Using Probabilistic Reasoning to Automate Software Tuning

David Sullivan
Margo Seltzer
and
Avi Pfeffer

TR-08-04

Computer Science Group
Harvard University
Cambridge, Massachusetts

Using Probabilistic Reasoning to Automate Software Tuning

David G. Sullivan, Margo I. Seltzer, and Avi Pfeffer
Harvard University
{sullivan, margo, avi}@eecs.harvard.edu

ABSTRACT

Manually tuning the parameters or "knobs" of a complex software system is an extremely difficult task. Ideally, the process of software tuning should be automated, allowing software systems to determine their own optimal knob settings and to reconfigure themselves as needed in response to changing conditions. We demonstrate that probabilistic reasoning and decision-making techniques can be used as the foundation of an effective, automated approach to software tuning. In particular, we present a methodology that uses a probabilistic, graphical model known as an influence diagram to determine the knob settings that optimize a system's performance for a given workload. Influence diagrams allow us to take advantage of expert knowledge about how the relevant variables are related and to exploit conditional independence relationships among these variables. To assess the effectiveness of our methodology, we have used it to simultaneously tune four knobs from the Berkeley DB embedded database system. We present results showing that an influence diagram can effectively generalize from training data for this domain, recommending optimal or near-optimal knob settings for a varied set of workloads and outperforming an alternative approach that employs regression models.

1. INTRODUCTION

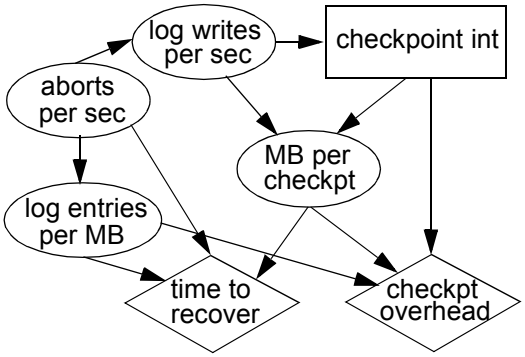
Complex software systems typically include a set of parameters that can be adjusted to improve the system's performance. System designers expose these parameters, which are often referred to as knobs, because they realize that no single configuration of the system can adequately handle every possible workload. However, tuning these knobs is not an easy task, and therefore many have called for the development of self-tuning software systems that are able to adjust their own knob settings in response to changing conditions [2, 3, 9].

The idea of self-tuning software systems is not new, but the need for them is increasing as a result of several trends. First, the complexity of software systems continues to grow, which makes manual tuning more and more difficult. Second, complex software systems increasingly are used by organizations and individuals who do not possess the necessary tuning expertise and who cannot afford to hire someone who does. Even large organizations suffer from both a shortage of qualified people and the growing human costs associated with computer systems. Third, software systems are increasingly being embedded in a wide variety of devices, and systems running in such contexts must, by necessity, be self-tuning [3, 29]. Given these trends, there is clearly a need for an automated approach to software tuning.

A number of prior research efforts have addressed the problem of automating software tuning; Section 6 includes an overview of this research. In addition, a number of commercial software developers have added a limited degree of self-tuning to their products [8, 33, *inter alia*]. However, much remains to be done before software systems are fully self-tuning. For one thing, most of the prior work on automated tuning has attempted to adjust individual knobs in isolation, whereas a comprehensive approach to automated tuning will need to be capable of tuning multiple knobs simultaneously, taking into account interactions between the knobs and ways in which they jointly affect the system's performance. In addition, most previous efforts have focused on a specific algorithm or module, without attempting to provide general guidance about how to automate the software-tuning process. To facilitate the construction of self-tuning systems, what is needed is a methodology for automated software tuning—a systematic approach that can be applied to an arbitrary software system.

In this paper, we propose just such a methodology, one based on the use of probabilistic reasoning and decision-making techniques that have been developed by researchers in artificial intelligence, operations research, and other related fields. In particular, the proposed methodology employs the influence diagram formalism [16] and related learning and inference algorithms [14, 31] to predict the system's performance under various combinations of workload characteristics and knob settings, and to thereby determine the optimal knob settings for a given workload. Influence diagrams are graphical, probabilistic models that allow decision-makers to reason about variables that influence the outcome of their decisions. In particular, influence diagrams model probabilistic dependencies among the relevant variables, and they exploit conditional independencies to simplify both the specification of the model and the process of inferring the expected outcomes of a particular set of decisions.

A tuning methodology based on influence diagrams takes advantage of the knowledge of domain experts about how the relevant variables are related—but this expertise is only needed once, when the structure of the influence diagram is designed by the software developer. Once the appropriate model structure is determined, it can be used as the foundation of a tuning module that runs without human intervention. This module can learn the initial parameters of the model using statistics gathered from actual or simulated workloads. Over time, newly gathered statistics can be used to modify these parameters, allowing the tuning module to refine its performance estimates and to adapt to changes in the environment in which the software system runs. And as the workload imposed on the system varies over time, the



P(MB per checkpt | 50 log writes/s, checkpt int)

MB per checkpt	checkpoint int 1 min	10 min	30 min	...
0-5	0.60	0.10	0.05	...
5-10	0.15	0.17	0.15	...
10-50	0.10	0.65	0.33	...
...	...	...	...	...

E(time to rec | log entr/MB, aborts/s MB/checkpt)

log entr/MB	aborts/s	MB/checkpt	time to rec
1000-5000	0-10	0-5	1.0 min
1000-5000	0-10	5-10	2.5 min
1000-5000	0-10	10-50	8.3 min
...	...	...	...

Figure 1. An example of an influence diagram and its associated parameters. The structure of the model is shown on the left-hand side of the figure. Also shown is an example of a portion of a conditional probability table for one of the chance nodes (*upper right*) and a portion of the table of expected values for one of the value nodes (*lower right*).

tuning module can use the influence diagram to determine the optimal knob settings for each workload—including previously unseen combinations of workload characteristics.

To assess the effectiveness of our methodology, we have used it to simultaneously tune four knobs from the Berkeley DB embedded database system, a programmatic toolkit that provides full support for concurrent accesses, transactions, and recovery [26, 32]. The influence diagram developed for this system recommends optimal or near-optimal knob settings for a varied set of workloads, and it outperforms both the default settings and an alternative approach that uses regression models to determine the optimal knob settings. In addition, the influence diagram is able to effectively generalize from experience, making effective tuning recommendations for previously unseen combinations of workload characteristics.

The following sections discuss various aspects of our work in greater detail, starting with a brief overview of influence diagrams (Section 2) and continuing with a description of our methodology for using an influence diagram for software tuning (Section 3) and of our influence diagram for tuning Berkeley DB (Section 4). Section 5 presents our experiments, Section 6 discusses related work, and Section 7 draws conclusions and discusses future directions for this work.

2. INFLUENCE DIAGRAMS

Influence diagrams are graphical, probabilistic models that were developed as a tool for solving difficult decision problems [16]. Tuning a software system may be viewed as just such a decision problem, in which we attempt to decide which knob settings will maximize the performance of the system for a given workload. The following paragraphs provide an overview of these models. For more detail, readers are encouraged to consult works from decision analysis and artificial intelligence [16, 20, 31].

2.1 Nodes, Arcs, and Parameters

In structure, influence diagrams are directed acyclic graphs that contain three types of nodes: rectangular *decision nodes* that represent the choices being made, oval *chance nodes* that represent random variables relevant to the decision problem, and diamond-shaped *value nodes* that represent aspects of the decision maker's utility function. For example, Figure 1 shows an influence diagram that could be used to tune the checkpoint interval of a database system.

