Application Classification through Monitoring and Learning of Resource Consumption Patterns

Jian Zhang and Renato J. Figueiredo
University of Florida
Dept. of Electrical and Computer Engineering Gainesville, FL 32611 USA
{jianzh,renato}@acis.ufl.edu

Abstract

Application awareness is an important factor of efficient resource scheduling. This paper introduces a novel approach for application classification based on the Principal Component Analysis (PCA) and the k-Nearest Neighbor (k-NN) classifier. This approach is used to assist scheduling in heterogeneous computing environments. It helps to reduce the dimensionality of the performance feature space and classify applications based on extracted features. The classification considers four dimensions: CPU-intensive, I/O and paging-intensive, network-intensive, and idle. Application class information and the statistical abstracts of the application behavior are learned over historical runs and used to assist multi-dimensional resource scheduling. This paper describes a prototype classifier for application-centric Virtual Machines. Experimental results show that scheduling decisions made with the assistance of the application class information, improved system throughput by 22.11% on average, for a set of three benchmark applications.

1. Introduction

Heterogeneous distributed systems that serve application needs from diverse users face the challenge of providing effective resource scheduling to applications. Resource awareness and application awareness are necessary to exploit the heterogeneities of resources and applications to perform adaptive resource scheduling. In this context, there has been substantial research on effective scheduling policies [38][33][37] with given resource and application specifications. There are several methods for obtaining resource specification parameters (e.g. CPU, memory, disk information from /proc in Unix systems). However, application specification is challenging to describe because of the following factors:

Numerous types of applications: In a closed environment where only a limited number of applications are running, it is possible to analyze the source codes of each application or even plug in codes to indicate the application execution stages for effective resource scheduling. However, in an open environment such as in Grid computing, the growing number of applications and lack of knowledge or control of the source codes present the necessity of a general method of learning application behaviors without source code modifications.

Multi-dimensionality of application resource consumption: An application's execution resource requirement is often multi-dimensional. That is, different applications may stretch the use of CPU, memory, hard disk or network bandwidth to different degrees. The knowledge of which kind of resource is the key component in the resource consumption pattern can assist resource scheduling.

Multi-stage applications: There are cases where long-running scientific applications exhibit multiple execution stages. Different execution stages may stress different kinds of resources to different degrees, hence characterizing an application requires knowledge of its dynamic run-time behavior. The identification of such stages presents opportunities to exploit better matching of resource availability and application resource requirement across different execution stages and across different nodes. For instance, with process migration techniques [29][15] it is possible to migrate an application during its execution for load balancing.

The above characteristics of grid applications present a challenge to resource scheduling: How to learn and make use of an application's multi-dimensional resource consumption patterns for resource allocation? This paper introduces a novel approach to solve this problem: application classification based on the feature selection algorithm, Principal Component Analysis (PCA), and K-Nearest Neighbor (k-NN) classifier [18][14]. The PCA is applied to reduce the dimensionality of application performance metrics, while preserving the maximum amount of variance in the metrics. Then, the k-Nearest Neighbor algorithm is used to categorize the application execution states into different classes based on the application's resource consumption pattern. The learned application class information is used to assist the resource scheduling decision-making in heterogeneous computing environments.

The application classifier is inspired by the VMPlant [26] project, which provides automated cloning and configuration of application-centric Virtual Machines (VMs). Problem-solving environments such as In-VIGO [11] can submit requests to the VMPlant service, which is capable of cloning an application-specific virtual machine and configuring it with an appropriate execution environment. In the context of VMPlant, the application can be scheduled to run on a *dedicated virtual machine*, which is hosted by a *shared physical machine*. Within the VM, system performance metrics such as CPU load, memory usage, I/O activity and network bandwidth utilization, reflect the application's resource usage.

The classification system described in this paper leverages the capability of summarizing application performance data by collecting system-level data within a VM, as follows. During the application execution, snapshots of performance metrics are taken at a desired frequency. A PCA processor analyzes the performance snapshots and extracts the key components of the application's resource usage. Based on the extracted features, a k-NN classifier categorizes each snapshot into one of the following classes: CPU-intensive, IO-intensive, memory-intensive, network-intensive and idle.

By using this system, resource scheduling can be based on a comprehensive diagnosis of the application resource utilization, which conveys more information than CPU load in isolation. Experiments reported in this paper show that the resource scheduling facilitated with application class composition knowledge can achieve better average system throughput than scheduling without the knowledge.

The rest of the paper is organized as follows: Section 2 briefly introduces VMPlant and motivates the need for a classifier. The PCA and the k-NN classifier are described in the context of application classification in Section 3. Section 4 presents the classification model and implementation. Section 5 presents and discusses experimental results of classification performance measurements. Section 6 discusses related work. Conclusions and future work are discussed in Section 7.

2. Virtual Machines and VMPlant

A "classic" virtual machine enables multiple independent, isolated operating systems to run on one physical machine, efficiently multiplexing system resources of the host machine [23]. A virtual machine is highly customizable, in terms of hardware (memory, hard disk, devices) as well as software resources (operating system, user applications and data). It provides a secure and isolated environment for application execution [20].

The VMPlant Grid service [26] builds upon VM technologies and provides support for automated creation and flexible configuration of virtual machine execution environments. Customized, application-specific VMs can be defined in VMPlant with the use of a directed acyclic graph (DAG) configuration. VM execution environments defined within this framework can then be cloned and dynamically instantiated to provide a homogeneous application execution environment across distributed resources.

