimodal since with the constraints imposed on the μ_i, $\frac{df_a}{dx_k} \geq 0$ and $\frac{df_b}{dx_k} \leq 0$, with a peak at $x_k = \mu_1$. Last, $f(x_k;\mu)$ is concave (i.e., the second derivative with respect to x_k is non-increasing) if $\mu_3 > 1$ and $0 < \mu_4 < 1$ (although some care is required since f may not be differentiable at $x_k = \mu_1$).

The second sub-model takes into account randomness. A central observation here is that in computing systems, the variance of a measurement is often a function of its mean value (e.g., [11]). Let ϵ_k be independent random variables with mean zero and standard deviation $f(x_k;\mu)\mu_6$. Then, the throughput estimate is $\hat{v}_k$, where:

$$\hat{v}_k = Min(\mu_2,\ Max(f(x_k;\mu) + \epsilon_k,\ 0)) \qquad (1)$$

The μ_i can be estimated in a straight-forward way from empirical data consisting of (x_k, v_k, M_k), where x_k is the concurrency level for the k-th measurement, v_k is the measured throughput, and M_k is the number of work items in the thread pool. Parameters μ_1, μ_2, μ_5 are readily obtained from these data (possibly using simple statistical tests identify the maximum of several sample means as in [16]). To estimate μ_3, we use least squares regression and consider only those data for which $x_k < \mu_1, v_k < \mu_2$ and so $v_k = f_a(x_k;\mu) + \epsilon_k$. Hence,

$$\begin{aligned} 1 - \tfrac{v_k}{\mu_2} \quad &= \quad \left(1 - \tfrac{x_k}{\mu_1}\right)^{\mu_3} + \epsilon'_k \\ ln(1 - \tfrac{v_k}{\mu_2}) \quad &= \quad ln\left(1 - \tfrac{x_k}{\mu_1}\right)\mu_3 + \epsilon''_k, \end{aligned}$$

where $\epsilon'_k, \epsilon''_k$ are random variables with distributions determined by transforming the ϵ_k. Thus, we have an expression that is linear in μ_3, and so μ_3 can be estimated using linear least squares regression. Note that the constraints imposed ensure positive arguments for ln. Estimating μ_4 is done in a similar way. To estimate μ_6, we compute $w_k = \frac{v_k}{f(x_k;\mu)}$. Under the assumptions of the system model, the w_k are independent and identically distributed with variance μ_6^2.

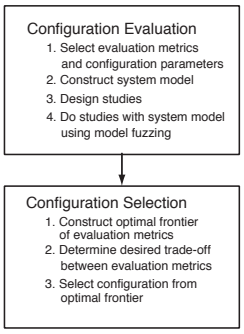


Figure 5. *Methodology for configuring resource managers.*

To assess the accuracy of the system model in Equation (1) and the effectiveness of the above procedure for estimating the parameters of this model, we apply the foregoing to measured values in Figure 2 (which are indicated by the open circles). The model estimates $\hat{v}_k$ are indicated by the asterisks. Observe, that there is a close correspondence between the model estimates and the measured values.

To gain insight into the variants of a managed system that can be characterized by model fuzzing, we vary the μ_i obtained in fitting Figure 2. Figure 4 displays the results of changing μ_i for $i \in \{3, 4, 6\}$ (pre-peak shape, post-peak shape, and variability). We see that even this limited, model fuzzing produces a wide range of concurrency-throughput curves.

IV. Configuration Methodology

This section describes our methodology for RM configuration using model fuzzing. The section focuses on the principles and algorithms. Section 5 provides a case study of applying the methodology to the .NET thread pool.

Figure 5 displays the steps in the methodology. There are two phases: (a) configuration evaluation and (b) configuration selection. Configuration evaluation obtains the data used to evaluate configurations. The challenges with configuration evaluation are scaling and accuracy. Model fuzzing plays a central role in addressing these challenges.

First, some notation. We denote the j-th configuration parameter for the m-th configuration by $p_{m,j}$, and so the m-th configuration is defined as $\mathbf{p}_m = (p_{m,1}, \cdots, p_{m,J})$. We consider multiple evaluation metrics for configurations. For example, in the .NET thread pool, we evaluate configurations based on throughput, number of excess threads, and variability of the number of threads. We denote the n-th evaluation metric for the m-th configuration by $y_{m,n}$ and $\mathbf{y}_m = (y_{m,1}, \cdots, y_{m,n})$.