The arcs in an influence diagram convey two different types of information. Arcs into chance or value nodes represent a possible probabilistic dependence relationship. If there is an arc from *A* to *B*, knowing the value of *A* may affect our probability assessments for *B*, and vice versa. These arcs are known as *conditional arcs*. In Figure 1, for example, increasing the checkpoint interval typically leads to an increase in the amount of data logged between checkpoints, and thus there is a conditional arc from *checkpoint int* to *MB per checkpt*.

Arcs into decision nodes represent known information. If there is an arc from *C* to *D*, the value of *C* is known when decision *D* is made. In Figure 1, for example, the arc from *log writes per sec* to *checkpoint int* indicates that the value of the former variable is available when the checkpoint interval is chosen. These arcs are referred to as *informational arcs*. If there is an arc from node *A* to node *B*, we will refer to *A* as a *parent* of *B*. If there is a directed path involving one or more arcs from *A* to *B*, we will refer to *A* as an *ancestor* of *B*.

A key feature of influence diagrams and related graphical models (e.g., Bayesian networks [27]) is the conditional independence relationships that they encode. In particular, given the values of its parents, any chance or value node is conditionally independent of its other ancestors. The encoding of such conditional independencies means that fewer parameters are needed to specify the model, and thus the process of learning these parameters and of using the model to perform inference is greatly facilitated. For example, the model in Figure 1 encodes the fact that the time needed to recover is conditionally independent of both the checkpoint interval and the number of log writes per second, given the amount of data written to the log between checkpoints. As a result, we only need to learn the expected value of *time to recover* for each combination of *log entries per MB*, *aborts per sec*, and *MB per checkpoint*, rather than for each combination of the entire set of chance and decision nodes.

Each node in an influence diagram has an associated set of parameters. For a decision node, there is the set of possible alternatives

for the corresponding decision. For a chance node with no parents (a *root chance node*), there are parameters that capture the marginal distribution over the possible values of its associated random variable. For a chance node with parents (an *intermediate chance node*), there are parameters that specify a collection of conditional distributions, one for each combination of the values of its parents (Figure 1, *upper right*). Similarly, a value node has one expected value for each combination of the values of its parents (Figure 1, *bottom right*). In Section 3.4, we discuss how to learn these parameters from data.

2.2 Determining the Optimal Decisions

Evaluating an influence diagram involves determining the decisions that maximize or minimize the total expected value of the value nodes, given the probability distributions over the chance variables and any observed values of these variables. For instance, in our simple tuning example, we would observe the value of *log writes per sec* for a given workload (because the informational arc from *log writes per sec* to *checkpoint int* indicates that the *log writes* value is known at the time of the decision) and solve the influence diagram to determine the value of *checkpoint int* that minimizes the expected cost for that workload.

A description of the standard algorithm for evaluating an influence diagram [31] is beyond the scope of this paper. Fortunately, there are a number of software toolkits that can be used to evaluate these models [17, 25, *inter alia*].

3. USING AN INFLUENCE DIAGRAM FOR SOFTWARE TUNING

3.1 High-Level Architecture

We will assume for the sake of argument that there is a distinct module known as the *tuner* that is responsible for determining the appropriate knob settings for the system. The tuner takes as input a description of the relevant aspects of the current state of the system and uses an influence diagram to determine the necessary adjustments to each of the knobs being tuned. As the system runs, a separate module known as the *monitor* periodically gathers statistics about the state of the system. When the monitor detects a significant state change, it feeds a description of the new state to the tuner. There are a number of issues that need to be addressed in implementing the monitor, such as distinguishing substantial changes in the state of the system from natural statistical fluctuations. For the most part, these issues have already been successfully addressed by others (e.g., by Brown [5]). Our methodology, which is described in the following sections, focuses exclusively on the design of the tuner.

3.2 Designing the Structure of the Model

Designing the structure of an influence diagram is a challenging task. It requires the knowledge of a domain expert—in this case, of someone familiar with the inner workings of the software system being tuned—to determine which variables and arcs to include in the model. However, the design process only needs to be performed once for a given software system. The resulting model structure can be hard-coded into the tuner associated with the software system, and the tuner itself can run without intervention.

3.2.1 Choosing Variables

When choosing the variables to include in an influence diagram for software tuning, it is helpful to begin with three subsets of the nodes: the decision nodes, which represent the knobs to be tuned; the value nodes, which represent the performance measures to be optimized; and the root chance nodes, which represent the relevant characteristics of workloads encountered by the software system.

The variables chosen for the value nodes should reflect the goal of the tuning process—for example, to maximize the throughput of the system. However, even if the tuning goal involves a single variable such as throughput, it may not be feasible to design the model with a single value node. The number of parameters associated with a node is exponential in the number of its parents, and a value node corresponding to a single, overarching performance metric will typically have too many parents—and thus too many parameters—to be practical. Therefore, it is typically preferable to use multiple value nodes, each of which reflects one aspect of the system's performance. In particular, it can be helpful to consider variables that reflect different types of performance *losses* that the system can incur. A given type of performance loss is typically affected by only a subset of the knobs and workload characteristics, which makes it easier to limit the number of its parents.

An influence diagram for automated software tuning does *not* need to include variables whose values are fixed or slowly changing, such as variables that describe either the platform on which the system runs (e.g., the amount of physical memory) or aspects of the system that are not being tuned. Such variables can be excluded because they are implicitly captured by the parameters of the model. If any of the excluded variables change, the process of updating the parameters over time (Section 3.7) will capture the changes.

3.2.2 Adding Arcs to the Model

The conditional arcs—arcs into chance or value nodes that indicate probabilistic relevance—can typically be added on the basis of intuitive notions of causality. For example, in Figure 1, the time to recover depends at least in part on the amount of data written to the log since the last checkpoint, so it makes sense to include an arc from *MB per checkpoint* to *time to recover*.

Informational arcs—arcs into decision nodes that indicate the information known when a decision is made—must be used to impose an ordering on the decisions. In general, the ordering chosen can affect the decisions recommended by the influence diagram. However, because we assume that the tuner knows the values of the observable workload characteristics before tuning the knobs, it turns out that *any* ordering of the decision nodes produces the same knob-setting recommendations for a given workload [35]. Informational arcs are also drawn from the observable workload characteristics to the decision node that comes first in the imposed ordering; doing so indicates the observations that are made before the knobs are tuned.

3.2.3 Limiting the Number of Parents of a Node

As discussed in Section 3.2.1, the number of parameters associated with a node is exponential in the number of its parents. As the number of parameters associated with a node increases, so does the amount of data needed to train the model and the cost of evaluating

the model. Therefore, it is important to limit the number of parents of each node. One technique for doing so is to introduce an intermediate chance node between the node in question and two or more of its parents, provided that the node in question is conditionally independent of these parents given the intermediate node. For example, if there were five values for each of the nodes in Figure 1, *time to recover* would have $5^4 = 625$ parent combinations if it inherited directly from *log writes per sec* and *checkpoint int.* Adding the intermediate *MB per sec* node to the model reduces the number of *time to recover*'s parent combinations to $5^3 = 125$ while also decreasing the total number of parameters in the model.

