Symphony: A Scheduler for Client-Server Applications on Coprocessor-based Heterogeneous Clusters

M. Mustafa Rafique[†‡], Srihari Cadambi[‡], Kunal Rao[‡], Ali R. Butt[†], Srimat Chakradhar[‡]
[†]*Dept. of Computer Science, Virginia Tech*; [‡]*NEC Laboratories America*
Email: {mustafa,butta}@cs.vt.edu; {cadambi,kunal,chak}@nec-labs.com

***Abstract*—Coprocessors such as GPUs are increasingly being deployed in clusters to process scientific and compute-intensive jobs. In this work, we study if GPU-based heterogeneous clusters can benefit *client-server applications*. Specifically, we consider the practical situation where multiple client-server applications share a heterogeneous cluster (*multi-tenancy*), and experience unpredictable variations in incoming client request rates, including steep *load spikes*. Even for "compute-intensive" client-server applications, it is unclear if a GPU-based cluster can seamlessly deliver acceptable response times in the presence of multi-tenancy and load spikes. We argue that a cluster-level scheduler that is aware of application load, request deadlines and the heterogeneity is necessary in this situation. We propose a novel scheduler called Symphony that enables efficient, dynamic sharing of a GPU-based heterogeneous cluster across multiple concurrently-executing client-server applications, each with arbitrary load spikes. Symphony performs three key tasks: it (i) monitors the load on each application, (ii) collects past performance data and dynamically builds simple performance models of available processing resources and (iii) computes a priority for pending requests based on the above parameters and the requests' *slack*. Based on this, it reorders client requests across different applications to achieve acceptable response times. We also define how client-server applications should interact with a scheduler such as Symphony, and develop an API to this end. We deploy Symphony as user-space middleware on a high-end heterogeneous cluster with dual quad-core Xeon CPUs and dual NVIDIA Fermi GPUs. An evaluation using representative applications shows that in the presence of load spikes (i) Symphony incurs $2-20\times$ fewer requests that do not meet response time constraints compared with other schedulers, and (ii) in order to achieve the same performance as Symphony, other schedulers need $2\times$ more cluster nodes.**

I. Introduction

GPUs are increasingly being used to accelerate non-graphical compute kernels, providing a $10-100\times$ performance boost for workloads such as linear system solvers, physical simulations, partial differential equations and flow visualizations [1]–[5]. With improved programming support [6], GPUs are now being introduced in mainstream clusters [7]–[9]. At the same time, client-server applications which have traditionally been IO- or network-intensive, are becoming more computationally intensive. Examples of such computationally-intensive client-server workloads are semantic search [10], video transcoding [11], financial option pricing [12] and visual search [13]. As in any client-server application, an important metric is *response time*, or the latency per request. For applications with enough parallelism within a single client request, latency per request can be improved by using GPUs.

However, latency per request by itself is not enough. Multiple applications must be able to concurrently run and share a GPU-based heterogeneous cluster[1], i.e., the cluster must support *multi-tenancy*. Further, client-server applications in practice experience varying rates of incoming client requests, sometimes even unpredictable *load spikes*. Thus, any practical heterogeneous cluster infrastructure must handle multi-tenancy and varying load, including load spikes, while delivering an acceptable response time for as many client requests as possible.

In this paper, we focus on the problem of a multi-tenant GPU-based heterogeneous cluster delivering acceptable response times (i.e., a response time that is less than or equal to the pre-specified response time) in the presence of load spikes. Response time is an important part of the system's Quality-of-Service (QoS), and is also the main concern of the client. We argue that for a heterogeneous cluster to handle client-server applications with load spikes, a scheduler that enables dynamic sharing of heterogeneous resources is necessary. As an example, client requests of applications incurring load spikes should be processed by faster resources like the GPU, while requests of other applications could be deferred, or processed by slower resources. Without such a scheduler, decisions made for one application may adversely affect another. For instance, sending one application's client request to the non multi-tasking GPU could block a more critical application.

We present a novel cluster-level scheduler, *Symphony*, that enables efficient sharing of heterogeneous cluster resources while delivering acceptable client request response times despite load spikes. Symphony manages client requests of different applications by assigning each request a priority based on the load and estimated processing time on different processing resources like the CPU and GPU. It then directs the highest priority application to issue requests to suitable processing resources within the cluster nodes. If necessary, the scheduler also directs applications to *consolidate* their requests (pack and issue multiple requests together to the same resource), and load-balances by directing client requests to specific cluster nodes.

Specifically, this paper makes the following contributions:

- We propose Symphony, a cluster-level scheduler, that enables multiple client-server applications to run on a GPU-based heterogeneous cluster. Such clusters accelerate

[1]The term GPU-based heterogeneous cluster is used to denote a cluster with node-level heterogeneity (the cluster nodes have both CPUs and GPUs), and cluster-level heterogeneity (the cluster nodes do not have to be identical to each other).

large scientific jobs, but it is not clear if a GPU's massive parallelism can benefit multi-tenant client-server applications with varying load conditions in a realistic setting. Symphony handles multi-tenancy and delivers acceptable response times while being largely immune to load spikes.
- We underline a key property of Symphony in that it uses user specification and information about the application to make scheduling decisions. It has three characteristics: it (i) monitors the load on each application, (ii) collects past performance data and dynamically builds performance models of available processing resources and (iii) computes a novel priority metric for pending requests based on the load, estimated performance and the request's *slack* (the time left before it must be processed).
- We define how client-server applications should interact with a scheduler such as Symphony.
- We implement Symphony as user-space middleware on a high-end heterogeneous cluster with CPUs and GPUs, and compare it to other schemes such as first-come-first-served (FCFS) and earliest-deadline-first (EDF).

We experimented with Symphony on a 7-node cluster with each node consisting of two quad-core Xeons, and two NVIDIA Fermi GPUs. We run four client-server applications simultaneously under different request load profiles injected with random load spikes, and show that Symphony incurs fewer QoS misses than other scheduling schemes. We also show that other scheduling schemes require more cluster nodes to deliver the same QoS, thereby demonstrating that they do not share heterogeneous cluster resources efficiently under multi-tenancy and varying load conditions.