The first step in configuration evaluation is selecting the evaluation metrics. These should be chosen so that they can be computed from the performance metrics produced by the

Inputs

- Results of numerical studies $S = \{(\mathbf{p}_m, \mathbf{y}_m)\}$

Steps

1) $F = \emptyset$
2) For each $(\mathbf{p}_m, \mathbf{y}_m) \in S$
   a) If there is $(\mathbf{p}, \mathbf{y})$ such that $\mathbf{p} \in F$ and $\mathbf{y} \prec \mathbf{y}_m$, then $F = F - \{\mathbf{p}\}$.
   b) If there is no $(\mathbf{p}, \mathbf{y})$ such that $\mathbf{p} \in F$ and $\mathbf{y}_m \prec \mathbf{y}$, then $F = F \cup \{\mathbf{p}_m\}$.

Figure 6. *Algorithm for finding configurations on the optimal frontier.*

system model and/or the RM resource allocations. Next, a system model is constructed. Following this we determine how to change the parameters of the system model to study many variants of the managed system. Last, as described in Section 3, model fuzzing is used to produce evaluation metrics for a wide range of configurations.

The second phase of our methodology is configuration selection. A core challenge here is dealing with multiple evaluation metrics. For example, in the .NET thread pool, some configurations produce high throughputs but they also result in a large number of excess threads and/or high variability of the number of threads. Other configurations minimize the number of excess threads, but they also result in low throughputs.

A challenge with having multiple evaluation criteria for the configurations is that in general there is no "best" configuration. However, there is a subset of configurations that dominates the other configurations as determined by their vector of evaluation metrics. Specifically, the evaluation vector $\mathbf{y}_1$ dominates $\mathbf{y}_2$ if, and only if, no evaluation metric $y_{1,n}$ is worse than $y_{2,n}$, and there is an evaluation metric $y_{1,n'}$ that is preferred to $y_{2,n'}$. We denote this by $\mathbf{y}_2 \prec \mathbf{y}_1$. For example, if the metric is throughput, "preferred to" means "greater than"; however, if the metric is standard deviation of number of threads, "preferred to" means "less than".

With this notation, we define the **optimal frontier** of a set of configurations. Let P be the set of all configurations considered, and let $F \subseteq P$ be the optimal frontier. Then, $\mathbf{p}_1 \in F$ if, and only if there is no $\mathbf{p}_2 \in P$ such that $\mathbf{y}_1 \prec \mathbf{y}_2$, where $\mathbf{y}_i$is the vector of evaluation metrics for $\mathbf{p}_i$. Figure 6 displays a simple algorithm for computing the optimal frontier.

By focusing on the optimal frontier, we obtain the best trade-off between evaluation metrics. Thus, the next step in configuration selection is to determine what trade-offs are acceptable. For the .NET thread pool, these trade-offs are between throughput, number of excess threads, and variability in the number of threads. Based on where we want to be in the multi-dimensional space of evaluation metrics, a configuration is selected by searching for configurations whose evaluation metrics are close to the desired point in the space.

V. Thread Pool Case Study

This section presents a case study of applying the methodology described in Figure 5 to the .NET thread pool.

The first phase of our methodology is configuration evaluation, and the first step of this phase is to select evaluation metrics. These evaluation metrics should be computed from the performance metrics and resource allocations obtained from fuzzing studies.

For the .NET thread pool, we conduct studies using the system model in Equation (1) that result in a collection of concurrency levels and throughput estimates $(x_k, \hat{v}_k)$. Since Equation (1) specifies the maximum throughput μ_2, our first evaluation metric is **normalized throughput**, $y_1 = \frac{1}{K}\sum_k \frac{\hat{v}_k}{\mu_2}$. This metric takes on values in $[0, 1]$ with 1 being the ideal value. Our second metric is **number of excess threads**, $y_2 = max(0, x_k - \mu_1)$ with 0 being the ideal value. Last, we are concerned about the frequency with which the number of threads changes since this increases overheads, and so use the **standard deviation of the number of threads** $y_3 = \sqrt{\frac{\sum(x_k - \bar{x})^2}{K-1}}$, with 0 being the ideal value.