3.3 Gathering the Training Data

The parameters of an influence diagram for software tuning are learned from training data. The process of gathering this data involves measuring the performance of the system for various combinations of knob settings and workload characteristics. On many systems, it may be impractical to experiment with knob settings while the system runs, because a randomly selected knob combination could lead to significant performance losses. Therefore, it may be preferable to gather the training examples offline—either on the system itself before it is deployed, on a version of the system running on a separate but similar hardware platform, or on the deployed system during periods of idleness. In order to perform this offline training, it may be necessary to develop a *workload generator* that simulates the workloads that the system will encounter. This is the approach we have taken in our work with Berkeley DB (Section 5.1), and we discuss the design of such workload generators elsewhere [35].

When using a workload generator, the characteristics of the simulated workloads are drawn from distributions over the possible values of these variables. These distributions can be estimated or anticipated by someone familiar with the system, or they can simply be uniform distributions over the ranges of possible values. Each of the selected workloads is run using one or more of the possible knob configurations. Once the system reaches a steady state for a given run, the values of the variables in the influence diagram are measured over some interval and together constitute a training example. Neither the workload distributions from which the simulated workloads are drawn nor the simulated workloads themselves need to precisely match the actual workloads run on the system. The training examples can be seen as providing reasonable starting estimates of the model's parameters, and the parameters can be refined over time as the system uses the influence diagram for tuning (Section 3.7).

It is worth noting that it is not necessary to see all possible combinations of workload characteristics and knob settings during training. Provided that the training instances include most of the possible combinations of the values of the parents of each chance and value node in the model, the model should be able to recommend optimal or near-optimal knob settings for workloads that are not present in the training data.

Because the parameters of the model may depend upon the hardware platform on which the software system runs, the process of gathering the training data will typically need to be performed on a platform-specific basis. One possible approach would be for the software developer to gather training data for a wide range of hardware platforms and to precompute the initial parameters of the model for each of these platforms. Alternately, a workload generator and a script for driving it could be included with the software system and run by the end-user on the appropriate platform. As in the design of the influence diagram, no expertise would be needed by the end-user. The workload generator—like the tuner itself—would run "out of the box."

3.4 Discretizing the Continuous Variables

Our discussion thus far has assumed that each variable in the influence diagram has a discrete number of possible values. For a continuous variable—or a discrete variable with enough possible values that it is effectively continuous—we need to *discretize* the range of possible values so that we can treat the variable as if it were discrete. Discretizing a variable involves specifying a series of increasing cutoff values that can be used to divide the range of possible values into subranges referred to as bins. Values of the continuous variable that fall into the same bin are given the same value in the discretized version of the variable.

The most commonly used discretization methods—such as equal-height bins, in which the cutoffs are chosen so that each bin contains approximately the same number of observed values—do not tend to perform well when applied to variables in an influence diagram. Such methods discretize each variable in isolation, without considering interactions between neighboring variables in the model. To overcome this limitation, we have developed an algorithm for discretizing variables in an influence diagram that is based on the prior work of Friedman and Goldszmidt [13]. The proposed algorithm learns the discretizations from training data, and it uses the minimum description length principle to balance the complexity of the model with the degree to which the learned discretizations capture interactions between related variables. The details of this algorithm can be found in a related report [35].

3.5 Learning the Parameters

If training data is available for all of the random variables, we can determine the probability parameters using *maximum likelihood estimation*, finding the parameter values that maximize the likelihood of the training data. For discrete or discretized variables, this simply involves maintaining counts of the number of times that each combination of the values of a node and its parents occur together. More precisely, the probability θ_{ijk} that chance variable i will take on its kth value given the jth combination of values for its parents is estimated using the following equation:

$$\theta_{ijk} = N_{ijk} / N_{ij} \qquad (1)$$

where N_{ijk} is the number of times that i takes on its kth value and i's parents take on the jth combination of their values, and N_{ij} is the total number of times that i's parents take on the jth combination of their values (i.e., $N_{ij} = \Sigma_k N_{ijk}$). Similarly, marginal probabilities for nodes without parents are determined by computing the percentage of training examples that use each value.

The expected values of the value-node variables are calculated as simple averages: the expected value E_{ij} of value-node variable i given the jth combination of the values of its parents is estimated using the following equation:

$$E_{ij} = \Sigma_k V_{ijk} / N_{ij} \quad (2)$$

where V_{ijk} is the value of i in the kth training example in which i's parents take on the jth combination of their values, and N_{ij} is again the total number of times that i's parents take on the jth combination of their values.

If one or more of the nodes in the influence diagram are unobservable (so-called *hidden nodes*) or if the training data is missing values, the parameters can usually still be estimated using techniques such as the expectation maximization (EM) algorithm [11].

As discussed in Section 3.2.1, it is often impractical to use a single value node to represent the performance metric that the tuner is attempting to optimize, and therefore it may be preferable to employ multiple value nodes that each reflect one aspect of the system's performance. In such cases, it is necessary to determine appropriate weights for the value nodes so that optimizing their sum is equivalent to optimizing the overall performance metric. In our work with Berkeley DB, we have used standard regression techniques to learn these weights from the training data.

3.6 Using the Model

Given the observed characteristics of a given workload, the tuner evaluates the influence diagram to determine the optimal knob settings for that workload. If the influence diagram recommends knob settings that differ from the ones that are currently in use, the tuner decides whether the knobs should be retuned. If there is a cost associated with adjusting the knobs, the tuner weighs the expected gain in performance from retuning—which can be estimated by using the influence diagram to determine the expected utility of both the current settings and the recommended ones—against an estimate of the cost of adjusting the knobs. If the tuner decides that the benefits of retuning outweigh the costs, it modifies the knobs accordingly.

3.7 Updating the Model Over Time

The statistics gathered as the system runs can be used to update the parameters of the model over time, provided that the tuning system stores the counts and value-node totals that appear in equations 1 and 2. In one possible approach, new training data would simply be used to increment these counts and totals, and the probabilities and expected values would be recomputed accordingly. However, to allow the system to quickly adapt to changes in its environment, newer training examples should be given more weight. This can be done by employing a real-valued *fading factor* from the interval (0,1) [20]. If f is the value of this factor, then all counts and totals that are affected by a given training example are updated as follows:

$$v \longleftarrow f \cdot v + 1 \quad (3)$$

In addition to updating the model's parameters, it may also make sense to periodically reconsider the discretizations of the continuous variables. Doing so requires access to the undiscretized training data, and it may thus be necessary for the tuner or monitor to log the undiscretized training examples—including the measurements gathered as the system runs. To improve the performance of the model and to fully capture the impact of environmental changes, it may also be necessary to periodically gather additional training data offline as described in Section 3.3. During this supplemental training, special focus could be given to workload characteristics that have been encountered most frequently, allowing the tuner to improve its performance on these workloads.

4. AN INFLUENCE DIAGRAM FOR TUNING BERKELEY DB

Figure 2 shows the influence diagram that we designed to tune four knobs from the Berkeley DB embedded database system [26, 32]. The model was created using information from the Berkeley DB documentation [32] and input from one of the system's designers. This section provides some necessary background information about Berkeley DB, as well as a brief overview of the model itself.

4.1 Relevant Background

We used Berkeley DB's Btree access method as the basis of the databases in the experiments. A Btree is composed of nodes, each of which contains data and zero or more pointers to other nodes that are referred to as its children. Nodes with children are known as internal nodes; nodes without children are known as leaf nodes. Berkeley DB organizes its databases in units called pages, and each internal node or leaf node in a Btree database corresponds to a single page.