The application behavior learning proposed in this paper is designed to capture the behavior of an application running in a dedicated virtual machine created by a VM scheduler. The *physical* machine upon which it is instantiated, however, is time- and space-shared across many VM instances. Through the decoupling between virtual and physical machines, the system performance metrics collected using existing mechanisms (e.g. /proc) *within* a VM summarize and reflect the resource consumption of an application independently from others.

The application behavior knowledge gained from the learning process over its historical runs can be used to assist the resource reservation on the virtual machine's host (physical) servers for VM platforms such as [1]. The classifier described in this paper can assist VMPlant or other VM-based schedulers by providing information that can be used to determine what resources are needed from the physical host to match an application's desired quality of service.

3. Learning Algorithms

Application behavior can be defined by its resource utilization, such as CPU load, memory usage, network and disk bandwidth utilization. In principle, the more information a scheduler knows about an application, the better scheduling decisions it can make. However, there is a tradeoff between the complexity of decision-making process and the optimality of the decision. The key challenge here is how to find a representation of the application, which can describe multiple dimensions of resource consumption, in a simple way. This section describes how the pattern classification techniques, the PCA and the K-NN classifier, are applied to achieve this goal.

A pattern classification system consists of pre-processing, feature extraction, classification, and post-processing. The pre-processing and feature extraction are known to significantly affect the classification, because the error caused by wrong features may propagate to the next steps and stays predominant in terms of the overall classification error. In this work, a set of application performance

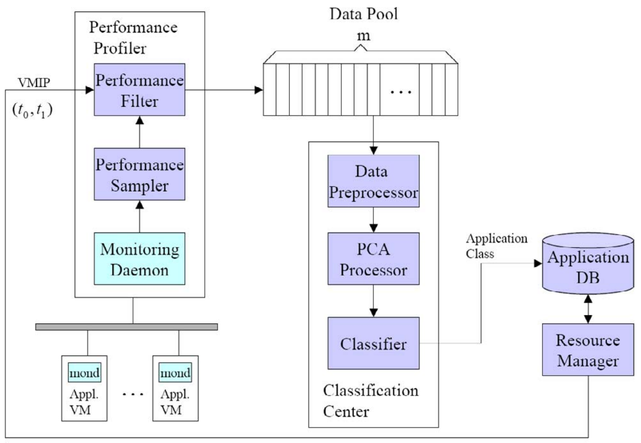


Figure 1. Application classification model
The *Performance profiler* collects performance metrics of the target application node. The *Classification center* classifies the application using extracted key components and performs statistic analysis of the classification results. The *Application DB* stores the application class information. (m is the number of snapshots taken in one application run, t_0/t_1: are the beginning ending times of the application execution, *VMIP* is the IP address of the application's host machine).

metrics are chosen based on expert knowledge and the principle of increasing relevance and reducing redundancy [39].

Principal Component Analysis (PCA) [18] is a linear transformation representing data in a least-square sense. When a set of vector samples are represented by a set of lines passing through the mean of the samples, the best linear directions result in eigenvectors of the scatter matrix - the so-called "principal components". The corresponding eigenvalues represent the contribution to the variance of data. When the k largest eigenvalues of n principal components are chosen to represent the data, the dimensionality of the data reduces from n to k.

K-Nearest Neighbor classifier (k-NN) is used in this paper. The k-NN classifier decides the class by considering the votes of k (an odd number) nearest neighbors. The nearest neighbor is picked as the training data geometrically closest to the test data in the feature space.

In this work, a vector of the application's resource consumption snapshots is used to represent the application. Each snapshot consists of a chosen set of performance metrics. The PCA is used to preprocess the raw data to independent features for the classifier. Then, a 3-NN classifier is used to classify each snapshot. The majority vote of the snapshots' classes is used to represent the class of the applications: CPU-intensive, I/O and paging-intensive, network-intensive, or idle. A machine with no load except for background load from system daemons is considered as in idle state.

4. Application Classification Model and Implementation

The application classifier is composed of a performance profiler, a classification center, and an application database (DB) as shown in Figure 1. In addition, a monitoring system is used to sample the system performance of a computing node running an application of interest.

4.1. The Performance Profiler

The performance profiler is responsible for collecting performance data of the application node. It interfaces with the resource manager to receive data collection instructions, including the target node and when to start and stop.

The performance profiler can be installed on any node with access to the performance metric information of the application node. In our implementation, the Ganglia [30] distributed monitoring system is used to monitor application nodes. The performance sampler takes snapshots of the performance metrics collected by Ganglia at a predefined frequency (currently, 5 seconds) between the application's starting time t_0 and ending time t_1. Since Ganglia uses multicast based on a listen / announce protocol to monitor the machine state, the collected samples consist of the performance data of all the nodes in a subnet. The performance filter extracts the snapshots of the target application for future processing. At the end of profiling, an application performance data pool is generated. The data pool consists of a set of n dimensional samples $\mathbf{A}_{n\times m} = (\mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_m)$, where $m = (t_1 - t_0)/d$ is the number of snapshots taken in one application run and d is the sampling time interval. Each sample $\mathbf{a}_i$ consists of n performance metrics, which include all the default 29 metrics monitored by Ganglia and the 4 metrics that we added based on the need of classification, including the number of I/O blocks read from/written to disk, and the number of memory pages swapped in/out. A program was developed to collect these four metrics (using *vmstat*) and the metrics were added to the metric list of Ganglia's *gmond*.

