Active and Accelerated Learning of Cost Models for Optimizing Scientific Applications

Piyush Shivam
Duke University
shivam@cs.duke.edu

Shivnath Babu
Duke University
shivnath@cs.duke.edu

Jeff Chase
Duke University
chase@cs.duke.edu

ABSTRACT

We present the *NIMO* system that automatically learns cost models for predicting the execution time of computational-science applications running on large-scale networked utilities such as computational grids. Accurate cost models are important for selecting efficient plans for executing these applications on the utility. Computational-science applications are often scripts (written, e.g., in languages like Perl or Matlab) connected using a workflow-description language, and therefore, pose different challenges compared to modeling the execution of plans for declarative queries with well-understood semantics. NIMO generates appropriate training samples for these applications to learn fairly-accurate cost models quickly using statistical learning techniques. NIMO's approach is *active* and *noninvasive*: it actively deploys and monitors the application under varying conditions, and obtains its training data from passive instrumentation streams that require no changes to the operating system or applications. Our experiments with real scientific applications demonstrate that NIMO significantly reduces the number of training samples and the time to learn fairly-accurate cost models.

1. INTRODUCTION

High-performance computing has become a key requirement for rapid advances in a range of sciences including astrophysics, bioinformatics, systems biology, and climate modeling [15, 31]. This new area of *computational science* has given rise to many resource-intensive scientific applications. For example, modern high-energy particle detectors generate up to 10^{15} bytes of data for analysis per year [4]. Other sources of important scientific applications include BIRN [6], GEON [14], and SDSS [30].

The typical scientific application can be represented as a *workflow* consisting of one or more *batch tasks* linked in a directed acyclic graph (DAG) representing task precedence and data flow (e.g., [5]). Complex scientific workflows are often run on networked computing utilities—systems that allocate compute, network, and storage resources on demand from a large heterogeneous resource pool. Examples of networked utilities include clusters of machines [8], computational grids [13], utility data centers, PlanetLab, and outsourced storage services.

A number of researchers have recently pointed out the critical need for automated systems to manage scientific workflows [15, 31]. Database technology is well-suited to handle many aspects of workflow management as evident from the number of *WorkFlow Management Systems (WFMSs)* [31]—e.g., Griddb [23], GriPhyn [16], Kepler [2], and Zoo [20]—that exist to provide functionality such as modeling, execution, provenance, auditing, and visualization for workflows.

One aspect of workflow management where database technology can help significantly is *workflow planning* that involves finding an efficient *plan* for executing a workflow on a networked utility. Workflow planning is both important and challenging. Many scientific workflows perform complex computations, process very large amounts of data, or both. The difference in completion time can be on the order of days between a good execution plan for a workflow and a poor one [5]. These differences get magnified when workflows run on networked utilities composed of highly heterogeneous pools of geographically-distributed resources.

Example 1. Consider a motivating scenario where three sites A, B, and C comprise a networked utility. Suppose a user at site A wants to run a workflow composed of a single task G on the utility. The input data for G is stored at A. Site B has the fastest compute resources, but insufficient storage to store G's input data locally. Site C has faster compute resources than A and sufficient local storage for G's data. Candidate plans to execute G include:

P_1: Run G locally at A

P_2: Run G at B, so G gets the best compute resource available, but incurs remote I/O to A for data access

P_3: Stage G's data to C from A, and run G locally at C

The performance of these plans can vary significantly depending on G's characteristics and the underlying resource characteristics. For example, plan P_2 can be much more efficient than plans P_1 and P_3 if G does a lot of computation, but relatively little I/O. $\square$

A plan for a workflow G specifies a *resource assignment* $\vec{R}$ for each batch task in the workflow. A task may be a batch application or a *data-staging task* interposed between a pair of application tasks. $\vec{R}$ comprises the hardware resources—compute, network, and disk storage—that are assigned simultaneously to run G. G's performance can vary significantly across different resource assignments. To construct an effective resource assignment for G, the WFMS must predict the interaction of G's application-level characteristics (e.g., compute-to-communication ratio) with resource attributes (e.g., processor speed, cache size, and I/O system behaviors). Specifically, the system needs a cost model that can predict G's total execution time on $\vec{R}$, which is the most common performance metric for scientific workflows. Accurate cost models are important for selecting efficient resource assignments.

Workflow planning is similar to query optimization in database systems, but it poses an entirely new set of challenges. A task in a workflow G is typically a script in a programming language like Perl or Matlab. Hence, a WFMS usually has no prior knowledge about G's resource usage characteristics, or its performance sensitivity to the diverse hardware platforms comprising the underlying networked utility. Studies indicate that it is almost impractical to ask scientists to use a declarative language like SQL, or a single programming language, or even to add instrumentation code to tasks to help with modeling and workflow planning [15, 31]. Consequently, G is a *black-box* to the WFMS, making it challenging to generate an accurate cost model for G.

In previous work [32] we showed that accurate cost models can be learned using statistical techniques if the right training data is given. We transformed the problem of generating a cost model for a task G to that of learning a regression model that fits a set of m sample data points collected by running G on different resource assignments. Each sample s_i ($1 \leq i \leq m$) is a point in a high-dimensional space. s_i represents a complete run of G on a resource-assignment $\vec{R}$, and has the general form $\langle \rho_1, \rho_2, \ldots, \rho_k, T\rangle$, where each ρ_j is a hardware attribute of $\vec{R}$ (e.g., processor speed or disk seek time), and T is the total execution time of G on $\vec{R}$. Given the set of m samples $s_1, \ldots, s_m$, an appropriate regression model can be fitted to the training data to predict the execution time T from the values of attributes $\rho_1, \rho_2, \ldots, \rho_k$.

However, the challenge of acquiring the right training data remains unresolved. Three challenges arise in this setting:

- *Cost of sample acquisition:* Acquiring each sample may have high overhead. For example, a sample $\langle \rho_1, \ldots, \rho_k, T\rangle$ 'costs' time T to acquire, which may be on the order of hours or days for long-running scientific tasks.
- *Curse of dimensionality:* As the dimensionality of the data increases, the number of samples needed to attain a given level of accuracy can increase exponentially.
- *Operating range of samples:* The training sample set must represent the entire operating range of the system. A model learned from samples that cover only a limited range may not give accurate predictions across the entire system operating range.

Example 2. Acquiring samples corresponding to a mere 1% of a 5-dimensional space with 10 distinct values per dimension and average sample-acquisition time of 10 minutes, takes around 7 days. If the space becomes 8-dimensional, them the overall time becomes 19 years! However, if the system knows or learns, e.g., that the task is CPU-intensive for most resource assignments because it performs complex computations per unit of input data, then the dimensionality and the learning time can be reduced significantly. □

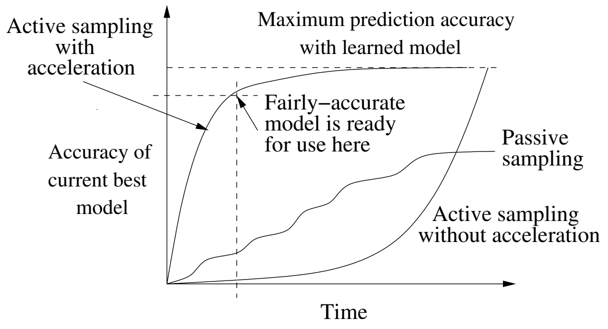

Figure 1: Active and accelerated learning

1.1 Contributions

- We present the *NIMO* system that learns cost models automatically for predicting task execution time on heterogeneous resources. NIMO performs *active sampling* of resource assignments to *accelerate* convergence to an accurate cost model for a task G. Active sampling acquires data to expose the relevant range of G's behavior by planning *experiments*. Each experiment runs G on a candidate resource assignment deployed in a *workbench* composed of heterogeneous resources. Active sampling with acceleration seeks to reduce the time before a reasonably-accurate cost model is available, as depicted in Figure 1. (The x-axis in Figure 1 shows the progress of time for collecting samples and learning models, and the y-axis shows the accuracy of the best model learned so far.)
- While NIMO actively deploys and monitors the task under varying conditions, NIMO is noninvasive in that it obtains its training data from passive instrumentation streams that require no changes to systems or applications. Specifically, NIMO can be applied to applications without changing application source or binary.
- We present experimental results that demonstrate how NIMO reduces the time to learn fairly-accurate cost models for real scientific applications.

