MassConf: Automatic Configuration Tuning By Leveraging User Community Information *

Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen
Department of Computer Science, Rutgers University
{*wzheng,ricardob,tdnguyen*}*@cs.rutgers.edu*

Technical Report DCS–TR–664, January 2010

Abstract

Configuring modern enterprise software can be extremely difficult, because its behavior often depends on large numbers of configuration parameters. We argue that vendors can simplify the configuration process for new users of their software by collecting and using configuration information from the existing user community. Our proposed approach is based on the observations that (1) a "good" configuration may work well for many different users, and (2) multiple good configurations may work well for each user. We demonstrate our idea by designing MassConf, a system that collects and uses existing configurations to automatically configure new software installations. To evaluate MassConf, we use it to configure the Apache Web server to achieve a response-time target. Our results confirm our observations and show that MassConf successfully reaches the targets of many more new installations than an existing efficient optimization algorithm. Even when we consider only the installations that can be configured with this efficient algorithm, our results show that MassConf reaches the desired targets running many fewer experiments on average.

1 Introduction

Enterprise software is becoming increasingly complex. A single piece of server software may include hundreds of configuration parameters, as software vendors and contributors (collectively called "vendors" hereafter) want their systems to be as flexible and adaptable as possible. For example, there may be configuration parameters to specify the number of server threads to create, the amount of memory to use for caching disk data, or the timeout before an idle network connection is destroyed. Selecting proper values for configuration parameters is critical, since they may affect the software's behavioral correctness, performance, availability, and/or energy consumption. While vendors often provide default configuration files that work well for many users, configurations need to be tuned in many other cases.

Unfortunately, configuring modern software can be extremely difficult in those cases. The reason is that a good configuration depends (at least) on the hardware environment, the workload, the load intensity, and the target behavior (e.g., some level of performance or availability) the user wants to achieve. Moreover, besides the sheer number of parameters that may affect the software's behavior, there can be relationships between the parameters that are not made explicit by the software documentation. Thus, it is very hard (if not impossible) for users to completely understand the configuration-hardware-workload-intensity-target relationship.

Due to the size and complexity of this configuration space, previous research has focused on approaches and tools to detect misconfigurations and/or troubleshoot them [1, 12, 13, 18, 22, 23, 24], to study the resilience of systems in the face of configuration errors [11], to automatically configure a large number of machines in a single installation [2, 3, 4], and to automatically tune configurations for best performance [21, 25].

Although these efforts have been useful, a user's ability to configure her software to achieve a certain target behavior is still far from ideal in practice. For example, a user who wants her server software to produce an average response time of 50 milliseconds is left clueless, when the default configuration reaches only 100 milliseconds. As long as the parameters that affect performance are identified, this user can run existing algorithms (e.g., [14]) to optimize the server's performance by experimentation with different configuration settings. However, tuning performance may involve a very large number of time-consuming experiments [21, 25]. For example, each tuning experiment with a database server may involve restoring a large database and its indexes back to a specific initial state.

We argue that vendors need to do more to help users configure their software. One approach vendors could take is to create automatic configurers that run locally at the users' sites and select the best values for the parameters, either through experimentation or modeling. A simpler and cheaper approach, the one we advocate here, is for the vendor to collect configuration information from the existing user community of its software and use it in configuring the software for new users.

Our approach is based on two key observations. First, *a configuration may actually work well for many users*, i.e. it may work well for many workloads, load intensities, and target behaviors, especially when the users use similar hardware platforms. For example, the default configuration often produces acceptable behavior for many users. This observation means that popular (i.e., frequently used) existing configurations may work well for many new users of the software. Figure 1 illustrates this observation with a simple example. Configuration C_9 is more popular than C_2 and C_4, as it is used by more users (2 instead of 1).

The second observation is that *multiple configurations may actually work well for each user*, i.e. they may all meet the

*This research was partially supported by NSF grant #CNS-0448070.

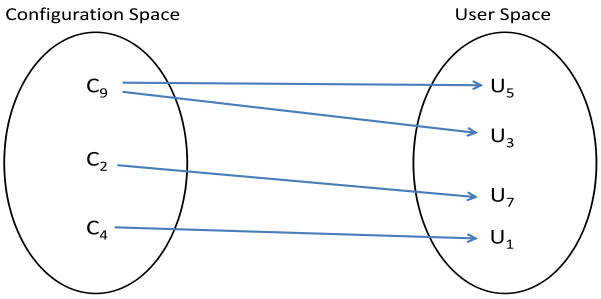

Figure 1: Many users may use the same configuration. An arrow from A to B means that configuration A is used by user B. Solid arrows represent information that MassConf has.

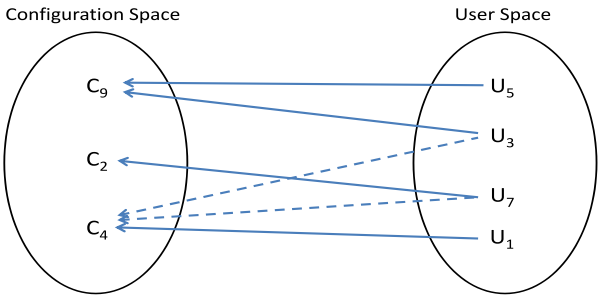

Figure 2: Many configurations may work for each user. An arrow from A to B means that user A can use configuration B. Dashed arrows represent data that MassConf cannot obtain.

user's target behavior. This observation means that there is flexibility in which existing configuration to select for each new user; even configurations that are unpopular may work well for her. For example, a user seeking an average response time of 50 milliseconds for a light workload may be able to use many different configurations, including the default one and a configuration that has enabled a single user with a particularly heavy workload to achieve high performance. Similarly, many other users may be able to use the latter configuration. Figure 2 illustrates this observation, showing that users U_3 and U_7 could also use configuration C_4 (besides C_9 and C_2, respectively).

Taken together, these observations mean that any work that users may do to tune their configurations can benefit new users of the software as well. Thus, in this paper we propose to leverage the existing users' configurations to find a good configuration for each new user. To demonstrate this idea, we designed MassConf, a system that automatically collects configuration and environment information from existing users, clusters users according to environment, produces an ordered (ranked) list of possible configurations, and tests each configuration in turn at the new user's site until the target behavior is met. After the configuration of each new user is complete, MassConf may change the ranking of configurations. MassConf seeks to (1) reach the target behavior for as many new users as possible and (2) minimize the average number of experiments required at the new users' sites. Because users are sometimes reluctant to divulge information about their systems, MassConf stores as little data as possible about them: only their environment descriptions and the non-sensitive parts of their configurations.

The most interesting technical aspect of MassConf (and the main focus of this paper) is its ranking of configurations. Faced with our first observation above, one would be tempted to rank configurations based on popularity; more popular configurations would be tried first at the new users' sites. The popularity information is readily available from the existing users' deployed configurations. In our example, the popularity information corresponds to the relationships depicted in Figure 1. However, as our experiments shall demonstrate, the popularity-based ranking is not the best choice. The reason is that particularly effective but difficult-to-find configurations would tend to appear towards the end of the list. Ranking them higher would allow more new users to be configured with fewer experiments.

To account for this effect, MassConf would require information about how every deployed configuration would do for every existing user. In our example, it would need to know at least about the dashed arrows in Figure 2. Knowing this information, MassConf would rank C_4 higher than any other configuration, since C_4 can satisfy more users than any other configuration. Unfortunately, *such information is obviously not obtainable.* Thus, MassConf adapts its behavior over time, by moving the configuration that is selected for each new user toward the front of the ranked list, regardless of its actual popularity. This adaptation increases the chance that another new user will also experiment with (and hopefully benefit from) the selected configuration.

Our optimized version of MassConf, which we call MassConf+, improves ranking further by cutting off the ranked list of configurations after an initial "learning" period. Shortening the list rids MassConf+ of configurations that are unlikely to satisfy a large number of new users, thereby reducing the average number of experiments. (Hereafter, we only refer to MassConf+ explicitly when discussing material that does not apply to MassConf. When the context does not require such a distinction, we refer to both systems simply as MassConf.)

