

An Optimization Framework for Web Farm Configuration

David Bartholomew Stewart

Microsoft Research Limited
7 JJ Thompson Avenue
Cambridge, CB3 0FB, UK
+44 1223 479718

davidst@microsoft.com

Efstathios Papaefstathiou

Microsoft Research Limited
7 JJ Thompson Avenue
Cambridge, CB3 0FB, UK
+44 1223 479768

efp@microsoft.com

Jonathan Hardwick

Microsoft Research Limited
7 JJ Thompson Avenue
Cambridge, CB3 0FB, UK
+44 1223 479769

jch@microsoft.com

ABSTRACT

A common problem that sales consultants face in the field is the selection of an appropriate hardware and software configuration for web farms. Over-provisioning means that the tender will be expensive while under-provisioning will lead to a configuration that does not meet the customer criteria. Indy is a performance modeling environment which allows developers to create custom modeling applications. We have constructed an Indy-based application for defining web farm workloads and topologies. The paper presents an optimization framework that allows the consultant to easily find configurations that meet customers' criteria. The system searches the solution space creating possible configurations, using the web farm models to predict their performance. The optimization tool is then employed to select an optimal configuration. Rather than using a fixed algorithm, the framework provides an infrastructure for implementing multiple optimization algorithms. In this way, the appropriate algorithm can be selected to match the requirements of different types of problem. The framework incorporates a number of novel techniques, including caching results between problem runs, an XML based configuration language, and an effective method of comparing configurations. We have applied the system to a typical web farm configuration problem and results have been obtained for three standard optimization algorithms.

Keywords

Optimization, Modeling, Infrastructures, Indy, Measurement, Performance, Design, Experimentation, Simulation.

1. INTRODUCTION

Large web services typically run on clusters of machines, so that they can handle the load of many clients. When people design these services they typically use rules of thumb to decide how to provision the system. Wrong decisions can lead to poor performance or excessive cost. Indy is a performance modeling environment that can simulate clusters used for web farms, making it possible to estimate the performance of a web farm before making costly purchasing decisions. However, a consultant

commissioning a web farm must still try many different configurations; this paper describes a framework that automatically searches through possible configurations, attempting to find a solution that meets criteria that the purchaser has specified.

We designed the optimization framework so that we could quantify a number of different optimization algorithms' performance when finding the best configuration. The framework allows these algorithms to be implemented with the minimum of problem-specific knowledge and statistical expertise. The framework was specifically designed to be used to optimize Indy configurations, but is also usable for different simulation environments. We will next describe Indy, and then discuss the sorts of problem that we can use the framework to solve.

1.1 Indy

Indy is a general-purpose toolkit that can be used to create specialized tools for individual modeling purposes, and is based on the concept of *modeling infrastructures* [17]. Furthermore, it is not limited to a fixed set of components – users can contribute new components to extend the toolkit's capabilities and the range of tools that can be produced using it (e.g. [12]).

Indy differentiates between *tool developers* and *end users*. A tool developer interacts directly with the modeling infrastructure, choosing from a set of parameters (modeling technique, tool role, abstraction level, output method, etc) to produce a tool for a particular role. The developer may choose to plug together preexisting components supplied in a library as part of the infrastructure, or to develop new components to further extend the capabilities of the infrastructure. An end user then uses the resulting tool to solve a particular performance problem. They can modify problem parameters, workloads, and configurations within the limits set by the developer of the tool, but cannot further extend it.

Indy provides the following capabilities to tool developers:

1. *Interfaces*: The developer can incorporate their own workload descriptions, device models, and problem configurations for creating a custom modeling application.
2. *Integration Mechanisms*: The Indy kernel is responsible for regulating the data flow between the components of the model. It also calculates the overall performance of the system by integrating individual device predictions.

