Automating Enterprise Application Placement in Resource Utilities

J. Rolia[1], A. Andrzejak[2], and M. Arlitt[1]

[1] Hewlett Packard Laboratories, Palo Alto, CA 94304, USA,
jerry_rolia@hp.com,martin_arlitt@hp.com,
[2] Zuse Institute Berlin (ZIB),
Takustraße 7, 14195 Berlin-Dahlem, Germany,
andrzejak@zib.de

Abstract. Enterprise applications implement business resource management systems, customer relationship management systems, and general systems for commerce. These applications rely on infrastructure that represents the vast majority of the world's computing resources. Most of this infrastructure is lightly utilized and incurs high operation's management costs. Server and storage consolidation are the current best practices for decreasing costs of ownership in such environments. However, capacity related decisions about which applications should be placed on a consolidated server are often made informally. This paper presents an approach for automating such exercises. We characterize the complex time varying demands of such applications and then assign them to a small number of servers such that their capacity requirements are satisfied. The approach can be repeated on an on-going basis to ensure the continued efficient use of resources. A case study using data from 41 data center servers is used to demonstrate the effectiveness of the technique.

1 Introduction

Today's enterprise infrastructure is lightly utilized. Computing and storage resources are often cited as being less than 30% busy. This is in significant contrast to most large scientific computing centers which run at much higher utilization levels. Unfortunately, the complex resource requirements of enterprise applications, and the desire to provision for peak demand are reasons for such low utilization.

Infrastructure consolidation is the current best practice for increasing asset utilization and decreasing operations costs in enterprise environments. *Storage consolidation* replaces disks within hosts with virtual disks supported by storage area networks and disk arrays. This greatly increases the quantity of storage that can be managed per operator. *Server consolidation* identifies groups of applications that can execute together on an otherwise smaller set of servers without causing significant performance degradations or functional failures. This can greatly reduce the number, heterogeneity and distribution of servers that must be managed. For both forms of consolidation, the primary goal is typically to decrease operations costs.

Recent advances in utility computing make it possible for resource utilities to offer secure server and storage resources to enterprise applications on-demand [1][9]. Each resource utility requires a Resource Management System (RMS) to govern access to and automate the configuration of its resources [10]. Ideally, such an RMS should also automate the process of assigning applications to servers. A set of RMS may then cooperate as part of a grid for enterprise applications. By supporting more applications on fewer servers, such a grid can help to decrease operations and infrastructure costs.

In this paper, our focus is on a specific class of enterprise applications. We define an *application* of this class as a group of operating system processes that are assigned to the same server. These applications operate continuously and have time-varying demands but often require only a fraction of the capacity of their server.

Our contributions in this paper are as follows. We describe the workload characteristics of the above class of enterprise applications and a model for characterizing their resource demands. We then introduce two techniques for assigning applications to a consolidated set of servers. The first is based on a linear integer programming model, the second on a genetic algorithm. We apply the techniques in a case study involving data from 41 data center servers. Our results indicate that the genetic algorithm behaves well with respect to the integer programming approach, requires less computation, and provides greater opportunities for enhancement.

Section 2 gives an overview of related work. Section 3 describes our system under study, our workload model, and our experimental design. Assignment algorithms are introduced in Section 4. A case study is given in Section 5. Section 6 offers concluding remarks.

2 Background and Related work

Virtualization features are becoming common within today's enterprise infrastructures. Examples of virtualization features include: virtual Local Area Networks, Storage Area Networks and disk arrays that support virtual disks, and virtual machines and other partitioning technologies the enable resource sharing within servers. These features make it possible for resource management systems to offer joint, secure, and programmatic configuration and control for computing, networking and storage infrastructure [6][9].

Monitoring infrastructures provide the ability to measure the demands of applications. Once applications are assigned to a server, server scheduling mechanisms can ensure each application receives access to its required portion of server resources [2].