Pages needed by an application are read from an on-disk database file into a shared memory cache known as the memory pool. Ideally, the entire database will fit in the memory pool so that accesses can be satisfied without the overhead of going to disk. In addition to the memory pool, the operating system maintains its own buffer cache of recently accessed file blocks. When a page requested by an application is not found in the memory pool (an event known as a miss), Berkeley DB asks the operating system to read a page from the database file, and the operating system first checks in the buffer cache for the corresponding block or blocks from the file. On many operating systems—including the one that we have used in our experiments—the buffer cache is integrated with the virtual memory system. As a result, a failure to find a file block in the buffer cache is recorded as a page fault, and we can use the operating system's page-fault statistics to count the disk reads that an application performs.

Locks in Berkeley DB are acquired on a per-page basis. A read lock is acquired before a page is read, and a write lock is acquired before a page is modified. Berkeley DB enforces multiple-reader, one-writer semantics: threads performing read-only operations can access the same pages simultaneously by acquiring read locks for the pages, but a thread that wishes to modify the database must obtain exclusive access to the affected pages by acquiring write locks for the pages. As a result of this need for exclusive access, threads may be forced to wait before they can acquire a lock.

When locking is used, it is possible for two or more threads to deadlock if they each hold a lock that prevents one of the other threads from making forward progress. Berkeley DB can detect when two or more operations are deadlocked, and it forces one of the offending operations to surrender its locks and return an error.

Like many database systems, DB allows a set of operations to be grouped together in a unit known as a transaction. If the transaction is committed, all of the changes made by the transaction are

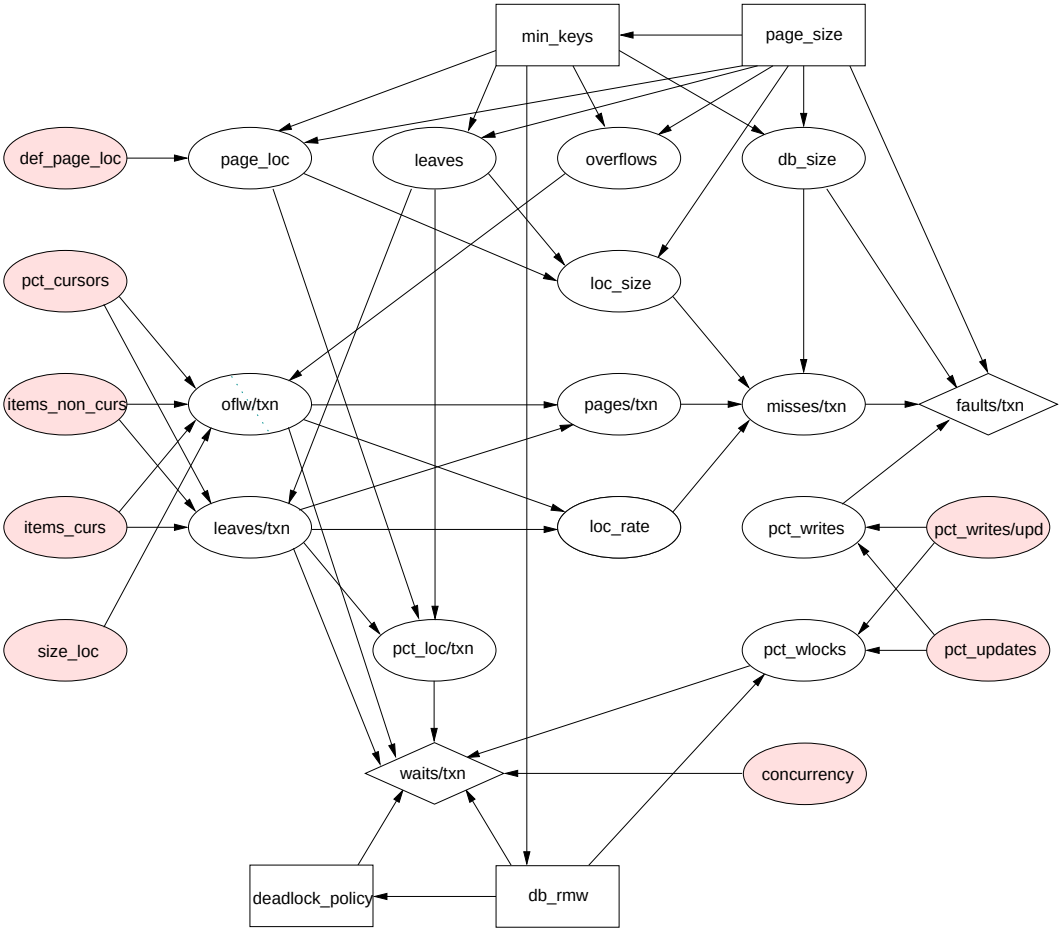


Figure 2: An influence diagram for tuning four knobs from the Berkeley DB database system. The shaded nodes are the root chance nodes, which represent characteristics of the workloads encountered by the system. All of these characteristics are observed before a tuning decision is made, but the informational arcs that would make this fact explicit have been omitted for the sake of readability. Brief descriptions of the nodes in the influence diagram are given in Table 1, and additional detail can be found in Sections 4.2, 4.3, and 4.4.

applied together. If the transaction is aborted, none of the changes are applied. When a set of operations is performed in the context of a transaction, all locks are acquired on behalf of the transaction. If one of the operations returns an error because of deadlock, the application must abort the transaction.

4.2 Decision Nodes: The Knobs Being Tuned

The four knobs that we tune are represented by the rectangular decision nodes in Figure 2. The first knob controls the database's *page size*. Smaller page sizes enable finer-grained concurrency, because having fewer records on a page reduces contention for any given page. However, smaller page sizes may also reduce performance by forcing items that are too large to fit on a leaf page into one or more *overflow pages*. Overflow pages tend to reduce performance by increasing the number of pages accessed, reducing the efficiency of the memory pool, and creating contention for a metadata page that is locked whenever an overflow page is updated. Page size can also impact I/O performance, because if the page size is smaller than the filesystem's block size, it may be necessary to read the contents of a block before writing a page so that the unmodified portions of the block can be maintained.

The second knob controls the *minimum keys per page* setting (*min_keys*), which indicates the smallest number of keys that can exist on a single page. In combination with *page size*, this knob governs the maximum size of an item that can be stored in a regular database page, and all items larger than this maximum size are forced into overflow pages. Despite the undesirable properties of overflow pages, there are situations in which it makes sense to decrease the maximum size of leaf-page items by increasing the value of *min_keys*. In particular, if a database has large items that are accessed infrequently, increasing *min_keys* in order to force these items into overflow pages can allow more of the smaller, frequently accessed items to fit in the memory pool.

The third knob is a binary *db_rmw* knob that specifies whether threads should acquire write locks when reading items that they may subsequently modify. Doing so eliminates the lock conflicts and deadlocks that can occur when a thread attempts to upgrade a read lock to a write lock, but it can also lead to increased lock contention and more aborts because threads end up holding write locks for longer intervals.

The final knob selects the policy used to determine which lock

Table 1. Summary of the nodes in the influence diagram in Figure 2. See Sections 4.2, 4.3, and 4.4 for more detail.