To evaluate MassConf's ranking of configurations in an interesting (yet understandable) case study, we investigate its use for automatically configuring the Apache Web server to achieve a response-time target. Despite the popularity of Apache, evaluating MassConf poses a challenge for academics, namely the unavailability of massive user configuration datasets in the public domain for any piece of software. Instead of being discouraged by this challenge, we decided to evaluate MassConf using a synthetic user population. In this context, we study different speeds for moving a selected configuration to the front of the ranking, as well as the popularity-based ranking of configurations. As a baseline for comparison, we use Simplex, an efficient algorithm [14] that has been successfully used to optimize server software [6, 25].

The results of our case study show that adaptive ranking requires many fewer experiments than the popularity-based ranking to configure a population of new users. Regarding adaptation speeds, we find that the faster we move a selected configuration to the front of the ranking, the better on average. In fact, the best approach is to always move such a configuration straight to the front of the ranked list. Popularity-based ranking is only faster on average than the slowest moving adaptation. We also find that MassConf can configure many more new users than Simplex. Moreover, MassConf requires many

fewer experiments than Simplex, even when we consider only those new users that both systems can configure. MassConf+ reduces our average number of experiments per new user even further. Based on our experience with the case study, we qualitatively extrapolate from it to identify the general conditions under which MassConf is most effective.

Our experience and results illustrate that software configuration can be significantly simplified by having users contribute parts of their configurations and use them to configure the software for other users. Because of its simplicity and effectiveness, we conclude that MassConf and its adaptive ranking of configurations have great potential to work well in practice.

The remainder of the paper is organized as follows. The next section describes MassConf in detail, including its configuration ranking algorithms and Simplex. Section 3 introduces our Apache case study and experimentally evaluates MassConf. Section 4 extrapolates from the Apache study. Section 5 overviews the related works. Finally, Section 6 concludes the paper and discusses some opportunities for future work.

2 MassConf: Automatic Configuration

In this section, we first detail the design of MassConf. We then describe its ranking approaches and discuss its bootstrapping. After that, we describe the Simplex algorithm. The last subsection discusses some potential refinements to MassConf.

2.1 MassConf Design

Figure 3 illustrates the MassConf design. The next few paragraphs detail each part of the design in turn.

Data collection from existing users. MassConf is run by the software vendor. First (step 1 in the figure), it collects configuration and environment information from each existing user that is willing to participate. (Although some users may refuse to provide this information, many would likely be willing to contribute to the community since they can benefit from it as shall be clear below.) This information is extracted by instrumentation in the server software itself and sent to the vendor.

The configuration information describes the settings of each configuration parameter of the software. The settings can be of any type, e.g. boolean, numeric, or character strings. When the configuration information may include sensitive data, only a few relevant parameter values may be collected. (The vendor should know which parameters may include sensitive data.)

As part of the configuration information, MassConf must be informed about the users' *high-level goals* when they selected their configurations. For example, the goal may have been to improve performance, improve performability (performance + availability), or lower energy consumption.

MassConf stores the parameter settings it receives without modification, except in the case of numeric parameters. For each numeric parameter, MassConf breaks the range of possible values into 10 evenly sized chunks. Two configurations are grouped together if their values for each parameter fall in the same chunk. For example, suppose that each configuration has two parameters, p_1 and p_2, with possible values ranging from 1 to 200 (chunks of size 20) and from 1 to 100 (chunks of size 10), respectively. Further, suppose that the values of these parameters for configurations C_4 and C_5 are: $C_4(p_1)$ = 10 (first chunk), $C_4(p_2)$ = 18 (second chunk), $C_5(p_1)$ = 16 (first chunk), and $C_5(p_2)$ = 12 (second chunk). Because the chunks match for each parameter, C_4 and C_5 would be grouped together.

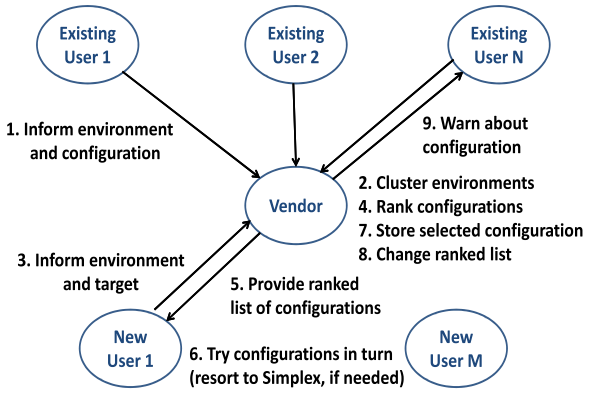

Figure 3: MassConf overview.

The configurations in each group are represented by a single "average" configuration. In the average configuration, each parameter is given the average of the values seen for that parameter in the corresponding group. For example, the average configuration C_{avg} for the cluster formed by configurations C_4 and C_5 above would have $C_{avg}(p_1)$ = 13 and $C_{avg}(p_2)$ = 15.

The environment information is a description of the hardware (e.g., number of CPU cores, amount of memory) and possibly the low-level software (e.g., operating system, settings for relevant environment variables) at the user's site. This information is necessary because the behavior of the software to be configured may depend heavily on the environment.

Clustering existing users according to environment. Using the environment information, MassConf then clusters the existing users (step 2) as was done in Mirage [7] for software upgrade deployment. The idea is to cluster users that have similar environments together, so that their configuration information can be used for new users with similar environments. For example, the vendor of a multithreaded server may want to separate out user sites with vastly different numbers of cores or threading libraries, as these aspects of the environment may have a significant effect on the ideal number of threads with which to configure the server. Conversely, user sites with similar numbers of cores and thread libraries should be clustered together. A number of algorithms can be used for clustering, but we prefer the Quality Threshold (QT) algorithm [10]. QT starts with one site per cluster. It then iteratively adds sites to clusters (effectively merging clusters) while trying to achieve the smallest average inter-site distance and not to exceed a predefined maximum cluster diameter. The algorithm stops when no more clusters can be merged together. Our distance metric involves the aspects of the environment that differ between clusters. Each aspect is weighted by the vendor, according to its importance to the software configuration.

Collecting information from a new user. After clustering some existing users, MassConf is ready to configure a new user. It first deploys the software to the new user's site and collects

its environment information (step 3). Then, it requests from the user a description of the software's target behavior (step 3). The target behavior reveals the high-level goal for the configuration tuning. With the new user's environment information, MassConf can now identify the best cluster for it.

Ranking configurations. Using the configuration information from this cluster, MassConf produces a ranked list (or ranking) of configurations to be tried at the new user's site (step 4). The list is formed by the configurations of the existing users that had the same goal as the new user. (For example, we do not want to use information about configurations that were selected to lower energy consumption when configuring servers for maximum throughput.) The exact ordering of the list is influenced by the order of the users' (both new and existing) arrivals, as described below. The list is transferred to the new user's site (step 5). At this point, MassConf can run experiments with each configuration, until the desired behavior is met or it runs out of configurations to try (step 6).

Testing configurations at the new user's site. These experiments are run under the user's actual workload and load intensity, so the user herself may have to provide a realistic test harness to exercise the software. If all experiments are run and the desired behavior is never achieved, the user is warned. MassConf's inability to reach a target may mean that the target is unrealistic for the workload and load intensity, or that it still does not have enough information (i.e., enough existing users) to produce a large enough coverage of the possible configurations. (We discuss this bootstrapping problem in a later subsection.) If the user confirms that the target is achievable and the parameter values are numeric, MassConf resorts to Simplex, *starting from the best configuration it has found so far*. However, we expect that MassConf would rarely have to resort to other approaches in practice; in most cases, the new user would relax the target. In these cases, MassConf would most likely have already found an appropriate configuration.