The above technologies help to enable a control-loop that can continuously re-evaluate and realize assignment alteratives. To close the loop, applications must provide interfaces that support their migration from one server to another while maintaining their qualities of service.

Assignment must deal with both the long term and short term consequences of consolidating applications on a server. Long term issues pertain to capacity planning. Shorter term issues deal with arbitration, for example, who should get access to resources when demand exceeds supply.

Long term consequences include interactions and correlations between the time varying demands of the applications. The demands of two applications interact when they both have similar patterns of demand. For example, both applications may have their heaviest demands between 9 am and 11 am on weekdays. Each application may have a range of demands that it typically experiences within that time frame. The demands are correlated if they tend to move together within their ranges.

The notion of how *long* long term is depends on how often applications are permitted to migrate and technologies that enable migration [4]. It is expected to be on the order of tens of minutes to many months. Rolia *et al.* consider the long term consequences of supporting multi-tier enterprise applications within a resource utility while providing a statistical assurance that resources will be available when needed [11].

Short term consequences deal with the scheduling of resources to applications at specific time instants in a manner that best meets the goals of the resource utility. For example, a scheduler must decide which applications get resources when demand exceeds supply. There are many techniques being proposed for dealing with such issues [3][8][12]. Short term decisions typically operate on time scales of tens of seconds to hours. In this paper, our focus is on long term issues.

3 Workload Characterization

This section describes our system under study, workload characterization, and the experimental design for our study. Our explanation of the assignment algorithms in Section 4 depends on values we introduce here in the experimental design.

3.1 System under Study

For the purpose of our study we were able to obtain CPU utilization information for 41 servers in a data center. The data was collected between September 2, 2001 and October 24, 2001. For each server, a trace of the average CPU utilization across all CPUs in the server was reported for each five minute interval. This information was collected using HP MeasureWare (Openview Performance Agent). We consider the work on each server as a single application and model per-interval CPU demand as the product of the number of CPUs on the server and their reported aggregate average utilization.

We identified four groups of servers that had similar hardware configurations. We treat these groups separately. Table 1 provides a breakdown of the groups of servers. The most homogeneous is Group one, where all 10 servers have the same number (six) and speed (240 MHz) of CPUs, and the same amount of memory

Group	Number of Servers	Server Class	CPU per Server	MHz	Memory (GB)
1	10	K-class	6	240	4
2	8	N-class	2-6	550	4-8
3	16	N-class	2-8	440	2-10
4	7	L-class	2-4	440	2-8

Table 1. Server Groups

(4 GB). The other groups consist of servers from the same family with the same CPU speed, but vary in the number of CPUs and the amount of memory per server.

For the purpose of our study, we treat the demand of each server as the demand of one enterprise application. We characterize each application using its trace of per interval CPU utilization. In general, we can also consider other attributes such as measures of bandwidth to and from the server and the amount of real memory used.