For step 2, we use the system model described in Equation (1). For step 3, we employ the design in Figure 7. The design is a fuzzing of a model fit to the performance benchmarks considered later in this section. The fitted model has pre-peak shape $\mu_3 = 1$ (i.e., linear), post-peak shape $\mu_4 = 0.3$, and variability $\mu_6 = 0.1$. Fuzzing is done to explore a range of settings of parameters of the system model for which the concurrency-throughput curve is concave.

The fourth step is to do evaluation studies. There are a total of 64 different parameterizations of the system model and 1,800 different RM configurations. Hence, there are $1,800 \times 64 = 115,200$ studies. Each is repeated ten times, and the mean value is reported. So, there are a total of 1,152,000 runs. This takes approximately 10 hours on a dual core 2GHz machine. If instead of model fuzzing, we used a measurement-only approach as in Figure 3(a) and each benchmark only ran for one minute (a very optimistic assumption), producing the evaluation data would take in excess of two years. Although the measurement-only approach can be scaled by running benchmarks in parallel, this is inadequate to address the computational demands as we scale our studies from four configuration parameters to one dozen configuration parameters.

The next phase in our methodology is configuration selection. The approach outlined in Section 4 involves some complexity to deal with multiple evaluation metrics and interactions between configuration parameters. We begin by considering a simpler, ad hoc alternative to motivate why this complexity is required.

The simpler, ad hoc approach selects each configuration parameter in isolation. To this end, Figure 8 plots the effect of the value of each configuration parameter on the evaluation metrics. The figure is a matrix of plots. The upper-left plot displays how values of control gain affect normalized throughput. The horizontal axis is control gain, and the vertical axis is normalized throughput. The error bars indicate the standard deviation of the evaluation metric across all studies. The other plots are organized in a similar way with rows in the figure plotting the same configuration parameter and columns

Parameter	Initial Value	Value Increment	Number of Increments
Pre-peak shape (μ_3)	1.0	0.5	4
Post-peak shape (μ_4)	0.1	0.1	4
Variability (μ_6)	0.1	0.1	4
Control gain	1.0	1.0	10
Signif. level	1.0	2.0	5
Mag. of move	0.5	0.5	6
Min. num. of completion	0.0	2.0	6

Figure 7. *Design of experiments for model fuzzing.*

plotting the same evaluation metric.

As an alternative to configuration selection in Figure 5, we use an ad hoc approach based on Figure 8 to select a configuration. The first configuration we consider, which is referred to as Configuration A, is intended to maximize throughput. Configuration A is constructed by considering each configuration parameter in isolation. We see that normalized throughput increases with control gain, and so the control gain of Configuration A is set to 10. Significance level does not seem to impact normalized throughput as much, but a slightly larger value seems desirable. So, we choose 7 for the significance level parameter of Configuration A. There seems to be even less of an impact on normalized throughput due to magnitude of a random move, and so we choose the smallest value, 0.5. We set the minimum number of completions to 2 based on some informal observations that it is important to be robust to variations in completion rates. Configuration B uses a similar kind of reasoning as A, but it also considers excess threads and the standard deviation of the number of threads. For Configuration B: control gain is 8; significance level is 1; the magnitude of random move equal is 3; and the minimum number of completions is 0.

To evaluate these configurations, we use performance benchmarks that are calibrated to a set of web and email scenarios. The scenario starts with forty work items that have significant memory contention. After 300 seconds, the workload changes to work items that have very low memory demands and so higher throughputs can be achieved. At 500 sec, the workload returns to the high memory contention scenario. The benchmark runs for approximately 11 minutes, with 100 sec of warm-up.

Figure 9 plots the results of these benchmarks. The solid line is the number of threads, and the line with "+" symbols is throughput. The results for Configuration A are quite poor in that it is slow to find a good concurrency level with the first workload, and it never adapts to the second workload. The average throughput of Configuration B (18.5) is much larger than A's (8.2), which is disconcerting since we intended to select A so that it had high throughput. This suggests that the ad hoc approach is not a reliable way to select configurations.

