An Approach to Benchmarking Configuration Complexity

Aaron B. Brown and Joseph L. Hellerstein
IBM T. J. Watson Research Center
{abbrown,hellers}@us.ibm.com

Abstract

Configuration is the process whereby components are assembled or adjusted to produce a functional system that operates at a specified level of performance. Today, the complexity of configuration is a major impediment to deploying and managing computer systems. We describe an approach to quantifying configuration complexity, with the ultimate goal of producing a configuration complexity benchmark*. Our belief is that such a benchmark can drive progress towards self-configuring systems. Unlike traditional workload-based performance benchmarks, our approach is* process-based*. It generates metrics that reflect the level of human involvement in the configuration process, quantified by interaction time and probability of successful configuration. It computes the metrics using a model of a standardized human operator, calibrated in advance by a user study that measures operator behavior on a set of parameterized canonical configuration actions. The model captures the human component of configuration complexity at low cost and provides representativeness and reproducibility.*

1. Introduction

Driven by the insatiable demand for features and performance, modern server systems have become incredibly complex. Much of this complexity is exposed as *configuration complexity*, with the result that these systems require significant investments of time and skill on the part of human system administrators to set them up and keep them running over time.

The magnitude of the problem is best illustrated by example. Figure 1 depicts the basic tasks in configuring a web service application using IBM's DB2 and WebSphere Application Server along with the data flows between these tasks. This particular configuration process requires that the human administrator perform over 60 individual configuration steps—and it is one of the simplest possible web service configuration scenarios.

If we continue to build systems that require such complicated configuration processes, we will soon reach a point where our systems are completely unmanageable. Researchers have begun to realize this, and there is burgeoning interest in tackling the problem of configuration complexity, or "futz" in the terminology established at the 1999 HotOS workshop [8]. IBM's autonomic computing effort is a major industrial effort [5], and academic researchers have identified research agendas in reduced-futz computing as well [4]. But a key missing piece in all this work is a quantitative way to measure futz—a repeatable benchmark that can quantify the level of configuration complexity for a given system, evaluate the impact of new technology on configuration complexity, and prove the value of futz-reducing techniques.

Building a benchmark for measuring configuration complexity requires a radically different approach from what is used in traditional performance benchmarks. This is apparent in Figure 1 in that the primary focus is on process complexity rather than runtime performance. Whereas traditional performance-oriented benchmarks are *workload-driven*, configuration complexity benchmarks must be *process-driven* as shown in Figure 2, with metrics based on an analysis of a captured process rather than on a system's response to a workload. Process-driven benchmarks demand novel methodologies and produce different categories of metrics than the usual throughput and latency metrics we are used to seeing.

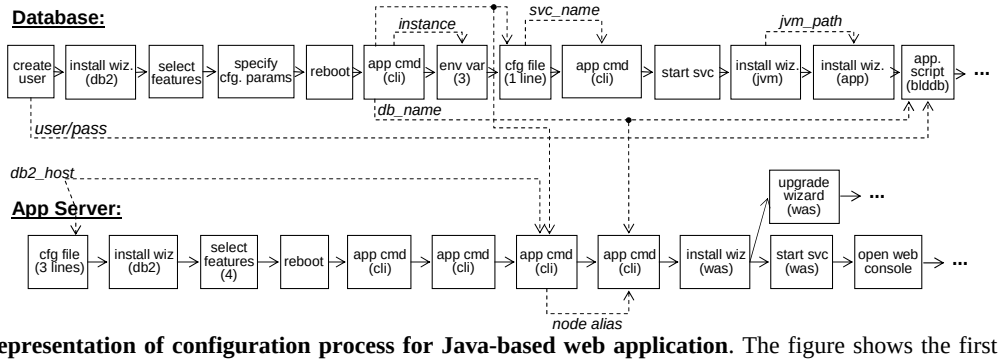
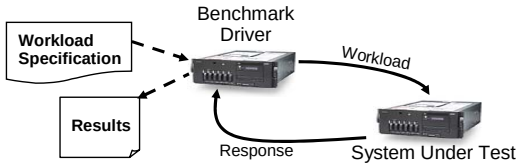
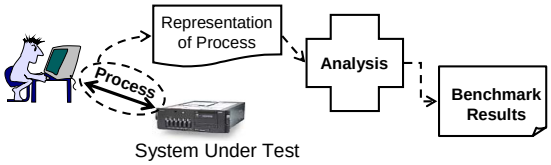


Figure 1. Representation of configuration process for Java-based web application. The figure shows the first few steps in the configuration process for a simple enterprise Java web application. Each box represents a configuration step, and dotted arrows represent information that the operator must pass between tasks.