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

WOSP '02, July 24-26, 2002 Rome, Italy

© 2002 ACM ISBN 1-1-58113-563-7 02/07 ...\$5.00

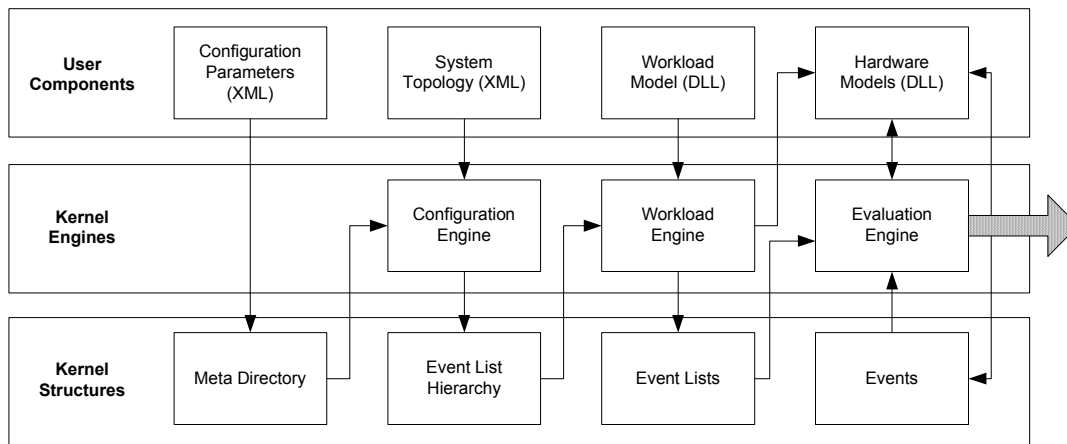


Figure 1: Indy Architecture Diagram

3. *Abstractions*: To simplify the model development process and provide a shared infrastructure for modeling tools, Indy defines a number of abstractions, including ones for configuring the system topology, configuring devices, and defining workloads.
4. *Services*: Indy is responsible for calculating the most complex parts of the model including resource contention and queuing delays. These services allow developers with little modeling background to create sophisticated applications.

Individual Indy components are Win32 DLLs that can be dynamically loaded into a running process. The components communicate either via XML according to standardized schemas, or via well-defined C++ APIs. The kernel is a linkable library containing the algorithms and data structures necessary for the evaluation of performance models, and for the coordination of the whole system. Any required user interface can be put on top of this kernel. One example is IndyView, a front-end to the Indy system that can be used both as a development environment for the creation of new tools, and as a production environment to run them in.

A diagram of information flow in the Indy architecture is shown in Figure 1. As shown, a tool created with the Indy system adds the following user-contributed components to the kernel:

- *Configuration parameters* are simple name/value variables that are defined by the creator of a performance study, and can then be adjusted by a user to modify the behavior of a model.
- The *system topology* lists the devices in the system being modeled, their interconnections, and the hardware models that correspond to these devices.
- A *hardware model* describes the behavior of a device listed in the topology. Note that “device” is used here in a loose sense, in that it may be a virtual entity (such as a thread) instead of a physical one (such as a CPU or disk). A device represents an active entity that provides a certain capability – for example, a disk can process disk operations, and a network interface can process incoming and outgoing messages.

- The *workload model* is the most complicated part of a performance study. It defines the flow of events through the system being modeled. The workload model therefore combines the tasks of generating the simulated input, and describing the causal relationships between events that take place on system devices because of that input.

At the next level down, the Indy kernel itself includes three engines that hide the complexity of the internal modeling algorithms by providing well-defined abstractions.

The *system configuration engine* processes the system topology script and creates an *event list* for each of the active devices modeled in the system. An event is the internal representation of an action being performed, such as computation, communication, or disk I/O. The event lists are then populated by the *workload engine*, which determines which events are run on which hardware devices, taking into account the incoming timelines from the workload generator, the system topology, and the underlying hardware models.

Finally, the *evaluation engine* coordinates the evaluation of each of the events using their assigned hardware models, and combines individual event timings to determine the overall performance of the system. The current evaluation engine uses event-based simulation together with novel scheduling and resource tracking algorithms that can efficiently model the contention happening on a real system.

The result of the evaluation is an output trace showing the interactions between the components of the system as it processes the workload. This can then be visualized directly using a tool such as IndyView, or it can be post-processed for use by other tools.

The next section describes the problems that we wish to solve, using optimization algorithms and the Indy toolkit.

1.2 Problem

Our motivation for building an optimization tool is to allow consultants to find the best configuration for a web farm to cope with a given workload, whilst meeting constraints that clients may

specify. In later sections, we will look at a simple example problem, and see how a variety of algorithms solve it.

In defining a problem, clients typically express constraints on system values such as response times, the utilization of system components, or its overall price. Given these constraints, they wish to maximize or minimize another property of the system, the objective function. They are also interested in the steady-state behavior of the system, and wish to ignore the transient effects of workloads starting up or winding down.

The constraints and objectives that clients specify can be very diverse, and the tool must cope with this: it may suggest a useless solution if not all constraints can be specified. Less obvious constraints include only buying from certain vendors, or constraining the amount of rack space used by the solution. A flexible specification of input parameters, constraints and objectives is required to express all these problems.

Different problems may also be suited to different optimization algorithms, depending on how many dimensions the problems have, and how changes in parameters cause changes in results. Our framework provides a simple API for implementing optimization algorithms, so each problem can use an appropriate strategy.