Node name	Description
concurrency	number of concurrent threads performing transactions
deadlock_policy	policy used to select transaction that will be aborted to resolve a deadlock
def_page_loc	percent of leaf pages in the locality set under the default *page_size* and *min_keys* settings
db_rmw	indicates whether to acquire write locks when reading items
db_size	size of the database
faults/txn	mean number of page faults in the OS buffer cache per transaction
items_curs	mean number of items accessed by transactions that use a cursor
items_non_curs	mean number of items accessed by transactions that do not use a cursor
leaves	total number of leaf pages
leaves/txn	mean number of leaf pages accessed per transaction
loc_rate	percent of the leaf- and overflow-page accesses that go to the locality set
loc_size	size of the locality set
min_keys	minimum number of keys per page
misses/txn	mean number of misses in the memory pool per transaction
overflows	total number of overflow pages
oflw/txn	mean number of overflow pages accessed per transaction
page_loc	percent of leaf pages in the locality set
page_size	size of each page in the database
pages/txn	mean number of leaf and overflow pages accessed per transaction
pct_cursors	percent of transactions that use a cursor
pct_loc/txn	mean percentage of the locality set accessed per transaction
pct_wlocks	percent of accesses using write locks
pct_writes	percent of accesses modifying an item
pct_writes/upd	percent of items accessed in an update transaction that are modified
pct_updates	percent of transactions that are update transactions
size_loc	percent of accesses to items that are forced into overflow pages under one or more knob settings
waits/txn	mean number of lock waits per transaction

request to deny when a deadlock occurs (*deadlock_policy*). We considered four policies in our experiments: *minlocks* (reject the request by the transaction holding the fewest locks), *minwrite* (reject the request by the transaction holding the fewest write locks), *random* (reject a random request), and *youngest* (reject the request by the transaction with the newest ID). Because several of these policies are based on the number and type of locks held by a transaction, and because the locks that a transaction holds typically reflect the operations that it has performed, the value of this knob can affect the number and types of operations that must be redone when aborted transactions are retried.

As mentioned in Section 3.2.2, it is necessary to impose an ordering on the decisions in an influence diagram, even though the ordering will not affect the recommended knob settings. We selected one ordering (*page_size*, *min_keys*, *db_rmw*, *deadlock_policy*) and added the resulting informational arcs.

4.3 Value Nodes

The goal of our tuning is to maximize throughput. However, given the problems associated with using a single value node (Section 3.2.1), we chose to use the two diamond-shaped value nodes shown in Figure 2, rather than a single value node corresponding to throughput. The *faults/txn* node represents the average number of page faults per transaction; it captures the impact of the knobs on I/O performance because of the connection between disk reads and page faults (Section 4.1). The *waits/txn* node represents the average number of times that a transaction waits to acquire a lock; it captures the impact of the knobs on lock contention. The values of these nodes are weighted so that minimizing their sum is equivalent to maximizing throughput (Section 5.3).

4.4 Chance Nodes and Conditional Arcs

4.4.1 Workload Characteristics

The eight root chance nodes characterize the workloads for which the database is being tuned. Five of the workload characteristics describe the transactions that run on the system: the percentage of transactions that use a database cursor to perform sequential accesses (*pct_cursors*), the average numbers of items accessed by transactions that use a cursor (*items_curs*) and those that do not (*items_non_curs*), and two variables that specify the degree to which transactions modify items in the database. We define an update transaction to be any transaction that *considers* modifying the items that it accesses. Such transactions read a set of the items, but they may not modify all of them, perhaps because they need to see the contents of an item before they can determine whether it should be modified. The *pct_updates* variable represents the percentage of transactions that are update transactions, and the *pct_writes/upd* variable represents the average percentage of the items accessed by an update transaction that are actually modified.

Two of the workload characteristics measure the degree to which accesses to the database exhibit locality by accessing certain items or pages in the database more often than others. We consider both locality based on the position of the accessed items in the database (*page locality*) and locality based on the size of the accessed items (*size locality*). To capture page locality, we make use of an abstraction known as the locality set, which we define as the collection of leaf pages that receive 90% of the accesses. Because the composi-

tion of the locality set depends on the configuration of the database, we define the *def_page_loc* workload characteristic as the percentage of leaf pages that are in the locality set when the default knob settings for *page_size* and *min_keys* are used. To capture size locality, we define *size_loc* as the percentage of accesses made to items that are large enough to be forced into overflow pages in one or more of the database configurations being considered.

The final workload characteristic, *concurrency*, measures the average number of transactions that simultaneously access the database.

4.4.2 Other Chance Nodes and Arcs

The remaining chance nodes mediate the impact of the knob settings and workload characteristics on the value nodes. Using these intermediate chance nodes allows us to maintain a reasonable number of parents for each node in the model (Section3.2.3). For example, the *pct_writes* node represents the percentage of transactions that modify an item. As such, it captures the impact of both the *pct_writes/upd* and *pct_updates* nodes on our beliefs about the *faults/txn* node (Figure 2), and it thereby reduces the number of parents of this node. Provided that there are fewer values for *pct_writes* than there are combinations of the values of *pct_writes/upd* and *pct_updates*, the number of parameters associated with *faults/txn* will also be reduced.

The conditional arcs into the chance and value nodes were added on the basis of intuitive notions of causality (Section 3.2.2). For example, the lock waits in a Berkeley DB application are "caused" by multiple threads attempting to acquire locks for the same database pages at the same time. As the number of threads accessing the database increases, it becomes increasingly likely that lock waits will occur. Therefore, it makes sense to include an arc from *concurrency* to *waits/txn*, as shown in Figure 2.

Space constraints preclude us from explaining all of the intermediate chance nodes and conditional arcs in the model. Table 1 provides a brief description of each node, and more information about the nodes and arcs can be found in a related report [35].

5. EXPERIMENTS

In order to assess the effectiveness of our methodology, we developed a workload generator for Berkeley DB and used it to run a variety of workloads on a database implemented using Berkeley DB. Data collected from one set of the workloads was used to train the model presented in Section 4, and the remaining workloads were used to assess the accuracy of the trained model's recommendations. Below, we briefly describe the significant features of our experiments and summarize our results.

5.1 The *db_perf* Workload Generator

To produce a variety of workloads, we developed a workload generator, *db_perf*, that allows us to easily specify characteristics of both the database and the transactions. After running the specified workload, *db_perf* outputs statistics that capture the steady-state performance of the system. To ensure that a steady state is reached, *db_perf* monitors the relevant statistics over time, storing *n* equally spaced samples of each statistic from the past *t* seconds. It thus effectively maintains a sliding, *t*-second window of measurements. Once the variance in each collection of samples is small enough, the run stops and the averages of the statistics over the current measurement window are recorded.

5.2 Gathering the Data

5.2.1 Test Platform

We ran *db_perf* on five 333-MHz Sun UltraSPARC IIi machines with Seagate ST39140A Ultra ATA hard drives and 128 MB of RAM; they each ran Solaris 7 and were configured more or less identically. We configured Berkeley DB to use a 64-MB memory pool and to lock the memory pool and the other memory-mapped files used by its subsystems into memory. To accommodate a large number of concurrent transactions, we also used a non-default value of 4000 for both the maximum number of locks and the maximum number of lock objects in the lock subsystem. With the exception of the knobs being tuned, we used the default configuration of the remaining aspects of Berkeley DB.

5.2.2 Test Database

We configured *db_perf* to use a database consisting of 490,000 key/item pairs, all of which have small (6 or 7 byte) keys. One-seventh of the items have lengths drawn from a normal distribution with a mean of 450 and a standard deviation of 10 (the *large items*), and the remaining item lengths are drawn from a normal distribution with a mean of 100 and a standard deviation of 5 (the *small items*). The large items are uniformly distributed across the leaf pages.