4.2. Classification Center

The classification center has three components: the data preprocessor, the PCA processor, and the classifier. To reduce the computation intensity and improve the classification accuracy, it employs the PCA algorithm to extract the principal components of the resource usage data collected and then performs classification based on extracted data of the principal components.

$$\overbrace{\begin{pmatrix} a_{11} & \cdots & a_{1m}\\ a_{21} & \cdots & a_{2m}\\ \vdots & \ddots & \vdots\\ \vdots & \ddots & \vdots\\ a_{n1} & \cdots & a_{nm}\end{pmatrix}}^{\mathbf{A}_{n\times m}} \xrightarrow[n\geq p]{\text{Preprocess}} \overbrace{\begin{pmatrix} a'_{11} & \cdots & a'_{1m}\\ a'_{21} & \cdots & a'_{2m}\\ \vdots & \ddots & \vdots\\ a'_{p1} & \cdots & a'_{pm}\end{pmatrix}}^{\mathbf{A'}_{p\times m}} \xrightarrow[p\geq q]{\text{PCA}} \overbrace{\begin{pmatrix} b_{11} & \cdots & b_{1m}\\ \vdots & \ddots & \vdots\\ b_{q1} & \cdots & b_{qm}\end{pmatrix}}^{\mathbf{B}_{q\times m}} \xrightarrow[q\geq 1]{\text{Classify}} \overbrace{(c_{11}\cdots c_{1m})}^{\mathbf{C}_{1\times m}} \xrightarrow{\text{Vote}} Class$$

m: The number of snapshots taken in one application run, n: The number of performance metrics,
$\mathbf{A}_{n\times m}$: All performance metrics collected by monitoring system,
$\mathbf{A'}_{p\times m}$: The selected relevant performance metrics after the zero-mean and unit-variance normalization,
$\mathbf{B}_{q\times m}$: The extracted key component metrics, $\mathbf{C}_{1\times m}$: The class vector of the snapshots,
$Class$: The application class, which is the majority vote of snapshots' classes.

Figure 2. Performance feature space dimension reductions in the application classification process

4.2.1. Data Preprocessing Based on Expert Knowledge

Based on the expert knowledge, we identified 4 pairs of performance metrics as shown in Table 1. Each pair of the performance metrics correlates to the resource consumption behavior of the specific application class and has limited redundancies. For example, performance metrics of CPU_System and CPU_User are correlated to CPU-intensive applications; Bytes_In and Bytes_Out are correlated to Network-intensive applications; IO_BI and IO_BO are correlated to the IO-intensive applications; Swap_In and Swap_Out are correlated to Memory-intensive applications. The data preprocessor extracts these eight metrics of the target application node from the data pool based on our expert knowledge. Thus it reduces the dimension of the performance metric from $n = 33$ to $p = 8$ and generates $\mathbf{A'}_{p\times m}$ as shown in Figure 2. In addition, the preprocessor also normalizes the selected metrics to zero-mean and unit-variance.

4.2.2. PCA Based Feature Selection

The PCA processor takes the data collected for the performance metrics listed in Table 1 as inputs. It conducts the linear transformation of the performance data and selects the principal components based on the predefined minimal fraction variance. In our implementation, the minimal fraction variance was set to extract exactly two principal components. Therefore, at the end of processing, the data dimension gets further reduced from $p = 8$ to $q = 2$ and the vector $\mathbf{B}_{q\times m}$ is generated, as shown in Figure 2.

4.2.3. Training and Classification

The 3-Nearest Neighbor classifier is used for the application classification in our implementation. It is trained by a set of carefully chosen applications based on expert knowledge. Each application represents the key performance characteristics of a class. For example, an I/O benchmark program, PostMark [2], is used to represent the IO-intensive

Performance Metrics	Description
CPU_System / User	Percent CPU_System / User
Bytes_In / Out	Number of bytes per second into / out of the network
IO_BI / BO	Blocks sent to / received from block device (blocks/s)
Swap_In / Out	Amount of memory swapped in / out from / to disk (kB/s)

Table 1. Performance metric list

class. SPECseis96 [19], a scientific computing intensive program, is used to represent the CPU-intensive class. A synthetic application, Pagebench, is used to represent the Paging-intensive class. It initializes and updates an array whose size is bigger than the memory of the VM, thereby inducing frequent paging activity. Ettcp [3], a benchmark that measures the network throughput over TCP or UDP between two nodes, is used as the training application of the Network-intensive class. The performance data of all these four applications and the idle state are used to train the classifier. For each test data, the trained classifier calculates its distance to all the training data. The 3-NN classification identifies only three training data sets with the shortest distance to the test data. Then the test data's class is decided by the majority vote of the three nearest neighbors.

4.3. Post Processing and Application Database

At the end of classification, an m dimension class vector $\mathbf{c}_{1\times m} = (c_1, c_2, \cdots, c_m)$ is generated. Each element of the vector $\mathbf{c}_{1\times m}$ represents the class of the corresponding application performance snapshot. The majority vote of the snapshot classes determines the application Class. The complete performance data dimension reduction process is shown in Figure 2. In addition to a single value (*Class*) the application classifier also outputs class composition, which can be used to support application cost models

(Section 4.4). The post processed classification results together with the corresponding execution time ($t_1 - t_0$) are stored in the application database and can be used to assist future resource scheduling.