The rest of this paper is organized as follows: Section 2 describes previous work in the field of optimization frameworks. Section 3 explains the optimization framework that we designed to optimize web farm configurations. Section 4 describes the algorithms that we have implemented within this framework, and compares their applicability to the problem. Section 5 presents the results of using these algorithms to solve a sample problem, and compares the algorithms in terms of quality of solution and number of simulations needed. Finally, Section 6 summarizes the work and describes future directions.

2. RELATED WORK

Much of the prior work on optimizing problems where results are available only through discrete event simulation has concentrated on producing new optimization strategies. This paper is about the construction of a framework in which optimization algorithms can be implemented so that a variety of methods can be easily used to optimize a discrete-event simulation model.

A framework for optimizing models that are simulated using terminating simulations was designed and evaluated by Joshi et al [14]. Terminating simulations are those of situations that have a natural termination point, so the transients at the beginning and end of the simulation are part of the valid results; in web farm simulations we are interested in the steady state behavior, not the transient behavior.

The problem that Joshi et al investigate is otherwise similar to that described here: the simulation results must meet a certain set of constraints, and must minimize the value of the objective function. Their methods are likewise similar: they transform their constraints from probabilistic to deterministic using the method of chance constraints [5], and they provide a tool to compare simulation results. Our framework provides much more functionality: for example, we provide tools for modifying simulation input, and insulation of the optimization algorithms from constraints and objective functions.

Thompkins is working on an optimization framework for simulated models that provides an interface between various statistical tools, optimization tools and simulators [19]. Rather than attempting to interface between a wide variety of disparate tools we provide specialized statistical code and interface to a single simulator. The environment that we provide makes it easy to implement optimization algorithms.

“Scenario Seeker” has a similar framework to that described here [3]. This framework separates the problem into three phases: problem specification, solution generation, and selection of the best solution. The last two phases are performed multiple times until an acceptable solution has been found. As in the work described here they have clear boundaries between the “solution generator” (optimization algorithm), the simulator and the comparison of results. They also hold the results in a database so that simulations are not run unnecessarily.

However their framework uses a particular simulator (“AweSim!”), cannot re-use results between different problems, and provides only one style of optimization using genetic algorithms. Additionally, their problem specification does not include constraints, only an objective function. Objective functions are implemented in C++, and therefore can be any function of simulation outputs. For our purposes, requiring consultants to be proficient in C++ is optimistic.

A number of approaches allow only a single type of optimization strategy. Neddermeijer et al. produced a framework for optimization using variants of Response Surface Methodology (RSM) [16]; Chong et al. simulated network control algorithms and using hindsight guess at optimal parameter values [6].

Gehlsen & Page designed a framework for distributed simulation optimization that allows the use of many optimization algorithms, but they prototype only genetic algorithms [9]. Their system provides less functionality than ours: they require the user to specify start and end transient times, they do not ensure results are not generated unnecessarily, and they do not support constraints.

In summary: previous frameworks have not provided all of the facilities described here. No framework has enabled problems to share results, or has been designed to use multiple simulators; and many have not allowed sufficiently complex problem specification for all the problems that we need to solve. The following section describes our optimization framework, designed to use any simulator and any optimization algorithm.

3. OPTIMIZATION FRAMEWORK

The framework we have designed allows optimization algorithms to be implemented with a minimal amount of understanding of the problem to be solved, the simulator that is used to solve it, and the statistical knowledge needed to compare the results. It also enables a variety of simulators to be used with only a small amount of additional implementation work. To achieve this, the framework cleanly separates the following functions, each of which is implemented by a separate module:

1. *Problem Manager*: handles problem specification and processes results.
2. *Solution Manager*: compares solutions and generates results.