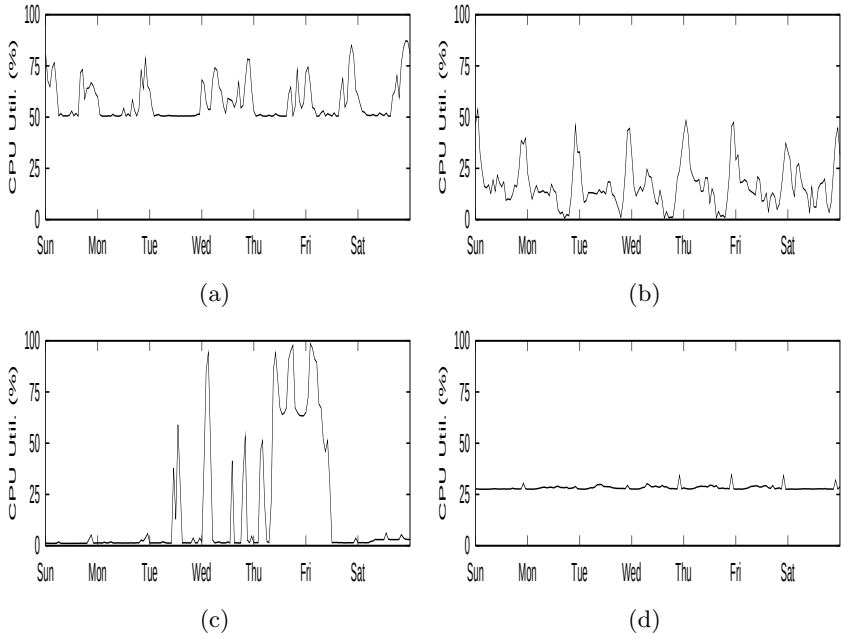


Fig. 1. One Week Traces of Application CPU Utilizations

Figure 1 illustrates the per-CPU utilization of four applications, each over a one week period. These applications portray the kinds of behaviour we have observed. Figures 1(a) and (b) illustrate clear time of day effects. The utilizations

are similar for the other weeks of data. The first has some constant background load, the other's utilization drops to near zero levels at certain times of day. Figure 1(c) shows the behaviour of an application that is less predictable. It has a surge of work towards the end of the week. For this application, some other weeks were basically idle, and others had surges that were more or less pronounced. Figure 1(d) illustrates a server with a flat CPU utilization. There were several of these, some with higher and some with lower CPU utilizations.

Traces provide us with a historical view regarding application behaviour. They must be long enough to capture behaviour that is representative. We assume that such behaviour will repeat itself; therefore we use the traces as input for our assignment algorithms.

When consolidating applications, we assume that an application from a server group is assigned to a consolidated server with a similar configuration. We also assume that the consolidated server has sufficient memory to support the aggregate demand of its assigned applications.

3.2 Experimental Design

This section describes our experimental design. In each case the goal is to minimize the number of servers needed for the entire time under study. Our design explores the sensitivity of assignment to several experimental factors as described below.

We consider the following scenarios for assignment:

A. Large multi-processor server: we assume that the target server is a single server with a large number of CPUs (e.g., 32, 64). Each CPU's worth of *load* from an application is assigned to a CPU for exactly one measurement interval. We assume processor affinity, i.e. the load cannot be served by any other CPU in this measurement interval. However, the CPU hosting the load is re-evaluated at the boundaries of measurement intervals.

B. Small servers without fast migration: here the target is a cluster of identical smaller servers with a small number of CPUs (e.g., 8). Each application is assigned to one server for the whole computation, but we do not assume processor affinity, i.e. the load might receive concurrent service from its server's CPUs.

C. Small servers with fast migration: this case is the same as case **B**, but the applications are assigned to the target servers for each separate measurement interval. This implies that an application may migrate between servers at the measurement interval boundaries. We do not consider the overhead of migration in our comparison.

For the above cases, we consider several additional factors.

– *Target utilization M* per CPU. We assign the applications to CPUs or servers in such a way that a fixed target utilization M of each CPU is not exceeded. The value of M that we use is either 50% or 80%, so each CPU retains an *unused capacity* that provides a soft assurance for *quality of service*. Since

our input data is averaged over 5-minutes intervals, the unused capacity supports the above-average peaks in demand within an interval. M can also be specified to set aside resources for expected server virtualization overheads. Finally, if per-CPU input utilization data exceeds the value of M for a measurement interval, it is truncated to M. Otherwise the mathematical programs can not be solved since no allocation is feasible. With our assignment algorithms, if a utilization is truncated, the application gets assigned to its own server but doesn't fully benefit from the $1 - M$ in soft assurance.
- *Interval duration* D. In addition to the original data with measurement intervals of 5 minutes duration, we average the CPU demands of three consecutive intervals for each original server, creating a new input data set with measurement interval length of 15 minutes. This illustrates the impact of measurement time scale and, for the case of fast migration, the impact of migrating applications less frequently.
- s, the number of CPUs in each small server. For cases **B** and **C**, we assume that the number s of CPUs of a single server is 6, 8 or 16.

