INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Virtual Machine Resource Allocation for Service Hosting on Heterogeneous Distributed Platforms

Henri Casanova — Mark Stillwell — Frédéric Vivien

N° 7772

October 2011

Distributed and High Performance Computing

ISSN 0249-6399 ISRN INRIA/RR--7772--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Virtual Machine Resource Allocation for Service Hosting on Heterogeneous Distributed Platforms

Henri Casanova, Mark Stillwell, Frédéric Vivien

Theme : Distributed and High Performance Computing
Équipe-Projet GRAAL

Rapport de recherche n° 7772 — October 2011 — 44 pages

Abstract: We propose algorithms for allocating multiple resources to competing services running in virtual machines on heterogeneous distributed platforms. We develop a theoretical problem formulation and compare these algorithms via simulation experiments based in part on workload data supplied by Google. Our main finding is that vector packing approaches proposed in the homogeneous case can be extended to provide high-quality solutions in the heterogeneous case, and combined to provide a single efficient algorithm. We also consider the case when there may be errors in estimates of performance-related resource needs. We provide a resource sharing algorithm and prove that for the single-resource, single-node case, when there is no bound on the error, its performance ratio relative to an omniscient optimal algorithm is $\frac{2J-1}{J^2}$, where J is the number of services. We also provide a heuristic approach for compensating for bounded errors in resource need estimates that performs well in simulation.

Key-words: scheduler, virtual machines, heterogeneous vector bin packing, grid computing, cloud computing

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Allocation de ressources pour l'hébergement de services sur machines virtuelles déployées sur plates-formes hétérogènes distribuées

Résumé : Nous proposons des algorithmes pour l'allocation des ressources aux services concurrents s'exécutant dans des machines virtuelles déployées sur des systèmes hétérogènes et distribuées. Nous développons une formulation théorique du problème et comparons les algorithmes proposés via des simulations basées en partie sur des traces mises à disposition par Google. Notre principale conclusion est que les approches de *bin packing* vectoriel proposées pour le cas homogène peuvent être étendues afin de fournir des solutions de qualité dans le cas hétérogène. Ces approches peuvent également être combinées entre elles pour fournir un seul algorithme efficace. Nous considérons également le cas où les estimations des besoins en ressources sont connus de manière imparfaite. Nous montrons, quand il y a un unique nœud et une unique ressource, et quand l'erreur maximale sur l'estimation des besoins en ressource n'est pas bornée, que l'on peut définir un algorithme avec un facteur de compétitivité de $\frac{2J-1}{J^2}$, où J est le nombre de services. Quand il existe une borne sur l'erreur maximale sur l'estimation des besoins, nous définissons une heuristique qui obtient de très bonnes performances même en présence de telles erreurs.

Mots-clés : ordonnancement, machines virtuelles, bin packing vectoriel hétérogène

1 Introduction

The trend toward increasing use of virtual machine technology in the data center, both leading to and reinforced by recent innovations in the private sector aimed at providing low-maintenance cloud computing services, has driven a great deal of research into developing algorithms for automatic instance placement and resource allocation on virtualized platforms [1, 2], including our own previous work [3]. Most of this research has assumed a platform consisting of homogeneous nodes connected by a fast local network, e.g., a cluster. However, there is a need for algorithms that are applicable to heterogeneous platforms.

Heterogeneity comes about when collections of homogeneous resources formerly under different administrative domains are *federated*, leading to a set of resources that belong to one of several classes. This is the case when federating multiple clusters at one or more geographical locations (e.g., grid computing, sky computing). There is some concern that, in these scenarios, slow communication links between sites could lead to degraded performance. However studies have shown that multi-cluster scheduling can improve performance at the workload level, even for workloads that contain parallel applications with communicating tasks [4]. The production cycle is also a common source of heterogeneity –when the time comes to purchase new machines, it generally makes more economic sense to purchase a more powerful machine than was used previously. An initially homogeneous platform thus can evolve to eventually comprise an arbitrary number of different classes of machines. Finally, in budget shops or when a large collection of individually owned (desktop) machines is assembled, one obtains a highly heterogeneous environment in which no two machines may even share the same hardware specification.