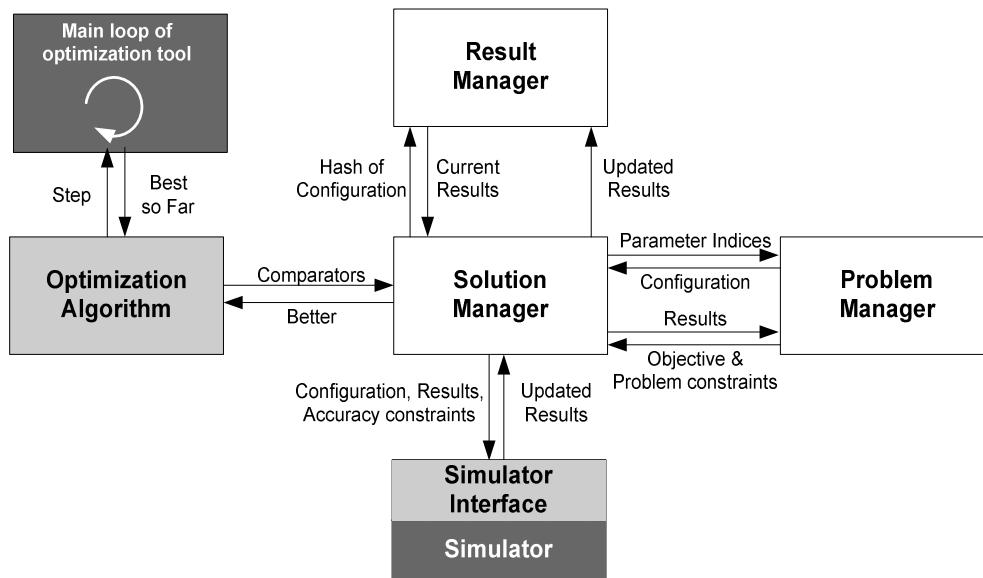


Figure 2: Optimization Framework

Table 1: Information contained in XML problem specification

Information	Description	Example
Input parameters and values	A name that appears in the simulator configuration files, and the list of values that it can be substituted with.	"Web server CPU" = "450MHz Pentium", "500MHz Pentium", "750MHz Pentium"
Variables	A name that appears in the simulator configuration files, and a function of the parameters that it is substituted with.	"# Database servers" = 16 - "# Web servers". "#Web servers" is an input parameter. The function lets us specify a maximum size for the web farm.
Constraints	Constraints on functions of input parameters and simulator outputs.	"Web server CPU Utilization" < 60% "Web server CPU Utilization" is an output from the simulator.
Objective Function	A function of input parameters and simulator outputs to be maximized or minimized.	Minimize "Response Time" "Response Time" is an output from the simulator.

3. *Simulator Interface*: provides a generic interface to simulators.
4. *Result Manager*: stores the simulation results.
5. *Optimization Algorithm*: selects possible solutions.

Figure 2 shows how these modules interact; the white boxes are modules which provide generic functionality; the light grey boxes represent pieces that can be implemented in a number of ways (the simulator interface and the optimization algorithm); and the dark grey boxes represent external components (the actual tool driving the optimization, and the simulator which is being used).

The framework and modules have been implemented in C++, and currently optimization algorithms and simulator interfaces also must be implemented in that language. The modules are discussed in the following subsections.

3.1 Problem Manager

The problem manager parses the problem specification, creates simulator configurations, and compares simulation results. The XML problem specification contains four types of information listed in Table 1.

Using the problem specification, and a set of skeleton simulator configuration files, the problem manager can then create valid simulator configurations from parameter values, and objective and constraint values from results.

The optimization algorithms and solution manager are unaware of the meaning of optimization parameters; they just see the solution space as having a number of dimensions, each of which can take a number of values. They specify the parameter values to the problem manager by the index of the value in the list supplied in the problem specification. For example if the problem had just one parameter, that used in the above table, to set "Web

server CPU” to “500MHz Pentium” the solution manager would ask for a configuration in which the first parameter had its second value.

The variables and parameters are substituted into a skeleton set of simulator configuration files; these contain the name of the parameters and variables wherever the values should be placed.

The optimization algorithms and solution manager are also unaware of the mechanisms and simulator outputs used to calculate the constraints and objective values. When the solution manager wishes these values, it passes the results gained from running simulations. The results are a set of pairs of output variable name and value; the value consists of the mean observed value, its standard deviation and the number of degrees of freedom it has. The problem manager then calculates the values of the constrained functions and the objective function, again with their standard deviation and degrees of freedom. The solution manager can then compare these using confidence intervals.

3.2 Solution Manager

The solution manager is the core of the optimization framework: it interacts with all the other components on behalf of the optimization algorithm, to which it provides a simple API. It decides when simulations need to be run, and deals with comparing the relative merits of different points in the solution space. The API provided to the optimization algorithms consists of two sets of functions:

1. Those that query the size of the search space: the number of parameters, and the number of possible values for each.
2. Those that compare possible positions in the solution space to see which is better.

These functions allow optimization algorithms to be implemented easily without needing to deal with managing results, or understand the problem or the simulators.

The values of the problem’s input parameters define points in the solution space. As stated previously, these are represented by the indices of the values each parameter takes.

The solution manager presents the optimization algorithms with the illusion that they are just attempting to minimize the value of the results by providing =, >, <, > comparators. The problem however has constraints as well as an objective, so results are compared in a novel way.