Table 2 summarizes the the factors and levels, and indicates when they apply.

Factor	Symbol	Level	Applicable Cases
Target Utilization	M	50%, 80%	**A, B, C**
Interval Duration	D	5 min, 15 min	**A, B, C**
Fast migration	-	false, true	**B, C**
# of CPUs per server	s	6, 8, 16	**B, C**

Table 2. Summary of Factors and Levels

4 Assignment Methods

This section presents two assignment methods. The first is a linear integer programming approach. It takes traces of application demand as input and packs them onto as small as set of servers as is possible with a limited computation time. This is a compute intensive process but gives us a baseline set of results that can be used for comparison. Our second approach is a genetic algorithm.

4.1 Linear Integer Programming Approach

Our goal is to assign the offered applications to as small a set of CPUs or servers as is possible. To encode constraints for this goal, we introduce an *idle application*. Each CPU or server either hosts an idle application or at least one of the offered applications. The demand of the idle application is set to M for case **A** and $s \cdot M$ for cases **B** and **C**.

We designate:

$I = \{0, 1, \ldots\}$ as the index set of applications, where 0 is the index of the idle application, these are the applications under consideration;
$J = \{1, \ldots\}$ as the index set of CPUs (case **A**) or target servers (cases **B** and **C**);
$T = \{1, \ldots\}$ as the index set of measurement intervals (with durations of either 5 or 15 minutes); and,
$u_{i,t}$ as the demand of an application with index $i \in I$ for measurement interval with index $t \in T$. As mentioned above, for all $t \in T$ we set $u_{0,t} = M$ for case **A** and $u_{0,t} = s \cdot M$ for cases **B** and **C**.

Furthermore, for $i \in I$ and $j \in J$ let $x_{i,j}$ be a 0/1-variable with the following interpretation: 1 indicates that the application with index i is placed on a CPU or server with index j, 0 when no such placement is made. Note that $x_{0,j} = 1$, $j \in J$ indicates that the CPU or server with index j is hosting the idle application, i.e. it is unused.

The following class of 0/1 integer programs cover our cases. The objective function is:

- Minimize the number of CPUs (case **A**) or servers (cases **B** and **C**) used by maximizing the number of idle applications:

$$\text{maximize} \quad \sum_{j \in J} x_{0,j}.$$

With the following constraints:

- Let the unused CPUs (case **A**) or servers (cases **B** and **C**) be those with highest indices. For each $j \in J \backslash \{1\}$:

$$x_{0,j-1} \leq x_{0,j},$$

where $\backslash$ denotes removal from a set.
- Each non-idle application $i \in I$ is assigned to exactly one CPU (case **A**) or server (cases **B** and **C**). For all $i \in I \backslash \{0\}$:

$$\sum_{j \in J} x_{i,j} = 1.$$

- For case **A**, each target CPU must not exceed target utilization M in each time interval t. Each $t \in T$ gives rise to a new integer program to be solved separately. For all $j \in \mathbf{J}$:

$$\sum_{i \in \mathbf{I}} u_{i,t} x_{i,j} \leq M.$$

- For case **B**, each target server must not exceed target utilization M for each CPU for all measurement intervals $t \in T$ *simultaneously*. For all $j \in J$ *and* all $t \in T$:

$$\sum_{i \in \mathbf{I}} u_{i,t} x_{i,j} \leq sM.$$

– For case **C**, each target server must not exceed target utilization M for each CPU for each measurement interval t. Each $t \in T$ gives rise to a new integer program. For all $j \in J$:

$$\sum_{i \in \mathbf{I}} u_{i,t} x_{i,j} \leq sM.$$

As mentioned above, for cases **A** and **C** we solve one integer program for each measurement interval. Case **B** requires one integer program that includes all time intervals.