The rest of the paper is organized as follows. Section II discusses related research and puts our work in perspective. Section III describes characteristics of the applications we consider, our cluster architecture and elaborates how applications interacts with Symphony. Section IV discusses Symphony in detail, including its system architecture, different components and the priority-based scheduling heuristics. Section V presents a comparison of Symphony with other scheduling schemes, and shows that the absence of a scheduler like Symphony necessitates the provisioning of more cluster resources, thereby increasing cost. We conclude in Section VI.

II. Related work

To the best of our knowledge, existing schedulers in the literature do not address the specific problem of managing client requests of multi-tenant client-server applications on heterogeneous clusters with the goal of delivering acceptable response times in the presence of load spikes. In this section, we discuss closely related work on fair scheduling, heterogeneous resource sharing and the use of accelerators and coprocessors in distributed settings.

Fair scheduling policies for homogeneous clusters involve allocating each job a fair share of the resources. For instance, if a job takes time t to execute all by itself, then in the presence of n jobs, it should take time nt. Many proposals offer modifications to fair scheduling. Deadline Fair Scheduling [14] provides processes with proportionate-fair CPU time in multiprocessor servers. Delay Scheduling [15] provides a cluster-level fair scheduling scheme that exploits data locality for MapReduce and Dryad, but their context does not cover processor heterogeneity or load spikes in user requests. A time-sharing based fair scheduling mechanism is presented in [16], which is developed for DryadLINQ cluster. Quincy [17] provides a fair scheduling scheme that preserves and leverages data locality for homogeneous clusters under MapReduce, Hadoop, and Dryad where static application data is stored on the computing nodes. All of these efforts target homogeneous clusters in specific settings, and do not consider varying application load and response times to make scheduling decisions. In a sense, our work could be seen as a modification of fair scheduling taking into account load spikes and resource heterogeneity.

Heterogeneous resource scheduling has been studied for distributed and grid systems, mostly addressing issues that arise from performance asymmetry instead of architectural heterogeneity. Most of these efforts addressing heterogeneity at the node-level [18]–[21] and distributed system-level [22]–[25] represent scientific kernels as graphs with nodes representing interdependent tasks of the entire job. Mesos [26] is a substrate for sharing cluster resources across multiple frameworks, such as Hadoop and MPI, and uses two-level scheduling where it first offers resources to a framework, and the framework selects some of the offered resources and uses its own scheduling policy. This approach enables the resource sharing at the granularity of frameworks, but does not incorporate application performance requirements such as deadlines and response times in the presence of load spikes. AJAS [27] provides an adaptive job allocation strategy for heterogeneous clusters, but does not consider varying application load and response time to make decisions.

The use of computational accelerators such as IBM Cell/BE and GPUs in distributed settings have also been explored [28]–[31]. Traditional approaches for using these accelerators exploit their use for scientific applications and kernels, but not client-server applications. Similarly, hosting multiple applications in a cloud using cooperative scheduling has also been explored [32]–[34]. In contrast, we present an approach for utilizing heterogeneous coprocessor resources such as GPUs in clusters for client-server computing.

III. System Overview

In this section, we describe our application characteristics, the cluster architecture, and define how applications interact with Symphony. Figure 1 shows a high-level overview of the system. It consists of a heterogeneous cluster hosting multiple client-server applications.

A. The Applications

We focus on applications that adhere to the client-server model and process remote client requests. Each application specifies an acceptable response time for its requests. We assume that all requests are of the same type, and only differ in

API	Arguments	Description
void newAppRegistration(...)		**Application registers with scheduler.**
	float `response_time`	Expected latency (ms) for each client request.
	float `average_load`	Average number of requests expected every second.
	int * `nodelist`	Possible cluster nodes on which a client request could be processed.
	int `nodelist_size`	Size of above nodelist.
	int `num_nodes`	Number of nodes necessary to process a client request.
	int `consolidate`	Number of requests that can be consolidated by application.
void newRequestNotification(...)		**Application notifies scheduler of the arrival of a new request.**
	int `size`	Size of data sent by request.
bool canIssueRequests(...)		**Application asks scheduler if requests can be issued.**
	int * `num_reqs`	Number of consecutive requests that can be consolidated and issued.
	int * `id`	Unique scheduler ID for this set of requests.
	int * `nodes`	Which cluster nodes to use.
	int * `resources`	Which resources to use within each cluster node.
void requestComplete(...)		**Application informs scheduler that issued requests have completed processing.**
	int `id`	Scheduler ID pertaining to the completed requests.

TABLE I: List of APIs exposed by Symphony.

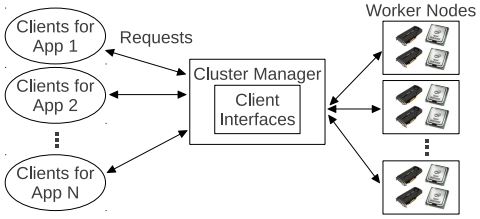


Fig. 1: High-level system architecture.

size, e.g., semantic search processes text queries, but the queries can range in size from a single word to a large sentence. We make no assumptions about inter-dependency of client requests; after interfacing with Symphony, applications will still process requests in the order in which they were received.

All applications have a client interface and a server portion. The server portion, along with static application data, is mapped to specific cluster nodes, and is expected to be online and communicating with Symphony. When a client request arrives, it may be processed by one or more nodes where the application data is pre-mapped. Some applications may require all nodes to process each request, while others may just need any one node. Applications specify this information to Symphony, as we explain soon.

Since we specifically target GPU-based heterogeneous clusters, we focus on applications whose request processing involves executing parallelizable compute kernels. We assume that applications already have optimized CPU and GPU implementations available in the form of runtime libraries with well-defined interfaces for such kernels. This enables Symphony to intercept calls to these kernels at runtime, and dispatch it to either CPU or GPU resources, as described later.

Finally, some applications may have the ability to *consolidate* requests, i.e., pack and process multiple independent client requests together to achieve better throughput via increased parallelism. Symphony leverages this to drain pending requests faster.