Constraints must be met with a particular confidence, so the solution manager transforms these stochastic constraints into deterministic constraints using the method of chance constraints [5]. All constraints must be met, so we place an arbitrary ordering on the constraints, and we compare the constraints in that order. Consider a problem with simulation results r and with n constraints, $l_i(r) < h_i(r), 1 \leq i \leq n$ and objective function $f(r)$; without loss of generality we wish to minimize f (if we wish maximize it we simply reverse some of the inequalities). $f(r)$, $l_i(r)$, $h_i(r)$ are all functions of the results and inputs to the simulator. The solution manager deems a point r_1 in the solution space better, i.e. less than, another point r_2 using the following rules:

1. $(\forall i, 1 \leq i \leq n, ((l_i(r_1) < h_i(r_1)) \wedge (l_i(r_2) < h_i(r_2)))) \wedge (f(r_1) < f(r_2)) \Rightarrow (r_1 < r_2)$, i.e., if both solutions satisfy all constraints then the objective function distinguishes the points.
2. $(\exists i_2, 1 \leq i_2 \leq n, (l_{i_2}(r_2) \geq h_{i_2}(r_2))) \wedge \neg(\exists i_1, 1 \leq i_1 \leq n, (l_{i_1}(r_1) \geq h_{i_1}(r_1))) \Rightarrow (r_1 < r_2)$, i.e., if r_1 meets all constraints but r_2 does not then r_1 is better.
3. $(\exists i_2, 1 \leq i_2 \leq n, (l_{i_2}(r_2) \geq h_{i_2}(r_2))) \wedge (\forall i, 1 \leq i < i_2, (l_i(r_2) < h_i(r_2))) \wedge (\exists i_1, 1 \leq i_1 \leq n, (l_{i_1}(r_1) \geq h_{i_1}(r_1))) \wedge (\forall i, 1 \leq i < i_1, (l_i(r_1) < h_i(r_1))) \wedge (i_1 > i_2) \Rightarrow (r_1 < r_2)$ i.e., if r_1 meets more early constraints than r_2 then r_1 is better.
4. $(\exists b, 1 \leq b \leq n, (l_b(r_1) \geq h_b(r_1)) \wedge (l_b(r_2) \geq h_b(r_2))) \wedge (\forall i, 1 \leq i < b, ((l_i(r_1) < h_i(r_1)) \wedge (l_i(r_2) < h_i(r_2)))) \vee ((l_i(r_1) \geq h_i(r_1)) \wedge (l_i(r_2) \geq h_i(r_2))) \wedge ((l_i(r_1) - h_i(r_1)) = (l_i(r_2) - h_i(r_2)))) \wedge ((l_b(r_1) - h_b(r_1)) < (l_b(r_2) - h_b(r_2))) \Rightarrow (r_1 < r_2)$ i.e., if r_1 and r_2 meet the first $b-1$ constraints equally well, then the amount that they fail to meet constraint b by is used to distinguish the points.

When the optimization algorithm compares two points, the solution manager performs the following steps:

1. For each of the points:
 - a. Use the problem manager to get the simulator’s configuration with the parameter values inserted.
 - b. Hash the simulator configuration.
 - c. Get the current result set from the result manager (may be empty if this is the first request for this set of results).
 - d. Use the problem manager to get the constraint and objective function values from the results.
2. Compare the points in the solution space using the rules described above. If the results are too inaccurate to differentiate these points, and the accuracy is below a certain threshold, then it is necessary to run more simulations to differentiate the results. If the accuracy is above the threshold then the solution manager deems the points indistinguishable.
3. Estimate the accuracy of results required to distinguish the points.
4. For each of the parameter sets:
 - a. Pass the configuration, this constraint, and the current set of results to the simulator interface; run simulations to improve the accuracy.
 - b. Get the new result set from the simulation.
 - c. Go to 1d.
5. Pass the updated results back to the result manager.

The solution manager returns the best solution if results can be generated to sufficient accuracy to distinguish them; otherwise

it returns them as being indistinguishable to within a certain threshold of accuracy.

By providing $<$, $>$, $=$, $>$, $=$, and C which can differentiate between solutions which fail to meet the same constraints, optimization algorithms such as hill-climbing algorithms can be used. Without this mechanism, too many points in the space would be indistinguishable, and only algorithms that scatter solutions around the space would work well.

3.3 Simulator Interface

The simulator interface insulates the rest of the system from the particular simulator being used, though only an interface to Indy has been implemented.