4.4. Applications in Cost-based Scheduling

The application class output provided by the proposed system can be used by resource providers and users to establish cost-based scheduling models. For example, a cost model may be conceived where the unit application execution time cost is calculated as the weighted average of the units costs of different resources: UnitApplicationCost = α.cpu% + β.mem% + γ.io% + δ.net% + ϵ.idle%, where $\alpha, \beta, \gamma, \delta$, and ϵ are the unit costs of CPU, memory, I/O, and network capacity, which are defined by the resource providers. The cpu%, mem%, io%, net% and idle% are the application class compositions, i.e. the outputs of the application classifier. The model gives the resource provider the flexibility to define their individualized pricing schemes.

5. Empirical Evaluation

We have implemented a prototype for application classification including a Perl implementation of the performance profiler and a Matlab implementation of the classification center. In addition, Ganglia was used to monitor the working status of the virtual machines. This section evaluates our approach from the following three aspects: the classification ability, the scheduling decision improvement and the classification cost.

5.1. Classification Ability

The application class set in this experiment has four classes: CPU-intensive, I/O and paging-intensive, network-intensive, and idle. Application of I/O and paging-intensive class can be further divided into two groups based on whether they have or do not have substantial memory intensive activities. Various synthetic and benchmark programs, scientific computing applications and user interactive applications are used to test the classification ability. These programs represent typical application behaviors of their classes. Table 2 summarizes the set of applications used as the training and the testing applications in the experiments [4][19][5][2][6][3][34][7][8][9][10]. The 3-NN classifier was trained with the performance data collected from the executions of the training applications highlighted in the table. All the application executions were hosted by a VMware GSX virtual machine (VM1). The host server of the virtual machine was an Intel(R) Xeon(TM) dual-CPU 1.80GHz machine with 512KB cache and 1GB RAM. In addition, a second virtual machine with the same specification was used to run the server applications of the network benchmarks.

Initially the performance profiler collected data of all the thirty-three ($n = 33$) performance metrics once every five seconds ($d = 5$) during the application execution. Then the data preprocessor extracted the data of the eight ($p = 8$) metrics listed in Table 1 based on the expert knowledge of the correlation between these metrics and the application classes. After that, the PCA processor conducted the linear transformation of the performance data and selected principal components based on the minimal fraction variance defined. In this experiment, the variance contribution threshold was set to extract two ($q = 2$) principal components. This helps to reduce the computational requirements of the classifier. Then, the trained 3-NN classifier conducts classification based on the data of the two principal components.

The training data's class clustering diagram is shown in Figure 3 (a). The diagram shows a PCA-based two-dimensional representation of the data corresponding to the five classes targeted by our system. After being trained with the training data, the classifier classifies the remaining benchmark programs shown in Table 2. The classifier provides outputs in two kinds of formats: the application class-clustering diagram, which helps to visualize the classification results, and the application class composition, which can be used to calculate the unit application cost.

Figure 3 shows the sample clustering diagrams for three test applications. For example, the interactive VMD application (Figure 3(d)) shows a mix of the idle class when user is not interacting with the application, the I/O-intensive class when the user is uploading an input file, and the Network-intensive class while the user is interacting with the GUI through a VNC remote display. Table 3 summarizes the class compositions of all the test applications. These classification results match the class expectations gained from empirical experience with these programs. They are used to calculate the unit application cost shown in section 4.4.

In addition, the experimental data also demonstrate the impact of changing execution environment configurations on the application's class composition. For example, in Table 3 when SPECseis96 with medium size input data was executed in VM1 with 256MB memory (SPECseis96_A), it is classified as CPU-intensive application. In the SPECseis96_B experiment, the smaller physical memory (32MB) resulted in increased paging and I/O activity. The increased I/O activity is due to the fact that less physical memory is available to the O/S buffer cache for I/O blocks. The buffer cache size at run time was observed to be as small as 1MB in SPECseis96_B, and as large as 200MB in SPECseis96_A. In addition, the execution time gets increased from 291 minutes and 42 seconds in the first case to 426 minutes 58 sec-

Table 2. List of training and testing applications

Expected Application Behavior	Application	Description
CPU Intensive	SPECseis96[1][2]	A seismic processing application [19]
	SimpleScalar	A computer architecture simulation tool [4]
	CH3D	A curvilinear-grid hydrodynamics 3D model [5]
IO & Paging Intensive	PostMark[1][2]	A file system benchmark program [2]
	PageBench[1]	A synthetic program which initiates and updates an array whose size is bigger than the memory of the virtual machine
	Bonnie	A Unix file system performance benchmark [6]
	Stream	A synthetic benchmark program that measures sustainable memory bandwidth and the corresponding computation rate for simple vector kernels [9]
Network Intensive	Ettcp[1]	A benchmark measuring network throughput over TCP/UDP between two nodes [3]
	Autobench	A wrapper around httperf to work together as an automated web server benchmark [10]
	NetPIPE	A protocol independent network performance measurement tool [34]
	PostMark_NFS	The Postmark benchmark with a NFS mounted working directory
	Sftp	A synthetic program which uses sftp to transfer a 2GB size file
Interactive	VMD	A molecular visualization program using 3-D graphics and built-in scripting [7]
	XSpim	A MIPS assembly language simulator with an X-Windows based GUI [8]
Idle	Idle[1]	No application running except background daemons in the machine

[1] Application is used as a training application.
[2] Application is used as a test application but with different data size.