In this work we propose virtual machine placement and resource allocation algorithms that, unlike previous proposed algorithms, are applicable to virtualized platforms that comprise heterogeneous physical resources. More specifically, our contributions are:

- We give a formulation of the service placement and resource allocation problem in heterogeneous virtualized platforms. This formulation is in fact more general, even for homogeneous platforms, than our previously proposed formulation in [3], and allows for specifying minimum allocations of arbitrary resources to satisfy Quality-of-Service (QoS) constraints.
- Using this problem formulation, we extend previously proposed algorithms to the heterogeneous case.
- We evaluate these algorithms via extensive simulation experiments, using statistical distributions of application resource requirements based on a real-world dataset provided by Google. We find that combining heterogeneous vector bin packing approaches leads to a practical and efficient solution.
- Most resource allocation algorithms rely on estimates regarding the resource needs of virtual machine instances. We study the impact of estimation errors, propose different approaches to mitigate these errors, and identify a strategy that works well empirically.

This paper is organized as follows. In Section 2 we formalize the resource allocation problem. In Section 3 we provide an overview of our heuristic algorithms. We describe our experimental procedure for evaluating these algorithms in Section 4 and present the obtained results in Section 5. In Section 6 we discuss the problem of scheduling in the presence of errors in estimated needs, and provide theoretical and experimental results. Section 7 discusses related work. Section 8 concludes the paper with a summary of our results and a discussion of future directions.

2 Problem Statement

We consider a service hosting platform composed of H heterogeneous hosts, or *nodes*. Each node comprises D types of different resources, such as CPUs, network cards, hard drives, or system memory. For each type of resource under consideration a node may have one or more distinct resource *elements* (e.g., a single real CPU, hard drive, or memory bank). In this work we assume that all resource elements of the same type on a given host are homogeneous. That is, we do not consider resources that comprise heterogeneous sub-units, such as the Cell™ processor [5].

Services are instantiated within virtual machines that provide analogous *virtual elements*. For some types of resources, like system memory or hard disk space, it is relatively easy to pool distinct elements together at the hypervisor or operating system level so that hosted virtual machines can effectively interact with only a single larger element. For other types of resources, like CPU cores, the situation is more complicated.

These resources can be partitioned arbitrarily among virtual elements, but they cannot be effectively pooled together to provide a single virtual element with a greater resource capacity than that of a physical element. For example, with current technology, a virtual machine instance with 3 virtual processors can run on a dual-core system with each of the virtual CPUs receiving 66.6% of the processing power of a single physical CPU [6], but it is impossible to pool two physical CPUs together to provide a single virtual CPU that is more powerful than the individual physical CPUs. For these types of resources, it is necessary to consider the maximum capacity allocated to individual virtual elements, as well as the aggregate allocation to all virtual elements of the same type.

Thus, each node is represented by an ordered pair of D-dimensional vectors. The *elementary capacity* vector gives the capacity of a single element in each dimension while the *aggregate capacity* vector gives the total resource capacity counting all elements. In practice, each value of the aggregate capacity vector is expected to be an integer multiple of the corresponding value in the elementary capacity vector, but our proposed approach does not rely on this assumption.

An *allocation* of resources to a virtual machine specifies the maximum amount of each individual element of each resource type that will be utilized, as well as the aggregate amount of each resource of each type. Expectedly, an allocation is thus represented by two vectors, a *maximum elementary allocation* vector and an *aggregate allocation* vector. Note that in a valid allocation it is not necessarily the case that each value in the second vector is an integer multiple of the corresponding value in the first vector, as resource demands may be unevenly distributed across virtual resource elements. For example, consider an application that uses two virtual CPUs: The application consists of one CPU-bound process that can utilize 100% of a physical CPU, and of one I/O-bound process that uses at most 10% of a physical CPU. The aggregate CPU resource allocation needed to run this service without degrading performance would then be 110% (easily achieved on a dual-core system), which is not an integer multiple of the maximum elementary resource allocation of 100%.

When determining how much of each resource to allocate to a virtual machine it is necessary to consider both the rigid *requirements* of the service as well as its fluid *needs*. The requirements for service j are given by an ordered vector pair (r_j^e, r_j^a) that represents the resource allocation needed to run the service at the minimum acceptable service level. If this requirement cannot be met, then resource allocation fails. The needs of the service are given by a second ordered vector pair (n_j^e, n_j^a) that represents the *additional* resources required to run the service at its maximum level of performance when it is by itself on a reference machine. While in general the most sensible reference machine is the best one for running the service on the target platform, this is not imposed. There are situations where it may make sense to set the needs based on some other criteria: the system or pricing structure may impose maximum virtual machine allocations that are less than the maximum of what is physically available, or it may be desirable to compare the relative performance of the same workload across a variety of platforms.