5.2.3 Knob Settings

The experiments considered two possible values for each of the *page_size* and *min_keys* knobs: *page_size* values of 2K and 8K, and *min_keys* values of 2 (for both 2K and 8K pages) and either 4 (for 2K pages) or 16 (for 8K pages). The database layouts that result from the (2K, 2) and (8K, 2) knob combinations fit all of the items on leaf pages. The (2K, 4) and (8K, 16) layouts force the large items into overflow pages, which may improve the performance of workloads that primarily access the small items. The experiments did not consider a (2K, 16) combination because it forces both the small and large items into overflow pages, and they did not consider an (8K, 4) combination because it leads to the same layout as the (8K, 2) combination. The experiments also considered both of the values of the binary *db_rmw* knob and the four values of the *deadlock_policy* knob, as described in Section 4.2. Thus, there were 32 combinations of knob settings in all.

5.2.4 Workload Characteristics

The workload characteristics were drawn from distributions that were designed to generate a wide range of workloads. In particular, an effort was made to include workloads that benefit from non-default knob settings. For example, to ensure that the non-default settings of the *page_size*, *db_rmw*, and *deadlock_policy* knobs would be optimal for some of the generated workloads, we included distributions that tend to produce workloads with large amounts of lock contention. Doing so allowed us to test the model's ability to accurately recommend different knob settings for different workloads.

5.2.5 Running the Workloads

Two sets of *db_perf* runs were conducted: the training runs, which generated the data used to train the models used in the experiments; and the test runs, which generated the data used to assess the accuracy of the models' tuning recommendations. For each of the 18717 randomly selected training workloads, *db_perf* was run using a single, randomly selected combination of knob settings. For each of the 443 randomly selected test workloads, *db_perf* was run 32 times, once for each possible combination of knob settings.

Of the 443 test workloads, 92 of them (the validation set) were used to refine the influence diagram, and the remaining 351 workloads (the test set) were used to assess the performance of the models. None of the test-set workloads have combinations of undiscretized workload characteristics that also appear in the validation set. Even when the workloads are compared on the basis of their *discretized* characteristics, 89% of the test-set workloads do not appear in the validation set, and the remaining 11% were not singled out in any way during the process of refining the influence diagram.

5.3 Training and Using the Influence Diagram

Given the data gathered during the training runs, we used the techniques described in Sections 3.4 and 3.5 to learn both the discretizations of the continuous variables in the influence diagram and the parameters associated with the chance and value nodes. In using linear regression to learn weights for the value nodes, we used the natural logarithm of throughput as the dependent variable rather than throughput itself because throughput can never be negative. In addition, an analysis of the training data revealed a small number of outliers with extremely high *waits/txn* values, and we removed these training examples before running the regression.

The Netica toolkit (version 2.15 for Solaris) [25] was used to construct and manipulate the resulting influence diagram. In particular, we used Netica to determine the influence diagram's recommended knob settings for each of the test-set workloads.

5.4 Results

5.4.1 Assessment Methodology

We assess the accuracy of a model's tuning recommendations for a given workload by comparing the throughput achieved using the model's recommended knob settings with the maximal throughput achieved during the 32 test runs for that workload. We define the *slowdown* of the recommended settings as the percent difference between the throughput achieved using the recommended settings and the maximal throughput. Because time constraints limited us to one test run for each combination of knob settings, we are not able to determine whether the observed slowdowns are statistically significant. However, we have observed that multiple runs using the same knob settings can produce throughputs differing by 5% or more. Therefore, we say that a model's recommendations for a given workload are *optimal* if they have a slowdown of 0 and *near-optimal* if they have a slowdown that is greater than zero but less than five percent, and that the recommendations are accurate if they are either optimal or near-optimal.

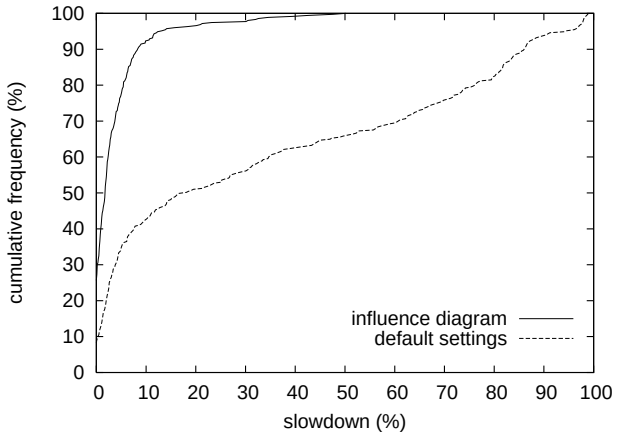


Figure 3. Performance of the influence diagram on the test workloads. The above graph shows the cumulative distribution of slowdown values associated with the influence diagram's recommended knob settings for the test-set workloads, and it compares the performance of the influence diagram's recommendations to the performance obtained using the default settings. The cumulative frequency value for a given slowdown value, *s*, represents the percentage of tuning recommendations with slowdowns that are less than or equal to *s*.

5.4.2 Performance of the Influence Diagram

The influence diagram recommends optimal or near-optimal knob settings for 270 of the 351 test-set workloads (76.9%); 90 of the recommended settings are optimal, and 180 are near-optimal. The throughputs achieved by the non-optimal knob recommendations (including the near-optimal cases) are 5.04% slower than optimal on average, and 93.3% of the model's recommendations have throughputs that are within 10% of optimal. Ideally, the model would have made optimal or near-optimal recommendations for all of the test-set workloads. The fact that it fails to do so is disappointing, but models of complex systems are seldom perfect. Moreover, the influence diagram still offers considerable performance improvements over the default knob settings, which produce optimal or near-optimal throughputs on only 33.9% of the test workloads (119 workloads, with 29 optimal cases and 90 near-optimal ones), and that have non-optimal throughputs that are an average of 37.1% slower than optimal. Figure 3 compares the overall performance of the influence diagram's recommendations with the performance of the default knob settings.

5.4.3 Ability of the Influence Diagram to Generalize

Of the 351 test workloads, none have exactly the same combination of workload characteristics as the workloads in the training set. However, in assessing the ability of the influence diagram to generalize, it is perhaps more meaningful to consider only the test workloads that involve previously unseen combinations of the values of the *discretized* workload variables. Using this criterion, 41 of the 351 test workloads are previously unseen. The influence diagram recommends optimal or near-optimal knob settings for 33 of these workloads (80.5%), and its non-optimal recommendations have an average slowdown of 6.94%. By comparison, the model recommends optimal or near-optimal settings for 237 of the 310 previously seen test workloads (76.5%), with non-optimal recom-

mendations that are an average 4.76% slower.

Although the knob settings recommended for the unseen workloads have a larger average slowdown than the settings recommended for the seen workloads, the model's overall performance on the unseen workloads still demonstrates that it is able to generalize from training data. Moreover, most of the seen test workloads are seen in conjunction with only a fraction of the 32 possible knob settings, and thus the model also needs to generalize to a certain degree to obtain knob recommendations for these workloads.

5.4.4 Comparison with a Regression-Only Approach

One possible alternative to using our methodology for tuning is to employ models based solely on regression. To compare the two approaches, 32 equations were derived from the training data using multiple linear regression; each equation maps workload characteristics to throughput for one of the 32 possible configurations of the knobs. These equations were then used to determine the optimal knob settings for each of the test-set workloads.

To learn the regression equation for a given combination of knob settings, we performed regression on the subset of training examples that used those settings. The regression process began with a set of possible independent variables and iteratively removed any variable that was deemed irrelevant because the 95% confidence interval for its coefficient included zero. If there were multiple irrelevant variables after a given round of regression, the one with the highest P-value was discarded. The regression was then rerun on the remaining variables. This process continued until all of the remaining variables were relevant. The list of possible independent variables included all of the workload characteristics, as well as combinations of these characteristics that seemed relevant to the throughput of the system. Because throughput can never be negative, the natural logarithm of throughput was used as the dependent variable rather than throughput itself.