2. NIMO

NIMO (NonInvasive Modeling for Optimization) is a workflow planning system that generates effective resource assignments for scientific workflows running on large-scale networked utilities. Figure 2 shows NIMO's overall architecture consisting of: (i) a *scheduler* that enumerates, selects, and executes plans for workflows; (ii) a modeling engine that consists of an *application profiler*, a *resource profiler*, and a *data profiler* that learns cost models for plans; and (iii) a workbench where NIMO proactively runs plans to collect samples for learning cost models. We describe each component in turn.

2.1 Scheduler

NIMO's scheduler is responsible for generating and executing a plan for a given workflow G. The scheduler enumerates candidate plans for G, estimates the cost of each plan, and chooses the execution plan with the minimum total execution time. A *plan* P for workflow G is an execution

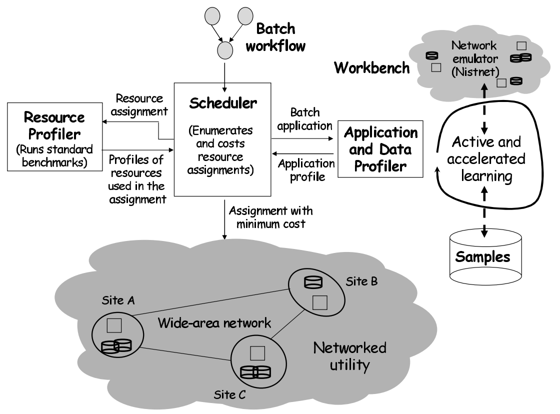
Batch workflow
Workbench
Network emulator (Nistnet)
Resource Profiler (Runs standard benchmarks)
Resource assignment
Profiles of resources used in the assignment
Scheduler (Enumerates and costs resource assignments)
Batch application
Application profile
Application and Data Profiler
Active and accelerated learning
Samples
Assignment with minimum cost
Site A
Site B
Site C
Wide-area network
Networked utility

Figure 2: Architecture of NIMO

strategy that specifies a resource assignment for each task in G. In addition to the batch tasks in G, P may also interpose additional tasks for staging data between each pair of batch tasks in G. For example, a staging task G_{ij} between tasks G_i and G_j in the workflow DAG, copies the parts of G_j's input dataset produced by G_i from G_i's storage resource to that of G_j. Example 1 illustrates such a *staging task*.

Let G_i, $1 \leq i \leq l$ be the tasks—including both batch and staging tasks—in a plan P. Let $\vec{R}_i = \langle C_i, N_i, S_i \rangle$ be the resource assignment made by P to task G_i. That is, when the scheduler schedules G_i on the networked utility, G_i executes on the compute resource C_i and accesses its input and output datasets from the storage resource S_i over the network resource N_i. (N_i will be null if S_i is local to C_i.) In this paper, we focus on individual tasks G_i or graphs G consisting of a single task. Our approach extends naturally to workflows with known structure.

The scheduler uses a cost model $M(G, I, \vec{R})$ to estimate the execution time of G with input dataset I on a resource assignment $\vec{R}$. Recall that the tasks in a plan P have a DAG relationship that restricts when each task can be scheduled. From this DAG and the estimated execution time of each task in P, the overall execution time of P can be estimated in a straightforward manner.

2.2 Workbench

NIMO's modeling engine automatically learns the cost model for G by deploying G on selected resource assignments, either to serve a real request, or proactively to use idle or dedicated resources (a 'workbench'; see Figure 2). Currently, NIMO uses a workbench that consists of a heterogeneous pool of compute, network, and storage resources to realize a variety of resource assignments. More details of the specific resources in NIMO's current workbench are given in Section 4.1. The details of instantiating a specific resource assignment in the workbench and conducting a run of G are described in Algorithm 2 in Section 3.1.

NIMO's modeling engine actively initiates new runs of G on selected resource assignments in the workbench to obtain sufficient training data for learning an accurate cost model for G in the shortest possible time. Instrumentation data is collected during a run, then aggregated to generate a sample data point as soon as the run completes (Algorithm 3 in Section 3.1). In keeping with NIMO's objective of being noninvasive, the collection of instrumentation data requires no changes to the tasks or the underlying system. Instead, NIMO relies only on high-level metrics collected by commonly-available monitoring tools: (i) processor and disk usage data is collected using the popular `sar` utility [29]; and (ii) network I/O measures are derived from the `nfsdump/nfsscan` tools [11].

2.3 Modeling Engine

NIMO's modeling engine is responsible for generating a cost model automatically for G. NIMO models the execution of task G as an interleaving of *compute phases*, in which the compute resource C is doing useful work, and *stall phases*, in which the compute resource is stalled on I/O. For the execution of task G with input dataset I on the resource assignment $\vec{R}$, we define:

- G's *compute occupancy* to process I on $\vec{R}$, denoted o_a, is the average time spent computing per unit of data flow processed by G.
- G's *stall occupancy* to process I on $\vec{R}$, denoted o_s, is the average time for which the compute resource is idle per unit of data flow. Stall occupancy $o_s = o_n + o_d$, where o_n and o_d capture the portions of occupancy caused by delays in the network and storage (disk) resources respectively.

G's total execution time to process I on $\vec{R}$ is:

$$Execution\ Time = D \times (o_a + o_s) = D \times (o_a + o_n + o_d) \quad (1)$$

Here, D is the total *data flow* processed by G, i.e., the total number of units of data read and written between the compute and storage resources allocated to G. The goal of the cost model $M(G, I, \vec{R})$ is to predict the occupancies o_a, o_n, o_d and the total data flow D for G processing dataset I on a resource assignment $\vec{R}$, so that the execution time can be estimated from Equation 1. Intuitively, task G's execution time to process I on $\vec{R}$ depends on:

- $\vec{R}$'s hardware characteristics, that NIMO captures by $\vec{R}$'s *resource profile*.
- I's data characteristics, that NIMO captures by I's *data profile*.
- G's application-level characteristics, that NIMO captures by G's *application profile*.

The cost model $M(G, I, \vec{R})$ uses these three profiles as inputs to estimate G's execution time to process I on $\vec{R}$. We describe each of the profiles in turn.

Resource Profile: A resource profile $\vec{\rho}$ is a vector $\langle \rho_1, \rho_2, \ldots, \rho_k \rangle$ where each ρ_i measures the value of some performance attribute of $\vec{R}$. For example, for a compute resource, the attributes may include processor speed, memory size, memory latency, and memory bandwidth. In the single task cases considered in this paper, the resource profile of a resource assignment $\vec{R} = \langle C, N, S \rangle$ represents the compute, network, and storage resources assigned to run the task.

Data Profile: The data profile $\vec{\lambda}$ of a task's input dataset I captures I's data characteristics such as total size and data distribution.

Application Profile: G's application profile consists of four *predictor functions* $\langle f_a(\vec{\rho}, \vec{\lambda}), f_n(\vec{\rho}, \vec{\lambda}), f_d(\vec{\rho}, \vec{\lambda}), f_D(\vec{\rho}, \vec{\lambda}) \rangle$

that predict G's occupancies $\langle o_a, o_n, o_d \rangle$ and total data flow D on a resource assignment $\vec{R}$ and input dataset I, as a function of $\vec{R}$'s resource profile $\vec{\rho}$ and I's data profile $\vec{\lambda}$.

Cost Model $M(G, I, \vec{R})$: Given task G's application profile $\langle f_a, f_n, f_d, f_D \rangle$, the resource profile $\vec{\rho}$ of a candidate resource assignment $\vec{R}$, and input dataset I's data profile $\vec{\lambda}$, G's execution time to process I on $\vec{R}$ is:

$$Execution\ Time = f_D(\vec{\rho}, \vec{\lambda}) \times (f_a(\vec{\rho}, \vec{\lambda}) + f_n(\vec{\rho}, \vec{\lambda}) + f_d(\vec{\rho}, \vec{\lambda})) \quad (2)$$

2.4 Problem Setting and Assumptions

In this paper we focus on how NIMO automatically generates the cost model $M(G, I, \vec{R})$ for a task. Our current prototype of NIMO makes some limiting assumptions that we plan to relax in future work:

- NIMO associates a specific dataset I along with a cost model for a task G. That is, a separate cost model is built for each task-dataset combination. The advantage is that the variable parameters in the predictor functions in G's application profile now consist of the resource-profile attributes only, and not the data-profile attributes. That is, the predictor functions have the simpler form $f(\vec{\rho})$ instead of $f(\vec{\rho}, \vec{\lambda})$. While the problem of automatically learning G's predictor functions is simplified, it largely remains the same and non-trivial, as seen in Section 3. The disadvantage is that NIMO has to learn a new cost model for each new input dataset for G. However, many scientific applications are often run repeatedly on the same input dataset, and these runs tend to have similar resource-usage behavior [25].
- Workflow planning using cost models in NIMO requires that any resource that is shared simultaneously among applications (e.g., a central storage server) is *virtualized*, so that we can control what fraction of the resource is used by each task. (This assumption is made implicitly in the cost models in most database systems.) While current sharing mechanisms do not provide full performance isolation, or provide it for only a subset of resources, resource virtualization is an active research area and the deployable software continues to improve (e.g., [10, 22]). Developing cost models that account for the complex interactions generated by shared access to resources remains an area for future work.
- NIMO requires that the resources assigned to a task remain constant throughout the execution of the task. This assumption is made implicitly in the cost models in most database systems that we are aware of, and it gives an interesting direction for future work.