In this formulation of the problem, rather than attempting to maximize the performance of individual applications we choose to focus on maintaining an appropriate proportional allocation of resources, as resource allocations are known to be transformable into higher level service objectives [7]. Toward this end, we define the *yield* of a service to be a value between 0 and 1, representing the relative performance of that service. A service with an assigned yield of 0 would be running at the lowest acceptable rate of service for a resource allocation to not be considered a failure, while a service with an assigned yield of 1 would be running at the maximum performance possible on the reference machine. The allocation needed to run a service j with yield y_j is then defined by the couple $(r_j^e + y_j n_j^e, r_j^a + y_j n_j^a)$. This linear correlation between resource consumption levels in different dimensions is established in the literature [8] and easily justified for a wide class of applications. Benchmarking studies have validated the approach for scientific workload applications [9].

We can now define the service placement and resource allocation problem precisely: *maximize the minimum yield over all services*. This amounts to making the least satisfied service as satisfied as possible, thus promoting both performance and fairness. This objective is directly motivated by known results derived in the context of stretch optimization [10, 11]. In particular, it is well known that simply minimizing the average stretch (and thus maximizing the average yield) is prone to starvation. For the single-node case maximizing the minimum yield addresses the problem of fair-sharing of multiple resources, potentially in the presence of multiple bottlenecks as discussed in Dolev et al. [8], though we choose to focus on

maximizing the performance of the most-penalized service, rather than ensuring that services receive their fair share of bottleneck resources.

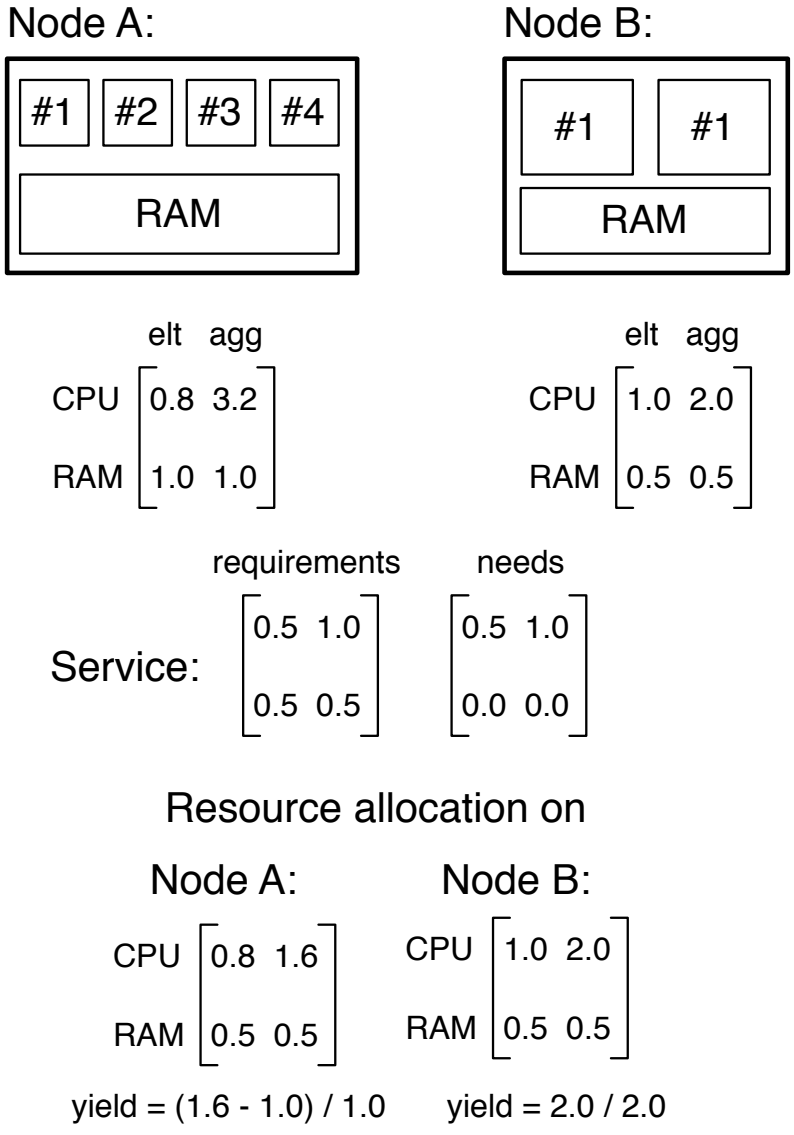

Figure 1: Example problem instance with two nodes and one service, showing two possible resource allocations.