(a) Workload-driven benchmark



(b) Process-driven benchmark

Figure 2. Workload- vs. Process-driven benchmarks.

Furthermore, quantifying configuration complexity inherently involves a human component. We typically try to avoid introducing people into benchmarks, since they add significant variability and cost. But it is unavoidable when benchmarking configuration complexity, since what we are trying to quantify is exactly the amount of human effort and skill required to achieve a configuration goal. An essential challenge is therefore to incorporate an understanding of human factors into the benchmark while minimizing the actual human involvement and resulting cost and variability.

Finally, good benchmarks produce metrics that are representative of real-world observable quantities; they are based on methodologies that are reproducible, widely-applicable, and resistant to nefarious optimizations designed solely to improve the benchmark score. Over time, we have figured out how to achieve these qualities for workload-driven performance benchmarks, but there is little experience to draw on for process-driven configuration complexity benchmarks. Reproducibility and representativeness are particularly challenging aspects, especially given the need to capture the human component of the configuration process.

While these technical challenges are daunting, we do not believe them to be insurmountable. In this paper, we outline a research agenda that we believe will lead us to solid benchmarks for configuration complexity, and present our initial approaches to constructing such benchmarks. We begin in Section 2 by focusing on the configuration process, then expand our horizons in Sections 3 and 4 by situating those processes in the context of system lifecycle. We treat related work in Section 5, and conclude in Section 6.

2. Quantifying the configuration process

At the core of our proposed benchmarks is a methodology for quantifying the complexity of a *configuration process*—the set of interactions between human operator and computer system that achieve a configuration goal. Systems with high configuration complexity force their administrators to carry out intricate, difficult, and error-prone configuration processes. In contrast, self-configuring, futz-free systems have simple configuration processes in which administrators only specify the configuration goal; the systems do the rest.

We quantify process complexity in terms of human interaction time and the probability that the configuration process is completed without error, parameterized by the skill level of the human operator. For example, the benchmark might report that an expert operator could configure a system in 25 minutes with a 95% likelihood of success, but that a novice operator would take 45 minutes and would only succeed 60% of the time.

A direct way of obtaining these metrics for a configuration process is through human factor studies. However, that approach is costly and difficult to reproduce. Figure 2(b) suggests an alternative: we can construct an optimal configuration process from one or more versions captured from expert human operator(s), then derive from this process appropriate metrics that quantify its complexity. This approach consists of the following steps:

1. **process capture:** The expert operator performs the configuration process on the system under test while the benchmark infrastructure records her interactions in a *response file*. This step can be repeated as needed to ensure the response file reflects the best-case configuration process.
2. **process decomposition and analysis:** The benchmark infrastructure maps each recorded interaction in the response file into a parameterized *canonical configuration action*. Examples of canonical configuration actions include selecting installable features and entering a configuration variable.
3. **process scoring:** The process is scored as to its complexity by estimating the interaction time and success probability for the aggregate set of canonical actions it encompasses. This scoring will be based on a model of the complexity of the canonical actions and the interrelationships between them, calibrated by studies of how human operators perform. Note that once these calibration studies have been done, they can be applied repeatedly to many benchmarks.
4. **process validation:** The benchmark verifies that the captured process has achieved its configuration goal subject to lifecycle-based quality constraints. We discuss this aspect in depth in Sections 3 and 4.

Our process-capture methodology is inspired by the model-human-processor (MHP) technique developed in the human factors community. MHP predicts human behavior on low-level stimulus-response tasks like moving a mouse pointer to click an on-screen button [2]; it works by identifying a set of basic, parameterized actions (like moving a mouse a certain distance) and uses a calibrated model of typical human behavior to estimate

the time it would take a person to perform those actions. We want to achieve the same thing, but at a much higher level where the basic actions are steps in a configuration process and where the model assigns both time and success probability to a set of those actions. Thus our methodology hinges on defining the right set of parameterized canonical configuration actions, and on building the model that maps them to human interaction time and error rate. These are open research challenges; we outline our initial approaches in the following subsections.