B. Cluster Architecture

The heterogeneous cluster has a cluster manager, which is a dedicated general-purpose multicore server node. It runs the cluster-level scheduler and application client interfaces. It manages a number of back-end servers, or *worker nodes*. The worker nodes contain heterogeneous computational resources comprising conventional multicores and CUDA-enabled GPUs. All cluster nodes are interconnected using any standard interconnection network.

C. Scheduler-Application Interface

We now define how client-server applications can communicate with a scheduler such as Symphony by making simple modifications. An application initially registers itself with Symphony and sends a notification each time it receives a client request. It then waits to receive the "go-ahead" from Symphony to process pending requests. Once requests have completed processing, the application informs Symphony. Applications can use two threads to do this: one to notify the scheduler of new requests, and the other to ask if requests can be issued, and inform the scheduler of completion. This is not a major change since most client-server applications already do this to simultaneously fill buffers with incoming requests, and drain requests from the other end. The application modifications only require linking with the scheduler library and adding a few lines of code, with no reorganization or rewriting.

Table I provides the lists of the APIs exposed by Symphony. First, the application registers with the scheduler (`newAppRegistration()`) and specifies its expected response time for each client request (`latency`). The application also specifies the average number of client requests it expects to receive each second (`average_load`), the set of cluster nodes onto which its static data has been mapped (`nodelist`), and how many nodes each request will require for processing (`num_nodes`). For example, an application's data may have been mapped to 4 cluster nodes, but any of those 4 nodes can process a request. In this case, `nodelist` will contain names (or other descriptor) of the 4 nodes, and `num_nodes` will be 1. Finally, when an application registers with Symphony, it must also specify how many requests it can consolidate together (`consolidate`). For example, in the case of the Semantic Search, several user queries can be packed and executed simultaneously on a single worker node.

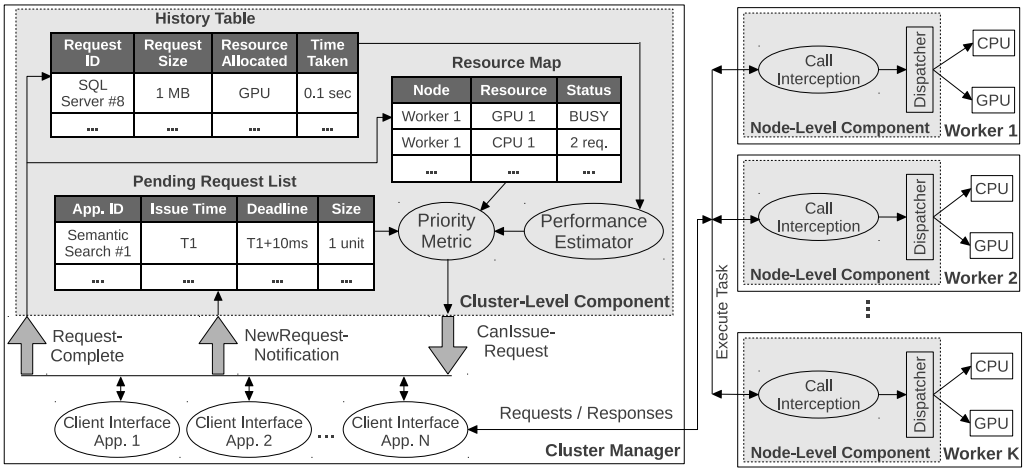


Fig. 2: Architecture of Symphony.

Applications notify Symphony of each new client request (`newRequestNotification()`) and specify the size of the request. In parallel, the application polls the scheduler to receive the go-ahead for processing pending requests (`issueRequests()`). Symphony tells the application how many requests to consolidate (`num_reqs`) and provides a unique identifier (`id`) for this set of requests. The application then processes the requests, and informs Symphony after they complete using `requestComplete()`.

IV. Architecture of Symphony

Symphony is a request scheduler for multi-tenant client-server applications on heterogeneous clusters. Its goal is to deliver acceptable response times in the presence of load spikes. It combines application-specified parameters with its own inferences to make scheduling decisions. Symphony consists of cluster-level and node-level components. Figure 2 shows the manager node running the cluster-level component and client interfaces for the applications. The figure also shows the worker nodes running the node-level components. Both components are implemented as user-space middleware.

A. Cluster-level Component of Symphony

This is the primary orchestrator in our system. Given client requests for different applications, it decides:

- which application should issue requests;
- how many requests should the application consolidate;
- to which cluster nodes should the requests be sent; and
- which resource (e.g., CPU or GPU) in the node should process the requests.

The architecture of the cluster component of Symphony consists of six portions as shown in Figure 2: (i) Pending Request List, (ii) Resource Map, (iii) History Table, (iv) Performance Estimator (v) Priority Metric Calculator and (vi) Load Balancer. We describe each of these below.

1) Pending Request List: Each application notifies Symphony upon the arrival of a client request. As shown in Figure 2, the scheduler stores certain information pertaining to pending requests, so that it can prioritize them and direct the applications to consolidate and dispatch the requests for processing. It does not store actual request data, but maintains for each request, the application that received the request, the time at which the request was received, the deadline by which the request should complete and the size of the request data.

2) Resource Map: Symphony monitors current cluster resource usage using a map of the CPU and GPU resources on each cluster node. For the CPU resource, it maintains a count of the number of requests being processed, while for the (non-multitasking) GPU, a BUSY/IDLE tag is maintained. This information is used to determine resource availability as well as to balance the load across the cluster. The resource map is updated each time the scheduler asks an application to issue requests (`issueRequests()`), and when an application informs the scheduler that it has completed requests (`requestComplete()`).

3) History Table and Performance Model: The history table stores details of recently completed requests of each application. Each entry of the history table contains a recently completed client request, resources that processed it, and the actual time taken to process the request. The history table is updated each time client requests complete (`requestComplete()`).

The information in the history table is used to build a simple linear performance model, the goal of which is to quickly estimate performance on the CPU or GPU so that the right requests can be issued with minimal response time failures. After collecting request sizes and corresponding execution times, we fit the data into a linear model to obtain CPU or GPU performance estimations based on request sizes. The model is dependent on the exact type of CPU or GPU; in our case we only have a single type of CPU and GPU, but if different generations of CPUs and GPUs exist, a model can be developed for each specific kind.