Figure 1 illustrates the above for a simple example with two nodes and one service. Node A comprises 4 cores and a large memory. Its resource capacity vectors show that each core has elementary capacity 0.8 for an aggregate capacity of 3.2. Its memory has capacity 1.0, with no difference between elementary and aggregate values because the memory, unlike cores, can be partitioned arbitrarily (e.g., no single virtual CPU can run at 0.9 CPU capacity on this node). Node B has two cores, more powerful, cores, each of elementary capacity 1.0, and a smaller memory. The service has a 0.5 elementary CPU requirement, and a 1.0 aggregate CPU requirement. For instance, it could contain two threads that must each saturate a core with 0.5 capacity. The memory requirement is 0.5. The elementary CPU need is 0.5 and the aggregate is 1.0. The memory need is 0.0, meaning that the service simply requires a 0.5 memory capacity. The figure shows two resource allocations one on each node. On both nodes the service can be allocated the memory it requires. If the service is placed on Node A, then the elementary requirements and needs can be fully satisfied as they are both 0.5 and this is less than the elementary allocation of 0.8. However, since the

aggregate capacity is 1.6 and the service has a CPU requirement of 1.0 that must be fully satisfied in order for the resource allocation to be successful, only 0.6 of the aggregate CPU capacity can be used to satisfy needs and improve performance. As the aggregate CPU need is 1.0, the maximum yield possible for the service with this allocation is $0.6/1.0 = 0.6$. On Node B, the service can fully saturate two cores, leading to an aggregate CPU allocation of 2.0. The service's yield is then $(2.0 - 1.0)/1.0 = 1.0$. If there is only one service to consider, then placing this service on Node B maximizes the (minimum) yield.

3 Algorithms

The minimum yield maximization problem defined in the previous section is NP-hard in the strong sense via a straightforward reduction to vector-packing (in fact, the reduction to vector packing also denies the existence of an asymptotic polynomial-time approximation scheme) [12]. We therefore seek heuristic algorithms that work well in practice for a wide range of relevant scenarios.

3.1 Linear Program Formulation

Our problem can be framed as a linear program (LP) with both integer and rational variables, i.e., a Mixed Integer Linear Program (MILP). Using $0 \leq j < J$ to index the services, $0 \leq h < H$ to index the nodes, and $0 \leq d < D$ to index the resource dimensions, we define the following variables: e_{jh} is set to 1 if service j is placed on node h, or 0 otherwise; and y_{jh} is the yield of service j on host h. We use r^e_{jd} (resp. r^a_{jd}) to denote the elementary (resp. aggregate) resource requirement of service j for resource d, n^e_{jd} (resp. n^a_{jd}) to denote the elementary (resp. aggregate) resource need of service j for resource d, and c^e_{hd} (resp. c^a_{hd}) to denote the elementary (resp. aggregate) resource capacity of node h for resource d. Given these notations, the constraints of the MILP are given by Equations 1-7.

$$\forall j, h \quad e_{jh} \in \{0, 1\}, \tag{1}$$
$$\forall j, h \quad y_{jh} \in [0, 1], \tag{2}$$
$$\forall j \quad \sum_{h=1}^{H} e_{jh} = 1, \tag{3}$$
$$\forall j, h \quad y_{jh} \leq e_{jh} \tag{4}$$
$$\forall j, h, d \quad e_{jh} r^e_{jd} + y_{jh} n^e_{jd} \leq c^e_{hd} \tag{5}$$
$$\forall h, d \quad \sum_{j=1}^{J} (e_{jh} r^a_{jd} + y_{jh} n^a_{jd}) \leq c^a_{hd} \tag{6}$$
$$\forall j \quad \sum_{h=1}^{H} y_{jh} \geq Y \tag{7}$$

Constraints 1 and 2 define the ranges of the variables. Constraint 3 states that a service must be placed on exactly one node. Constraint 4 states that a service has a non-zero yield only on the node on which it is placed. Constraint 5 states that the elementary resource capacities of the nodes are not overcome, and Constraint 6 states that the aggregate resource capacities of the nodes are not overcome. Finally, Constraint 7 defines variable Y as the minimum of the yields of all services. The optimization objective is thus to maximize Y.

3.2 Exact and Relaxed Solutions

We have implemented a resource allocation problem solver that can use either the Gnu Linear Programming Toolkit (GLPK) [13] or the IBM iLog CPLEX Optimization Studio [14] as back-end MILP solvers. Since solving a MILP takes exponential time, for large instances we relax the problem by assuming that all variables are rational. The MILP becomes a rational LP, which can be solved in polynomial time in practice. The solution of the rational LP may be infeasible but has two important uses. First, the achieved minimum yield is an upper bound on the solution of the MILP. Second, the rational solution may point the way toward good feasible solutions as seen in the next section.