2.5 Learning Profiles Proactively

We describe how NIMO automatically learns data profiles for input datasets and resource profiles for hardware resources. The more challenging problem of learning application profiles automatically is discussed in Section 3.

The data profile for an input dataset I in NIMO is currently limited to I's total size in bytes. We obtain resource profiles by running standard benchmark suites that are designed to expose the differences that are most significant for the performance of real applications. We use *whetstone* [9] to calibrate processor speeds, *lmbench* [24] to calibrate memory latency and bandwidth, and *netperf* [28] to calibrate the network latency and bandwidth between compute and storage resources. We found in previous work that these benchmarks are sufficient for making accurate predictions in our environment [32]. Our approach is independent of the specific benchmarks as long as they capture the underlying resource characteristics. Other researchers have: (1) confirmed that simple benchmarks can be used in profiling high-performance computing platforms [7]; (2) studied benchmark selection for comprehensive coverage [34]; and (3) devised strategies for robust resource profiling in the presence of competition for shared resources [33].

3. LEARNING APPLICATION PROFILES

NIMO's modeling engine uses active and accelerated learning to generate the predictor functions comprising G's application profile. Recall from Section 2.4 that the modeling engine builds a separate cost model for each task-dataset combination, so we consider the cost model for G that processes an input dataset I, denoted $G(I)$. To learn $G(I)$'s cost model, we have to learn $G(I)$'s predictor functions $\langle f_a(\vec{\rho}), f_n(\vec{\rho}), f_d(\vec{\rho}), f_D(\vec{\rho}) \rangle$, where $\vec{\rho}$ represents the resource profile of the resources $\vec{R} = \langle C, N, S \rangle$ assigned to run $G(I)$. Specifically, $f_a(\vec{\rho})$ predicts the compute occupancy o_a of $G(I)$ on $\vec{R}$ as a function of a subset of the resource-profile attributes $\langle \rho_1, \rho_2, \ldots, \rho_k \rangle$ of $\vec{R}$; similarly, $f_n(\vec{\rho})$ predicts the network-stall occupancy o_n, $f_d(\vec{\rho})$ predicts the disk-stall occupancy o_d, and $f_D(\vec{\rho})$ predicts the total data flow D.

If NIMO is given a reasonably large and representative set of samples of the form $\langle \rho_1, \rho_2, \ldots, \rho_k, o_a, o_n, o_d, D \rangle$, then the problem of learning accurate predictor functions $\langle f_a(\vec{\rho})$, $f_n(\vec{\rho})$, $f_d(\vec{\rho})$, $f_D(\vec{\rho}) \rangle$ reduces to a statistical-learning problem of fitting accurate regression functions to predict each of o_a, o_n, o_d, and D using subsets of attributes in $\rho_1, \rho_2, \ldots, \rho_k$. The challenge, however, is that NIMO does not initially have a representative sample set for training. Instead, NIMO has to collect each sample by running $G(I)$ to completion on a selected resource assignment in the workbench. The total overhead of collecting training samples can be extremely high because of the curse of dimensionality—resource profile $\vec{\rho} = \langle \rho_1, \ldots, \rho_k \rangle$ may contain many attributes (i.e., k may be large)—and the high cost of data acquisition per sample—collecting a sample $\langle \rho_1, \ldots, \rho_k, o_a, o_n, o_d, D \rangle$ involves a complete run of G on $\vec{R}$, which could take up to hours or days for some scientific tasks.

Algorithm 1 illustrates the main steps that NIMO uses to learn $G(I)$'s application profile. (Details of these steps are discussed in Sections 3.1–3.6.) The algorithm consists of an initialization step and a loop. The loop continuously refines the accuracy of the predictor functions by learning from new samples acquired by running $G(I)$ on new resource assignments instantiated in the workbench.

3.1 Initialization

The initialization step of Algorithm 1 (Step 1) runs the task $G(I)$ on a designated *reference resource assignment* $\vec{R}_{ref} = \langle C_{ref}, N_{ref}, S_{ref} \rangle$. Based on this run, NIMO measures the compute, network-stall, and disk-stall occupancies—called the *reference occupancies*—and the total data flow—called the *reference data flow*—of $G(I)$ on $\vec{R}_{ref}$. The details of this step follow from: (i) Algorithm 2, which shows how

Algorithm 1: Active and accelerated learning of predictor functions $f_a(\rho_1, \ldots, \rho_k)$, $f_n(\rho_1, \ldots, \rho_k)$, $f_d(\rho_1, \ldots, \rho_k)$, and $f_D(\rho_1, \ldots, \rho_k)$ for task $G(I)$

```
1) Initialize: (Section 3.1) Obtain reference occupancies
   $o_{aref}$, $o_{nref}$, $o_{dref}$ and reference data flow $D_{ref}$ for
   $G(I)$ on a reference resource assignment $\vec{R}_{ref}$; Set
   $f_a(\vec{\rho}) = o_{aref}$, $f_n(\vec{\rho}) = o_{nref}$, $f_d(\vec{\rho}) = o_{dref}$, and
   $f_D(\vec{\rho}) = D_{ref}$ (constant functions);
2) Design the next experiment: (Sections 3.2–3.4)
   2.1 Select a predictor function for refinement,
       denoted $f$ (Section 3.2);
   2.2 Should an attribute from $\rho_1, \ldots, \rho_k$ be added to
       the set of attributes already used in $f$? If yes,
       then pick the attribute to be added (Section 3.3);
   2.3 Select new assignment(s) to refine $f$ using the set
       of attributes from Step 2.2 (Section 3.4);
3) Conduct the chosen experiment: (Section 3.5)
   3.1 Run $G(I)$ in the workbench using the
       assignment(s) picked in Step 2.3;
   3.2 After each run, generate the corresponding
       sample $\langle \rho_1, \ldots, \rho_k, o_a, o_n, o_d, D \rangle$, where $o_a, o_n, o_d$
       are the observed occupancies and $D$ is the
       observed total data flow;
   3.3 Learn $f$ (and other predictor functions) from the
       new sample set;
4) Compute current prediction error: (Section 3.6)
   Compute current prediction error of each predictor. If
   the overall error in predicting execution time is below
   a threshold, and a minimum number of samples have
   been collected, then stop, else go to Step 2.
```

Algorithm 2: Running $G(I)$ on $\vec{R} = \langle C, N, S \rangle$

```
1) Instantiate a Network File System (NFS) server on
   the storage resource $S$ in $\vec{R}$. Export a storage volume
   from $S$ containing $G$'s input dataset $I$, and mount this
   volume on $C$. Set $G(I)$ to access this volume;
2) Set routing tables in $C$ and $S$ so that all
   communication happens via a specific router $r$ running
   NIST Net [32]. $r$ is configured to emulate the network
   specifications (e.g., latency and bandwidth) of $S$;
3) Start monitoring tools (Section 2.2) to measure the
   execution time $T$ and $C$'s utilization $U$ (required by
   Algorithm 3) for this run;
4) Start $G(I)$ on $C$. When the task finishes, stop the
   monitoring tools, and compute $T$ and $U$.
```

Algorithm 3: Computing task $G(I)$'s occupancies on $\vec{R} = \langle C, N, S \rangle$

```
1) Using Algorithm 2, run $G(I)$ on $\vec{R}$, and measure $C$'s
   average utilization $U$, $G(I)$'s execution time $T$, and
   the total data flow D (using network I/O traces);
2) Solve for $o_a$ and $o_s$ from $U = \frac{o_a}{o_a+o_s}$, $\frac{D}{T} = \frac{1}{o_a+o_s}$;
3) Use network I/O traces to derive the average time
   spent per I/O in the network resource $N$ and in the
   storage resource $S$;
4) Split $o_s = o_n + o_d$ into $o_n$ and $o_d$ in proportion to the
   ratio of network and storage components of the
   average I/O time from Step 3, to obtain $\langle o_a, o_n, o_d, D \rangle$.
```

NIMO runs a task on a resource assignment instantiated in the workbench; and (ii) Algorithm 3, which shows how NIMO computes the occupancies and total data flow for a run. Specifically, the initialization step runs $G(I)$ on $\vec{R}_{ref}$ using Algorithm 2, then it measures $G(I)$'s occupancies and total data flow on $\vec{R}_{ref}$ using Algorithm 3.