We now proceed with the configuration selection phase in Figure 5. The first step is to use the algorithm in Figure 6 to

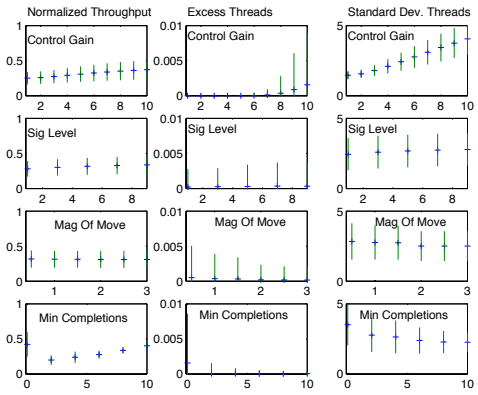


Figure 8. *Performance of individual configuration parameters.*

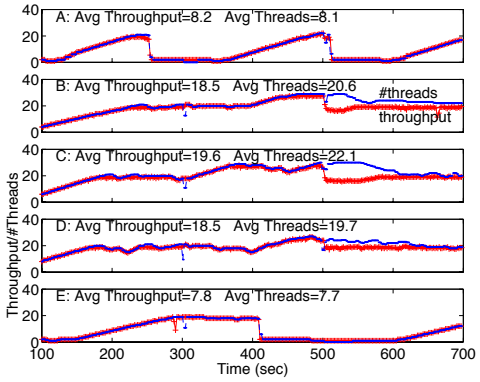


Figure 9. *Performance of Resource Manager Configurations. Good performance means a high throughput and low number of threads.*

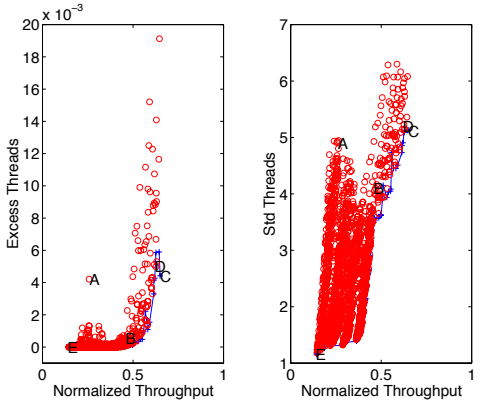


Figure 10. *Evaluation of configurations studied. Each point corresponds to a configuration. The blue plus signs indicate the optimal frontier, and the red circles indicate interior points. The black letters correspond to configurations listed in Figure 11.*

Config	Control Gain	Signif. Level	Mag. of Move	Min. Num. of Completions
A	10	7	0.5	2
B	8	1	3	0
C	10	9	2.5	0
D	10	5	2	0
E	1	1	0.5	2

Figure 11. *Details of configurations annotated in Figure 10.*

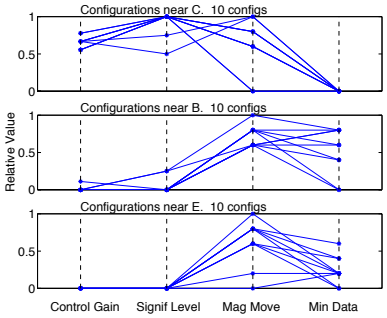


Figure 12. *Configurations near three configurations listed in Figure 11. The y-axis is the relative value of the metric normalized into the range from 0 to 1.*

construct the optimal frontier of evaluation metrics. Figure 10 displays the results. There are two plots. The left plot depicts the relationship between normalized throughput and excess threads; the right plot depicts normalized throughput versus the standard deviation of the number of threads. Each point corresponds to one of the 1,800 configurations defined in Figure 7. The blue line with the '+' markers defines the optimal frontier of the evaluation metrics. Of the 1,800 configurations, 98 lie on the optimal frontier. This is over a 10:1 reduction in the number of configurations to consider. The plots contain annotations with the letters A through E that correspond to configurations described in Figure 11.