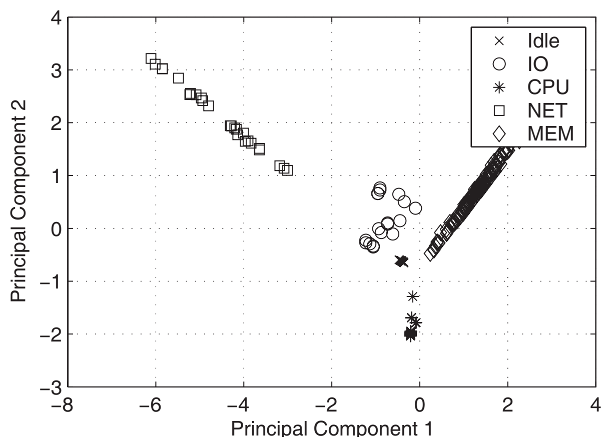
Table 3. Experimental data: Application class compositions

Application Class	Test Application	# of Samples	Idle	I/O	CPU	Network	Paging
CPU Intensive	SPECseis96_A[a]	3,434	–	0.26%	99.71%	–	0.03%
	SPECseis96_C[b]	112	–	–	100%	–	–
	CH3D	45	–	–	100%	–	–
	SimpleScalar	62	–	–	100%	–	–
IO & Paging Intensive	PostMark	52	–	96.15%	–	–	3.85%
	Bonnie	94	–	86.17%	4.26%	–	9.57%
	SPECseis96_B[c]	5,150	0.21%	42.87%	50.39%	–	6.52%
	Stream	96	1.04%	79.17%	–	–	19.79%
Network Intensive	PostMark_NFS	77	–	–	–	100%	–
	NetPIPE	74	4.05%	4.05%	–	91.89%	–
	Autobench	172	–	–	–	100%	–
	Sftp	46	–	2.17%	–	97.83%	–
Idle + Others[d]	VMD	86	37.21%	40.70%	–	22.09%	–
	XSpim	9	22.22%	77.78%	–	–	–

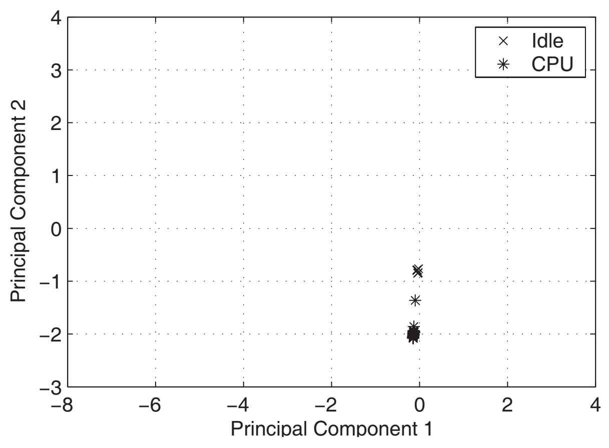
[a] SPECseis96 with medium data size running in a VM with 256MB virtual memory
[b] SPECseis96 with medium data size running in a VM with 32MB virtual memory
[c] SPECseis96 with small data size running in a VM with 256MB virtual memory
[d] User interactive applications show substantial idle states with a mixture of other activities

Table 4. System Throughput: Concurrent vs. Sequential Executions

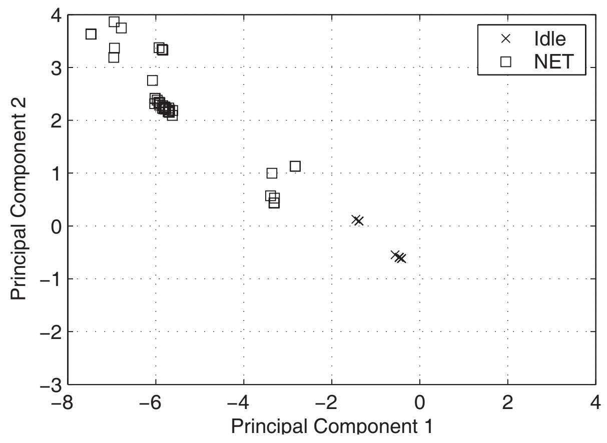
Execution Elapsed Time (sec)	CH3D	PostMark	Time Taken to Finish 2 Jobs
Concurrent	613	310	613
Sequential	488	264	752