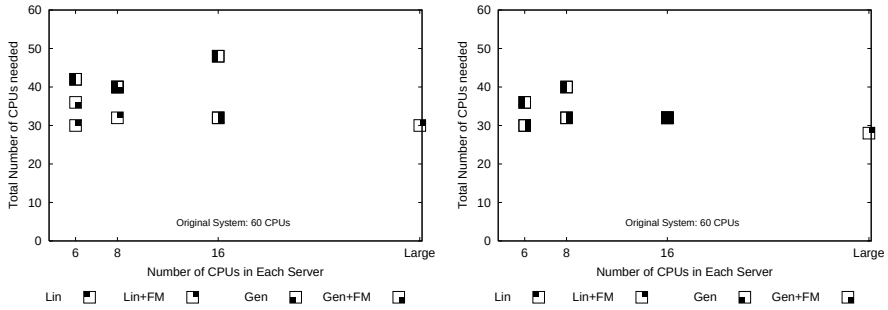
4.2 Genetic Algorithm Approach

We rely on the GALib genetic algorithm library as our genetic algorithm solver [7]. To use the library, it is necessary to prepare an initial assignment for the solver, to provide an objective function that is called by the solver to evaluate the utility of a next assignment, and to provide a mutation function and crossover method that are called by the solver to generate alternative assignments.

The encoding of a genome, a single assignment, is based on a one-dimensional array where a value v at position i means that application i is assigned to server v. For the initial assignment, we simply place all applications on a first server.

The solver aims to minimize an objective function. In our case, the objective is to minimize the number of servers used. When an assignment is presented to the objective function we compute the peak of aggregate demand on each server. To compute the peak of aggregate demand, we traverse the traces of all applications under consideration. By re-visiting these traces for each possible assignment, we take into account both interactions and correlations as described in Section 2. If the peak demand exceeds M per CPU for any server, then the assignment, i.e. the genome, is invalid. To correct the genome, we use a greedy algorithm that starts with the first server. The algorithm simply keeps a subset of applications on the first server such that the peak of their aggregate demand does not exceed M per CPU. Those applications that must be moved are moved to the next server, possibly causing a server to be added. This process is repeated with the remaining servers until all applications are assigned. The objective function returns the number of servers used. If an individual application requires more capacity than a server can offer, we mark the result of the optimization as infeasible and exit.

The solver invokes mutation and crossover routines to perturb the current assignment and arrive at the next assignment. Our code for the mutation routine implements three kinds of mutations. The first is to take a number of applications and re-assign them to the next higher server. This tends to add servers to the system. The second randomly swaps pairs of applications. The last takes all the applications from the last server and randomly assigns them to the other servers. This last step tends to decrease the number of servers used. We rely on the solver's built in uniform crossover method to generate next assignments based on earlier assignments.



(a) $D = 5$, $M = 50\%$ (b) $D = 15$, $M = 50\%$

Fig. 2. Numbers of CPUs Required for Group 1

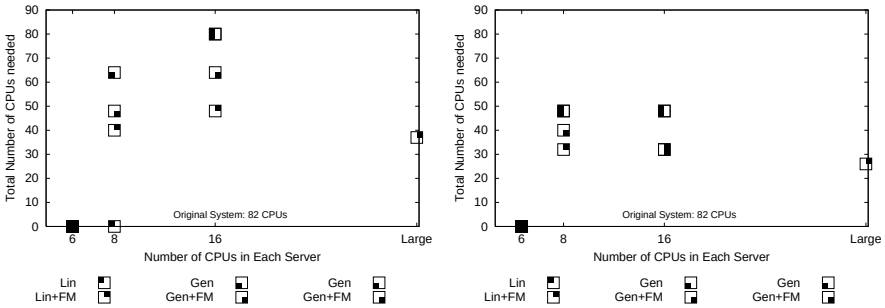
Finally, we describe the termination criteria used by the solver. The solver generates a population with sixty members (assignments); each member starts with the initial solution. The population is permitted forty generations (iterations). The solver returns the best assignment observed. When generating assignments, we specified that 80% of an assignment should be perturbed from one iteration to the next. The crossover probability and mutation probability were set to 0.75 and 0.1, respectively.