In addition to the dynamic performance model builder, existing analytical models can also be used to estimate the execution time of an application on available resources. Some analytical models such as [35] may require application and resource specific information at compile time to accurately

generate performance estimates. Although the performance model builder used by Symphony is simple, it requires no compile-time information to generate performance estimates.

4) Priority Metric: Symphony uses a priority metric to calculate the urgency of pending requests from the point of view of response time and overall load. Given N applications, where application A has n_A requests in the pending request list, the goal of the priority metric is to indicate (i) which of the N applications is most critical and therefore must issue its requests and (ii) which resource (e.g., CPU or GPU) should process the requests. Note that Symphony does *not* reorder requests within an application, but only across applications.

We assume that our heterogeneous cluster has r types of resources in each node, labeled R_1 through R_r. For example, if a node has 1 CPU and 1 GPU, r is 2. Furthermore, the application itself is responsible for actual request consolidation, but the scheduler indicates how many requests can be consolidated. To do this, the scheduler is aware of the maximum number of requests MAX_A that application A can consolidate (through `newAppRegistration()`). So if A is the most critical application, the scheduler directs it to consolidate the minimum of MAX_A or n_A requests.

If $DL_{k,A}$ is the deadline for request k of application A, CT the current time, and $EPT_{k,A,R}$ the estimated processing time of request k of application A on resource R, we define $slack$ for request k of application A on resource R as:

$$slack_{k,A,R} = DL_{k,A} - (CT + EPT_{k,A,R}) \quad (1)$$

Initially, in the absence of historical information, $EPT_{k,A,R}$ is assumed to be zero. Resource R is either the CPU or GPU; if the system has different types of CPUs and GPUs, then each type is a resource since it would result in a different estimated processing time (EPT). A zero slack indicates the request must be issued immediately, and a negative slack indicates the request is overdue. Given the slack, we define urgency of request k of application A on resource R:

$$U_{k,A,R} = -slack^p \quad (2)$$

The above is a polynomial urgency functions, and we find that an exponent such as $p = 3$ provides good performance for our applications. We compare linear, polynomial and also exponential urgency functions in the results section. The above is the urgency of issuing a single request, and it increases polynomially as the slack nears zero. To account for load spikes, Symphony calculates the load L_A for each application A, using the average number of pending requests in the queue and the average number of requests expected every second ($navg_A$) specified at the time of application registration:

$$L_A = n_A / navg_A \quad (3)$$

We define the urgency of issuing the requests of application A on R as the product of the urgency of issuing the first pending request of A and the load of A:

$$U_{A,R} = \begin{cases} U_{1,A,R} * L_A & , \ if \ R \ is \ available \\ -\infty & , \ otherwise \end{cases} \quad (4)$$

We only consider the first pending request for each application because all application requests are processed in the order they are received, while requests across applications may be reordered. All pending requests of an application will therefore be less urgent than the first request. Considerations such as preferred customers with prioritized requests are outside the scope of this paper.

Algorithm 1 Application Selection Algorithm.

Input: $appList$, $reqList$, $resList$, DL, EPT
Output: app, q, R

```
for all A in appList do
    k = getEarliestRequest(A)
    slack_{k,A,R} = calculateSlack(DL_{k,A}, EPT_{k,A,R})
    U_{k,A,R} = calculateReqUrgency(slack_{k,A,R})
    n_A = getAppReqCount(A)
    navg_A = getAvgAppReqCount(A)
    L_A = n_A / navg_A
    for all R in resList do
        if resAvailable(R) then
            U_{A,R} = U_{k,A,R} × L_A
        else
            U_{A,R} = -∞
        end if
    end for
    U_A = getMinimum(U_{A,R}, resList)
end for
/* Select application app to issue q requests to resource R */
app = getAppHighestUrgency()
q = MIN(MAX_app, n_app)
R = getResWithLowestUrgency(app)
```

The overall urgency U_A for issuing A's requests is the minimum urgency across all available resources R_i. If there are r different types of resources in each cluster node:

$$U_A = min_{i=1}^{r}(|U_{A,R_i}|) \quad (5)$$

Given the urgency for all applications, the scheduler will request application A to consolidate and issue q requests to resource R such that:

- Application A has the highest urgency among all applications;
- q is the minimum of MAX_A and n_A;
- Among all available resources, R is the resource when scheduled on which application A has minimum urgency.

We note the following about the priority metric:

- If request falls behind in meeting its deadline, its urgency sharply increases (Equation 2).
- If an application experiences a load spike, its urgency sharply increases (Equation 4).
- Request processing is predicated on resource availability (Equation 4).
- For an application, the resource with the lowest urgency is the one with the best chance of achieving the deadline, and is therefore chosen (Equation 5).

Algorithm 1 shows an approach to implement the priority metric described above. It returns the application (app) with the highest urgency, the number of requests (r) that should be consolidated together in the next dispatch, and the resources (R) on which the application should be executed. It is highly

Application	Description	Response Time	Data Layout	Data Size
Semantic Search	Supervised Semantic Indexing [10] (SSI) matching to search large document databases. It searches the indexed documents for the user queries and ranks the results based on their semantic similarity to the given queries.	5 msec/query	Document repositories distributed across worker nodes so that each document is available on at least two worker nodes.	2 million documents
Video Transcoding	An implementation [11] of x264 that converts the video streams into the H.264/MPEG-4 AVC format. Each cluster node executes an instance of Video Transcoding application to encode the given video stream.	500 msec/MB	Input video data accessible at each worker node through Network File System (NFS) [36].	$4-45$ MB
SQL Server	An implementation [37] of a subset of SQLite commands processor. Each worker executes an instance of SQL Server hosting the same database.	150 msec/query	Database replicated on all the worker nodes. Any worker node can process the given query.	512 MB
Option Pricing	An implementation [38] of Black-Scholes financial model to compute the evolution of future option prices. Each worker hosts an instance of Option Pricing and provides the option prices for user queries.	800 msec/query	Input data accessible at each worker node through NFS.	400 MB

TABLE II: Enterprise applications with execution resources and performance criteria.

scalable since we do not compute the slack and urgency for every request in the pending request list, but only for the first MAX_A requests of every application. This keeps Symphony's overhead small, as we show in Section V.