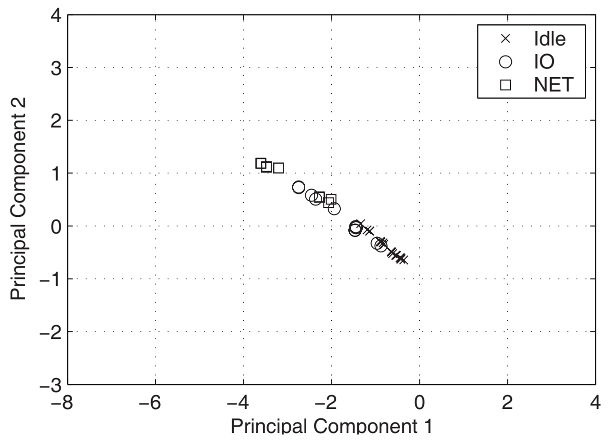
(a) Training data



(b) SimpleScalar



(c) Autobench

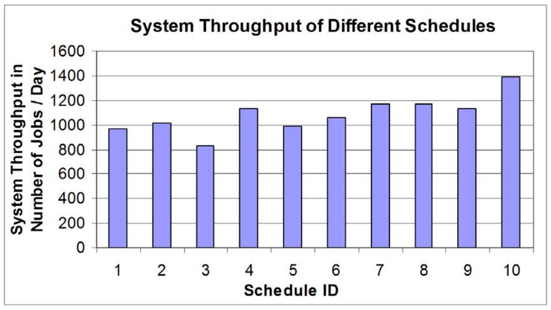


(d) VMD

Figure 3. Sample clustering diagrams of application classifications

CPU-intensive: (b) Network-intensive: (c) Interactive: (d)

Principal Component 1 and 2 are the principal component metrics extracted by PCA



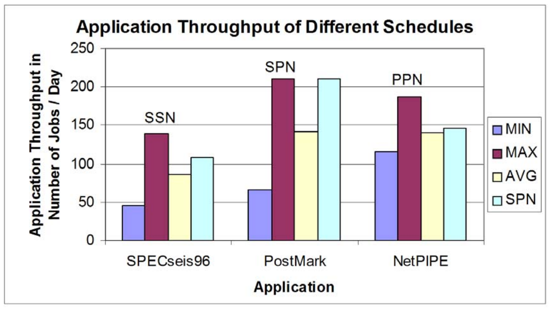


Figure 4. System throughput comparisons for ten different schedules

1:{(SSS),(PPP),(NNN)}, 2:{(SSS),(PPN),(PNN)}, 3:{(SSP),(SPP),(NNN)}, 4:{(SSP),(SPN),(PNN)}, 5:{(SSP),(SNN),(PPN)}, 6:{(SSN),(SPP),(PNN)}, 7:{(SSN),(SPN),(PPN)}, 8:{(SSN),(SNN),(PPP)}, 9:{(SPP),(SPN),(SNN)}, 10:{(SPN),(SPN),(SPN)}

S – SPECseis96 (CPU-intensive), P – PostMark (I/O-intensive), N – NetPIPE (Network-intensive).

Figure 5. Application throughput comparisons of different schedules

MIN, MAX, and AVG are the minimum, maximum, average application throughput of all the ten possible schedules. SPN is the proposed schedule 10 {(SPN), (SPN), (SPN)} in Figure 4.

onds in the second case.

Similarly, in the experiments with PostMark, different execution environment configurations changed the application's resource consumption pattern from one class to another. Table 3 shows that if a local file directory was used to store the files to be read and written during the program execution, the PostMark benchmark showed the resource consumption pattern of the I/O-intensive class. In contrast, with an NFS mounted file directory, it (PostMark_NFS) was turned into a Network-intensive application.

5.2. Scheduling Performance Improvement

Two sets of experiments are used to illustrate the performance improvement that a scheduler can achieve with the knowledge of application class. These experiments were performed on 4 VMware GSX 2.5 virtual machines with 256MB memory each. One of these virtual machines (VM1) was hosted on an Intel(R) Xeon(TM) dual-CPU 1.80GHz machine with 512KB cache and 1GB RAM. The other three (VM2, VM3, and VM4) were hosted on an Intel(R) Xeon(TM) dual-CPU 2.40GHz machine with 512KB cache and 4GB RAM. The host servers were connected by Gigabit Ethernet.

The first set of experiments demonstrates that the application class information can help the scheduler to optimize resource sharing among applications running in parallel to improve system throughput and reduce throughput variances. In the experiments, three applications – SPECseis96 (S) with small data size, PostMark (P) with local file directory and NetPIPE Client (N) – were selected, and three instances of each application were executed. The scheduler's task was to decide how to allocate the nine application instances to run on the 3 virtual machines (VM1, VM2 and VM3) in parallel, each of which hosted 3 jobs. The VM4 was used to host the NetPIPE server. There are ten possible schedules available, as shown in Figure 4.

When multiple applications run on the same host machine at the same time, there are resource contentions among them. Two scenarios were compared: in the first scenario, the scheduler did not use class information, and one of the ten possible schedules was selected at random. The other scenario used application class knowledge, always allocating applications of different classes (CPU, I/O and network) to run on the same machine (Schedule 10, Figure 4). The system throughputs obtained from runs of all possible schedules in the experimental environment are shown in Figure 4.

The average system throughput of the schedule chosen with class knowledge was 1391 jobs per day. It achieved the highest throughput among the ten possible schedules – 22.11% larger than the weighted average of the system throughputs of all the ten possible schedules. In addition, the random selection of the possible schedules resulted in large variances of system throughput. The application class information can be used to facilitate the scheduler to pick the optimal schedule consistently. The application throughput comparison of different schedules on one machine is shown in Figure 5. It compares the throughput of schedule ID 10 (labeled SPN in Figure 5) with the minimum, maximum, and average throughputs of all the ten possible schedules. By allocating jobs from different classes to the machine, the three applications' throughputs were higher than average by different degrees: SPECseis96 Small by 24.90%, Postmark by 48.13%, and NetPIPE by 4.29%. Figure 5 also shows that the maximum application throughputs were achieved by sub-schedule (SSN) for SPECseis96 and (PPN) for NetPIPE instead of the proposed (SPN). However, the low throughputs of the other applications in the sub-schedule make their total throughputs sub-optimal.