For the **B** cases, the assignments in the initial solution are based on the peak of the aggregate demand as computed over the traces. For the **C** cases, we take into account the ability of applications to migrate after each measurement interval. Our problem is to deduce how many servers are needed to support the applications in the presence of such migration. To do this, we consider a *daily profile* of demand. Time of day is partitioned into slots and each slot has a duration equal to that of a measurement interval D. We create an initial solution for each slot and apply the solver to each slot separately. When computing peak demands for an assignment for a slot, we consider only the subset of measurement intervals from the traces that apply to the slot. To compare the solution with the results of the linear integer programming approach, we report the maximum number of servers that are needed over all slots.

5 Case Study

This section presents the results of the assignment methods for the experimental design. Due to space constraints, a subset of results for cases **A, B,** and **C** are presented. We note that there is no guarantee that either of the assignment algorithms offers an optimal assignment.

Figure 2 presents results for Group 1 with $D = 5$ minutes and $M = 50\%$. Case **A** is the Large server case as identified on the x-axis. Cases **B** and **C** correspond to the six, eight, and sixteen CPUs per server cases, without and with fast migration, respectively. Squares in the figure identify the number of CPUs



(a) $D = 5$, $M = 50\%$ (b) $D = 5$, $M = 80\%$

Fig. 3. Numbers of CPUs Required for Group 3

required on the y-axis versus the number of CPUs per server. The quadrants of each square show the corresponding solution method and whether fast migration is considered. For example, consider Figure 2(a) and its upper-left most square. It shows that for the six CPU per server case, without fast migration, both the linear integer program and genetic algorithm required forty two CPUs. The squares below it show that for the fast migration case, the genetic algorithm required thirty six CPUs while the linear integer program only required thirty CPUs.

As the number of CPUs per server increases, we expect consolidation to be more effective. However, as Figure 2(a) shows, each server, in particular the last server, isn't always well utilized. For this reason, the sixteen CPU per server case shows a jump in the required number of CPUs. The Large case required only thirty CPUs. It has the least waste. The results for the Large case were computed using the linear integer program in a manner similar to the fast migration case. Figure 2(b) shows the impact of increasing D to fifteen minutes. This *smooths* reported bursts in utilization, in some cases, letting us pack more work onto fewer CPUs. For example, we report fewer CPUs as required for the six CPU per server case without fast migration, and the sixteen CPU per server case without fast migration.

Figure 3(a) and (b) show results for Group 3 with $D = 5$ minutes and $M = 50\%$ or $M = 80\%$, respectively. As the utilization threshold increases, clearly more work can be added to each server. We note that in both Figure 3(a) and (b), solutions were not possible for the six CPU per server cases. This is because some applications required more than six CPUs even after truncation of demand with respect to M. These are identified as squares at $y = 0$ (i.e. 0 CPUs). Note that given a limited computation time, the linear integer program did not return a valid solution for the eight CPU per server case without fast migration in Figure 3(a). The genetic algorithm was able to find a solution. Computation times are discussed in the next subsection.

The results for Groups 2 and 4 were nearly identical for both the linear integer program and the genetic algorithm. Valid solutions were found for all cases in Groups 1, 2, and 4.

From our case study, we find that the genetic algorithm typically finds that the same number of servers are required as the linear integer programming approach. In some cases an additional server is required. These cases usually required an additional CPU or two that force the addition of an entire server.