3.3 Algorithms Based on Relaxed Solutions

We propose two algorithms, RRND and RRNZ, that use a solution of the rational LP as a basis and then round off rational e_{ij} values. Due to licensing restrictions, we could not run large numbers of CPLEX solvers. Consequently, we use the solutions produced by GLPK.

3.3.1 Randomized Rounding (RRND)

For each service j (taken in an arbitrary order), this algorithm allocates it to node h with probability e_{jh}. If the service cannot fit on the selected node because of memory constraints, then probabilities are adjusted and another attempt is made. If all services can be placed in this manner then the algorithm succeeds. Such a probabilistic approach has been used successfully in previous work [15].

3.3.2 Randomized Rounding with No Zero probabilities (RRNZ)

One problem with RRND is that a service, j, may not fit (in terms of resource requirements) on any of the nodes, h, for which $e_{jh} > 0$, in which case the algorithm would fail to generate a solution. To remedy this problem, we first set each zero e_{jh} value to ϵ, where $\epsilon << 1$ (we use $\epsilon = 0.01$). For those problem instances for which RRND provides a solution RRNZ should provide nearly the same solution, but RRNZ should also provide a solution for some instances for which RRND fails.

3.4 Greedy Algorithms

In general greedy algorithms operate by making a series of fast, locally-optimal decisions. They run quickly, but may provide poor results for some problem instances. One way to deal with this issue is to run several greedy algorithms and choose the best solutions from those computed. In our previous paper [3], in the context of homogeneous platforms, we proposed a family of greedy algorithms that first sort the services under consideration and then go through the list in sorted order, selecting the best node for each item by various criteria. We considered the following service sorting strategies: S1: no sorting; S2: decreasing order by maximum need; S3: decreasing order by sum of needs; S4: decreasing order by maximum requirement; S5: decreasing order by sum of requirements; S6: decreasing order by the maximum of sum of requirements and sum of needs; and S7: decreasing order by the sum of requirements and needs. We also proposed a number of strategies for selecting a node from the set of those capable of running the current service: P1: choose the node with the most available resource capacity in the dimension of maximum need; P2: choose the node with the minimum ratio of sum of loads over all dimensions to sum of resource capacities over all dimensions after service placement, P3: choose the node with the least remaining capacity in dimension of largest requirement (best fit), P4: choose the node with the least aggregate available capacity (best fit), P5: choose the node with the most capacity remaining in dimension of largest requirement (worst fit), P6: choose the node with the most total available resource (worst fit), P7: choose the first node (first fit). Each sorting strategy can be paired with any selection strategy to obtain a greedy algorithm, for a total of $7 \times 7 = 49$ combinations. Given that these 49 algorithms can be executed quickly, we simply define the METAGREEDY algorithm which runs all 49 greedy algorithms and pick the solution that results in the largest minimum yield.

3.5 Vector-Packing (VP) Algorithms

As has been noted in the literature, resource allocation is closely related to bin packing [16]. Bin-packing can be extended to vector-packing [17], also called multi-capacity bin-packing by some authors [18], in the case where multiple resource dimensions are under consideration. In our previous work we discussed a straightforward method of applying any vector packing heuristic to our resource allocation problem with the objective of maximizing the minimum yield: Since the amount of each resource that needs to be allocated to a given service is fixed for a particular yield value, it is possible to determine whether or not a given heuristic can find a solution for that yield value. Since we seek to maximize the minimum yield value, without loss of generality we can assume that all services have the same yield. Thus, we perform a binary search for the

largest yield for which the heuristic can find a solution. We stop when the upper and lower bounds of the binary search are within some threshold distance of each other (0.0001 in our simulation experiments).

The largest source of difficulty in designing vector-packing heuristics is that there is no single unambiguous definition of vector "size" or "order". Any mapping from vectors to a scalar metric can be used, and the use of different metrics must be evaluated in simulation as analytical comparison is typically not feasible. We consider the following metrics: size of maximum dimension (MAX), sum of all dimensions (SUM), ratio of maximum and minimum dimension (MAXRATIO), and the difference between maximum and minimum dimensions (MAXDIFFERENCE). We also consider ordering the vectors lexicographically (LEX), though the ordering of the dimensions is necessarily arbitrary. Some strategies may also specify NONE, to leave items in their natural order and not attempt sorting. Since there are 5 mappings of vectors to scalars under consideration that can be used to sort vectors in either ascending or descending order, and one option to not sort vectors, we consider 11 distinct strategies for vector sorting. In general, we should expect algorithms that sort items in decreasing order by size and bins in increasing order by capacity (e.g., algorithms that first try to put the largest items into the smallest bins) to have the best performance. For each of the sorting options, we can now define vector packing algorithms as follows.