5) Load Balancer: As stated earlier, we assume static application data are pre-mapped to the cluster nodes. Client requests can be processed by a subset of these nodes, and the application tells the scheduler how many nodes are required to process a request (through `newAppRegistration()`). When the scheduler directs an application to issue requests, it provides a list of cluster nodes where the request can be processed by simply choosing the least loaded cluster node. The application is expected to issue its requests to these nodes and thus maintain overall load balancing.

B. Node-level Component

Besides the cluster-level scheduler that runs on the cluster manager node, separate node-level dispatchers run on each worker node. The node level dispatcher is responsible for receiving an issued request and directing it to the correct resource (CPU or GPU) as specified by the cluster-level scheduler. In order to do this, we assume that parallelizable kernels in the applications have both CPU and GPU implementations available as dynamically loadable libraries. The node-level dispatcher intercepts the call to the kernel, and at runtime directs it to either the CPU or GPU. For example, if processing a Semantic Search request requires a call to matrix multiplication, we assume that CPU and GPU library implementations are available for a specified function name, say `sgemm`. The node-level component intercepts `sgemm`, and looks for a directive from the cluster-level component. When the request was issued, the cluster component directly intimates the node-level component that `sgemm` in this instance of Semantic Search should be directed to, say the GPU. Further details of the node-level dispatcher implementation are presented in [39], [40], and are outside the scope of this paper.

V. Evaluation

In this section we describe our evaluation methodology and present results. We run four, full-fledged client-server applications concurrently on a high-end heterogeneous cluster with Intel Xeon CPUs and NVIDIA Fermi GPUs over a period of 24 hours. We subject the applications to load spikes, where the duration and size of each spike are taken from published observations. Using our implementation of the scheduler as user-space middleware, we present the following results:

- *Priority Metric*: A comparison of Symphony's performance under different priority metrics and establish a "good" working metric for the following experiments.
- *Scheduler Performance Comparison*: A comparison of Symphony and baseline FCFS and EDF schedulers considering the number of dropped client requests, i.e., requests that do not meet response time constraints.
- *Efficient Cluster Sharing*: Empirical data showing that, compared to other schedulers, Symphony needs a smaller cluster to achieve the same performance.
- *Sensitivity to Load Spike Profile*: Unlike the baseline schedulers, Symphony performs well across a range of load spikes, i.e., spikes with varying height and width.
- *Scalability*: Data showing the running time of Symphony itself increases only marginally with increasing number of cluster nodes and applications.

For the first four set of results, the common metric of comparison is the number of client requests that do not meet response time constraints (QoS). We also call this "dropped requests".

A. Methodology

Our methodology consists of different sized heterogeneous clusters, with four real, end-to-end applications concurrently running on each cluster. We compare Symphony with two scheduling mechanisms, *First Come First Served* (FCFS) and *Earliest Deadline First* (EDF). In FCFS, client requests are processed in the same order as they arrive at the cluster manager. In EDF, the client request with the closest deadline is processed first. Both FCFS and EDF incorporate application placement and pre-mapped data while making scheduling decisions. Furthermore, FCFS and EDF consider GPUs as well as CPUs while scheduling application requests on the available nodes. However, GPUs are preferred: requests are processed on the CPUs only if all GPUs are busy.

Spike Height	Polynomial ($-slack^3$)	Linear ($-slack$)
Spike Width: $5\ minutes$		
1.25	0.02	0.19
1.5	0.05	0.32
2.0	0.8	3.31
Spike Width: $15\ minutes$		
1.25	0.02	0.16
1.5	0.04	0.29
2.0	0.81	3.16

TABLE III: Average percentage of requests per minute not meeting the specified QoS under different slack functions.

We now describe the applications, cluster configurations and spike introduction mechanisms.

1) Applications: Emerging cluster computing workloads consist of a mix of short- and long-running jobs. We have selected four representative applications from different domains covering the spectrum of latency- and throughput-intensive workloads. We choose Semantic Search, SQL Server, Video Transcoding and Option Pricing as our representative workload set. Table II provides the brief description of each application along with their performance requirements. Some of these applications, i.e., Semantic Search, and SQL Server, execute short-running tasks, while others execute long-running jobs. The table also describes the application's static data layout. Based on data layout, a client request can be processed on a subset of worker nodes (e.g., Semantic Search), or on any available worker node (e.g., Video Transcoding, SQL Server, Option Pricing) of the heterogeneous cluster.

2) Cluster Configurations: Our cluster consists of seven high-end worker nodes, and a manager node. Each worker node has two Intel Xeon E5620 processors (2.4 GHz each), with QPI and 48 GB main memory. A worker node also has two 1.3 GHz NVIDIA Fermi C2050 GPUs with 3 GB internal memory, connected as coprocessors on PCI Express slots. The worker nodes are interconnected together, and with the manager using gigabit ethernet.

3) Spike Introduction Mechanism: According to published observations, typical spike durations vary from $10-30$ minutes, while the peak of the spikes can be as high as $1.5\times$ of the normal load [41]–[44]. We introduce random load spikes with duration and heights in this range, but extend our evaluation to a broader range of spikes. Our spike introduction mechanism injects spikes at random time points for each application, independent of the other applications running at the same time.

B. Priority Metric

Before we present results with Symphony, we explore the priority metric used in order to empirically establish a good enough heuristic for the rest of the experiments. Specifically, we replace the polynomial urgency function $-slack^3$ from Equation 2, with exponential and linear functions, and compare the final performance of the system in terms of the number of client requests dropped. The polynomial function ($-slack^3$) was replaced with exponential (2^{-slack}) and linear ($-slack$) functions. We find out that the average percentage of dropped requests are similar for the exponential and polynomial urgency functions, while the simple linear urgency function underperforms. Table III compares polynomial and linear functions for different load spike heights and widths, where the height of the spike is the ratio between the spike and normal application load, and the width is the duration of the spike in minutes. The data shows that using the linear urgency function table incurs far more requests dropped than the polynomial (or exponential) urgency functions.