2.1. Defining canonical configuration actions

The boxes in Figure 1 are examples of canonical configuration actions. Included are actions such as selecting installable features from a list, setting environment variables, and editing configuration files. Each type of action may be parameterized. For example, "select features" could be parameterized by the number of features that must be selected, the total number of features available, and the number of changes made from the defaults.

The actions are also affected by the configuration information supplied by the operator, which is represented by the dashed arrows in Figure 1. Our hypothesis is that one of the major factors in configuration complexity is that the human operator needs to generate, remember, and later re-supply configuration variables, for example to input the IP address and port number of an earlier-configured database server when later setting up an application server. We can capture this complexity by characterizing each configuration action by the number, type, and source of its configuration variables, and by tracking what configuration variables must be retained in memory at each step of the process. Thus we can distinguish an action that reuses a variable that appeared 5 steps earlier in the process from one that uses a never-before-seen variable, and from one that uses a variable generated in the immediately-preceding step.

2.2. Building the complexity model

The complexity model predicts human interaction time and success probability for a set of canonical actions. In doing so, it defines the behavior of a *standardized human operator*. We envision a modeling approach that first assigns complexity scores to each canonical action based on its type, parameterization, and use of configuration variables. These low-level scores are then weighted, aggregated, and mapped to process-wide estimates of interaction time and success probability. To calibrate the weights and mappings, we use a human user study performed up front during process of designing and specifying the benchmark; the calibrated model becomes a reusable part of the benchmark specification.

The design of the user study is straightforward. We first generate a number of test cases, each of which includes a set of canonical actions arranged in a sequence that exposes a certain level of interrelatedness through shared configuration variables. These test cases should be chosen to cover the set of actions and their parameterizations, and to expose different levels of interrelatedness. Next, we recruit trained system operators to work through the test cases, and we measure their interaction times and success rates. We are exploring the possibility of performing these experiments using a web-based simulated configuration environment to reduce the cost of gathering the data. From the collected data, we can determine which action parameters and relationship metrics have a statistically-significant effect on complexity, and can calculate expected interaction times and success rates for various values of those parameters, along with associated confidence intervals. If we achieve tight confidence intervals, we know that we have chosen our actions and relationship metrics well; if not, then we must refine them and repeat the calibration process.

A key challenge in creating the complexity model will be to control the variance in the subject pool by selecting subject operators with similar skill levels; we expect that well-established screening techniques from human factors research will be sufficient. Ideally, the modeling process should be repeated with different subject pools—expert, newly-certified, and novice operators, at minimum—to produce a set of models indexed by the skill level of the operators used to calibrate them.

2.3. Discussion

Our approach satisfies most of the properties of a good benchmark. If the model is accurate, we produce results that are *representative* of real-world configuration complexity, namely predicted human interaction time and success probability. By using a model, we achieve representativeness without the cost and variability of a separate large-scale human trial for each use of the benchmark. Our approach is *widely-applicable*, because it is based on analyzing canonical configuration actions, rather than directly processing system system-specific actions. Its results are *reproducible* since they are derived from a standardized operator model included in the benchmark. Furthermore, the standardized model and action set ensures those results can be *directly compared* across systems, assuming the same configuration goal was specified for each. Our approach provides *guidance for improvement*, since it is easy to identify the steps of the configuration process that contributed the most to the overall complexity score; the benchmarker can use the model to estimate the impact of specific improvements. Finally our approach is reasonably *low cost*: other than the cost of the expert operator and process capture, the benchmark is entirely analytical. Note that the cost of the operator is not an issue in many practical benchmarking scenarios, as at least one expert operator