Once NIMO computes the reference occupancies in Step 1 of Algorithm 1, it initializes the predictor functions to constant functions that predict the compute, network-stall, and disk-stall occupancies and the total data flow of $G(I)$ on any resource assignment $\vec{R}$ as equal to the corresponding reference values; which is a reasonable thing to do based on the single run of $G(I)$ so far. NIMO refines the predictor functions in Algorithm 1 as it collects more sample data points.

There are many ways in which NIMO can choose the reference assignment $\vec{R}_{ref} = \langle C_{ref}, N_{ref}, S_{ref} \rangle$ from the different candidate assignments available in the workbench:

- *Random assignment (Rand)*: Pick each of C_{ref}, N_{ref}, and S_{ref} at random from among the corresponding resources in the workbench.
- *High-capacity assignment (Max)*: Pick the compute resource with the fastest processor speed, the network resource with minimum latency, and the storage resource with maximum transfer rate.
- *Low-capacity assignment (Min)*: Pick the compute resource with the slowest processor speed, the network resource with maximum latency, and the storage resource with minimum transfer rate.

We evaluate these strategies experimentally in Section 4.2.

3.2 Guiding the Sequence of Exploration for the Predictor Functions

In each iteration of Algorithm 1, Step 2.1 picks a specific predictor function to refine by collecting more sample points for training. We consider both static and dynamic schemes to guide this sequence for exploring the predictor functions across iterations.

Static Schemes: A static scheme first decides a *total ordering* of the predictor functions $f_a(\vec{\rho})$, $f_n(\vec{\rho})$, $f_d(\vec{\rho})$, and $f_D(\vec{\rho})$, then defines a fixed *traversal plan* for picking the predictor function to refine in each iteration. NIMO currently supports two techniques each for ordering and for traversal. The two ordering techniques are:

- *Domain-knowledge-based* where a domain expert specifies a total order of the predictor functions to NIMO. For example, the expert may know that the scientific task $G(I)$ is likely to be CPU-intensive for most resource assignments because $G(I)$ performs complex computations per unit of data in I, so $f_a(\vec{\rho})$ should come first in the total order and be refined first.
- *Relevance-based* where NIMO estimates the relevance of the predictor functions on $G(I)$ using the classic

Algorithm 4: Dynamic scheme for picking the predictor function to refine in an iteration of Algorithm 1

```
1) Let $s_1, \ldots, s_m$ be the $m$ training samples of the form
   $\langle \rho_1, \rho_2, \ldots, \rho_k, o_a, o_n, o_d, D \rangle$ collected so far;
2) Supplement each of the $m$ samples with the predicted
   compute occupancy $o_{ap}$ from the current
   $f_a(\rho_1, \ldots, \rho_k)$; similarly, the predicted network-stall
   occupancy $o_{np}$ from the current $f_n(\rho_1, \ldots, \rho_k)$, the
   predicted disk-stall occupancy $o_{dp}$ from the current
   $f_d(\rho_1, \ldots, \rho_k)$, and the predicted data flow $D_p$ from
   the current $f_D(\rho_1, \ldots, \rho_k)$;
3) Use the $m$ actual and predicted value-pairs $\langle o_a, o_{ap} \rangle$ to
   compute the current prediction error $E_a$ of
   $f_a(\rho_1, \ldots, \rho_k)$ (see Section 3.6); similarly, compute $E_n$,
   $E_d$, and $E_D$ from the respective $\langle o_n, o_{np} \rangle$, $\langle o_d, o_{dp} \rangle$,
   and $\langle D, D_p \rangle$ pairs;
4) Pick for refinement the predictor function with
   maximum current prediction error.
```

Plackett-Burman design with foldover (*PBDF*) statistical technique [34]. NIMO orders the predictor functions in decreasing order of effect. To order the four predictor functions using PBDF, NIMO performs eight runs of $G(I)$ on predefined resource assignments. Appendix A describes the PBDF technique.

The two techniques to traverse a given total order are:

- *Round-robin* where NIMO chooses the predictor functions to refine across iterations in a round-robin fashion from the given total order.
- *Improvement-based* where NIMO traverses the total order from beginning to end, and keeps refining the current predictor function until the reduction in the prediction error obtained in the last iteration drops below a predefined threshold. (The computation of the current prediction error of a predictor function is described in Section 3.6.) When the reduction in error drops below the threshold, NIMO moves on to the next predictor function in the total order. When it exhausts all predictor functions, it resumes at the beginning of the total order.

Dynamic Schemes: Dynamic schemes do not use a static ordering of the predictor functions. Instead, the function to refine in each iteration is based on the training samples collected so far. NIMO currently considers one dynamic scheme that, in each iteration, chooses to refine the predictor function with the maximum current prediction error, as illustrated in Algorithm 4.
We evaluate the different schemes in Section 4.3.

3.3 Adding New Attributes to Predictor Functions

Step 2.2 of Algorithm 1 decides when to add a new resource-profile attribute to a predictor function $f(\vec{\rho})$, and if so, which of the k attributes from $\rho_1, \ldots, \rho_k$ to add for maximum potential reduction in $f(\vec{\rho})$'s prediction error. (Recall from Step 1 of Algorithm 1 that $f(\vec{\rho})$ is initially set to a constant function having no variable parameters.) As in Section 3.2, NIMO's twofold strategy is to first define a total order over the $\rho_1, \ldots, \rho_k$ attributes with respect to $f(\vec{\rho})$, and then to define a traversal plan based on this order to select attributes for inclusion in $f(\vec{\rho})$.

Following a approach similar to the one in Section 3.2, a total ordering of the resource-profile attributes $\rho_1, \ldots, \rho_k$ for predictor function $f(\vec{\rho})$ can be:

- *Domain-knowledge-based* where a domain expert specifies a total ordering of $\rho_1, \ldots, \rho_k$ for $f(\vec{\rho})$. For example, the expert may know that the task has a purely sequential I/O pattern. Thus, the memory-size attribute may have minimal effect on the compute occupancy o_a, so this attribute can be placed towards the end of the total order for $f_a(\vec{\rho})$.
- *Relevance-based* where NIMO first estimates the effect of each resource-profile attribute on the occupancy predicted by $f(\vec{\rho})$ using PBDF (Appendix A). Then, it orders the resource-profile attributes in decreasing order of effect.

Based on the total ordering of attributes $\rho_1, \ldots, \rho_k$ for a predictor function $f(\vec{\rho})$, NIMO decides when to add the next attribute in the total order to the current set of attributes in $f(\vec{\rho})$. The *improvement-based* approach that NIMO uses here adds the next attribute in the order when the reduction achieved in prediction error during an iteration with the current $f(\vec{\rho})$ (i.e., with the current set of attributes) falls below a predefined threshold. When NIMO exhausts all attributes, it resumes at the beginning of the total order. We evaluate these alternatives in Section 4.4.

3.4 Selecting New Sample Assignments

Step 2.3 of Algorithm 1 chooses new assignments to run task $G(I)$ and collect new samples for learning. To create a new assignment $\vec{R}$, NIMO needs to select a value of each attribute ρ_i in $\vec{R}$'s resource profile, while accounting for:

1. Covering the full operating range of ρ_i to avoid learning functions that are accurate only for a narrow range of ρ_i's values. For example, limited variation of processor speed may fail to expose latency-hiding behavior due to prefetching [32]. If the processor speed is sufficiently low, then the rate of I/O requests from the processor may be low enough that prefetching can hide the I/O latency completely.
2. Capturing the important interactions among attributes. In the above example, the point where latency-hiding behavior shows up as the processor speed is reduced depends on the network latency between the compute and storage resources. Therefore, the effect of changing processor speed on the occupancy o_a of the compute resource may depend on the value of network latency [32], representing an interaction between the processor speed and network latency attributes of the resource profile. This interaction may or may not have a significant effect on overall execution time depending on the task's I/O characteristics.