The second set of experiments illustrates the improved throughput achieved by scheduling applications of different classes to run concurrently over running them sequentially. In the experiments, a CPU intensive application (CH3D) and an I/O intensive application (PostMark) were scheduled to run in one machine. The execution time for concurrent and sequential executions is shown in Table 4. The experiment results show that the execution efficiency losses caused by the relatively moderate resource contentions between applications of different classes were offset by the gains from the utilization of idle capacity. The resource sharing of applications of different classes improved the overall system throughput.

5.3. Classification Cost

The classification cost is evaluated based on the unit sample processing time in the data extraction, PCA, and classification stage. Two physical machines were used in this experiment: The performance filter in Figure 1 was running on an Intel(R) Pentium(R) 4 CPU 1.70GHz machine with 512MB memory. In addition, the application classifier was running on an Intel(R) Pentium(R) III 750MHz machine with 256MB RAM.

In this experiment, a total of 8000 snapshots were taken with five-second intervals for the virtual machine, which hosted the execution of SPECseis96 (medium). It took the performance filter 72 seconds to extract the performance data of the target application VM. In addition, it took another 50 seconds for the classification center to train the classifier, perform the PCA feature selection and the application classification. Therefore the unit classification cost is 15 ms per sample data, demonstrating that it is possible to consider the classifier for online training.

6. Related Work

Feature selection [24][39] and classification techniques have been applied to many areas successfully, such as intrusion detection [28][22][13][27], text categorization [21], and image and speech analysis. Kapadia's evaluation of learning algorithms for application performance prediction in [25] shows that the nearest-neighbor algorithm has better performance than the locally weighted regression algorithms for the tools tested. Our choice of k-NN classification is based on conclusions from [25]. This paper differs from Kapadia's work in the following ways: First, the application class knowledge is used to facilitate the resource scheduling to improve the overall system throughput in contrast with Kapadia's work, which focuses on application CPU time prediction. Second, the application classifier takes performance metrics as inputs. In contrast, in [25] the CPU time prediction is based on the input parameters of the application. Third, the application classifier employs PCA to reduce the dimensionality of the performance feature space. This is especially helpful when the number of input features of the classifier is not trivial.

Condor uses process checkpoint and migration techniques [29] to allow an allocation to be created and preempted at any time. The transfer of checkpoints may occupy significant network bandwidth. Basney's study in [16] shows that co-scheduling of CPU and network resources can improve the Condor resource pool's goodput, which is defined as the allocation time when a remotely executing application uses the CPU to make forward progress. The application classifier presented in this paper performs learning of application's resource consumption of memory and I/O in addition to CPU and network usage. It provides a way to extract the key performance features and generate an abstract of the application resource consumption pattern in the form of application class. The application class information and resource consumption statistics can be used together with recent multi-lateral resource scheduling techniques, such as Condor's Gang-matching [32], to facilitate the resource scheduling and improve system throughput.

Conservative Scheduling [37] uses the prediction of the average and variance of the CPU load of some future point of time and time interval to facilitate scheduling. The application classifier shares the common technique of resource consumption pattern analysis of a time window, which is defined as the time of one application run. However, the application classifier is capable to take into account usage patterns of multiple kinds of resources, such as CPU, I/O, network and memory.

The skeleton-based performance prediction work introduced in [35] uses a synthetic skeleton program to reproduce the CPU utilization and communication behaviors of message passing parallel programs to predict application performance. In contrast, the application classifier provides application behavior learning in more dimensions.

Prophesy [36] employs a performance-modeling component, which uses coupling parameters to quantify the interactions between kernels that compose an application. However, to be able to collect data at the level of basic blocks, procedures, and loops, it requires insertion of instrumentation code into the application source code. In contrast, the classification approach uses the system performance data collected from the application host to infer the application resource consumption pattern. It does not require the modification of the application source code.

Statistical clustering techniques have been applied to learn application behavior at various levels. Nickolayev et al applied clustering techniques to efficiently reduce the processor event trace data volume in cluster environment [31]. Ahn and Vetter conducted application performance analysis by using clustering techniques to identify the representative performance counter metrics [12]. Both Cohen and Chase's [17] and our work perform statistical clustering using system-level metrics. However, their work focuses on system performance anomaly detection. Our work focuses on application classification for resource scheduling.

Our work can be used to learn the resource consumption patterns of parallel application's child process and multi-stage application's sub-stage. However, in this study we focus on sequential and single-stage applications.

7. Summary

The application classification prototype presented in this paper shows how to apply the Principal Component Analysis and K-Nearest Neighbor techniques to reduce the dimensions of application resource consumption feature space and assist the resource scheduling. In addition to the CPU load, it also takes the I/O, network, and memory activities into account for the resource scheduling in an effective way. It does not require modifications of the application source code. Experiments with various benchmark applications suggest that with the application class knowledge, a scheduler can improve the system throughput 22.11% on average by allocating the applications of different classes to share the system resources.

In this work, the input performance metrics are selected manually based on expert knowledge. We plan to automate this feature selection process to support online classification. In addition, further research is needed to illustrate how to make full use of the classification result and the stochastic information of application behavior for efficient scheduling. We believe that the application classification approach proposed in this paper is a good complement to related application run-time prediction approaches applied to resource scheduling.