It is insightful to see where the results of the ad hoc approach lie in Figure 10. We see that Configuration A is not on the optimal frontier. Indeed A has a fairly low normalized throughput, a relatively high number of excess threads, and a large standard deviation of threads. On the other hand, Configuration B is fairly close to the optimal frontier. Its normalized throughput is considerably larger than that for Configuration A, and B's excess threads and standard deviation of threads are both much lower than A's. These observations are consistent with the benchmark results in Figure 9.

Proceeding with the second step in configuration selection, we consider three configurations. C is a high throughput configuration on the optimal frontier. We also consider Configuration D that is in the interior but very close to C so as to gain insight into how performance changes as we move away from the optimal frontier. Last, we consider Configuration E that minimizes excess threads and standard deviation of threads, but it also has low throughput.

We see that the benchmark results plotted in Figure 9 are largely consistent with the evaluation metrics produced by our system model. Configurations C and D both have high throughput and more threads (although very few excess threads). Note that the throughput of Configuration C, which is selected by our methodology, has throughput that is 240% larger than Configuration A, which is chosen in an ad hoc manner. The throughput of configuration E is considerably smaller than the other configurations, and it has the fewest number of threads as well.

These observations give us assurance that selecting a configuration from the optimal frontier provides the best tradeoffs between the evaluation metrics. Where we want to be in the frontier depends on our relative weighting of the metrics. Certainly, Configuration C is a good choice since it produces the highest throughput.

To conclude the case study, we examine the parameters of configurations that produce similar normalized throughputs. Figure 12 displays details of the ten configurations that have normalized throughputs closest to each of configurations C, B, and E. Consider the top plot, which studies configurations near C. The horizontal axis is the categorical variable "configuration parameter", and has the values "Control Gain", "Significance Level", Magnitude of Random Move", and "Minimum Number of Completions". The vertical axis is the relative values of each of these configuration parameters. For example, Control Gain has the discrete values 1 through 10. Thus, if Control Gain is 10, its relative value is 1, and if Control Gain is 1, its relative value is 0. A configuration is a line with a point of intersection (configuration parameter) for each categorical variable on the horizontal axis.

We see that the high throughput configurations near C have large Control Gains and large Significance Levels. One intuition is that a large Control Gain allows the thread pool controller to adapt quickly, and the large Significance Level means that small changes are sensed quickly. Most, but not all of these configurations, have a large Magnitude of Random Move. The advantage of a large Magnitude of Random move is that the thread pool controller explores more concurrency levels. All of the configurations near C have a low value for the Minimum Number of Completions. Such a setting avoids slow starts.

The foregoing configuration settings are quite different from the configurations near B. These configurations still produce large throughputs, but they do not have as many excess threads, and they have a smaller standard deviation of threads. Quite likely, this is due to using a small Control Gain and small Significance Level, both of which make the controller conservative. However, the values of these parameters are even smaller for configurations near E, which is consistent with its having extremely low throughput.

VI. Related Work

There is considerable existing work on configuration and tuning. One theme is model-based approaches to predicting system performance. Examples include the Microsoft SQL Server Tuning Advisor [1] and configuration of Virtual Private Networks [13]. In all cases, a model is constructed that relates configuration parameters to system performance. Unfortunately, it is not always possible to construct such models. There have also been proposals for policy-based frameworks for configuration (e.g. [15], [8], [9]). However, these approaches require substantial knowledge of the relationship between configuration parameters and performance of the managed system. Last, there has been work with tools for modeling the relationship between configuration parameters [18], a conceptual framework for determining requirements for

configuration tools [4], and on-line reasoning about the configuring software components [17]. Unfortunately, these tools and techniques are helpful only if it is possible to construct a model that relates configuration to performance. In contrast, we show that when such a model cannot be constructed, considerable efficiencies are obtained by just modeling the relationship between resource allocation and performance, and doing numerical studies to relate configuration parameters to resource allocations.

A second body of related work is in software testing. Indeed, the term "model fuzzing" is motivated by work on fuzz testing [14]. A common approach to performance testing is to develop workload characterizations, and then run performance benchmarks using these characterizations [3]. However, a direct application of this approach is impractical for configuration evaluation because of the time required to run performance benchmarks and the large number of configurations that must be evaluated (especially because of interactions between configuration parameters). Another approach is performance testing on selected subsets of system features [6]. In essence, this is our approach, but we exploit the ability to model the relationship between resource allocation and performance of the managed system to achieve computational efficiencies.