3.5.1 First and Best Fit

Once the items are sorted using one of the sorting strategies, the First Fit algorithm places each item in this order in the first bin in which it fits using an arbitrary bin order. The Best Fit algorithms considers bins in descending order of the sum of their loads across all dimensions.

3.5.2 Permutation-Pack and Choose-Pack

The Permutation-Pack and Choose-Pack heuristics for multi-capacity bin packing, also known as vector-packing, were first proposed by Leinberger et al. [18] The basic strategy used by these algorithms is to go bin-by-bin and select items that go against the current "capacity imbalance" in the first w most-loaded dimensions. The idea behind this strategy is to keep a bin from becoming full in one dimension while it still has remaining capacity available in other dimensions. The value of w is called the "window size" for the algorithm.

Let us use D to denote the number of dimensions of bin and item vectors. To select an item to put into the current bin, the Permutation Pack algorithm ranks the dimensions of the current bin in ascending order by their load and ranks the dimensions of each of the items in descending order by their size. The items are then ordered by how well their dimension ranking matches against that of the current bin in the first w dimensions. An item that has a permutation matching that of the bin will have its largest requirement in the dimension where the load on the bin is smallest and its smallest requirement in the dimension where the load on the bin is largest. The Choose Pack algorithm relaxes this ordering slightly and only considers which dimensions fit within the window, but not their relative ordering. It should be noted that when the window size is 1, the Permutation Pack and Choose Pack algorithms operate identically.

The implementation proposed by Leinberger et al. in [18] separates the items into $D!$ lists–one for each potential permutation of item dimensions imposed by ordering the dimensions in descending order by their size. The items in these lists are further sorted by one of the vector sorting criteria discussed previously. For each bin it selects items by going through these lists in a lexicographic order determined by the permutation imposed on the bin dimensions by sorting them in ascending order by their current load. That is, it first looks for an item in the list with a permutation matching that of the current bin. If no such item can be found then it looks for an item in the list where the positions of the two least important dimensions are reversed. This continues until an item that fits in the current bin is found, with items whose dimensional permutation is opposite of the current bin considered last. If the algorithm cannot find any item that fits in the current bin, then it moves on to the next one.

In the case that $D! \leq J$ (i.e., for small D) the algorithm essentially makes multiple passes through all of the items, selecting one to place in the current bin at each step, and so is order $O(J^2)$. For the case when $D! >> J$, Leinberger et al. note that searching for the matching list can require $O(D!)$ operations; In fact, if these permutations are not first mapped into an easily comparable set then each comparison of permutations is an $O(D)$ operation, potentially leading to an overall cost of $O(J^2 + JDD!)$ operations for this heuristic.

Their proposal to use a window to check only the first few, most important, dimensions reduces the cost of the search to $O(DD!/(D-w)!)$, for an overall complexity of $O(J^2 + JDD!/(D-w)!)$.

To reduce the high complexity of this algorithm, we have made the following improvement. Instead of needlessly splitting the items into $D!$ lists, many of which may be empty when D is large, and checking each list in order, we simply assign each item a sorting key by mapping its dimension permutation into the permutation space defined by the bin's dimensional ordering, a $O(JD)$ operation. For example, consider a bin with dimensional ordering is $(4, 2, 3, 1)$ in a 4-dimensional case (meaning that its largest capacity is its 4th dimension, the next largest capacity is its 2nd dimension, etc.), and an item with ordering $(3, 1, 4, 2)$ (i.e., its largest requirement is its 3rd dimension, its next largest demand is its 1st dimension, etc.). The key assigned to this item is then $(3, 4, 1, 2)$ (the 1st dimension of the item ordering is the 3rd dimension of the bin ordering, the 2nd dimension of the item ordering is the 4th dimension of the bin ordering, etc.). Once these permutations are computed for each item, we simply chose the first item in the lexicographic order of the permutations. In our example, an item with permutation $(1, 2, 3, 4)$ would be perfectly fitted for the bin. No sorting of the services is involved and a single scan through the list is necessary, or $O(JD)$ operations. Thus, our implementation is $O(J^2D)$, which is a significant improvement when D is large, or $O(J^2w)$ if a window is used.

3.5.3 METAVP

At each step of the binary search, this algorithm iteratively applies all of the above vector packing strategies until a solution is found, including all of the options for sorting items–$3 \times 11 = 33$ strategies overall for each step of yield optimization. METAVP necessarily performs at least as well as all previously discussed algorithms, but also has much longer run time.