Figure 3 shows the range of techniques for selecting new samples that we are currently experimenting with in NIMO. The techniques are shown in terms of their general performance and tradeoff on the two metrics above, namely, covering the

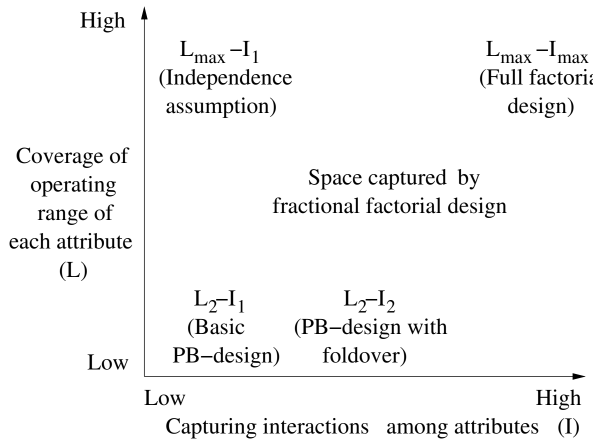
High
Low
Coverage of operating range of each attribute (L)
L$_{max}$–I$_1$ (Independence assumption)
L$_{max}$–I$_{max}$ (Full factorial design)
Space captured by fractional factorial design
L$_2$–I$_1$ (Basic PB–design)
L$_2$–I$_2$ (PB–design with foldover)
Low
High
Capturing interactions among attributes (I)

Figure 3: Techniques for selecting new sample assignments (PB = Plackett-Burman)

Algorithm 5: Selecting the next sample assignment using L_{max}-I_1

```
1) Let $f(\vec{\rho})$ be the current predictor function chosen for
   refinement in Step 2.1 of Algorithm 1. Let $\rho_r$ be the
   attribute most recently added to $f(\vec{\rho})$ in Step 2.2 of
   Algorithm 1. Let $IN \subset \vec{\rho}$ be the set of attributes
   already considered for addition in $f(\vec{\rho})$, and $OUT \subset \vec{\rho}$
   be the set of attributes not considered for addition in
   $f(\vec{\rho})$. Let $\vec{R}_{ref}$ be the designated reference assignment
   chosen in the initialization step of Algorithm 1;
2) $L_{max}$-$I_1$ chooses the new assignment as follows:
    1. All attributes in $IN$ and $OUT$ are set to the
       corresponding values in $\vec{R}_{ref}$;
    2. The value of $\rho_r$ is set to the next unselected value
       from the binary-search sequence—$lo$, $hi$, $\frac{lo+hi}{2}$,
       $\frac{3lo+hi}{4}$, $\frac{lo+3hi}{4}$, $\frac{7lo+hi}{8}$, $\frac{5lo+3hi}{8}$, and so on—where
       $lo$ and $hi$ are the minimum and maximum values
       $\rho_r$ can take. That is, $lo$ is chosen for the first
       assignment, $hi$ is chosen for the next assignment,
       and so on.
```

operating range of attributes, and capturing significant interactions among attributes. We use an L_α-I_β naming format, where (i) α represents the number of significant distinct values, or *levels*, in the attribute's operating range covered by the technique, and (ii) β represents the largest degree of interactions among attributes guaranteed to be captured by the technique. Among these techniques, the ones for which we report experimental results in Section 4.5 are:

- L_{max}-I_1: This technique, described in Algorithm 5, systematically explores all levels of a newly-added attribute using a binary-search-like approach. However, it assumes that the effects of attributes are independent of each other, so it chooses values for attributes independent of one another.
- L_2-I_2: This technique is an adaptation of PBDF which we describe in Appendix A. (PBDF is an example of the popular *fractional factorial design* in statistics [21, 34].) Given the total number of attributes, L_2-I_2 specifies the number of samples required and the values of attributes in each sample. L_2-I_2 captures two levels (e.g., *low* and *high* [34]) per attribute and up to pairwise interactions among attributes.

Algorithm 6: Learning $G(I)$'s compute occupancy predictor function $f_a(\vec{\rho})$

```
1) Suppose $m$ runs of $G$ have been conducted, where run
   $i$ is on $\vec{R}_i$. Let $\langle \rho_{1_i}, \ldots, \rho_{j_i} \rangle$ be the subset of $\vec{R}_i$'s
   resource-profile attributes added to $f_a(\vec{\rho})$ so far;
2) Use Algorithm 3 to generate $m$ training data points
   where the $i$th point is $\langle \rho_{1_i}, \ldots, \rho_{j_i}, o_{a_i} \rangle$;
3) Normalize the training points using a *baseline*
   assignment $\vec{R}_b$ with resource profile $\vec{\rho}_b$. (Currently,
   NIMO chooses $\vec{R}_b = \vec{R}_{ref}$.) Let $G$'s compute
   occupancy on $\vec{R}_b$ be $o_{a_b}$, so the $i$th normalized
   training data point is $\langle \frac{\rho_{1_i}}{\rho_{1_b}}, \ldots, \frac{\rho_{j_i}}{\rho_{j_b}}, \frac{o_{a_i}}{o_{a_b}} \rangle$;
4) Use regression on the training data to learn a function
   $F(\vec{\rho})$ that predicts the value of $\frac{o_a}{o_{a_b}}$ from the
   normalized values of $\rho_1, \ldots, \rho_j$. Set $f_a(\vec{\rho}) = o_{a_b} \times F(\vec{\rho})$.
```

3.5 Performing the Selected Experiment

In Step 3.1 of Algorithm 1, NIMO instantiates the assignment selected in Step 2.3 in the workbench and runs the task; details of running task $G(I)$ on a resource assignment $\vec{R}$ are given in Algorithm 2. The compute, network-stall, and disk-stall occupancies, and the total data flow, are collected from the run as described in Algorithm 3. These measures give a new sample data point of the form $\langle \rho_1, \ldots, \rho_k, o_a, o_n, o_d, D \rangle$. NIMO then analyzes all the sample data points collected so far, including the one collected most recently, to refine the predictor function chosen in Step 2.1 of Algorithm 1 with its current attribute set as chosen in Step 2.2. If the latest run provides a new sample for another predictor f based on the current set of attributes included in f, then NIMO refines f as well. The details of this step are given in Algorithm 6 for f_a; f_n, f_d, and f_D can be learned similarly.

3.6 Computing Current Prediction Error

NIMO considers two techniques for computing the current prediction error of a predictor function $f(\vec{\rho})$:

1. *Cross-validation*: In this technique, NIMO uses *leave-one-out* cross-validation to estimate the current prediction error of $f(\vec{\rho})$ [1]. For each sample s out of the m samples collected so far, NIMO learns $f(\vec{\rho})$ using all samples other than s (using Algorithm 6). NIMO then uses $f(\vec{\rho})$ to predict the corresponding occupancy for s, and computes the *absolute percentage error*. For example, if the predictor function is $f_a(\vec{\rho})$, and the actual and predicted occupancies for s are o_a and o_{ap} respectively, then the absolute percentage error is $\frac{|o_a - o_{ap}|}{o_a} \times 100\%$. The average of the m individual values of absolute percentage error, denoted *Mean Absolute Percentage Error (MAPE)*, is the current prediction error.
2. *Fixed test set*: In this technique, NIMO designates a small subset of resource assignments in the workbench

as an *internal test set*. The test assignments may be a random subset of the possible assignments in the workbench, or chosen more robustly; different choices for internal test set selection are evaluated in Section 4.6. When a fixed test set is used, the initialization step of Algorithm 1 begins by running the task on each assignment in the test set. NIMO computes the current prediction error of $f(\vec{\rho})$ as the MAPE in predicting occupancy on each assignment in the test set. Note that the samples collected for this test set are never used as training samples for any predictor function.

4. EXPERIMENTAL EVALUATION

Our experimental evaluation has two goals: (i) to evaluate the different algorithmic choices introduced in Section 3; and (ii) to show that NIMO significantly reduces the overall time to learn fairly-accurate cost models.

4.1 Setup

Applications We consider four biomedical scientific tasks to evaluate active and accelerated learning in NIMO. The four tasks are *BLAST* [3], *NAMD* [26], *CardioWave* [27], and *fMRI* [19]. *BLAST*, *NAMD*, and *CardioWave* are typically CPU-intensive, while *fMRI* is typically I/O-intensive.[1] In this paper, we use *BLAST* by default for demonstrating the performance of our algorithms.