Acknowledgement

This work was supported in part by a grant from Intel Corporation ISTG R&D Council and the National Science Foundation under grants: EIA-0224442, EEC-0228390, ACI-0219925 and NSF Middleware Initiative (NMI) grants ANI-0301108 and SCI-0438246. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Intel and NSF.

References

[1] http://www.vmware.com/pdf/esx25_admin.pdf.
[2] http://www.netapp.com/tech_library/3022.html.
[3] http://sourceforge.net/projects/ettcp/.
[4] http://www.cs.wisc.edu/ mscalar/simplescalar.html.
[5] http://users.coastal.ufl.edu/ pete/CH3D/ch3d.html.
[6] http://www.textuality.com/bonnie/.
[7] http://www.ks.uiuc.edu/Research/vmd/.
[8] http://www.cs.wisc.edu/ larus/spim.html.
[9] http://www.cs.virginia.edu/stream/ref.html.
[10] http://www.xenoclast.org/autobench/.
[11] S. Adabala, *et al.* From virtualized resources to virtual computing grids: the in-vigo system. *Future Generation Comp. Syst.*, 21(6):896–909, 2005.
[12] D. H. Ahn and J. S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *SC'02*, p.1–16, Baltimore, MD, Nov. 16–22, 2002.
[13] M. Almgren and E. Jonsson. Using active learning in intrusion detection. In *IEEE Computer Security Foundations Workshop*, p.88–98, June 28–30, 2004.
[14] C. G. Atkeson, A. W. Moore, and S. Schaal. Locally weighted learning. *Artif. Intell. Rev.*, 11(1-5):11–73, 1997.
[15] A. Barak, O. Laden, and Y. Yarom. The now mosix and its preemptive process migration scheme. *Bull. IEEE Tech. Commit. OS Appl. Env.*, 7(2):5–11, 1995.
[16] J. Basney and M. Livny. Improving goodput by coscheduling cpu and network capacity. *Int. J. High Perform. Comput. Appl.*, 13(3):220–230, Aug., 1999.
[17] I. Cohen, *et al.* Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *SOSDI'04*, p.231–244, 2004.
[18] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley-Interscience, New York, NY, 2001. 2nd edition.
[19] R. Eigenmann and S. Hassanzadeh. Benchmarking with real industrial applications: the spec high-performance group. *IEEE Computational Science and Engineering*, 3(1):18–23, 1996.
[20] R. Figueiredo, P. Dinda, and J. Fortes. A case for grid computing on virtual machines. In *ICDCS'03*, p.550–559, May 19–22, 2003.
[21] G. Forman. An extensive empirical study of feature selection metrics for text classification. *J. Mach. Learn. Res.*, 3:1289–1305, 2003.
[22] A. K. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, p.51–62, Santa Clara, CA, Apr. 9–12, 1999.
[23] R. P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, June 1974.
[24] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, Mar. 2003.
[25] N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive application-performance modeling in a computational grid environment. In *HPDC'99*, page 6, Redondo Beach, CA, Aug. 3–6, 1999.
[26] I. Krsul, A. Ganguly, J. Zhang, J. Fortes, and R. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. *SC'04*, p.7–7, Washington, DC, Nov. 6–12, 2004.
[27] S. C. Lee and D. V. Heinbuch. Training a neural-network based intrusion detector to recognize novel attacks. *IEEE Transs. Systems, Man, and Cybernetics, Part A*, 31(4):294–299, 2001.
[28] Y. Liao and V. R. Vemuri. Using text categorization techniques for intrusion detection. In *USENIX Security Symp.*, p.51–59, San Francisco, CA, Aug. 5–9, 2002.
[29] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, Univ. of Wisconsin - Madison Computer Sciences Dept., April 1997.
[30] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(5-6):817–840, 2004.
[31] O. Y. Nickolayev, P. C. Roth, and D. A. Reed. Real-time statistical clustering for event trace reduction. *The Int. J. Supercomp. Appl. High Perform. Comp.*, 11(2):144–159, 1997.
[32] R. Raman, M. Livny, and M. Solomon. Policy driven heterogeneous resource co-allocation with gangmatching. In *HPDC'03*, page 80, Seattle, WA, June 22–24, 2003.
[33] J. M. Schopf and F. Berman. Stochastic scheduling. In *SC'99*, page 48, Portland, OR, Nov. 14–19, 1999.
[34] Q. Snell, A. Mikler, and J. Gustafson. Netpipe: A network protocol independent performace evaluator, June 1996.
[35] S. Sodhi and J. Subhlok. Skeleton based performance prediction on shared networks. In *CCGrid'04*, p.723–730, 2004.
[36] V. Taylor, X. Wu, and R. Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Perform. Eval. Rev.*, 30(4):13–18, 2003.
[37] L. Yang, J. M. Schopf, and I. Foster. Conservative scheduling: Using predicted variance to improve scheduling decisions in dynamic environments. In *SC'03*, page 31, Nov. 15-21, 2003.
[38] Y. Yang and H. Casanova. Rumr: Robust scheduling for divisible workloads. In *HPDC'03*, p.114–125, Seattle, WA, June 22-24, 2003.
[39] L. Yu and H. Liu. Efficient feature selection via analysis of relevance and redundancy. *Journal of Machine Learning Research*, 5:1205–1224, Oct. 2004.