Storing the selected configuration. When a configuration is selected, MassConf includes information about it in its central database of existing users (step 7). At that point, the new user becomes one of the existing users within the corresponding cluster. (As MassConf found the configuration for the new user, it already has all the information required from an existing user.) Thus, after the bootstrapping period, the population of existing users should exhibit similar characteristics (as a group) to the new users (also taken as a group).

Adapting the ranking. As it is impossible to predict the set of new users that will want to join the system, MassConf adjusts its ranking (step 8) by moving the configurations that have been selected for each new user towards the top of the ranking. This adjustment enables very good configurations to be chosen more often. When MassConf needs to resort to Simplex, the new configuration is added to the end of the ranking. We discuss these decisions in detail in the next subsection.

Providing feedback to existing users. Finally, MassConf warns existing users when their configurations seem suboptimal (i.e., new users with the same goal have selected other configurations) with respect to the rest of the users in the same cluster (step 9). This feedback to existing users is an incentive for them to provide their configuration and environment information, even when they had to configure their software entirely by hand or when the community was still small. Another important incentive may be to help these users configure an upgraded version of the software by leveraging information from the users who benefited most from MassConf to configure the existing version.

Discussion. Previous systems have also relied on information from their users [1, 19, 22, 23]. However, those systems seek to troubleshoot configurations, not tune them as MassConf does. In our context, the specific and diverse characteristics of the users' workloads and their behavior targets mean that configuration information is also diverse (i.e., coercion as in PeerPressure [22] does not apply) and prior actions from a user do not produce the same results for another user (i.e., local experiments are necessary). For these reasons, our main focus has been studying adaptive ranking algorithms and the number of tuning experiments to which they lead on average. Neither of these issues was considered by these prior works.

Although we have focused on the use of MassConf to configure software at the users' sites, our system can also be used for software that users deploy to Cloud Computing services such as Amazon's EC2. In this case, MassConf would require information about the *virtual* environment (and, possibly, the service) on which the software will be run. Every other aspect of MassConf would remain as described above.

2.2 Configuration Ranking

Dynamically adapting the ranking. As we mentioned before, a popularity-based ranking can be misleading. It is possible that unpopular configurations can actually satisfy many more new users than popular ones. The reason why these highly useful configurations are not more popular may be that they are more difficult to find, e.g. they are only needed for heavy workloads or hard-to-achieve target behaviors.

Instead of relying on popularity, MassConf dynamically adapts its rankings to eventually concentrate configurations that can satisfy many new users at the top. We study three approaches for promoting the selected configurations within a ranking: *slow*, *fast*, and *fastest*. The slow approach moves a selected configuration one slot up in the ranking. The fast approach moves the configuration to the halfway point between its current slot and the top of the ranking. The fastest approach moves the configuration directly to the first slot of the ranking. Figure 4 shows an example of how ranking (a) is adjusted after configuration C_4 and C_5 are selected by two consecutive new users, using the slow (b), fast (c), and fastest (d) adaptation approaches. For example, in the fast approach, C_4 is first moved from the 8th to the 4th slot in the ranking. This moves C_5, C_9, C_1, and C_8 one slot down the ranking. Then, when C_5 is selected by the next new user, it moves from the 5th to the 3rd slot. This moves C_3 and C_4 one slot down.

Regardless of the speed of promotion, any new configurations that are added to the system are appended to the end of the corresponding ranking. The reason is that we want to see more than one user benefit from a new configuration before we promote it up the ranking.

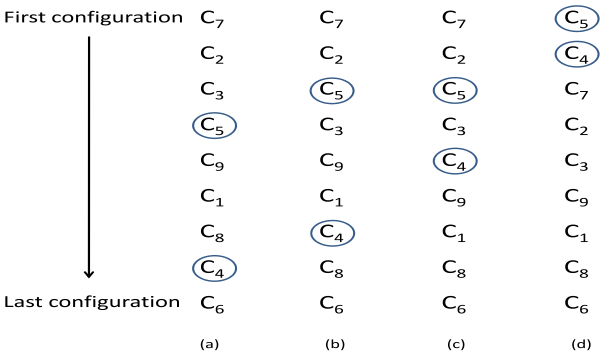

Figure 4: Original ranking (a) and slow (b), fast (c), and fastest (d) adaptation approaches, after configurations C_4 and C_5 are selected by two consecutive new users.

Note that configurations coming from existing users are treated the same as those selected for the new users, despite the fact that the former users select their configurations by means other than MassConf (i.e., the ranked configurations are not tested in turn for these users). We also considered the possibility of not altering the ranking when an existing user joins with a configuration that had already been seen. We ultimately decided against this approach because it would disregard the fact that the configuration satisfied an additional user.

Cutting off the tail of the ranking (MassConf+). After a period of adaptive ranking in MassConf, the configurations that satisfy the most users' targets will tend to rank high and reduce the average number of experiments per new user. Conversely, configurations that are not as widely useful will tend to be left at the tail of the ranking. This means that the likelihood that a configuration will satisfy a new user decreases rapidly as we move past the first set of configurations. Beyond this set, it may actually be more advantageous for MassConf to cut off the ranking and resort to Simplex right away, instead of trying a large number (potentially all) of the less useful configurations.

Based on this observation, we designed an optimized version of MassConf (called MassConf+) defining two thresholds: (1) the number of new users to see before cutting the ranking off; and (2) where the ranking should be cut off. MassConf+ uses heuristics to select these thresholds. For (1), it waits until the average number of experiments for configuring each new user has gone down many times in a row (10 times by default). Another option would have been to wait for a period with a stable average number of experiments per new user. We selected our current approach, because it allows MassConf+ to cut the list faster (before the average has stabilized). For (2), it cuts off the ranking at the number of configurations that has satisfied a large percentage (80% by default) of the new users seen so far.

Picking these thresholds properly is important, since any new configurations that are added to the system are *not* added to the corresponding shortened ranking. The reason is that adding these configurations to the shortened ranking could discard more useful configurations. A more robust approach could be to repeatedly prune the ranking, allow it to grow (which would happen ever more slowly) for a period, and then prune it again. Our simpler approach has worked well in our experiments, so we leave the more sophisticated one for future work.

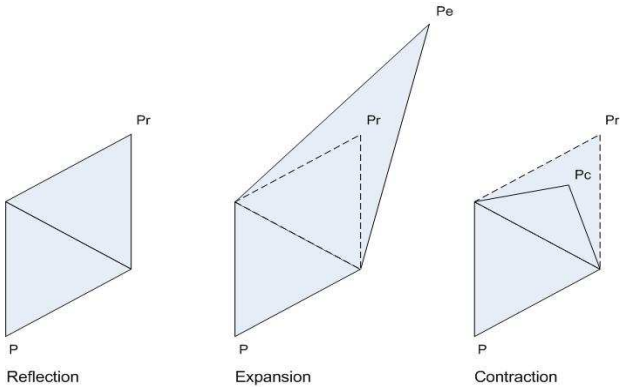

Figure 5: Main operations in the Simplex algorithm.

2.3 Bootstrapping the System

Any system that relies on other systems' information to make decisions faces a bootstrapping problem. MassConf is no different. It starts performing well when the existing users within each cluster become a good representation of the new users to come into the same cluster. Until that point, MassConf may be unable to meet the target behavior requested by new users without resorting to (experiment-intensive) Simplex. Instead of resorting to Simplex, the new user may also decide to optimize the configuration manually until the target behavior is achieved. Fortunately, these Simplex-derived or manually generated configurations contribute to MassConf just the same as the configurations of the existing users that join MassConf.