3.5.4 Heterogeneous Vector-Packing (HVP) Algorithms

These algorithms explicitly consider the heterogeneity of the platform and try to sort the vector bins. The Best-fit, Permutation-Pack algorithm is also modified to consider total remaining capacity rather than total load.

3.5.5 METAHVP

This algorithm tries all of the above heterogeneous bin-packing heuristics at each step of the binary search. Since Best-Fit and Permutation-Pack can consider bins in sorted order, but Best-Fit imposes its own criteria on bin selection, this is is $11 + 2 \times 11 \times 11 = 253$ strategies overall. METAHVP necessarily performs at least as well as any of the HVP algorithms, but its run time can be high.

4 Simulation Experiments

Though we provide a general framework capable of representing an arbitrary number of resource constraints, in practice the two most important resources to consider are processing power (CPU) and memory. Often, these are the only resources with consumption levels reported in log files, and thus the only ones for which it is possible to build reasonable statistical models of demand. These are also resources for which current virtualization technology provides accurate performance throttling and isolation. For these reasons in our simulation experiments we choose to focus on the two-dimensional problem with one CPU resource dimension and a memory resource dimension.

In order to control the relative levels of node heterogeneity, rather than reflecting the statistics of any actual set of machines we draw aggregate CPU and memory capacities from a normal distribution with a median value of 0.5, limited to minimum values of 0.001 and maximum values of 1.0. The coefficient of variation is varied from 0.0 (completely homogeneous) to 1.0. To generate consistent elementary capacities, we assume that despite differences in total computational power all machines are quad core, and therefore have CPU elements with $1/4$ the aggregate machine power.

To instantiate service resource requirements and needs, we use a production dataset provided by Google Inc. [19]. As this dataset only provides information about the number of requested cores and the percentage

Table 1: Major Heuristics, ordered pairs of $(\mathcal{Y}_{A,B}, \mathcal{S}_{A,B})$, with A given by the row and B by the column.

100 services					
A/B	RRND	RRNZ	METAGREEDY	METAVP	METAHVP
RRND		(66.7%, -19.7%)	(-41.8%, -22.2%)	(-71.6%, -22.2%)	(-71.6%, -22.2%)
RRNZ	(-40.0%, 19.7%)		(-69.7%, -2.5%)	(-85.2%, -2.5%)	(-85.3%, -2.5%)
METAGREEDY	(71.7%, 22.2%)	(230.3%, 2.5%)		(-50.9%, 0.0%)	(-51.0%, 0.0%)
METAVP	(252.4%, 22.2%)	(577.6%, 2.5%)	(103.6%, 0.0%)		(-0.2%, 0.0%)
METAHVP	(252.4%, 22.2%)	(578.9%, 2.5%)	(104.0%, 0.0%)	(0.2%, 0.0%)	
250 services					
A/B	RRND	RRNZ	METAGREEDY	METAVP	METAHVP
RRND		(76.9%, -41.3%)	(-38.2%, -52.8%)	(-73.3%, -52.4%)	(-73.3%, -52.9%)
RRNZ	(-43.5%, 41.3%)		(-69.4%, -11.5%)	(-86.5%, -11.1%)	(-86.7%, -11.6%)
METAGREEDY	(61.7%, 52.8%)	(227.2%, 11.5%)		(-55.4%, 0.4%)	(-56.4%, -0.1%)
METAVP	(273.9%, 52.4%)	(643.0%, 11.1%)	(124.3%, -0.4%)		(-2.1%, -0.4%)
METAHVP	(275.2%, 52.9%)	(652.6%, 11.6%)	(129.5%, 0.1%)	(2.1%, 0.4%)	
500 services					
A/B	RRND	RRNZ	METAGREEDY	METAVP	METAHVP
RRND		(74.5%, -49.2%)	(-32.7%, -59.6%)	(-69.0%, -59.6%)	(-69.3%, -59.6%)
RRNZ	(-42.7%, 49.2%)		(-62.8%, -10.4%)	(-82.3%, -10.4%)	(-83.0%, -10.4%)
METAGREEDY	(48.6%, 59.6%)	(168.6%, 10.4%)		(-51.5%, 0.0%)	(-54.1%, 0.0%)
METAVP	(222.9%, 59.6%)	(464.7%, 10.4%)	(106.3%, 0.0%)		(-5.2%, 0.0%)
METAHVP	(226.0%, 59.6%)	(489.1%, 10.4%)	(117.7%, 0.0%)	(5.5%, 0.0%)	

of system memory used, we assume that aggregate CPU needs of services are proportional to the number of requested cores, while elementary CPU requirements are equal to the same reference value for all services.