Last, there is work that seeks to eliminate configuration parameters altogether. One example is automatically setting configuration parameters of the Apache Web Server [7]. More generally, adaptive control [2] provides a framework for on-line optimization of configurations that eliminates parameters such as control gains. However, adaptive control typically adds new parameters such as forgetting factors and adaptation rates; so, the configuration problem remains. In the software community, there have been proposals for autonomous adaptive systems (e.g., [5]) and autonomic systems [10]. However, these are mostly architectural proposals and statements of requirements that do not address practical considerations such as configuring resource managers.

VII. Conclusions

It is common for RMs to expose parameters that must be configured properly to achieve good performance (e.g., database buffer pools, Apache Keep-Alive timeout). Finding good settings for these parameters is difficult because of the large number of configurations that must be evaluated. For example, in the studies of the .NET thread pool presented in this paper, 1,800 configurations are evaluated for 64 system variants, which is a total of 115,200 different studies. Clearly, it is impossible to collect this volume of data using performance benchmarks that run for minutes or hours since machine-years would be required to do configuration. On the other hand, the complexity of and interdependencies between configuration parameters make it difficult to construct accurate models that relate configurations to performance of the managed system.

We propose model fuzzing, a technique that combines measurement and modeling to provide accurate and scalable evaluation of RM configurations. Model fuzzing begins by constructing a system model that relates resource allocations to performance of the managed system. For the .NET thread pool, this is expressed as a unimodal concurrency-throughput curve, a function that is easy to estimate using simple equations. Then, studies are conducted in which the RM implementation interacts with the system model to estimate performance of the managed system for RM configurations. By "fuzzing" the parameters of the system model, we explore the performance of RM configurations for many variants of the managed system and its workloads. We develop a methodology for configuring RMs that uses model fuzzing and considers multiple evaluation criteria (e.g., high throughput, low number of threads). Applying this methodology to the .NET thread pool, we find a configuration that increases throughput by 240% compared with the throughput of a poorly chosen configuration. Further, model fuzzing reduces the time to do this analysis from machine-years to machine-hours.

References

[1] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005. *Proceedings of the 30th COnference on Very Large Databases*, 2004.

[2] K. J. Astrom and B. Wittenmark. *Adaptive Control*. Addison-Wesley, second edition, Jan. 1995.

[3] A. Avritzer, J. Kondek, E. Weyuker, and D. Liu. Software performance testing based on workload characterization. *International Workshop on Software and Performance*, 2002.

[4] M. Burgess and A. L. Couch. Modeling next generation configuration management tools. *LISA*, pages 131–147, 2006.

[5] K. Cooper, J. Cangussu, and E. Wong. An architectural framework for the design and anlysis of autonomous adaptive systems. *Internation Computer Software and Applications Conference*, 2007.

[6] G. Denaro, A. Polini, and W. Emmerich. Early performance testing of distributed software applications. *International Workshop on Software and Performance*, 2002.

[7] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. Tilbury. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server. In *IEEE/IFIP Network Operations and Management*, pages 219–234, April 2002.

[8] K. Heller and R. Wies. Policy driven configuration management of network devices. *IEEE Network Operations and Management (NOMS)*, 1996.

[9] A. Keller and H. Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, Mar. 2003.

[10] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[11] L. Kleinrock. *Queueing Systems*. Wiley-Interscience, 2nd edition, 1975.

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

[13] I. Luck, S. Vogel, and H. Krumm. Model-based configuration of vpns. *Network Operations and Management Symposium (NOMS)*, 2002.

[14] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[15] A. Sahai, S. Singhal, and V. Machiraju. Automated generation of resource configurations through policies. *IEEE Workshop on Policies for Distributed Systems and Networks*, 2004.

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

[17] M. SPear, T. Roeder, O. Hodson, G. Hunt, and S. Levi. Solving the staring problem: Device drivers as self-describing artifacts. *EuroSys*, 2006.

[18] Y. Yemini, A. Kostantinou, and D. Florissi. Nestor: An architector for network self-management and organization. *IEEE Journal on Selected Areas in Communications*, 18(5), 2000.