2.4 Simplex

The Simplex algorithm, as extended by Nelder and Mead [14], is an efficient method for nonlinear, unconstrained optimization problems. The algorithm optimizes (maximizes or minimizes) unknown functions $f(x)$ for $x \in I\!R^n$. A *simplex* is a set of $n+1$ points in $I\!R^n$, i.e. a triangle in $I\!R^2$, a tetrahedron in $I\!R^3$, and so on. The algorithm starts by selecting a random simplex and evaluating the function at each vertex of the simplex. Each iteration involves reflecting one of the vertices, but may also include expanding and contracting the simplex. These three operations are illustrated in Figure 5.

In more detail, each iteration involves the following steps: (1) Ordering – ordering the function values according to the optimization goal (e.g., descending order if the goal is to maximize the function); (2) Reflection – replace the vertex that leads to the worst function value (e.g., the smallest value when we are maximizing the function) by its mirror image in the centroid of the remaining n vertices. If the reflected value is better than the old value but not better than the best value currently, accept the reflected vertex and terminate the iteration; (3) Expansion – if the reflected value is better than the current best value, expand the reflected vertex away from the centroid. If the expanded value is better than the reflected value, take the former and terminate the iteration. Otherwise, take the latter and terminate; and (4) Contraction – if the reflected value is worse than the next to worst value, perform a contraction of the worst vertex.

If the contracted value is better than the worst value, take the former and terminate the iteration. When reflection, expansion, and contraction all fail to produce a better value, Simplex "restarts" by shrinking the parameter space around the current best vertex, picking new vertices (randomly) to form a new simplex, and starting another iteration.

In our context, each vertex of the simplex is a configuration. The operations done to each vertex involve operating with the corresponding parameters of the configuration. For example, a reflection involves reflecting each configuration parameter of the worst configuration independently with respect to its value in the centroid configuration.

In our experiments, we set Simplex to terminate when a target average response time is reached or the standard deviation of the vertices' response times is smaller than 5 milliseconds [14]. Under these stopping criteria, Simplex required between 7 and 174 experiments.

2.5 Potential Refinements

We have considered many refinements to MassConf. We describe some of them next.

Storing workload and load intensity information. We currently only collect configuration and environment information from users. However, we could also potentially collect workload and load intensity data and use it to improve our ranking of configurations. The obvious difficulty is how to characterize these new data in a manner that enables a meaningful comparison of different user sites. For example, we could collect resource utilization data summarizing behaviors at each user's site. However, the resource utilizations of two sites may be similar while the actual workload and load intensity are quite different (e.g., high CPU utilization may be the result of a light load that is computationally intensive or a high load that is network stack intensive). Configuration and environment information (including the users' tuning goals) are sufficiently well-defined that characterization is not a problem.

Storing the results of experiments. Right now, MassConf only stores the configuration that is selected for each user; the exact behavior (e.g., response time or energy consumption) to which this configuration leads is not used or stored. Another potential refinement would be to store all the configurations and actual experiment results on the way to meeting targets. The experiment results from the existing users could be compared to the new users' targets to potentially improve ranking further. Specifically, the configurations leading to results that are close to the targets could be ranked higher than others. However, this would require more complex ranking algorithms that consider the experiment results. Furthermore, it is unclear that experiment results obtained for the specific workload, load intensity, and target of one user would be useful in configuring another.

More aggressive re-ranking. Right now, MassConf does not change its ranking as a result of each experiment at a new user's site. Changes are only made after a good configuration has actually been selected for the new user. Another approach could be to change the ranking as we observe the behaviors of different configurations at the new site. For example, this more aggressive adaptation could be started after finding out that the configurations tested early on lead to very poor behavior compared to the target.

As Section 3.4 demonstrates, MassConf is very effective, so we decided that these refinements were not worth their additional complexity.

3 Case Study: Apache Configuration

To understand and validate MassConf and its adaptive ranking, we consider the Apache Web server as a case study. In particular, we focus on configuring Apache to achieve a target average response time. Because we lack real configuration data, we create a synthetic population of users. In this section, we first describe our approach for generating the populations of existing and new users. Then, we analyze the bootstrapping behavior and the characteristics of the configurations deployed by our users. Finally, we evaluate MassConf by comparing its adaptive ranking against the popularity-based ranking, as well as comparing its results to those of Simplex.

3.1 Methodology

Apache configuration and performance. Apache has five main configuration parameters that affect performance: StartServers, MinSpareServers, MaxSpareServers, MaxClients, and MaxRequestsPerChild [20]. StartServers specifies the number of server processes that should be started, MinSpareServers specifies the minimum number of server processes that should be kept in a spare pool, MaxSpareServers specifies the maximum number of server processes that should be kept in the spare pool, MaxClients defines the maximum number of server processes allowed to start, and MaxRequestsPerChild defines the maximum number of requests a server process may serve (0 means infinite). The default configuration assigns values of 5, 5, 10, 150, 0, respectively, to each of these parameters.

Workloads, intensities, and targets. Each user in our synthetic population represents a different combination of workload, load intensity, and response-time target.

We define each workload by its fraction of requests for three types of content: small static files (average size 13KB with 20% cache miss rate), a large static file (130KB in size with 0% cache miss rate), and dynamic CGI scripts (each consuming 14ms of CPU execution). Each of these types of requests stresses a different part of the system: the file system, networking, and the CPU, respectively. We refer to the fraction of CGI requests F_{CGI} as a fraction of the total number of requests. In contrast, we refer to the large-file component of the workload F_{LF} as a fraction of the static requests. The remaining percentage represents the requests to small files. We vary F_{CGI} and F_{LF} from 0% to 100%.

To define load intensities for each workload that do not overload the system, we experimentally find the intensity that leads to saturation assuming the default configuration. We call this the "maximum throughput" for the workload. Then, we assign load intensities for each workload from 50 requests/second to the maximum throughput with a step of 50 requests/second. Since the maximum throughput T_{max} is not always a multiple

of 50, the maximum load intensity L_{max} is

$$L_{max} = \begin{cases} \lfloor \frac{T_{max}}{50} \rfloor * 50 & T_{max}\%50 < 25, \\ T_{max} & T_{max}\%50 \geq 25. \end{cases}$$

For each workload, we select different targets along the reachable range. The targets are evenly distributed between the performance of the default configuration $P_{default}$ and the best performance we can achieve with Simplex $P_{simplex}$. Specifically, the targets are $[P_{default}, P_{default} * 0.95, \cdots, P_{simplex}]$. If $P_{simplex}$ is not exactly a multiple of 5% away from $P_{default}$, the last target we use is the lowest such value that is still higher than $P_{simplex}$. We choose 5% because it creates a good number of targets and poses a non-trivial challenge for configuration tuning.

User populations. We synthetically generated an initial set of "existing users" that is evenly spread in the 3D space of workloads, load intensities, and response-time targets. The existing users are given workloads of the form $F_{CGI} \in (0\%, 20\%, \cdots, 100\%)$ and $F_{LF} \in (0\%, 20\%, \cdots, 100\%)$. The load intensities and response-time targets are selected as described above.

We define the existing users' configurations by running Simplex. In particular, we set Simplex to start from the default Apache configuration and stop trying new configurations (new values for the different configuration parameters) when the response-time target is met. Only this last configuration is stored for each existing user.

The set of "new users" in our synthetic population is also evenly spread across the parameter space, but is completely distinct from the set of existing users. In particular, the new users' workloads are defined as $F_{CGI} \in (10\%, 30\%, \cdots, 90\%)$ and $F_{LF} \in (10\%, 30\%, \cdots, 90\%)$. The load intensities for these users are selected as described above. When setting targets for the new users, we select targets that are achievable by either MassConf or Simplex alone. Our goals are to select configurations for as many of these new users as possible, while using the smallest possible number of experiments on average.

Overall, we create 219 existing users. We start with 31 different workloads. After selecting acceptable load intensities for each of these workloads, we produce 91 combinations. By defining reachable targets for each workload, we get to 219 combinations. We also create 195 new users, starting with 25 different workloads. When load intensities are considered, we reach 66 combinations. Finally, the addition of the reachable targets brings us to 195 combinations.