Workbench NIMO's workbench (Section 2.2) in our experiments consists of five compute nodes with speeds: 451 MHz, 797 MHz, 930 MHz, 996 MHz, and 1396 MHz; cache sizes: 256 KB or 512 KB, and the Intel PIII architecture. We use the boot parameters on the compute nodes to vary the memory size of the nodes across 5 sizes ranging from 64 MB to 2 GB. The tasks run on the Linux operating system with the 2.4.25 kernel. We used `NIST Net` to impose 6 different network round-trip latencies in the range $0-18$ ms and 10 different network bandwidths in the range 20 Mbps to 100 Mbps. The underlying heterogeneity gives us a large sample space of resource assignments. For example, with 5 CPU speeds, 5 memory sizes, and 6 network latencies, we have a maximum of 150 candidate resource assignments for each batch task on our workbench. We choose assignments from these 150 candidates in our experiments.

Regression functions Recall from Algorithm 6 that NIMO uses regression to learn the predictor functions in the cost model. NIMO currently uses multivariate linear regression [21]. (More sophisticated regression techniques, e.g., *transform regression* [35] can be applied in NIMO without changing the overall approach, and we plan to do so in future work.) A typical predictor function in our experiments has the form: $f(\vec{\rho}) = a_1 g_1(\rho_1) + a_2 g_2(\rho_2) + \cdots + a_k g_n(\rho_k) + c$, where each a_i is a regression coefficient, each ρ_i is a resource-profile attribute, each g_i is a transformation function, and c is a constant. Apart from the default $g(\rho_i) = \rho_i$ transformation, we also consider reciprocal transformations. For example, a reciprocal transformation is applied to the CPU speed attribute because occupancy values are inversely proportional to CPU speed. The experiments reported in this section focus on learning the three occupancy predictor functions f_a, f_n, and f_d automatically, and assume that the data-flow predictor f_D is known.

[1]Technically, a task can be CPU- or I/O-intensive depending on the underlying resource assignment.

Step	**Alternatives**
Initialization	Min*, Rand, Max
Predictor refinement	Static + Round-Robin*, Static + Improvement-based, Dynamic
Attribute addition	Relevance-based (PBDF)*, Static
Sample selection	L_{max}-I_1^*, L_2-I_2
Prediction error	Cross-Validation*, Fixed Test Set (Random), Fixed Test Set (PBDF)

Table 1: Choices for steps of Algorithm 1. * denotes the default in experiments unless otherwise noted

Evaluation The metric we use to report the current accuracy of a cost model M in our experiments is M's Mean Absolute Percentage Error (Section 3.6) in predicting total execution time on an *external test set* of 30 resource assignments chosen randomly from the workbench. Note that the external test set is different from the internal test set used by NIMO to compute the current prediction error (Section 3.6), and is never exposed to NIMO for training or testing.

Accelerated learning of predictor functions depends on the choices made at the different steps of Algorithm 1 as explained in Section 3. We evaluate various alternatives for each of the following five steps:

1. *Initialization*: The reference assignment that decides the starting point in the resource assignment search space (Section 3.1)
2. *Predictor refinement*: The order and traversal NIMO uses to refine the predictor functions (Section 3.2)
3. *Attribute addition*: The order and traversal NIMO uses to add attributes to each predictor function (Section 3.3)
4. *Sample selection*: The choice of value for each resource-profile attribute to generate a new sample assignment for a run of the task (Sections 3.4 and 3.5)
5. *Prediction error*: The technique to compute the current prediction error at any point in time (Section 3.6)

While evaluating any of these 5 factors, we fix the choices for the other 4 factors to defaults as shown in Table 1.

4.2 Initialization

The reference assignment serves several purposes in NIMO: (i) starting sample assignment for the learning algorithm (Algorithm 1); (ii) baseline for normalizing training samples (Algorithm 6); and (iii) reference for setting attribute values during sample selection (Algorithm 5). Different reference assignments may lead to completely different training samples, and hence, different MAPE statistics. We begin our experimental results with the evaluation of alternatives from Section 3.1 for choosing the reference assignment. Note that we fix the choice for each other step of Algorithm 1 to the default given in Table 1.

Figure 4 shows the impact of three alternatives for choosing the reference assignment on the overall accuracy and convergence time of the automatically-learned cost model for *BLAST*: (a) a randomly chosen assignment (*Rand*); (b) a high capacity assignment (*Max*); and (c) a low capacity assignment (*Min*). Each point in the figure corresponds to the MAPE when a new sample is added to the training data or a new attribute is added to a predictor during learning.

We can make the following observations from Figure 4: (i) the plots start at different times; (ii) the MAPE values

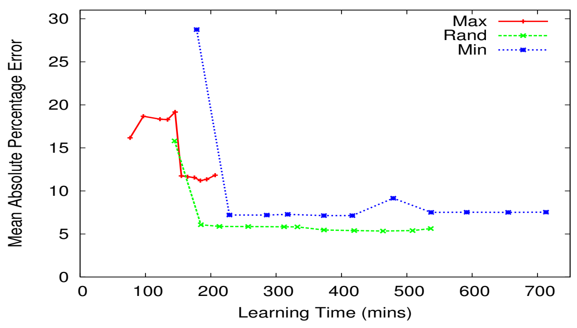

Figure 4: Impact of different alternatives for choosing the reference assignment (*BLAST* application)

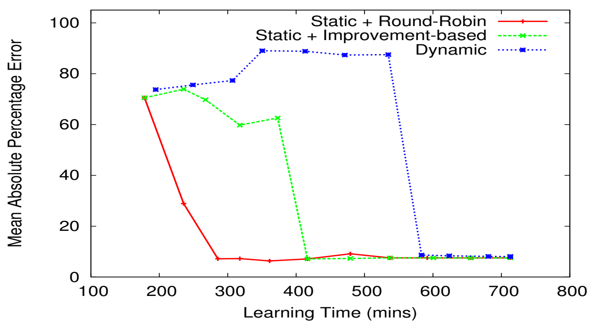

Figure 5: Impact of different alternatives for refining the predictor function (*BLAST* application)

do not converge smoothly, e.g., there may be a sharp drop when a new training point is added; and (iii) while *Max* converges in the shortest time to a reasonably-accurate cost model, *Min* and *Rand* converge to models with lower errors. We explain these observations next.

Among the three alternatives, the reference assignment in *Max* has the maximum resource capacity, so it results in the shortest time to finish the first run and generate a training sample. Also, note that in the default L_{max}-I_1 strategy for sample selection, only one attribute in any new sample assignment is set to a value different from the corresponding value in the reference assignment. Hence, *Max* will generate new training samples at a faster rate than *Min* or *Rand*.

The nonsmooth nature of the plots in Figure 4 is a consequence of NIMO's online exploration of the space of resource assignments to learn predictor functions with the right attributes. The prediction errors may drop sharply, e.g., when a relevant attribute is added to a predictor. Recall that the MAPE values in Figure 4 are based on an external test set that is never exposed to NIMO for training or testing.

Min and *Rand* converge to cost models with lower errors than *Max*. Our hypothesis is that the set of training samples produced when *Min* or *Rand* is used is more representative of the space of sample assignments than when *Max* is used. That is, *Min* and *Rand* may be leading to training sets that capture the operating range of relevant attributes and the significant interactions among attributes better.

4.3 Exploration Sequence for Predictors

The sequence in which NIMO explores predictor functions for refinement across iterations of Algorithm 1 determines the time to learn accurate cost models. In Figure 5 we evaluate the static and dynamic strategies from Section 3.2 for guiding predictor refinement through ordering and traversal. As usual, choices for the other steps are the defaults given in Table 1. The strategies we compare in Figure 5 are: (i) static order f_d, f_a, f_n + round-robin traversal; (ii) static order f_d, f_a, f_n + improvement-based traversal; and (iii) dynamic ordering and traversal.

The main observations from Figure 5 are: (i) round-robin traversal performs better than improvement-based traversal for static ordering; and (ii) the dynamic strategy takes the longest to converge and shows the most nonsmooth behavior.

Improvement-based traversal of predictors is sensitive to the order in which the predictors are refined as well as the improvement threshold used (Section 3.2). In Figure 5, the static order is the nonoptimal f_d, f_a, f_n order—the actual relevance order computed using PBDF is f_n, f_a, f_d—and the improvement threshold is 2%—i.e., we move to the next predictor in the order when the reduction in prediction error with the current predictor falls below 2%. The MAPE of the improvement-based strategy remains high until f_n starts being refined (around 400 minutes), when it drops sharply. On changing the static order to f_n, f_a, f_d, the improvement-based strategy learns an accurate cost model quickly (as shown by the *Min* plot in Figure 4). Round-robin traversal of the static order acquires samples for each predictor in turn, so it is less sensitive to the correctness of the order or the threshold.