The resulting regression equations recommend optimal or near-optimal knob settings for 219 of the 351 test workloads (62.4%); 61 of the recommended settings are optimal, and 158 are near-optimal. The throughputs achieved by the regression equations' non-optimal knob recommendations (including the near-optimal cases) are 11.38% slower than optimal on average.

As shown in Figure 4, the influence diagram clearly outperforms the regression equations on the test-set workloads. And while it might be possible to improve the performance of the regression equations by starting with a different set of terms for the independent variables, it is important to note that the process of determining which terms to consider involves challenges that are similar to the ones encountered in designing the structure of an influence diagram, but it must be performed without the advantages provided by the graphical structure of influence diagrams and their ability to exploit intuitive notions of causality. And unlike influence diagrams, regression equations fail to take advantage of the conditional independencies that exist between the variables, and they may thus require more training data to achieve comparable levels of accuracy.

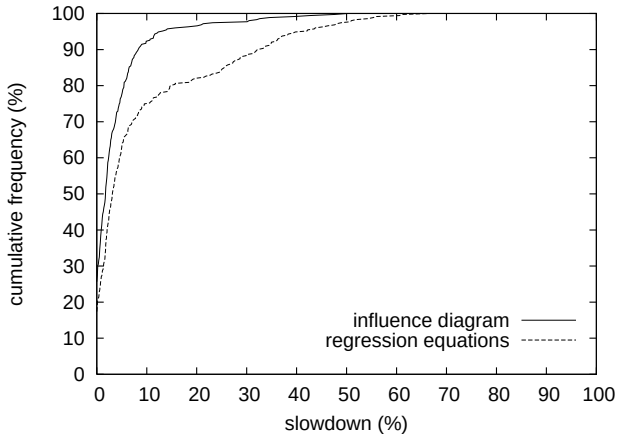


Figure 4. Comparing the influence diagram to a regression-only approach. The above graph shows cumulative distributions for the slowdown values associated with the knob settings recommended by both the final influence diagram for Berkeley DB and a set of regression equations derived from the training data. The cumulative frequency value for a given slowdown value, *s*, represents the percentage of tuning recommendations with slowdowns that are less than or equal to *s*.

5.4.5 Evaluating the Impact of Training-Set Size

The process of gathering training data is one of the costlier aspects of any model-based approach to software tuning. Therefore, we would like the model to be useful after only a limited amount of training. To evaluate the impact of the training-set size on the quality of the influence diagram's recommendations, variants of the model were derived from samples of the training data that had sizes ranging from 10 to 100 percent of the total training set. Figure 5 summarizes the accuracies of the influence diagrams derived from these samples. The mean accuracy of the models increases as the sample size increases from 10% to 60%, at which point it levels off. And although it is not shown, the mean slowdown of the models decreases as the sample size increases from 10% until 60% and then levels off. The 90% samples produce models with the highest mean accuracy and lowest mean slowdown; however, the resulting differences in the performance of the system are small enough that any sample size from 60 to 100% could be used.

The time needed to gather 60% of the training data is still quite substantial: it would take the five machines used in the experiments roughly 22 days. However, the influence diagrams derived from even the smallest samples still exceed the performance of the default knob settings, which is indicated by the dotted line in Figure 5. Therefore, it would be possible to deploy the tuner after gathering only a small amount of training data (e.g., after the four days needed to gather 10% of the data) and to have it begin tuning the system while additional training is carried out offline. Moreover, the bulk of the training time was devoted to warming the memory pool and the buffer cache. In situations in which cache-related effects are less significant, it should be possible to collect training data much more efficiently. In any case, the use of a workload generator allows training data to be collected offline without disrupting the performance of the software system, and the time costs associated with training are amortized over the life of the tuner.

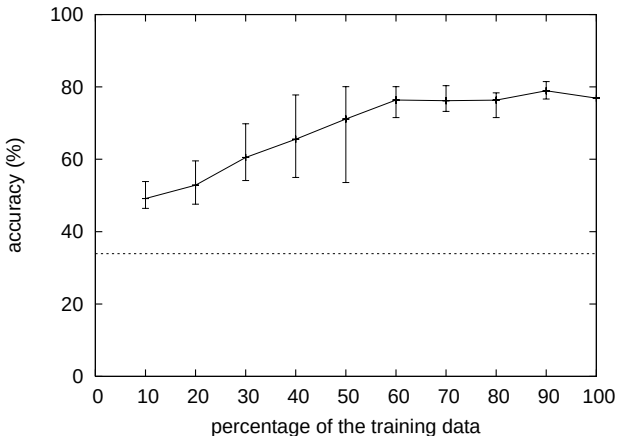


Figure 5. Varying the number of training examples. The above graph shows how the accuracy of the model is affected by the size of the training set. For sizes less than 100%, the points plotted are the average accuracy of ten models derived from random samples of that size, and the error bars show the minimum and maximum accuracies for that size. The dotted line indicates the performance of the default knob settings.

6. RELATED WORK

One approach that has been used to automate software tuning is to empirically compare the performance of some or all of the possible knob settings for a given workload and to choose the settings that yield the best performance [12, 28, 29, *inter alia*]. The chief limitation of tuners based on empirical comparisons is that they are unable to generalize from experience, and therefore they cannot make timely tuning recommendations for previously unseen workloads. The ability to generalize is one of the strengths of model-based tuning approaches like the one that we propose.

Many of the prior efforts to automate database tuning have employed a feedback-driven approach that iteratively adjusts the knobs to meet a specified criteria, such as a response-time goal [6] or an acceptable level of lock contention [36]. Such techniques have been used to tune other types of software systems as well [10, 19, 21, 34, 37, *inter alia*]. However, feedback-based techniques require a potentially lengthy series of iterative adjustments to reach their goal, and they typically have been limited to tuning single knobs in isolation. Brown et al. do use feedback to tune two interacting knobs [6], but Brown himself explicitly mentions the difficulty of doing so [5, Sect. 5.2]. The model-based approach presented in this paper is capable of quickly tuning multiple interacting knobs, as demonstrated by our influence diagram for Berkeley DB, which tunes four knobs.

Other types of models have also been used to automate software tuning, including regression models [4, 24], queueing-network models [23], and cost or cost-benefit formulas [1, 22]. Although these models have been shown to work well in certain domains, it is unclear whether they could be used as the basis of a methodology for tuning an arbitrary software system with multiple interacting knobs. We have shown that an influence diagram, with its ability to explicitly model interactions between the relevant variables, is able to outperform regression models in tuning one example of such a system.

To the best of our knowledge, the methodology presented in this paper represents the first application of probabilistic reasoning techniques to the problem of software tuning. The closest related applications that we are aware of are those of Horvitz et al., who apply graphical, probabilistic models to a variety of applications that involve software systems, including software debugging [7] and intelligent user interfaces [15]. The FAST expert system developed by Irgon et al. [18] also employs techniques from artificial intelligence to tune the performance of a complex software system. However, FAST uses frame-based knowledge representation and heuristic problem-solving techniques rather than an approach based on probabilistic reasoning, and it is focused primarily on the diagnosis and repair of performance problems rather than on finding the optimal knob settings for a given workload.