As one would expect, our population of users is quite diverse. For example, we have a user with F_{SF} = 100% and a load intensity of 400 requests/second that observes a response time of 125 ms, assuming the default Apache configuration. A second user requests F_{LF} = 100% at 87 requests/second for a response time of 278 ms, while a third user requests F_{CGI} = 100% at 80 requests/second for a response time of 174 ms.

Note that our evenly spread and non-overlapping populations of users represent a pessimistic scenario for MassConf. The reason is that any concentration of users in specific parts of the workload-intensity-target space would increase the likelihood that (1) many users would deploy the same configuration; and (2) many users could be satisfied by each configuration. These are the two basic premises behind MassConf, as mentioned in Section 1 and illustrated in Figures 1 and 2.

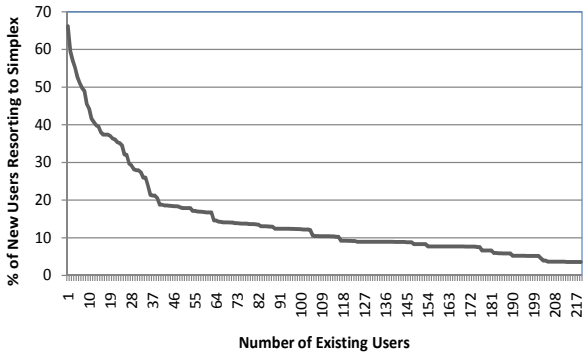

Figure 6: Bootstrapping in MassConf.

Finally, note that, for simplicity, we assume that all users have selected their configurations with the goal of lowering response time and belong to the same environment cluster. (In fact, in our experiments, all users use the same hardware and low-level software environment. Although it would have been interesting to investigate the effect of slight environment variations, this assumption does *not* skew our results. The reason is that, in the real world, there are many more users per environment than environments, just like in our experiments.) Because MassConf operates on clusters independently, our results extrapolate trivially to scenarios with multiple clusters.

Running experiments. All our experiments are run on two Dell 2650 machines. Each machine has one Intel Xeon CPU (2.80 GHz), 2 GB of memory, and a 7200 rpm disk. One machine hosts an HTTP client emulator and the other the Apache Web server (version v2.0.4). Both machines run Linux 2.6.18 and are interconnected by a 100-Mbit Ethernet switch.

Using the client emulator, multiple clients concurrently send requests to the server. During each experiment, a pre-defined workload is sent to the server at a fixed rate, i.e. the pre-defined load intensity for that experiment. The inter-request time follows a Poisson distribution. At the end of each experiment, the emulator reports the average response time and throughput.

3.2 Understanding Bootstrapping

MassConf becomes most useful when the population of existing users has "stabilized" as a good representation of the new users to come, i.e. the probability that MassConf will have to resort to Simplex has become relatively small. This point occurs only when MassConf has gathered enough configurations.

Figure 6 helps us visualize the bootstrapping process. The figure assumes that MassConf is about to configure a new user, after a certain number of existing users have joined the system. The figure plots the probability that MassConf will have to resort to Simplex for the new user, as a function of the number of existing users who have already joined. We compute each probability by assessing the fraction of our new users that would require Simplex to run given the set of existing users. Since the state of MassConf at each point depends on exactly which existing users have joined, we plot the average fraction from 10 different (random) arrival orders.

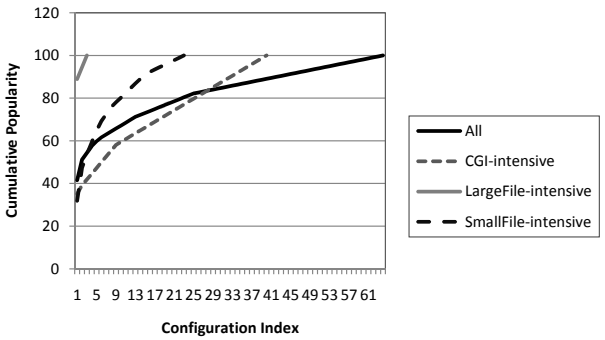

Figure 7: Popularity for CGI-intensive, large-file-intensive, small-file-intensive workloads, and all workloads.

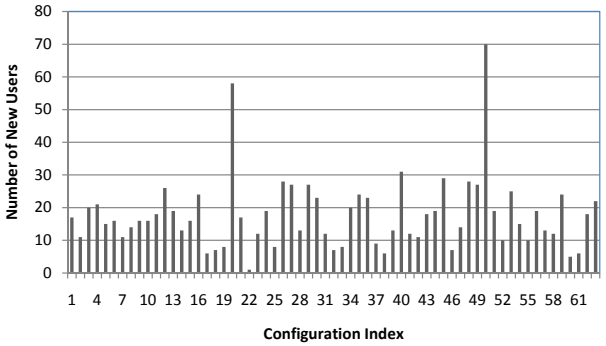

Figure 8: Popularity ranking and number of new users that can be satisfied by each configuration.

As one would expect, the probability of needing Simplex is high when the number of existing users who have joined the system is small. As this number increases, the probability falls sharply. Beyond roughly 37 existing users, the probability of needing Simplex falls below 20%. At that point, we can say that MassConf had been bootstrapped.

3.3 Understanding Ranking

Popularity. Popularity refers to how often an exact set of parameter values appears in a collection of configurations. For example, if the set of values $\{200, 17, 3, 1000\}$ for four relevant parameters appears in 12 out of 20 configurations, we say that this configuration has 60% popularity. Any configuration that appears a smaller percentage of times is considered less popular than this one. Popularity-based ranking ranks configurations based solely on their popularity.

Figure 7 illustrates the popularity of the configurations that met the performance targets for our existing users. The figure plots the popularity for workloads dominated by small files, large files, and CGI requests, as well as the popularity when all workloads are considered together. We define a workload to be CGI-intensive when $F_{CGI} \geq 50\%$. A workload is large-file-intensive when $F_{LF}(1 - F_{CGI}) \geq 50\%$, whereas it is small-file-intensive when $(1 - F_{LF})(1 - F_{CGI}) \geq 50\%$. On the X-axis, the figure shows the index of the unique configurations in decreasing order of popularity (from left to right). On the Y-axis, the figure shows the cumulative popularity of the configurations on the X-axis. The leftmost point of each curve is the default configuration.

The figure confirms one of the basic premises of MassConf, namely that certain configurations work well for many existing users, despite our pessimistic assumptions about the population of existing users. Specifically, we can see that the default configuration indeed works well for a large fraction of users. In addition, the fact that the curves are not straight lines shows that other configurations, besides the default one, are used by multiple users. Moreover, the figure shows that the configuration popularity of the three types of loads is quite different. Large-file-intensive workloads show the least amount of popularity, whereas small-file-intensive workloads show the most. These observations suggest that it is harder to configure the large-file-intensive workloads than others. Nevertheless, MassConf has potential to greatly benefit users with these types of workloads, as shown by the curve that accounts for all workloads together.

The other side of the coin. To fully understand ranking, we have to consider its impact given a population of new users. As we have suggested, ranking configurations based on popularity only may hide the fact that very good configurations just happen to be unpopular. Figure 8 illustrates this effect clearly. On the X-axis, the figure lists the index of each existing configuration in popularity-based order (the most popular on the left, the least popular on the right). Only the default configuration (index #0) is not listed. On the Y-axis, the figure lists the number of new users in our synthetic population that could be satisfied by each configuration.

The default configuration can satisfy the performance targets of 66 new users. As the figure shows, there is another configuration (#50) that can satisfy even more new users (70). Unfortunately, this configuration appears very late, i.e. it is very unpopular. This means that 49 other configurations would be tried before reaching this very good one. A similar observation can be made of configuration #20, which can satisfy 58 new users. Moving either (or both) of these configurations up the ranking would allow many new users to be configured with a smaller average number of experiments. In contrast, the two most popular configurations can only satisfy 17 and 11 new users. These observations clearly suggest that the popularity-based ranking can cause a higher than necessary number of experiments.