The accuracy-driven dynamic strategy performs the worst in Figure 5. Recall from Section 3.2 that the dynamic strategy chooses to refine the predictor with the maximum current prediction error. In Figure 5, the dynamic strategy gets stuck initially in a local minima where it keeps refining f_a until all samples for the attributes in f_a are exhausted, and f_n starts being refined (around 550 minutes). The problem with the dynamic strategy is that the current prediction error of a predictor f is not representative of f's relevance to the total task execution time.

4.4 Adding New Attributes to Predictors

Accurate learning of predictor functions can happen only when relevant attributes are added quickly to the functions. Recall from Section 3.4 that the attributes in the predictor function currently chosen for refinement dictate which sample assignments are selected for training. Our next experimental results show that adding attributes to predictors in an incorrect order can delay convergence to accurate cost models. We consider the two alternatives from Section 3.3 for determining the order in which attributes are added to predictors:

- Relevance-based ordering that determines relevant attributes and their order using PBDF as: (i) f_a—*cpu speed*, *memory size*, (ii) f_n—*network latency*, *memory size*, and (iii) f_d—*network latency*.
- Static ordering set as: (i) f_a—*network latency*, *memory size*, *cpu speed*, (ii) f_n—*cpu speed*, *memory size*, *network latency*, and (iii) f_d—*cpu speed*, *memory size*, *network latency*. The static ordering is kept different from the relevance-based ordering to show the importance of adding attributes in the right order.

Figure 6 compares the two alternatives. While the relevance-based order learns an accurate cost model quickly, the in-

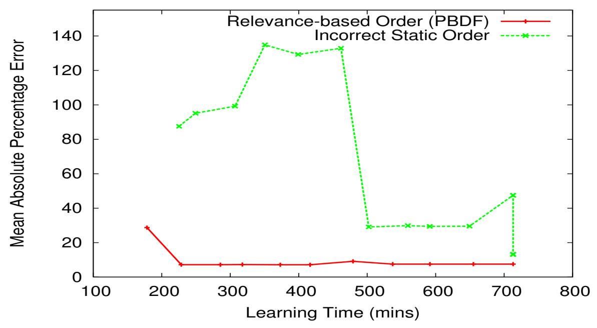

Figure 6: Impact of alternatives for adding new attributes to a predictor function (*BLAST* application)

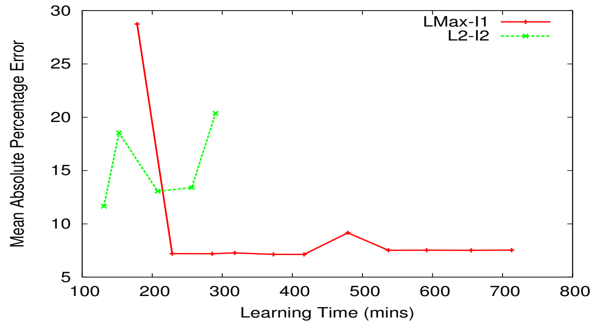

Figure 7: Impact of alternatives for selecting new sample assignments (*BLAST* application)

correct static order causes nonsmooth behavior and slow convergence.

4.5 Selecting New Sample Assignments

A good sample-selection strategy must cover the operating range of relevant attributes and expose all significant interactions among attributes while acquiring only a small number of samples. We evaluate two strategies from Section 3.4 for sampling new assignments: L_{max}-I_1 and L_2-I_2. Recall that the L_{max}-I_1 strategy covers the operating range of relevant attributes, but it may fail to expose significant interactions among attributes. The L_2-I_2 strategy adds training samples one at a time from the design matrix specified by PBDF (Appendix A). L_2-I_2 considers only two levels from the operating range of each attribute, but it can expose significant two-way interactions among attributes.

Figure 7 compares the two alternatives. Here we observe that L_{max}-I_1 converges quickly to an accurate cost model, while L_2-I_2 fails to converge. Our hypothesis is that the simple L_{max}-I_1 strategy is enough to expose any significant interactions among attributes for *BLAST*. On the other hand, with only two levels considered for each attribute, L_2-I_2 fails to obtain good regression functions for the predictors. We are now exploring sampling schemes that can guarantee the capture of significant interactions among attributes and also provide good coverage of the operating range of relevant attributes.

4.6 Computing Current Prediction Error

An important component of NIMO's accelerated learning algorithm is the computation of current prediction error for each predictor. This error is used by other steps of

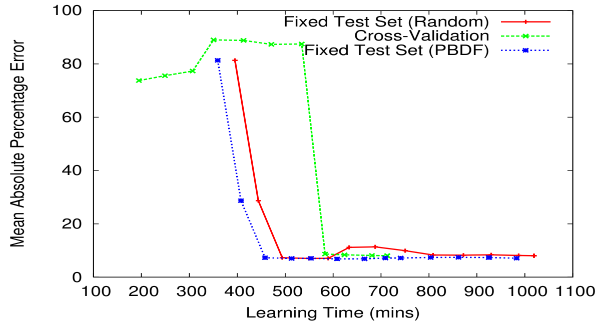

Figure 8: Impact of alternatives for computing the current prediction error (*BLAST* application)

Algorithm 1, e.g., by the improvement-based traversal and the dynamic strategy for choosing the predictor function to refine. We consider the two strategies from Section 3.6 for computing the prediction error: (i) leave-one-out cross-validation using all samples collected so far; and (ii) using a fixed internal test set. The fixed test set is chosen in two ways: (a) a set of 10 assignments chosen randomly from the space of possible assignments; and (b) a set of 8 assignments chosen from the samples specified by PBDF. The results are shown in Figure 8. Here, we use the accuracy-driven dynamic strategy for refining the predictor functions to study the impact of internal test sets on MAPE. All other factors are set to their defaults as shown in Table 1.

Figure 8 shows the strengths and weaknesses of the approaches. Compared to fixed test sets, cross-validation starts producing results earlier, but it shows nonsmooth behavior and slow convergence. Cross-validation produces its initial error estimates from the very few samples collected so far, causing the observed nonsmooth behavior. However, these estimates get more accurate over the course of active learning as more samples are collected. The fixed test set approach requires an upfront investment of time to obtain the test samples, which delays the start of the learning process. However, fixed test sets give more robust estimates of prediction error because these sets are representative of the total sample space in terms of capturing operating ranges and attribute interactions.

4.7 Summary of Experimental Results

We apply NIMO's active and accelerated learning to three real biomedical applications other than *BLAST*. Table 2 shows the time to learn an accurate cost model and the corresponding MAPE values for all four applications. The table shows that as the attribute space gets larger, NIMO reduces the time to learn fairly-accurate cost models by an order of magnitude compared to approaches that first sample a significant part of the entire space and then build models all-at-once (the active sampling without acceleration strategy in Figure 1).

We summarize the results of our experiments evaluating the algorithmic choices presented in Section 3:

- The *Min* approach tends to select reference assignments that produce training sets that are representative of the total sample space.
- Unlike the improvement-based strategy, round-robin traversal is not sensitive to the (static) ordering of predictors, nor does it require a predefined threshold.

Appl.	#Attrs	MAPE	NIMO's Learning Time (hrs)	Learning Time for All Samples (hrs)	Sample Space Used (%)
BLAST	3	10	12	130	10
fMRI	3	10	4	112	10
NAMD	2	4	2	16	25
C. Wave	2	10	2	16	25

Table 2: Gains from active and accelerated learning

Round-robin traversal also avoids the local-optima problem of dynamic approaches that are based on current prediction error.

- Adding attributes in relevance order based on PBDF is a good approach for adding new attributes to predictors. Other attribute orders may significantly delay convergence to accurate cost models.
- In our experiments, the L_{max}-I_1 strategy for selecting new sample assignments performs better than L_2-I_2 mainly because of the limited attribute operating range considered by L_2-I_2.
- A fixed internal test set, chosen randomly or using PBDF, is a reasonable choice for computing current prediction error. Cross-validation-based approaches show nonsmooth behavior and slow convergence.

5. RELATED WORK

The importance of applying database technology to manage the modeling, execution, provenance, auditing, and visualization of scientific workflows is emphasized in, e.g., [15, 31]. Traditional work on cost modeling in centralized and distributed relational databases (e.g., [17]) assume that the execution plans are composed of operators belonging to a small well-defined family of operators. This assumption does not hold for scientific workflows.