This is expected since the urgency function defines how responsive the system is to spikes. In a sense, the urgency function has to "follow" a spike closely; the faster a spike rises, the quicker the urgency of the those pending requests should become. Our data establishes that in general polynomial or exponential urgency functions follow random load spikes well. We thus use the polynomial urgency function for the rest of the experiments in this section.

C. Scheduler Performance Comparison

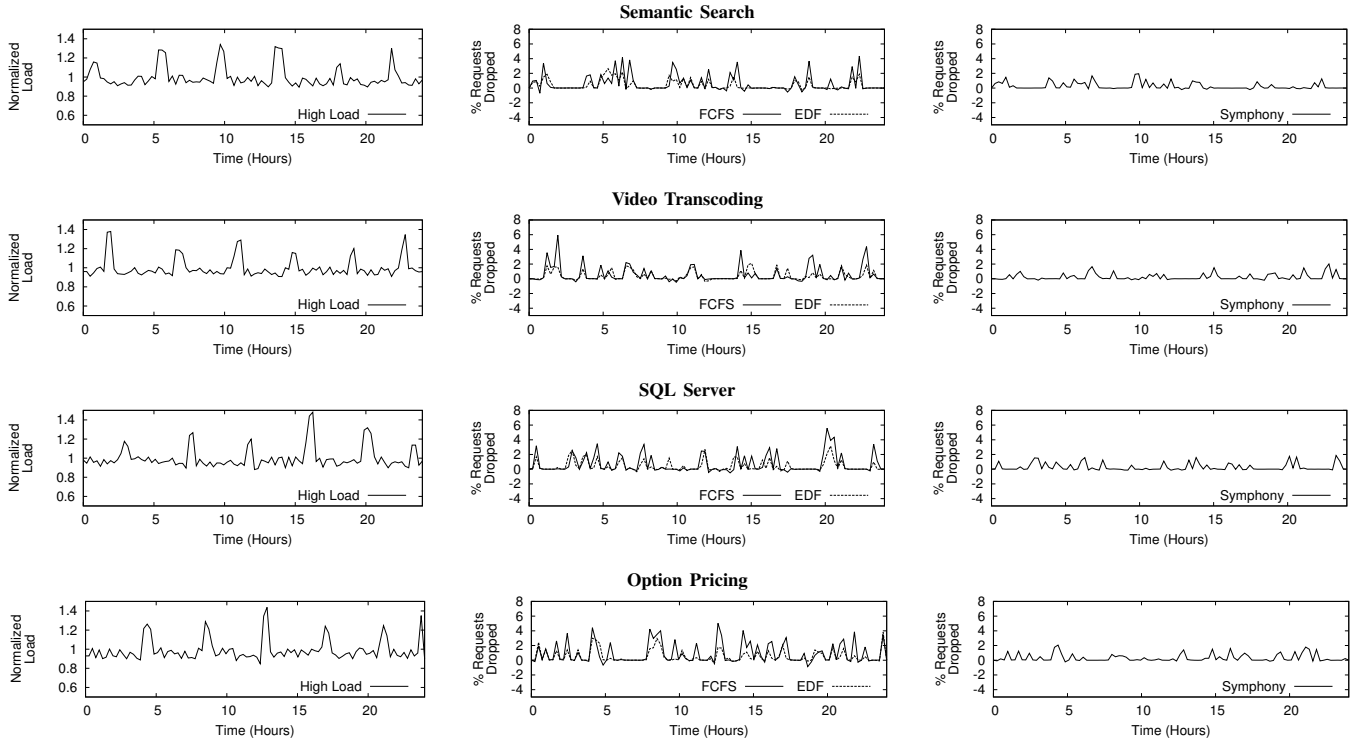
We now compare the performance of Symphony versus baseline FCFS and EDF schedulers using a 5-node heterogeneous cluster. The metric for the comparison is the number of client requests that do not meet response time constraints.

1) Low Application Load: We first run all four applications concurrently under low load conditions. Not surprisingly, we find that at low loads, all scheduling schemes perform well, i.e., the fraction of dropped requests is under 0.01% for FCFS, EDF and Symphony. For instance, for EDF and Symphony the maximum fraction of dropped requests at any instance is 0.012% for and 0.011% for Video Transcoding, respectively. Thus EDF and Symphony exhibit similar performance for our applications under low application load.

2) Realistic Application Load (with Spikes): Next we study the performance under varying load conditions using the realistic spike introduction model of Section V-A3. Figure 3 shows the results. For FCFS, the maximum observed percentage of dropped requests is around 7%, which occurs for Video Transcoding. On average FCFS drops 3.3% of requests across all applications over 24 hours. For EDF and Symphony, the maximum fraction of dropped requests is 3.5% and under 2%, respectively, occurring for Video Transcoding and Option Pricing. Across all four applications over 24 hours, EDF (which is better than FCFS) drops close to 2% of requests on the average, while Symphony drops under 1%. This shows that Symphony is effective in handling load spikes when given a fixed number of heterogeneous cluster nodes. This is inline with expectations since FCFS does not consider request deadlines at all, while EDF, although it prefers requests that are closest to missing deadlines, does not consider estimated request processing time.

D. Efficient Cluster Sharing

We now study how Symphony performs with limited resources. Specifically, we reduce the cluster size to 3 worker nodes, and experimentally determine the number of dropped requests for all three scheduling schemes. Then we gradually increase the cluster size, and find that the baseline scheduling



(a) Application load. (b) Performance with FCFS and EDF. (c) Performance with Symphony.

Fig. 3: Requests missing QoS under spike load conditions for the four concurrently running applications.

Application	No. of Requests Processed	Average Desired Latency/Request	3-Node Cluster			7-Node Cluster		
			FCFS	EDF	Symphony	FCFS	EDF	Symphony
Semantic Search	17.2 M	5 msec.	24.1	20.2	3.0	2.9	1.2	0.05
Video Transcoding	52 K	45 sec.	27.2	21.6	3.3	3.2	1.3	0.05
SQL Server	576 K	150 msec.	21.8	14.7	2.8	2.6	1.1	0.04
Option Pricing	108 K	800 msec.	25.9	20.9	3.2	3.1	1.4	0.05

TABLE IV: Percentage of requests dropped under different cluster configurations for 24 hour period.

schemes need a cluster with more worker nodes to provide the same fraction of requests dropped.