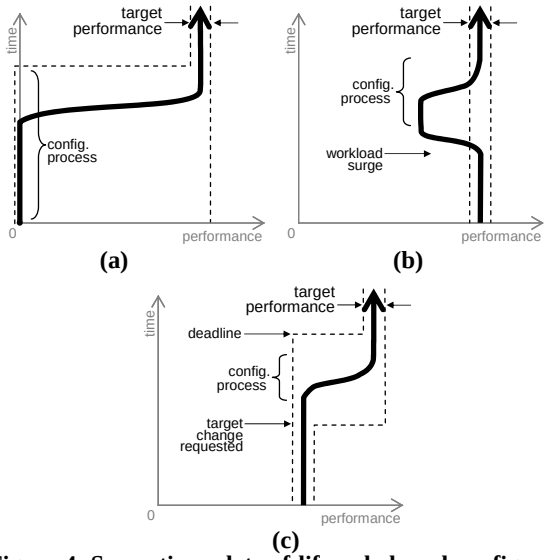


Figure 4. Space-time plots of lifecycle-based configuration scenarios. The diagrams show a simplified space with performance as the only dimension. The dashed lines define the region of desired performance. The solid line depicts the system's performance trajectory.

will already be employed to ensure that the system is optimally configured before collecting performance results.

Our approach does suffer some limitations. It assumes a single operator, and will have to be altered to scale to system installations that require multiple cooperating operators. It examines only one possible configuration process—the one captured in the response file. If the system supports multiple configuration paths, the benchmark must be repeated on each and cannot capture the complexity of choosing between those paths. Finally, as we have described it, our approach is not resistant to *benchmark-specific optimization* since we do not check to ensure that the configuration process produced the desired results. We address this concern in the next section by adding specific configuration goals and quality constraints to our process-evaluation methodology.

3. Adding lifecycle constraints

Configuration processes (and hence configuration complexity) enter at many different points in the lifecycle of a software system, including *initial setup*, *runtime reconfiguration*, and *decommissioning*. Each lifecycle phase has different goals and constraints: for example, an administrator might ignore system performance and availability while performing an initial configuration process, but might be very concerned about preserving them during runtime configuration. As a result, a configuration task performed on a running system could require a much more complex set of configuration actions than the same task performed during the system's initial setup. The implication for our benchmark is that it must consider more than one lifecycle scenario; it must also define and enforce lifecycle-specific quality constraints.

One way to approach quality constraints is to represent the system's lifecycle as a trajectory in a multidimensional space, where each dimension measures one aspect of the system's service. For example, a simple web server might trace a path through a 4-D space defined by axes of throughput, latency, functionality, and correctness; a clustered system might add an axis for fault-tolerance. At any point during the system's lifecycle, there is a target region in the space that defines acceptable quality, and thus establishes quality constraints. As a system moves through different phases in its lifecycle, it can move out of its target region, requiring configuration to bring it back into line.

Figure 3 gives a few examples for a very simple 1-dimensional space characterized by an unspecified performance metric. Figure 3(a) depicts initial setup of a system. The system starts out at the origin, since a set of undeployed components provides no performance. The target region is defined by the system's desired performance; the initial configuration process shifts the system from the origin to the target point. In this example, the constraint on the configuration process is that it must produce a system that falls within the target by the end of the process; the intermediate trajectory is irrelevant.

Figure 3(b) depicts a scenario that might be seen when a sudden workload surge hits a system. The surge pushes the system out of its target performance region, and some configuration process is needed to bring it back to normal (perhaps, deploying additional capacity). In this case, the configuration is reactive and ongoing, and the constraints are both to minimize the time it takes the system to return to its target window and to minimize any additional performance impact during that time. Figure 3(c) is similar, but here the target has moved rather than the system, representing a scenario where an administrator proactively reconfigures a running system, perhaps via an upgrade or by deploying capacity to meet a predicted upcoming load increase. The constraints for this scenario are to reach the new performance target by a pre-specified time and to minimize additional performance degradation in the meantime.

4. A complete benchmark scenario

In this section, we combine the methodology of Section 2 with the constraints of Section 3 to form a complete benchmark methodology for measuring the configuration complexity of initial system setup (from Figure 3(a)). To begin, we must pick a specific application context for the benchmark—this will help us set goals and constraints. For convenience, we hijack the scenario from an existing performance benchmark—say, the SPECjAppServer2002 Java Enterprise application benchmark [9]. This gives us a configuration goal, namely to assemble a system that can satisfy the SPECjAppServer workload at some performance level. We leave the level undefined to allow the benchmark to

scale across different system sizes, but require that the benchmark report the performance achieved. This reporting rule ensures that the configuration goal is met, and that an unscrupulous benchmarker cannot get away with optimizing away configuration complexity at the cost of system performance.