Recently, statistical learning methods have been used to develop cost models for: (i) complex user-defined functions, e.g., [18]; (ii) remote autonomous database systems in the multidatabase setting, e.g., [36]; and (iii) complex XML operators [35]. The general approach is to first identify a set of query and data features that potentially determine operator costs, and then to use given training data to learn the relationship among the values of the identified features and operator cost. NIMO differs from this category of work in two ways: (i) it addresses the problem of automatically acquiring the right training data to minimize the overall learning time; and (ii) it considers the applications as black-boxes and relies only on passive measurement streams to make our work more widely applicable.

The theory of *design of experiments (DOE)* is a well established branch of statistics that studies planned investigation of factors affecting system performance [21]. *Active learning* [12] from the machine-learning literature deals with the issue of picking the next sample that provides the most information to maximize the accuracy of an objective function. NIMO uses both DOE and active learning in an iterative fashion for learning cost models for scientific applications. More recently, the computer architecture community has also looked at DOE for improving system simulation methodology [34].

6. CONCLUSIONS AND FUTURE WORK

We presented the NIMO system that uses an active and accelerated approach for learning cost models for predicting the execution time of computational-science applications running on large-scale networked utilities. NIMO is noninvasive in that it uses training data from passive instrumentation streams collected using common profiling tools, requiring no changes to the operating system or applications. Our experimental results indicate that NIMO can learn fairly-accurate cost models quickly for real scientific applications. There are many avenues for future work:

- To be fully self-managing, NIMO needs an algorithm that can automatically select the best combination of choices for each step of Algorithm 1 for a given application.
- To handle an application A whose resource usage is highly data dependent, NIMO needs to capture the data dependency using attributes in the data profile of A's input dataset. Identifying the right set of attributes in the data profile for a black-box application is a challenging problem.
- NIMO currently makes some limiting assumptions regarding cost models that need to be addressed, e.g., predictors use multivariate linear regression with predetermined transformations and do not account for resource sharing.

7. REFERENCES

[1] A. Allen. *Probability, Statistics, and Queuing Theory with Computer Science Applications*. Academic Press, 1990.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *Proc. of SSDBM*, Jun 2004.

[3] S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[4] Grid Physics Network in Atlas. `www.usatlas.bnl.gov/computing/grid/griphyn`.

[5] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit Control in a Batch-Aware Distributed File System. In *Proc. of the USENIX Symp. on Networked Systems Design and Implementation*, Mar 2004.

[6] Biomedical Informatics Research Network. `www.nbirn.net`.

[7] L. Carrington, M. Laurenzano, A. Snavely, R. Campbell, and L. Davis. How Well Can Simple Metrics Represent the Performance of HPC Applications? In *Proc. of the ACM/IEEE Conf. on Supercomputing*, Nov 2005.

[8] Condor High Throughput Computing. `www.cs.wisc.edu/condor`.

[9] H. J. Curnow and B. A. Wichmann. A Synthetic Benchmark. *The Computer Journal*, 19(1):43–49, Feb 1976.

[10] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proc. of the ACM Symp. on Operating Systems Principles*, Oct 2003.

[11] D. Ellard and M. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proc. of the Annual Large Installation System Administration Conf.*, Oct 2003.

[12] V. Fedorov. *Theory of Optimal Experiments*. Academic Press, 1972.

[13] I. Foster and C. Kesselman. The Grid2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 2004.

[14] Cyberstructure for the Geosciences. `www.geongrid.org`.

[15] J. Gray, D. Liu, M. Nieto-Santisteban, A. Szalay, G. Heber, and D. DeWitt. Scientific Data Management in the Coming Decade. Technical Report MSR-TR-2005-10, Microsoft Research, Jan 2005.

[16] Grid Physics Network. `www.griphyn.org`.

[17] L. Haas, M. Carey, M. Livny, and A. Shukla. Seeking the Truth About Ad Hoc Join Costs. *VLDB Journal*, 3(6):241–256, 1997.

[18] Z. He, B. Lee, and R. Snapp. Self-Tuning UDF Cost Modeling Using the Memory-Limited Quadtree. In *Proc. of EDBT*, Mar 2004.

[19] S. A. Huettel, A. W. Song, and G. McCarthy. *Functional Magnetic Resonance Imaging*. Sinauer Associates, Inc., 2004.

[20] Y. Ioannidis, M. Livny, A. Ailamaki, A. Narayanan, and A. Therber. Zoo: A Desktop Experiment Management Environment. In *Proc. of SIGMOD*, Jun 1997.

[21] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, May 1991.

[22] W. Jin, J. S. Chase, and J. Kaur. Interposed Proportional Sharing for a Storage Service Utility. In *Proc. of the Joint Intl. Conf. on Measurement and Modeling of Computer Systems*, Jun 2004.

[23] D. Liu and M. Franklin. The Design of Griddb: A Data-Centric Overlay for the Scientific Grid. In *Proc. of VLDB*, Sep 2004.

[24] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proc. of the USENIX Annual Technical Conf.*, Jan 1996.

[25] B. K. Pasquale and G. C. Polyzos. A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload. In *Proc. of ACM/IEEE Conf. on Supercomputing*, Nov 1993.

[26] J. C. Phillips, R. Braun, et al. Scalable Molecular Dynamics with NAMD. *Journal of Computational Chemistry*, 26:1781–1802, 2005.

[27] J. Pormann, J. Board, D. Rose, and C. Henriquez. Large-Scale Modeling of Cardiac Electrophysiology. In *Proc. of Computers in Cardiology*, Sep 2002.

[28] Netperf: A Network Performance Benchmark. `www.cup.hp.com/netperf/NetperfPage.html`.

[29] *Performance Monitoring Tools for Linux*. `perso.wanadoo.fr/sebastien.godard`.

[30] Sloan Digital Sky Survey. `www.sdss.org`.

[31] S. Shankar, A. Kini, D. DeWitt, and J. Naughton. Integrating Databases and Workflow Systems. *SIGMOD Record*, 3(34):5–11, 2005.

[32] P. Shivam, S. Babu, and J. Chase. Learning Application Models for Utility Resource Planning. In *Intl. Conf. on Autonomic Computing*, Jun 2006.

[33] S. Vazhkudai and J. M. Schopf. Using Regression Techniques to Predict Large Data Transfers. *Intl. Journal of High Performance Computing Applications*, cs.DC/0304037, 2003.

[34] J. J. Yi, D. J. Lilja, and D. M. Hawkins. A Statistically Rigorous Approach for Improving Simulation Methodology. In *Proc. of Intl. Symp. on High Performance Computer Architecture*, Feb 2003.

[35] N. Zhang, P. Hass, V. Josifovski, G. Lohman, and C. Zhang. Statistical Learning Techniques for Costing XML Queries. In *Proc. of Intl. Conf. on VLDB*, Aug-Sep 2005.

[36] Q. Zhu and P. Larson. Building Regression Cost Models for Multidatabase Systems. In *Proc. of PDIS*, 1996.

APPENDIX

A. PLACKETT-BURMAN DESIGN

We describe the Plackett-Burman design with foldover (PBDF) technique to rank N independent parameters X_1, …, X_N based on their effect on a dependent parameter Y. PBDF requires that $N + 1$ be a multiple of 4, with dummy parameters added as needed. PBDF performs a set of runs where the configuration of $X_1, \ldots, X_N$ for each run is given by a specific design matrix. The general matrix is described in [34]. Here, we give an example. Table 3 shows a part of the design matrix when $N = 7$. This matrix has $2 * 8 = 16$ rows because 8 is the nearest higher multiple of 4 for $N = 7$, and PBDF requires twice that many rows. Each row specifies the configuration of $X_1, \ldots, X_N$ for a run. The value of Y given by the run is also shown. A "+" ("–") value for a parameter represents a value that is higher (lower) that the normal range of values for that parameter.

X_1	X_2	X_3	X_4	X_5	X_6	X_7	Y
+1	+1	+1	-1	+1	-1	-1	y_1
-1	+1	+1	+1	-1	+1	-1	y_2
-1	-1	+1	+1	+1	-1	+1	y_3
⋯	⋯	⋯	⋯	⋯	⋯	⋯	
-1	-1	+1	-1	+1	+1	-1	y_{15}
+1	+1	+1	+1	+1	+1	+1	y_{16}

Table 3: Design matrix for PBDF

After performing all 16 runs, the *effect* of each parameter on Y is computed by multiplying the parameter's value in each row with the value of Y for the row, taking the sum across all rows, and taking the absolute value of the sum. For example, the effect of parameter X_1 is $|y_1 - y_2 - y_3 \cdots - y_{15} + y_{16}|$. Finally, the parameters are ranked in decreasing order of their effect on Y.