3.4 Experimental Evaluation

We now turn to evaluating the use of MassConf for configuring new user installations. We study two scenarios. In the first, we initialize MassConf with configurations from our 219 existing users and use it to configure our 195 new users to meet their response-time targets. In the second scenario, we initialize MassConf with configurations from 1/3 of our existing user population chosen at random. After this initialization phase, we use MassConf to configure our new users and integrate another 1/3 of our existing users (again chosen randomly) at the same time. The order of these user arrivals is also random. To mimic the fact that some users may decide not to join, the final 1/3 of existing users are not used in this scenario.

To evaluate MassConf in these scenarios, we compare its ranking adaptation algorithms to popularity-based ranking. In popularity-based ranking, the ranking changes whenever the selection of an existing configuration causes the popularity or-

Number of Experiments	Popularity Ranking	MassConf Adapt-slow	MassConf Adapt-fast	MassConf Adapt-fastest	Optimal Static
Total	1519	2383	1380	1272	873
Avg.	11.8	18.5	10.7	9.9	6.8
Max.	84	84	84	84	84

Table 1: MassConf vs. popularity for 129 new users.

Number of Experiments	Popularity Ranking wo Simplex	MassConf wo Simplex Adapt-slow	MassConf wo Simplex Adapt-fast	MassConf wo Simplex Adapt-fastest	Optimal Static wo Simplex
Total	1023	1887	884	776	377
Avg.	8.4	15.5	7.2	6.4	3.1
Max.	64	60	59	59	12

Table 2: MassConf-without-Simplex vs. popularity for the 122 new users for which both approaches reached the targets.

dering to change; more popular configurations appear first. New configurations (those found by Simplex) are always added to the end of the ranking, as they are the least popular.

To put the results in context, we also compare them against Simplex (running on its own and starting from the default configuration). In addition, we present results for the "optimal" static ranking, i.e. the static ranking that generates the smallest possible number of experiments in configuring our population of new users. This ranking sorts the configurations in decreasing order of number of (unique) new users that they satisfy; selecting a configuration for a new user does not alter this order. Obviously, the optimal ranking can only be determined because we know the entire set of new users in advance, which is impossible in practice. We present results for the optimal ranking simply as a lower bound on the number of experiments.

Next, we discuss each of the arrival scenarios in turn.

3.4.1 First Scenario

We make several observations from our experiments with the first scenario:

1. MassConf successfully reached the performance targets of all new users. Out of our 195 new users, 66 were able to meet their response-time targets using the default configuration. MassConf was able to configure *all* 129 new users that could not use the default configuration. When MassConf is not allowed to resort to Simplex (MassConf-without-Simplex), it is able to configure 122 of these new users. Two out of the 7 new users that MassConf-without-Simplex cannot configure have light workloads and seek to achieve 5% better performance than the default configuration can produce. The other 5 new users had higher targets and missed them by several percentage points.

2 and 3. Adaptive ranking beats popularity-based ranking. The faster the adaptive algorithm promotes configurations, the better. Table 1 summarizes the statistics for our new users. Table 2 summarizes the statistics for the case in which MassConf (with popularity-based or adaptive ranking) is not allowed to resort to Simplex. For this latter table, we only show results for the 122 new users that MassConf-without-Simplex can configure. Since the behavior of the adaptation algorithms depends on the exact sequence in which new users join the system, for both tables we generated 10 random sequences and averaged the results.

Both tables show that two adaptive ranking approaches (Adapt-fast and Adapt-fastest) require fewer experiments on average than the popularity-based ranking. The analysis of adaptive ranking from the previous section suggested that we would find this result. The faster selected configurations are promoted up the ranking, the smaller the average number of experiments per user. *The best adaptive ranking (Adapt-fastest) runs up to 24% fewer experiments per user than popularity-based ranking on average.* In contrast, Adapt-slow actually requires up to 85% more experiments per user than popularity-based ranking on average. There are two effects at play here: (1) on the positive side, moving a good configuration up enables it to satisfy more users; and (2) on the negative side, it may increase the number of experiments required when a configuration that was moved down is selected. When moving up one slot at a time, only a few extra users can be satisfied by the promoted configuration, so the negative effect becomes more prominent. When moving configurations up faster, the good configurations can satisfy many extra users, making the positive effect more prominent.

The fact that Adapt-fastest is the best approach confirms the two observations that motivated our MassConf design: one configuration works well for multiple users and multiple configurations work well for each user. If only one configuration met each user's target, Adapt-fastest would make the worst decision. At the other extreme, if all configurations met all new users' targets, all approaches would produce the same number of experiments, i.e. 1.

When we compare MassConf Adapt-fastest to the optimal (but unrealistic) static ranking, we find that our system is 31% slower (Table 1). Nevertheless, as shall be seen, MassConf+ actually performs better than this optimal ranking, because it shortens the list of configurations to be tried. (We shall compare MassConf+ to a different optimal static ranking below.)

4. MassConf successfully reached the performance targets for many more users than Simplex. As mentioned above, MassConf was able to configure all 129 new users that could not use the default configuration. In contrast, *Simplex failed to*

Number of Experiments	Simplex Only	Popularity Ranking	MassConf Adapt-slow	MassConf Adapt-fast	MassConf Adapt-fastest	Optimal Static
Total	1023	818	978	762	730	639
Avg.	18.6	14.9	17.8	13.9	13.3	11.6
Max.	112	84	84	84	84	84

Table 3: MassConf vs. Simplex for the 55 new users for which Simplex reached the targets.

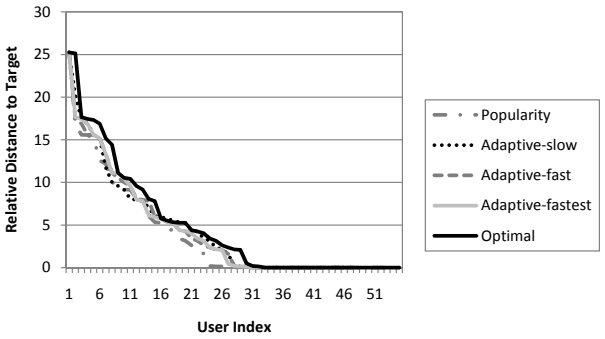

Figure 9: Relative distance to target if Simplex stops when MassConf does.

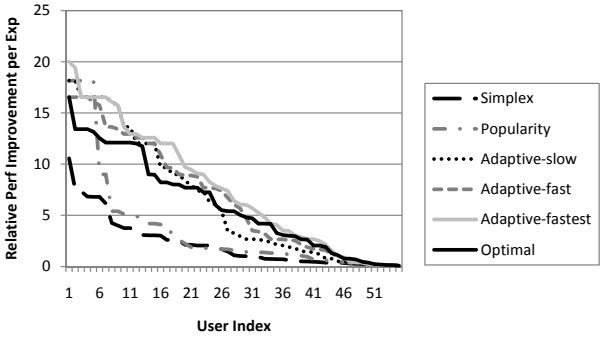

Figure 10: Average relative performance gain per experiment of MassConf and Simplex.

configure 74 of these new users. Even when MassConf is not allowed to resort to Simplex, it still can configure 67 more new users than Simplex (122 vs. 55). The reason Simplex cannot configure these new users is that it gets stuck at local minima, trying configurations that lead to very similar performance.

The ability of MassConf and MassConf-without-Simplex to configure many more new users than Simplex is particularly interesting since our existing user configurations were originally derived using Simplex. This result reinforces the point that Simplex has to search a large space of configurations each time it is used and so, for any particular search, it may miss some "good" configurations. MassConf is completely different in that it is guided by the tuning efforts of existing users and its adaptive ranking algorithms.