5.1 Computation times

The linear integer program computations were performed on an 8 CPU server with 440 MHz PA-RISC processors and 16 GB of memory running HP-UX. We relied on the AMPL 7.1.0 modeling environment and CPLEX 7.1.0 optimizer [5]. The environment only used one CPU at a time. CPU time limits were established for each run of the CPLEX tool. For each run, we report the best solution found within its time limit. For scenarios in cases **A** and **C**, time limits were generally 128 seconds. Case **B** scenarios had limits of 4096 seconds. There were a few scenarios in **B** that did not yield good allocations. These were cases where we deemed the bounds for the solution, as reported by CPLEX, to be too wide. They then received 131072 seconds of computation time. As illustrated in Figure 3(a), the eight CPU per server case without fast migration did not return a valid solution at all. Several case **C** scenarios received 2048 seconds of processing time. Unfortunately, we only had limited access to the modeling environment; some runs did not lead to satisfactory allocations.

The genetic algorithm was performed on a laptop with a 2GHz Pentium IV processor. It typically required tens of seconds per solution, and never required more than two minutes. Solutions were found for each scenario. Even after taking into account differences in processor speeds between the laptop and server, the genetic algorithm performed quickly, especially with respect to the time consuming case **B** scenarios.

6 Summary and Conclusions

In this paper, we describe two assignment algorithms for assigning a class of enterprise applications to servers in resource utilities. The first is a linear integer programming approach, the second is based on a genetic algorithm. The algorithms are used in a case study involving data from 41 data center servers.

For our data set, the linear integer programming approach offers the best solutions but had relatively large solution times. It seems most appropriate as an off-line algorithm. The genetic algorithm offered solutions that were nearly as good as the programming approach. However solution times were much lower. It appears to be a good candidate for an on-line assignment algorithm that can be used in a control-loop that automates the assignment and re-assignment of applications. The genetic algorithm permitted us to compute peak aggregate demands as assignments were considered. This helped to reduce the problem

size with respect to the linear integer programming approach. With the latter approach, the same strategy is not possible. Furthermore, the genetic approach can be easily enhanced. For example, the objective function can be modified to consider features of assignments that can not be expressed in a linear integer program.

We intend to extend our work in several ways. We plan to collect data from additional data centers for greater periods of time. With longer traces, we can use parts of the traces for deciding assignments and the remainder for validation. In addition to collecting measures of CPU utilization, we intend to collect information on network and storage utilization, in order to get a more complete picture of data center usage. We also plan to extend these techniques to consider multiple demand attributes and classes of service regarding access to resources [10][11] and to further study our use of the genetic algorithm.

References

1. K. Appleby, S. Fakhouri, L. Fong, M. Goldszmidt, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano – SLA based management of a computing utility. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, May 2001.
2. H. Brown. Application consolidation tutorial, 1999. `http://docs.hp.com/hpux/onlinedocs/ha/app_consolidation.pdf` .
3. J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.
4. Ejasent. Utility computing white paper, November 2001. `http://www.ejasent.com`.
5. R. Fourer, D.M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press / Brooks/Cole Publishing Company, 1993.
6. Hewlett-Packard. HP utility data center architecture. `http://www.hp.com/solutions1/infrastructure/solutions/utilitydata/architecture/`.
7. Illinois Genetic Algorithms Laboratory. Galib. `http://www-illigal.ge.uiuc.edu/index.php3`.
8. R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance management for cluster based web services. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, pages 247–261, March 2003.
9. J. Rolia, S. Singhal, and R. Friedrich. Adaptive Internet Data Centers. In *SSGRR'00*, L'Aquila, Italy, July 2000.
10. J. Rolia, X. Zhu, and M. Arlitt. Resource access management for a resource utility for enterprise applications. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, pages 549–562, March 2003.
11. J. Rolia, X. Zhu, M. Arlitt, and A. Andrzejak. Statistical service assurances for applications in utility grid environments. In *Proceedings of the Tenth IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 247–256, October 2002.
12. Sychron. Sychron Enterprise Manager, 2001. `http://www.sychron.com`.