When the simulator interface is initialized, it initializes all parts of the simulator aside from that which can be changed by selecting parameter values. When points in the solution space are compared the solution manager supplies the remainder of the configuration, which is produced by the problem manager. The simulator interface is then asked to run simulations, refining a set of results, until constraints on their accuracy are met. Currently we only run a fixed number of simulations, though it is possible to estimate how many runs of what length are required to reach a particular accuracy [4]. If multiple simulations are required then the interface may run these in parallel on several machines.

Before aggregating the results from a simulation with the results from previous runs, the periods of transient behavior at the start and end of the simulation must be removed, as it is the steady state behavior that is of interest. Code shared between all interfaces performs this operation. The transient period at the end of a simulation starts after the last operation in the workload is issued; at this point, the amount of work being serviced by the simulated system begins to decay. There are also mechanisms for identifying the length of the transient behavior at start of the simulation; we use changes in the mean of the outputs to identify when the behavior has reached a steady state [15].

3.4 Result Manager

The result manager stores the results of all simulations. Results are sets of name-statistic pairs, and are stored in a hash table keyed by a hash of the simulator configuration. When results for a particular configuration are requested, the optimization algorithm queries the result manager, which returns the results of all the simulations that have been run with that configuration so far. Caching the results of simulations in this way avoids unnecessary simulation runs.

These results are then used by the problem manager to calculate the constraint values and the objective function. When the solution manager compares the results from different configurations, they may be too inaccurate to determine which is better. If this is the case then the solution manager may run more simulations, aggregating the new results with the old. It then passes the old results to the result manager to store.

Since the result manager stores results based on the simulator configuration, and can store all the outputs of the simulator, these results can be shared between problems.

3.5 Optimization Algorithm

Any discrete space optimization algorithm can be implemented in our framework by implementing two functions: *Step* performs a

unit of optimization work and *Best* returns the best result found so far.

The algorithm simply picks points in the solution space and compares them; the solution manager does the rest. The solution manager informs it of the size of the search space by making calls to the problem manager. The algorithms that have been implemented are described later in this paper.

3.6 Summary

The framework consists of five modules, each of which provides a small and well-defined set of functionality. The framework allows easy integration with different simulators and different optimization algorithms, using simple APIs.

The framework has been implemented and tested on real-world problems, although the current tool is partially dependent on Indy. Those dependencies will be removed in the near future.

4. ALGORITHMS

We have implemented four standard optimization algorithms in this framework. Two simple algorithms, one of which evaluates all possible solutions and another which alters a single parameter at a time, are used as baselines. The two more complex algorithms are Tabu Search [10,11] and Revised Simplex Search [13].

4.1 Evaluate all solutions

Each step of this algorithm evaluates one more point in the solution space and compares it with the current best result. This is guaranteed to find the correct result; it also adds all the points in the solution space to the result manager. The other algorithms used the cached results generated by this algorithm.

4.2 One parameter at a time

This algorithm is the simplest of all hill-climbing strategies. It picks a parameter and increases it until it reaches the edge of the solution space or until the solutions start getting worse. It then moves to the next parameter. Once it is out of parameters it goes back to the start and decreases the first parameter. It continues in this way until all adjacent solutions are worse than the current one.

This algorithm cannot cope with spaces with several local optima, as it settles on the first it finds. Of all the algorithms implemented here only Tabu Search finds the global optimum.

4.3 Tabu Search

Tabu Search is designed to optimize a discrete space problem whilst avoiding being stuck at local optima [10,11]. It starts at a point specified by the user, and attempts to move to the best solution using the following steps:

1. Evaluate all solutions neighboring the current point that are not reached by "tabu moves".
2. Move to the best neighboring solution.
3. Add the opposite of that move to the list of tabu moves.
4. Remove the oldest tabu move.

The tabu move list must be long enough that the search is likely to reach a different local optimum in the search space, rather than turning around and returning to the previous maximum. There are two typical causes of local optima:

1. The results are stochastic; the uncertainty in results can cause a result to appear higher than its neighbors.

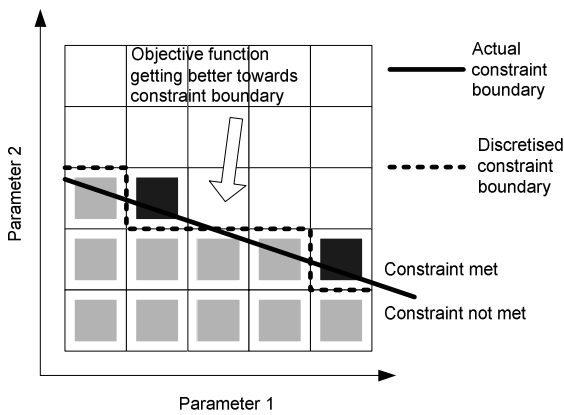


Figure 3: Example of Local Minima

- The solution space is discrete, with constraints that conflict with the objective function (e.g. minimizing price whilst bounding response time). In this case, points that just satisfy the constraint are likely to be the cheapest. Figure 3 shows such an example: the grey boxes do not meet the constraints, and the black boxes are local optima. The objective functions' improvement here is not orthogonal to the constraint: the right black box is the global optimum, but the left one could be deemed the best by other hill-climbers.