5. MassConf is faster than Simplex. To properly compare the number of experiments required by MassConf and Simplex, we consider only the subset of 55 new users for which *both* MassConf and Simplex were able to achieve the performance targets. Table 3 summarizes the statistics for these new users. Again, these results are the average over 10 random sequences. The table shows that the three adaptive ranking approaches require between 13.3 and 17.8 experiments per user on average. The best approach, Adapt-fastest, requires 13.3 experiments on average, which is *28% faster than Simplex.*

6. MassConf produces faster performance for Apache than Simplex. To understand the difference between the approaches in terms of Apache performance, Figure 9 plots the relative distance between the performance achieved by Simplex (R_S) and the target performance (R_T), when Simplex is stopped at the same number of experiments used by MassConf for a randomly selected sequence of new user arrivals. Relative distance is defined as $(R_S - R_T)/R_T$. Users on the X-axis are sorted in order of most-to-least relative distance. The results for MassConf Adapt-fastest show that 10 out of the 55 users would have been 10% to 25% away from reaching their targets. (Relative distances for users 31-55 are 0 because in 5 cases MassConf and Simplex performed the same number of experiments, and in 20 cases Simplex performed fewer experiments.) These results mean that Simplex would lead to substantially worse performance results if it were given the same budget of experiments as MassConf.

Figure 10 shows the average relative performance gain (with respect to the response time achieved by the default configuration) *per experiment* for MassConf and Simplex. This figure shows that our system can achieve substantial performance gains per experiment in many cases. In fact, MassConf Adapt-fastest provides 5-20% performance gain *per experiment* for 32 of the 55 new users. As expected, Simplex's performance gains per experiment are much smaller given the larger number of experiments that it needs.

7. MassConf+ improves significantly on MassConf. MassConf runs a significant number of experiments when the chosen configurations are either at the tail of the ranking or not in the ranking at all. As an example of the latter scenario, MassConf unsuccessfully tries all 64 configurations before resorting to Simplex for the 7 new users' targets that were not met by MassConf-without-Simplex. MassConf+ was designed exactly to reduce the number of unsuccessful experiments in MassConf.

To evaluate MassConf+, we investigated a number of sequences of new user arrivals. The results we discuss next represent a randomly selected such sequence. For that sequence, MassConf+ decided to cut off the tail of the ranking after having seen 28 new users. (Recall that MassConf+ selects this point after having seen 10 consecutive decreases in average number of experiments per new user.) After the 28th new user was configured, MassConf+ also decided to cut the ranking off at the 14th configuration. (Recall that the cut off point is the

Number of Experiments	Popularity Ranking	MassConf Adapt-fastest	MassConf+ Adapt-fastest	Optimal+ Static
Total	1529	1262	793	528
Avg.	11.9	9.8	6.1	4.1
Max.	84	84	37	34

Table 4: MassConf+ vs. MassConf for 129 new users. The popularity and MassConf results here are different than in Table 1 because this table only considers one sequence of arrivals.

Number of Experiments	Simplex Only	Popularity Ranking	MassConf Adapt-fastest	MassConf+ Adapt-fastest	Optimal+ Static
Total	1023	818	734	336	294
Avg.	18.6	14.9	13.3	6.1	5.3
Max.	112	84	84	34	34

Table 5: Simplex vs. MassConf+ vs. MassConf for the 55 new users for which Simplex reached the targets. The popularity and MassConf results here are different than in Table 3 because this table only considers one sequence of arrivals.

minimum size that was required to satisfy 80% of the 28 new users.) Starting with the 29th new user, MassConf+ only tries a maximum of 14 configurations for each new user before resorting to Simplex.

Using these thresholds, MassConf+ is able to find configurations that meet the targets of *all* the 129 new users that cannot use the default configuration. Table 4 summarizes the results of MassConf+, while comparing them against popularity-based ranking and MassConf. The Optimal+ system ranks configurations in the best possible order and cuts the ranking off at the optimal point (12 configurations). Again, Optimal+ is obviously unrealistic and is presented simply as a lower bound on the number of experiments.

The table shows that MassConf+ reduced the overall number of experiments by 469, compared to MassConf. As a result of this reduction, *MassConf+ finds configurations 37% and 48% faster than MassConf and popularity-based ranking, respectively, on average.* In addition, it cuts the maximum number of experiments for any new user to less than half of those performed by MassConf and popularity-based ranking. Comparing these MassConf+ results with the optimal ranking results of Table 1, we can see that our system actually performs better. The reason is that MassConf+ prevents a large number of experiments by cutting off the ranking. Compared to Optimal+ ranking, MassConf+ incurs 33% more experiments.

Table 5 compares MassConf+, MassConf, and Simplex for the 55 new users for which Simplex reached the performance targets. The table shows that *MassConf+ finds configurations 67% faster than Simplex*, again while significantly reducing the maximum number of experiments for any new user.

To understand the impact of its two thresholds, we performed a number of sensitivity experiments with MassConf+ Adapt-fastest, assuming the entire set of 129 new users and the same sequence of new user arrivals. For the first threshold, we considered 5 and 15 continuous decreases of the average number of experiments per new user, besides the default setting of 10 continuous decreases. For the second threshold, we considered cutting off the list at the size that would satisfy 70% and 90% of the new users, besides the default setting of 80%.

When the first threshold is set to 5, the numbers of new users resorting to Simplex are 88, 88, and 71, when the second threshold is set to 70%, 80%, and 90%, respectively. However, regardless of the setting of the second threshold, MassConf+ does *not* reach the targets of all new users. The reason is that the ranking had not been trained enough before it was cut off. In contrast, when the first threshold is 15, MassConf+ always reaches the new users' targets. For this first threshold, the total numbers of experiments for the different settings of the second threshold are 604, 607, and 846, respectively. In this case, MassConf+ achieves low experiment totals for 70% and 80% settings of the second threshold. Overall, these results suggest that it is most efficient to select a relatively low value for the second threshold (e.g., 70%), as long as the ranking is trained for long enough by picking a relatively high value for the first threshold (e.g., 10 or higher). In fact, note that picking a value of 15 for the first threshold would lead to significantly better MassConf+ results than those in Tables 4 and 5.

3.4.2 Second Scenario

One could argue that the results of our first scenario above were optimistic in the sense that all existing users joined MassConf before any new user had to be configured. To evaluate MassConf in more pessimistic circumstances, we now turn to our second scenario. Recall that, in this scenario, we initialize MassConf with configurations from only a random 1/3 of our existing user population. After this initialization, the new users start arriving concurrently with another random 1/3 of the existing users. The final 1/3 of the existing users never joins.

From this scenario, we can make the following observations:

8 and 9. The benefits of MassConf and MassConf+ remain significant. The comparisons between systems exhibit the same trends as before. Table 6 compares MassConf and MassConf+ with popularity-based ranking and Optimal+ ranking for our population of 129 new users. These results confirm the trends we observed from the first scenario. Specifically, (1) both MassConf and MassConf+ can configure all new users, even in the absence of a large fraction of existing users; (2) MassConf+ reduces the number of experiments by 42%, compared to MassConf; (3) MassConf and MassConf+ involve 14%

Number of Experiments	Popularity Ranking	MassConf Adapt-fastest	MassConf+ Adapt-fastest	Optimal+ Static
Total	1447	1250	717	528
Avg.	11.2	9.7	5.6	4.1
Max.	70	66	37	34

Table 6: MassConf+ vs. MassConf for 129 new users in the second scenario.

Number of Experiments	Simplex Only	Popularity Ranking	MassConf Adapt-fastest	MassConf+ Adapt-fastest	Optimal+ Static
Total	1023	747	702	365	294
Avg.	18.6	13.6	12.8	6.6	5.3
Max.	112	70	66	37	34

Table 7: Simplex vs. MassConf+ vs. MassConf for the 55 new users for which Simplex reached the targets in the second scenario.

and 50% fewer experiments than popularity-based ranking, respectively; and (4) MassConf+ runs only 27% more experiments than the unrealistic Optimal+ ranking algorithm.