In this scenario, the benchmark methodology is straightforward. First, the benchmarker documents the initial state of the system as an inventory of the components that will be assembled into the working system. Next, she performs the configuration process, applying the methodology of Section 2 to capture the final process. She then chooses a behavior model based on the skill level of her system's operators, and uses the model to analyze the captured process and produce a complexity score. She then runs the SPECjAppServer2002 performance benchmark on the configured system and records the performance result. Finally, she prepares a benchmark report that includes the complexity score, the skill level of the model used, the final performance score, the component inventory, and the response file corresponding to the captured process. Besides the ultimate complexity score, this report contains enough information to reproduce the analysis and audit the results for accuracy. While we have described the benchmark workflow as a manual process, our hope is that it can eventually be fully automated via technology for automated process capture and decomposition [6].

Note that we can extend our methodology to the other scenarios in Figure 3 by replacing the goal appropriately and by requiring that the SPECjAppServer workload be applied continuously during the configuration process. The continuous trace of performance results during the configuration process is sufficient to quantify the response time and performance impact of the process, much as is done in availability and dependability benchmarks to capture the impact of an injected failure [1] [7].

5. Related work

To our knowledge, there is very little existing work on benchmarking configuration complexity. Benchmarks exist that attempt to test properties of a static configuration, such as the CIS Security Benchmarks [3], but these do not measure the complexity of changing the configuration, and hence cannot evaluate complexity-reducing technology.

That said, our process-based benchmark approach borrows from existing techniques in other fields. As already mentioned, our model-based approach to assigning human-based complexity metrics was inspired by the model human processor concept from the human factors community [2]. And the idea of measuring a process itself is a direct extension of the traditional user-centric approach used by HCI practitioners to evaluate new interface designs; our contribution here is to distill that approach into a benchmark framework so that it can be applied reproducibly, and at low cost, across systems.

One domain where complexity has been studied in great depth is software engineering, where hundreds of metrics exist for quantify the static complexity of a piece of code (see, for example, Zuse's excellent survey [10]). Unfortunately, nearly all this work focuses on complexity as manifested in a static snapshot of the code (typically based on the control-flow graph), and does not translate to the process-centric view of configuration complexity that we have defined. While there may be value in adapting these software metrics to the task of measuring the static complexity of a system's configuration, we believe it far more important to quantify the complexity of *altering* the configuration, as that aspect is what contributes to system management cost.

6. Conclusion

The techniques that we have outlined in this paper represent a first step toward rigorous, reproducible configuration complexity benchmarks. But significant research challenges remain: in identifying the right set of canonical configuration actions and their parameterizations, in constructing and validating the model, in capturing different operator skill levels, and in automating the task of process capture and analysis, amongst others. While we are only starting to tackle these challenges by implementing proof-of-concept benchmarks, we do believe that our methodologies are sound and that they offer great promise toward building the evaluation tools that will prove the value of and drive progress toward low-futz, easily-configured systems.

References

[1] A. Brown and D. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. *USENIX Annual Technical Conference*, 2000.

[2] S. Card, T. Moran, and A. Newell. The Model Human Processor. In Boff, K., et al. (Eds.) *Handbook of Perception and Human Performance*, 1986.

[3] Center for Internet Security. http://www.cisecurity.org.

[4] D. Holland et al. Research Issues in No-Futz Computing. *Proc. 8th Workshop on Hot Topics in Operating Systems*. Schoss Elmau, Germany, 2001.

[5] P. Horn. *Autonomic Computing Manifesto*. http://www.ibm.com/autonomic/pdfs/autonomic_computing.pdf.

[6] T. Lau et al. Learning Procedures for Autonomic Computing. *Workshop on AI and Autonomic Computing (IJCAI 2003)*. Acapulco, Mexico, 2003.

[7] H. Madeira and P. Koopman. Dependability benchmarking: making choices in an n-dimensional problem space. *First Workshop on Evaluating and Architecting Systems for Dependability*, Göteborg, Sweden, 2001.

[8] M. Satyanarayanan. Digest of Proceedings. *7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, 1999.

[9] SPEC. *SPECjAppServer2002*. http://www.spec.org/jAppServer2002/.

[10] H. Zuse. *Software Complexity: Measures and Methods*. Berlin: Walter de Gruyter, 1991.