7. CONCLUSIONS AND FUTURE WORK

One aspect of the proposed methodology that remains to be tested is the process of updating the parameters of the model over time (Section 3.7). Experiments are needed to determine appropriate values for the fading factor used to reduce the impact of older training examples over time (equation 3), as well as to determine whether it is necessary to supplement the training examples gathered as the system runs with periodic offline training. Important insights could also be gained into all of the steps in the methodology by applying it to other software systems.

It would be useful to extend the methodology to handle numerical-valued knobs with many possible settings. To handle such knobs efficiently, it may be necessary to discretize them along with the continuous chance nodes. The influence diagram would then be used to recommend an interval of possible settings for the knob rather than a precise value. The challenge would be to devise a procedure for selecting a particular setting from the recommended interval. This may necessitate supplementing the influence diagram with an additional model of some sort, or empirically exploring possible values within the recommended interval using a feedback-based approach.

The need to automate software tuning has existed for some time because of the difficulties involved in manually tuning complex software systems. Current trends, including the growing complexity of software systems and the increasing deployment of software systems in settings where manual tuning is either impractical or overly costly, only exacerbate this need. The methodology presented in this paper uses probabilistic reasoning and decision-making techniques as the foundation of an effective, automated approach to software tuning. Our results show that an influence diagram created using the proposed methodology is able to recommend optimal or near-optimal tunings for a varied set of workloads running on the Berkeley DB embedded database system, including workloads that are not encountered during training. Although there are still issues that must be addressed before a tuner based on the proposed methodology can be deployed, the results demonstrate that this methodology has the potential to meet the need for a fully automated approach to software tuning.

8. REFERENCES

[1] Agrawal, S., Chaudhuri, S., and Narasayya, V. Automated selection of materialized views and indexes for SQL databases. In *Proc. of VLDB '00*, 2000.

[2] Barrera, J.S. Self-tuning systems software. In *Proc. of the 4th Workshop on Workstation Operating Systems*, 1993.

[3] Bernstein, P., Brodie, M., Ceri, S., DeWitt, D., Franklin, M., Garcia-Molina, H., Gray, J., Held, J., Hellerstein, J., Jagadish, H., Lesk, M., Maier, D., Naughton, J., Piradesh, H., Stonebraker, M., and Ullman, J. The Asilomar report on database research, *SIGMOD Record* 27(4):74-80, 1998.

[4] Brewer, E.A. High-level optimization via automated statistical modeling. In *Proc. of the 5th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1995.

[5] Brown, K.P. Goal-oriented memory allocation in database management systems. Ph.D. thesis, University of Wisconsin, Madison, 1995.

[6] Brown, K.P., Carey, M.J., and Livny, M. Goal-oriented buffer management revisited, In *Proc. of ACM SIGMOD* '96, 1996.

[7] Burnell, L., and Horvitz, E. Structure and chance: melding logic and probability for software debugging. *Communications of the ACM* 38(3):31-41, 1995.

[8] Chaudhuri, S., Christensen, E., Graefe, G., Narasayya, V., and Zwilling, M.J. Self-tuning technology in Microsoft SQL server. *Data Engineering Journal* 22(2):20-26, June 1999.

[9] Chaudhuri, S. and Weikum. G. Rethinking database system architecture: towards a self-tuning, RISC-style database system. In *Proc. of VLDB '00*, 2000.

[10]Corbato, F.J., Merwin-Daggett, M., and Daley, R.C. An experimental time-sharing system. In *Proc. of the AFIPS Fall Joint Computer Conference*, 1962.

[11] Dempster, A. P., Laird, N. M., Rubin, D. B. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, B, 39, 1-38, 1977.

[12] Feitelson, D.G., and Naaman, M. Self-tuning systems. *IEEE Software* 16(2):52-60, March/April 1999.

[13] Friedman, N., and Goldszmidt, M. Discretizing continuous attributes while learning Bayesian networks. In *Proc. of the 13th Int'l Conf.on Machine Learning (ICML),* 1996.

[14] Heckerman, D. A tutorial on learning with Bayesian networks. Technical report TR-95-06, Microsoft Research, March 1995 (revised November 1996).

[15] Horvitz, E., Breese, J., Heckerman, D., Hovel, D., and Rommelse, K. The Lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *Proc. of the 14th Conf. on Uncertainty in Artificial Intelligence*, 1998.

[16] Howard, R.A., and Matheson, J.E. Influence diagrams. In R. A. Howard and J. E. Matheson (eds.), *The Principles and Applications of Decision Analysis, Vol. II*, Strategic Decisions Group, Menlo Park, California, USA, 1984.

[17] Hugin Expert A/S. Hugin decision engine. www.hugin.dk.

[18]Irgon, A.E., Dragoni, A.H., and Huleatt, T.O. FAST: A large scale expert system for application and system software performance tuning. In *Proc. of ACM SIGMETRICS '88*, 1988.

[19]Jacobson, V. Congestion avoidance and control. In *Proc. of ACM SIGCOMM '88*, 1988.

[20] Jensen, F.V. *Bayesian Networks and Decision Graphs.* Springer Verlag, 2001.

[21] Massalin, H., and Pu, C. Fine-grain adaptive scheduling using feedback. *Computing Systems* 3(1):139-173, Winter 1990.

[22] Matthews, J.N., Roselli, D., Costello, A.M., Wang, R., and Anderson, T. In *Proc. of SOSP '97*, 1997.

[23] Menascé, D.A., Barbará, D., and Dodge, R. Preserving QoS of e-commerce sites through self-tuning: A performance model approach. In *Proc. of the 2001 ACM Conf. on Economic Commerce (EC '01)*, 2001.

[24] Narayanan, D., Flinn, J., and Satyanarayanan, S. Using history to improve mobile application adaptation. In *Proc. of the 3rd IEEE Workshop on Mobile Computing Systems and Applications*, 2000.

[25] Norsys Software. Netica API. www.norsys.com/netica.html.

[26] Olson, M., Bostic, K., and Seltzer, M. Berkeley DB. In *Proc. of the 1999 USENIX Technical Conf.*, 1999.

[27] Pearl, J. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

[28] Reiner, D., and Pinkerton, T. A method for adaptive performance improvement of operating systems. In *Proc. of ACM SIGMETRICS '81*, 1981.

[29] Seltzer, M.I., and Small, C. Self-monitoring and self-adapting operating systems. In *Proc. of the HotOS-VI*, 1997.

[30] Seltzer, M., Olson, M. Challenges in embedded database system administration. In *Proc. of the 1st Workshop on Embedded Systems*, 1999.

[31] Shachter, R.D. Evaluating influence diagrams, *Operations Research* 34(Nov-Dec):871-882, 1986.

[32] Sleepycat Software, Inc. *Berkeley DB*. New Riders Publishing, Indianapolis, IN, 2001.

[33] Spiro, P.M. Ubiquitous, self-tuning, scalable servers. In *Proc. of SIGMOD '98*, 1998.

[34] Steere,D.C., Goel, A., Gruenberg, J., McNamee, D., Pu, C., and Walpole, J. A feedback-driven proportion allocator for real-rate scheduling. In *Proc. of OSDI '99*, 1999.

[35] Sullivan, D.G. Using probabilistic reasoning to automate software tuning. Ph.D. thesis, Harvard University, September 2003.

[36] Weikum, G., Hasse, C., Monkeberg, A., and Zabback, P., The COMFORT Automatic Tuning Project, *Information Systems* 19(5):381-432, 1994.

[37]Welsh, W., Culler, D., and Brewer, E. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of SOSP '01*, 2001.