Moreover, we can see that the MassConf, MassConf+, and popularity-based ranking *results are actually better in absolute terms than those for the first scenario* (Table 4). The reason is that some of the configurations that cannot help many new users are never tried, as the corresponding existing users either join the system too late or not at all.

Table 7 compares MassConf and MassConf+ to Simplex for the 55 new users that the latter was able to configure. First, note that MassConf and MassConf+ can configure many more new users than Simplex, even under adverse conditions that have no effect on Simplex. In addition, the table shows that our systems perform 31% and 65% fewer experiments than Simplex, respectively. Again, these results exhibit the same trends as we observed with the first scenario.

4 Extrapolating Beyond The Case Study

The results from the previous section are very positive, but they are specific to our Apache case study. In this section, we qualitatively extrapolate from them by *abstracting away* the server software, the high-level tuning goal, the workloads, the load intensities, and the target behaviors.

The extrapolation is based on the observation that three aspects of the user-configuration space matter most in determining the number of experiments for a particular sequence of new user arrivals: (1) the number of new users that can be satisfied by each configuration; (2) the number of configurations that can satisfy each new user; and (3) the number of new users for which MassConf would have to resort to Simplex.

Adapt-fastest and Adapt-fast behave better than Adapt-slow and popularity-based ranking when there is significant potential for re-use of the configurations that are promoted forward in the ranking. This potential increases when aspects #1 and #2 above are skewed towards a subset of configurations and new users, respectively, and the tails of the distributions are short. In other words, the re-use potential increases when (a) a significant fraction of configurations can satisfy many new users and (b) a significant fraction of new users can be satisfied by many configurations. When the amount of skew is limited or tails are long, Adapt-slow should perform best.

Compared to Simplex, MassConf can benefit from configuration re-use to achieve a lower average number of experiments per new user. Moreover, only a small fraction of new users should require MassConf to resort to Simplex (aspect #3), since the existing and new user populations should have the same characteristics (after bootstrapping).

In our Apache study, we saw significant skew, short tails, and limited use of Simplex by MassConf. Specifically, aspect #1 can be approximated as a power-law function, in which the configuration that can satisfy the most new users satisfies 70 out of 129 such users. Aspect #2 of our Apache study can be approximated as an exponential function, in which the new user that can be satisfied by the most configurations can be satisfied by 52 out of 64 configurations. MassConf had to resort to Simplex for only 7 new users.

We expect real user populations to benefit from MassConf even more than in our Apache study. The reason is that our synthetic population is evenly spread across the workload-intensity-target parameter space; greater concentration in part of the space would increase the potential for configuration re-use. With high potential for re-use, either Adapt-fastest or Adapt-fast would be a good choice; the vendor can select the best approach after configuring a number of new users.

5 Related Work

We now discuss works in the two areas related to this paper: leveraging existing users' information or knowledge for configuration, and tuning of the configuration of server software.

Leveraging existing data on configurations. Several previous works have investigated how to leverage others' configurations to diagnose and troubleshoot misconfigurations [1, 19, 22, 23].

Even though MassConf also relies on configuration information from a population of users, it focuses on a completely different problem: configuration tuning; there are no misconfigurations to troubleshoot. As mentioned more extensively in Section 2, the impact of this key difference is that our main focus has been on issues that have not been addressed before, namely the study of adaptive ranking algorithms and the average number of experiments to which they lead.

Configuration tuning. Many works have considered the performance tuning of server configurations, e.g. [5, 6, 8, 9, 17, 21, 25]. Osogami *et al.* [15, 16] focused on shortening each experiment, rather than reducing the number of experiments.

MassConf differs from these works in four main ways: (1) it seeks to produce configurations that meet the users' target behaviors, rather than to find the best possible configuration; (2) it relies on configuration information from a population of systems, rather than a single system; (3) it relies on adaptive ranking algorithms to tune performance efficiently; and (4) unless it needs to resort to Simplex, it tests existing configurations for new users, rather than trying to use experience or dependencies to create new configurations.

6 Conclusions

In this paper, we addressed the problem of configuring enterprise software efficiently. Specifically, we proposed MassConf, a system that uses existing configurations to automatically configure the software for new users. The configuration process relies on dynamic adaptation of the order of configurations (ranking) to be tried. To evaluate MassConf, we used it to configure Apache for performance for a population of users. Our results compared three ranking adaptation algorithms to popularity-based ranking. The results showed that our fastest adaptation leads to the smallest number of experiments. The results also showed that MassConf is able to configure more users in fewer experiments than Simplex, an efficient optimization algorithm.

Our future work will address the potential benefits of MassConf for multi-tier services, rather than stand-alone servers. In particular, we will investigate whether configurations exhibit strong popularity across these systems and what is the best ranking approach for new service installations.

References

[1] B. Aggarwal et al. NetPrints: Diagnosing Home Network Misconfigurations Using Shared Knowledge. In *Proceedings of NSDI '09*, April 2009.

[2] P. Anderson et al. LCFG: The Next Generation . In *UKUUG Winter Conference*. UKUUG, 2002.

[3] P. Anderson et al. SmartFrog meets LCFG: Autonomous Reconfiguration with Central Policy Control. In *Proceedings of LISA '03*, October 2003.

[4] M. Burgess. Cfengine: A Site Configuration Engine. *USENIX Computing systems*, 8(3), 1995.

[5] H. Chen et al. Experience Transfer for the Configuration Tuning in Large Scale Computing Systems. In *Poster Session of SIGMETRICS '09*, June 2009.

[6] I. Chung et al. Automated Cluster-Based Web Service Performance Tuning. In *Proceedings of HPDC '04*, June 2004.

[7] O. Crameri et al. Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System. In *Proceedings of SOSP '07*, October 2007.

[8] Y. Diao et al. Managing Web Server Performance with AutoTune Agent. *IBM Systems Journal*, 42(1), 2003.

[9] S. Duan et al. Tuning Database Configuration Parameters with iTuned. In *Proceedings of VLDB '09*, August 2009.

[10] L. J. Heyer et al. Exploring Expression Data: Identification and Analysis of Coexpressed Genes. *Genome Research*, 1999.

[11] L. Keller et al. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *Proceedings of DSN '08*, June 2008.

[12] E. Kiciman et al. Discovering Correctness Constraints for Self-Management of System Configuration. In *Proceedings ICAC '04*, May 2004.

[13] K. Nagaraja et al. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of OSDI '04*, December 2004.

[14] J. A. Nelder et al. A Simplex Method for Function Minimization. *Computer Journal*, 7(4), 1965.

[15] T. Osogami et al. Finding Probably Better System Configurations Quickly. In *Proceedings of SIGMETRICS '06*, June 2006.

[16] T. Osogami et al. Optimizing System Configurations Quickly by Guessing at the Performance. *SIGMETRICS Perform. Eval. Rev.*, June 2007.

[17] A. Saboori et al. Autotuning Configurations in Distributed Systems for Performance Improvements Using Evolutionary Strategies. In *Proceedings of ICDCS '08*, June 2008.

[18] C. Stewart et al. Performance Modeling and System Management for Multi-component Online Services. In *Proceedings of NSDI '05*, May 2005.

[19] Y. Su et al. AutoBash: Improving Configuration Management with Operating System Causality Analysis. In *Proceedings of SOSP '07*, October 2007.

[20] The Apache Software Foundation. Apache HTTP Server Version 2.0. http://httpd.apache.org/docs/2.0/mod/prefork.html.

[21] R. Thonangi et al. Finding Good Configurations in High-Dimensional Spaces: Doing More with Less. In *Proceedings of MASCOTS '08*, September 2008.

[22] H. J. Wang et al. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of OSDI '04*, December 2004.

[23] Y. Wang et al. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of LISA '03*, October 2003.

[24] A. Whitaker et al. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of OSDI '04*, December 2004.

[25] W. Zheng et al. Automatic Configuration of Internet Services. In *Proceedings of Eurosys '07*, March 2007.