Table IV shows the results using the realistic load profile from Figure 3(a). With thousands to millions of requests and a 3-node cluster, FCFS and EDF end up dropping 14 to 27% of requests, while Symphony drops around 3%. FCFS and EDF attain the same level of performance as Symphony, i.e., 3% dropped requests, when provisioned with 7 cluster nodes. At that cluster size however, and the same load profile, Symphony drops only around 0.05% of requests. Therefore, with its load- and heterogeneity-sensitive scheduling, Symphony shares cluster resources more efficiently under varying load conditions, leading to better utilization of the cluster.

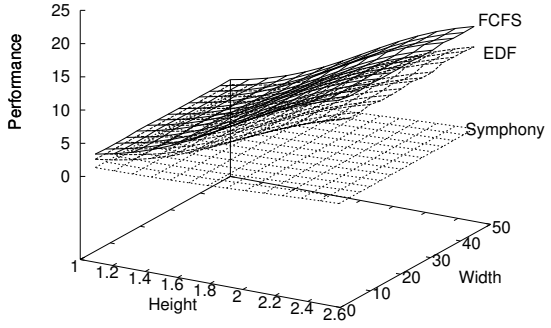


Fig. 4: Performance with increasing spike height and width.

E. Sensitivity to Load Profile

To study how Symphony performs under different types of load spikes, we introduced spikes of varying height and width and measured the average percentage of dropped requests on the 7-node heterogeneous cluster. Figure 4 shows the percentage of dropped requests versus the height and width of the spikes. The height of spike is relative to its height at low load, while the width is its duration in minutes. The performance of FCFS and EDF deteriorates sharply with spike height. The maximum percentage of dropped requests is 20.5%, 18.0% and 2.15% for FCFS, EDF and Symphony respectively. Across all points, the average percentage of dropped requests per minute is 9.2%, 7.2% and 0.7% for FCFS, EDF and Symphony respectively. This shows that Symphony is not very sensitive to the type of

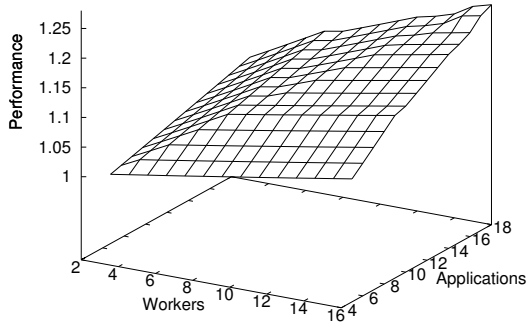


Fig. 5: Scalability with increasing workers and applications.

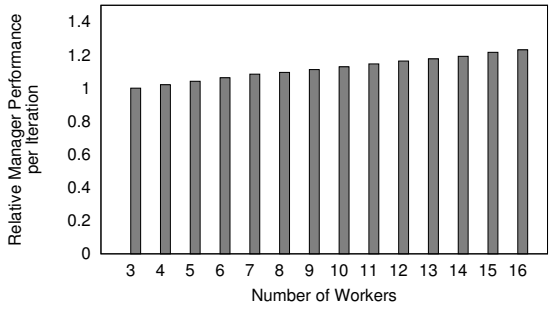


Fig. 6: Symphony overhead with increasing workers (diagonal from Figure 5).

load spike, and can handle a broader range of spikes that are outside previously published observations.

F. Scheduler Scalability

A concern with the single-manager design is the overhead due to introduction of additional middleware layers, as this could lead to performance bottlenecks when operating at scale. To this end, we observed how the performance of Symphony scales with the number of worker nodes and with the number of applications. We increase the number of worker nodes from 3 to 16 and the number of applications from 4 to 18. To run more than four applications, we replicate our existing applications.

Figure 5 shows the overhead of Symphony, i.e., the time taken by the scheduler itself, normalized to the overhead for a 3 node heterogeneous cluster executing four applications. We see that it scales nearly linearly (with a small slope) as the number of worker nodes and applications increase. The four corners of the 3-D plot represent: (i) 3-node cluster with four applications; (ii) 3-node cluster with 18 applications; (iii) 16-node cluster with four applications; and (iv) 16-node cluster with 18 applications. The scheduler takes longest when both the number of cluster nodes as well as the number of concurrent applications are high. However, the running time of Symphony at 16 cluster nodes with 18 concurrent applications is only 22% larger than the base case of 3 cluster nodes with four applications. The marginal increase and Symphony's scalability is apparent from Figure 6 (the diagonal from Figure 5), which shows the overhead versus cluster size where a cluster with m nodes hosts $m + 1$ applications.

In summary, we empirically demonstrated the performance of our scheduling framework vis-a-vis two baseline schedulers on a high-end heterogeneous cluster. We also demonstrate the scheduler's scalability as the cluster scales-out.

VI. Conclusion

Current GPU-based clusters mostly cater to large scientific workloads. Using such clusters to handle client-server workloads, especially those with GPU-friendly compute intensive portions, is a natural transition. However, for such applications, a key requirement is maintaining acceptable response times in the presence of unpredictable load spikes and multi-tenancy. We argue that this requires a cluster scheduler that enables efficient, dynamic sharing of heterogeneous resources across applications. We present a novel priority based scheduler called Symphony that is characterized by three key attributes. First, it continuously monitors the load on each application. Second, it collects past performance data and dynamically builds simple performance models of the available heterogeneous processing resources. Third, it computes a priority for pending requests based not only on user-specified parameters about the application but also information inferred from its performance models.

We show Symphony is largely immune to load spikes and maintains acceptable response times despite fairly large load variations. We deploy it as user-space middleware on a 7-node heterogeneous cluster with dual quad-core Xeon CPUs and NVIDIA Fermi GPUs, and show that in the presence of load spikes (i) Symphony incurs $2 - 20\times$ fewer requests that don't meet response time constraints compared with other schedulers, and (ii) in order to match the performance of Symphony, other schedulers need $2\times$ more cluster nodes.