4.4 Revised Simplex Search

Revised Simplex Search is a variant of Nelder & Mead's Simplex Search [13]. It is an efficient hill-climbing algorithm designed for continuous space problems. Tabu search can evaluate many solutions for each move, and each move may make very little progress: simplex search evaluates very few solutions, and quickly moves towards a local optima.

Simplex search maintains a *simplex* of points in the solution space. A simplex contains one more point than the number of parameters, which do not lie on a hyper-plane within the space. The initial simplex is formed by taking a user-defined point and projecting it along each of the parameters.

The algorithm repeatedly removes the worst point in the space and replaces it with a better one, which is generally the worst point reflected about the centroid of the others. Each step requires very few simulations to be run, and can make considerable progress towards the optimum. When the simplex collapses onto a hyper-plane, or insufficient progress is being made, the best point found so far is used as the basis of a new simplex, smaller than the first.

Simplex search was designed for continuous solution spaces; the problem we are tackling with this framework is discrete. This means that the transformation from continuous to discrete

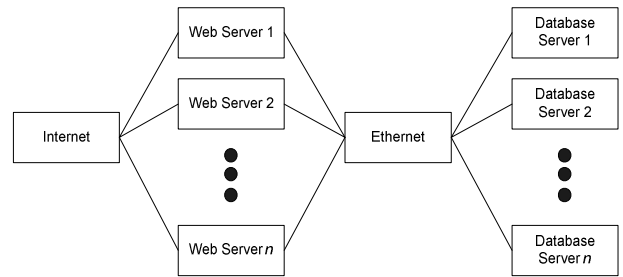


Figure 4: Two Tier Web Farm Topology

parameters may collapse the simplex to a hyper-plane.

4.5 Summary

The algorithms we implemented were chosen from a wide range of possible algorithms. Other algorithms can be easily implemented within this framework; the current implementations range between 150 lines of commented code for Tabu Search and 500 lines for Revised Simplex Search. Further algorithms and techniques for optimization by simulation are described in [1, 2, 4, 7, 8, 18]. The next section evaluates the performance of these algorithms on a typical small optimization problem.

5. RESULTS

The four optimization algorithms were evaluated using a small real-world problem: the aim is to minimize the cost of a two-tier web farm that services 10,000 users, but constrain the utilization of all components of the system to be below 60%, which leaves headroom for expansion.

For simplicity, we only allowed the optimizer to vary the number of web servers and database servers between 1 and 16. The configuration of the servers and the other components was fixed. The size of the solution space for this problem is smaller than many other typical problems, with only 256 possible solutions. This allowed us to quickly implement and debug the algorithms. To simulate 6 minutes of web farm operation, each simulation took 3 minutes on average to run on a Pentium III 600 Mhz processor. We aggregated the results of five simulations at each point to decrease the error bounds.

We compare each of the algorithms using the number of simulations it took to find the optimal solution: two database servers and ten web servers. All of the algorithms take a starting position as input, and this was set to a single web server and two database servers. This position is very close to the origin of the search space: the origin itself was not used because revised simplex search cannot use it as a starting point. We set the length of the tabu move list for Tabu search to five moves. The table below shows the number of simulations which each of the algorithms ran whilst attempting to find the best solution, and

Algorithm	Found correct solution	Number of simulations
Evaluate All Solutions	Yes	256
One Parameter at a Time	No — trapped by local optimum caused by inaccurate results.	6
Tabu Search	Yes	37
Revised Simplex Search	No — converged to nearby position due to discretisation rounding errors.	27

Table 2: Optimization Algorithm Comparison

whether they found it. The results are shown in Table 2.

The two more complex algorithms were much more successful: *Tabu Search* found the correct solution and *Revised Simplex Search* converged to a point adjacent to it, both in a substantially lower number of simulations than evaluating the entire solution space. *One Parameter at a Time* was trapped by the first local minima it came across, and did not find a solution that met the constraints; despite Revised Simplex Search being a hill-climbing algorithm as well, it found a much better solution as it considers points spread all over the solution space.

The benefits of using Revised Simplex Search increase with the number of input parameters—the dimension of the solution space—since it only requires one more point in the simplex than the number of dimensions. Refining the solution with Tabu Search after finding a good solution with Revised Simplex Search seems currently the most promising strategy for this problem.

6. CONCLUSIONS & FUTURE WORK

This paper has described a framework for writing simulation optimization tools; these tools could use almost any simulator and optimization algorithm. It has described how a tool implemented within this framework can use the Indy performance modeling environment with standard optimization algorithms to produce optimal configurations for web farms. A simple problem was used to evaluate the performance of the algorithms we implemented. The relatively simple dependence between the input parameters and the objective function, which is typical in this environment, means that hill-climbing algorithms can perform very efficiently.

In the future, we will investigate how the algorithms perform with bigger and more complex problems, for example, where the hardware configuration and number of the servers can be modified.

7. ACKNOWLEDGEMENTS

This work would have never been completed without the help of Gavin Smyth, or the discussions with Paul Barham, Richard Black and Neil Stratford.

8. REFERENCES

- [1] S. Andradóttir, *A Review of Simulation Optimization Techniques*, Proceedings of the 1998 Winter Simulation Conference, 151–158.
- [2] F. Azadivar, *Simulation Optimization Methodologies*, Proceedings of the 1999 Winter Simulation Conference, 93–100.
- [3] J. Boesel, B.L. Nelson and N. Ishii, *A Framework for Simulation-Optimization Software*, Technical Report, Department of Industrial Engineering and Management Sciences, Northwestern University, 1999.
- [4] J. Boesel, R.O. Bowden Jr., F. Glover and J.P. Kelly, *Future of Simulation Optimization*, Proceedings of the 2001 Winter Simulation Conference 1466–1469.
- [5] A. Charnes and W.W. Cooper, *Chance Constraint Programming*, Management Science Vol 6, No 1, 1959.
- [6] E.K.P. Chong, R.L. Givan, H.S. Chang, *A Framework for Simulation-based Network Control via Hindsight Optimization*, Conference on Decision and Control (CDC), 2000.
- [7] M.C. Fu, *Optimization via Simulation: A Review*, Annals of Operations Research, 53:199–247.
- [8] M.C. Fu, *A Tutorial Review of Techniques for Simulation Optimization*, Proceedings of the 1994 Winter Simulation Conference, 149–156.
- [9] B. Gehlsen and B. Page, *A Framework for Distributed Simulation Optimization*, Proceedings of the 2001 Winter Simulation Conference, 508–514.
- [10] F. Glover, *Tabu Search — Part I*, ORSA Journal of Computing, 190–206, Vol 1. No. 3, Summer 1989.
- [11] F. Glover, *Tabu Search — Part II*, ORSA Journal of Computing, 4–32, Vol 2. No 1, Winter 1990.
- [12] J.C. Hardwick, E. Papaefstathiou, and D. Guimbellot, Modeling the Performance of E-Commerce Sites, *Journal of Computer Resource Management*, Winter 2002 Edition, Issue 105, pp. 3–12.
- [13] D.G. Humphrey and J.R. Wilson, *A Revised Simplex Search Procedure for Stochastic Simulation Response-Surface Optimization*, Proceedings of the 1998 Winter Simulation Conference, 751–759.
- [14] B.D. Joshi, R. Unal, N.H. White and W.D. Morris, *A Framework for the Optimization of Discrete-Event Simulation Models*, Proceedings of the 17th ASEM National Conference, October 10–12 1996.
- [15] R. Jain, *The Art of Computer Systems Performance Analysis* (John Wiley & Sons, 1991).
- [16] H.G. Neddermeijer, G.J. van Oortmarssen and N.P.R. Dekker, *A Framework for Response Surface Methodology for Simulation Optimization*, Proceedings of the 2000 Winter Simulation Conference.
- [17] E. Papaefstathiou, *Design of a Performance Technology Infrastructure to Support the Construction of Responsive Software*, in Proceedings of the 2nd International Workshop on Software and Performance (WOSP 2000), Ottawa, Canada, Sep. 2000, pp. 96–104.
- [18] J.R. Swisher, P.D. Hyden, S.H. Jacobson, L.W. Schruben, *A Survey of Simulation Optimization Techniques and Procedures*, Proceedings of the 2000 Winter Simulation Conference, 119–128.
- [19] W.T. Thompkins, *Policy Driven, Multi-Objective Optimization of Simulated Stochastic Models*, http://www.ncsa.uiuc.edu/TechFocus/Projects/NCSA/Policy_Driven_Multi-Objective_Optimization_of_Simulated_Stochastic_Models.html.