Our study has opened directions for further investigation. For instance, Symphony can benefit from predicting client request patterns. We are also looking into mechanisms to harden Symphony against the system being gamed, especially because its behavior is determined in part by parameters (such as response time) specified by the application developer. Investigating latency and throughput trade-offs due to client request consolidation is another direction. Finally, we are in the process of extending Symphony to other heterogeneous resources and testing it on a cluster comprising thousands of compute nodes.

References

[1] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. IEEE/ACM SC.*, 2008.

[2] M. J. Zafont, A. Martin, F. Igual, and E. S. Quintana-Orti, "Fast development of dense linear algebra codes on graphics processors," in *Proc. IEEE IPDPS.*, 2009.

[3] J. Schneider, M. Kraus, and R. Westermann, "GPU-based euclidean distance transforms and their application to volume rendering," in *Proc. VISIGRAPP.*, 2009.

[4] J. A. Stuart, C.-K. Chen, K.-L. Ma, and J. D. Owens, "Multi-GPU volume rendering using MapReduce," in *Proc. ACM HPDC.*, 2010.

[5] K. Buerger, F. Ferstl, H. Theisel, and R. Westermann, "Interactive streak surface visualization on the GPU," *IEEE Trans. on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1259–1266, 2009.

[6] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[7] "Cloud services for GPU computing," 2011, http://www.hoopoe-cloud.com/.

[8] "Accelerator cluster," 2011, http://www.iacat.uiuc.edu/resources/cluster/.

[9] Amazon Inc., *Amazon Elastic Compute Cloud (Amazon EC2)*, May 2011, http://aws.amazon.com/ec2/.

[10] B. Bai, J. Weston, D. Grangier, R. Collobert, K. Sadamasa, Y. Qi, O. Chapelle, and K. Weinberger, "Learning to rank with (a lot of) word features," *Information Retrieval*, vol. 13, pp. 291–314, 2010.

[11] VideoLAN, "x264 - A Free h264/AVC Encoder," Nov 2010, http://www.videolan.org/developers/x264.html.

[12] C. Kolb and M. Pharr, "GPU gems 2." http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter45.html.

[13] "bing Visual Search," 2011, http://www.bing.com/browse.

[14] A. Chandra, M. Adler, and P. Shenoy, "Deadline fair scheduling: Bridging the theory and practice of proportionate fair scheduling in multiprocessor systems," in *Proc. IEEE RTAS.*, 2001.

[15] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc. ACM EuroSys*, 2010.

[16] S.-M. Park and M. Humphrey, "Predictable time-sharing for DryadLINQ cluster," in *Proc. ACM ICAC.*, 2010.

[17] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proc. ACM SOSP.*, 2009.

[18] H. Oh and S. Ha, "A static scheduling heuristic for heterogeneous processors," in *Proc. Euro-Par*, 1996.

[19] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro, "Predictive runtime code scheduling for heterogeneous architectures," in *Proc. HiPEAC.*, 2009.

[20] L. Wang, Y.-z. Huang, X. Chen, and C.-y. Zhang, "Task scheduling of parallel processing in CPU-GPU collaborative environment," in *Proc. IEEE ICCSIT.*, 2008.

[21] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task scheduling algorithms for heterogeneous processors," in *Proc. IEEE HCW.*, 1999.

[22] M. Maheswaran and H. J. Siegel, "A dynamic matching and scheduling algorithm for heterogeneous computing systems," in *Proc. IEEE HCW.*, 1998.

[23] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Softw. Eng.*, vol. 3, pp. 85–93, January 1977.

[24] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, pp. 175–187, February 1993.

[25] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, pp. 260–274, March 2002.

[26] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-87, May 2010.

[27] C.-T. Yang and K.-Y. Chou, "An adaptive job allocation strategy for heterogeneous multiple clusters," in *Proc. IEEE CIT.*, 2009.

[28] M. M. Rafique, A. R. Butt, and D. S. Nikolopoulos, "Designing accelerator-based distributed systems for high performance," in *Proc. IEEE/ACM CCGRID.*, 2010.

[29] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a message-driven parallel application to GPU-accelerated clusters," in *Proc. ACM/IEEE SC.*, 2008.

[30] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing," in *Proc. ACM/IEEE SC.*, 2004.

[31] M. M. Rafique, A. R. Butt, and D. S. Nikolopoulos, "A capabilities-aware framework for using computational accelerators in data-intensive computing," *J. Parallel Distrib. Comput.*, vol.71, pp. 185–197, Feb. 2011.

[32] Y. Chen, T. Wo, and J. Li, "An efficient resource management system for on-line virtual cluster provision," in *Proc. IEEE CLOUD.*, 2009.

[33] C. Fehling, F. Leymann, and R. Mietzner, "A framework for optimized distribution of tenants in cloud applications," *Proc. IEEE CLOUD.*, 2010.

[34] A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle, "Multi-tenant SOA middleware for cloud computing," *Proc. IEEE CLOUD.*, 2010.

[35] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proc. ACM ISCA.*, 2009.

[36] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "RFC3530: Network File System (NFS) Version 4 Protocol," 2004, http://www.ietf.org/rfc/rfc3530.txt.

[37] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proc. ACM GPGPU.*, 2010.

[38] V. Podlozhnyuk, "Black-Scholes option pricing," in *NVIDIA CUDA SDK Documentation*, June 2007.

[39] M. Becchi, S. Byna, S. Cadambi, and S. Chakradhar, "Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory," in *Proc. ACM SPAA.*, 2010.

[40] M. Becchi, S. Cadambi, and S. Chakradhar, "Enabling legacy applications on heterogeneous platforms," in *Proc. USENIX HotPar*, 2010.

[41] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload analysis and demand prediction of enterprise data center applications," in *IEEE IISWC.*, 2007.

[42] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *Proc. SoCC.*, 2010.

[43] R. Miller, "Go Daddy Ad Drives Huge Traffic Spike," February 2010, http://www.datacenterknowledge.com/archives/2010/02/08/go-daddy-ad-drives-huge-traffic-spike/.

[44] John Treadway, "The End of Over-Provisioning," June 2010, http://cloudcomputing.sys-con.com/node/1429706.