We consider scenarios with 64 hosts, 100, 250 and 500 services and coefficient of variation values 0.0 to 1.0 in increments of 0.025 For each scenario we generate 100 random instances, for a total of 12,300 base instances. Each base instance is then use to generate a family of scaled problem instances with specified *memory slack*, defined at the fraction of total memory that would remain free in a successful resource allocation. The memory slack is a value between 0 and 1 that quantifies the hardness of the instance in terms of memory bin packing, with a low value corresponding to a more difficult instance. We experiment with slack values between 0.1 and 0.9, in 0.1 increments. We also scale CPU needs so that the sum of all service CPU needs is equal to the sum of all CPU resources available.

5 Experimental Results

We evaluate a large number of algorithms over a large dataset, and these algorithms can vary both in terms of how often they are able to find solutions to problem instances (success rate) and how good their solutions are when found (minimum yield). It is thus difficult to provide a single metric that can be used as a basis for determining an overall winner. For this reason we first perform pairwise comparisons of the algorithms using the following metrics: (i) $\mathcal{Y}_{A,B}$: the average percent minimum yield difference between A and B, relative to the minimum yield achieved by B, computed on instances for which both algorithms succeed. (ii) $\mathcal{S}_{A,B}$: the percentage of instances for which A succeeds and B fails, minus the percentage of instances where B succeeds and A fails; and For both measures, a positive value means an advantage of A over B. Since no single simple vector packing heuristic can perform as well as the aggregate METAVP and METAHVP algorithms in terms of either failure rate or achieved minimum yield, for now we only consider the RRND, RRNZ, METAGREEDY, METAVP, and METAHVP algorithms.

Pairwise comparisons of the major heuristics under consideration are given by Table 1. Perhaps surprisingly, RRND can lead to much better performance than RRNZ, but it has an extremely low success rate relative to the other algorithms, so is dropped from further consideration. METAGREEDY widely outperforms RRNZ both in terms of success rate and minimum yield, but is in turn beaten my METAVP. The METAVP algorithm is itself outperformed by METAHVP.

While not large, the advantage of METAHVP over METAVP becomes more pronounced as the number of services per node increases. METAVP solves 15,376 of the 36,000 100-service instances, while METAHVP

only solves 1 additional instance (METAVP does not solve any instances not solved by METAHVP). METAHVP achieves yield values more than 0.002 greater than METAVP on 311 instances, while METAVP beats METAHVP by the same margin on only 1. The average yield values on instances solved by both algorithms are close together: 0.504 for METAHVP and 0.503 for METAVP. METAVP solves 30,596 of the 36,900 250-service instances, while METAHVP solves an additional 162 instances (METAVP does not solve any instances not solved by METAHVP). METAHVP achieves yield values at least 0.002 higher on 17,256 of those instances, while METAVP beats METAHVP by at least the same amount on 86 instances. The average yield values on instances solved by both algorithms is larger (0.820 for METAHVP and 0.803 for METAVP). Both algorithms solve 36,844 of the 36,900 500-task instances (neither algorithm solves any instances not solved by the other). METAHVP achieves yield values that are at least 0.002 higher on 28,680 of these instances, compared to the 44 where METAVP achieves a higher yield value by the same margin. METAHVP achieves an average yield of 0.897, compare to 0.850 for METAVP. While there is no commonly accepted target for virtual machines per node, 8 is probably reasonable in current production platforms, which corresponds to our 500-service instances. Higher levels of consolidation are likely to be expected, thus increasing the advantage of METAHVP over METAVP.

To gain more insight into our results, Figure 2 shows results for instances with 500 services and a memory slack of 0.3. Each data point (x, y) corresponds to one problem instance and one of RRNZ, METAGREEDY, or METAVP. x is the coefficient of variance of the CPU and memory capacities of the nodes in the platform, and y is the difference between the achieved yield and that achieved by METAHVP. Points on the $x = 0$ axis are for perfectly homogeneous platforms. Points below the $y = 0$ axis correspond to instances in which METAHVP does not achieve the best yield. Figure 3 shows the same information for problem instances where the CPU was held homogeneous (i.e., all servers have a CPU resource of 0.5), while Figure 4 does the same for memory. As expected based on our previous set of results, METAGREEDY never outperforms METAHVP, and is generally worse than METAVP. RRNZ is never a contender and leads to markedly poorer performance than the other algorithms on most instances, without ever outperforming METAHVP. METAHVP is the best algorithm in all three figures. The interesting observation is that while METAVP performs close to METAHVP over a wide range of problem instances, its performance relative to METAHVP decreases as the platform becomes more heterogeneous.

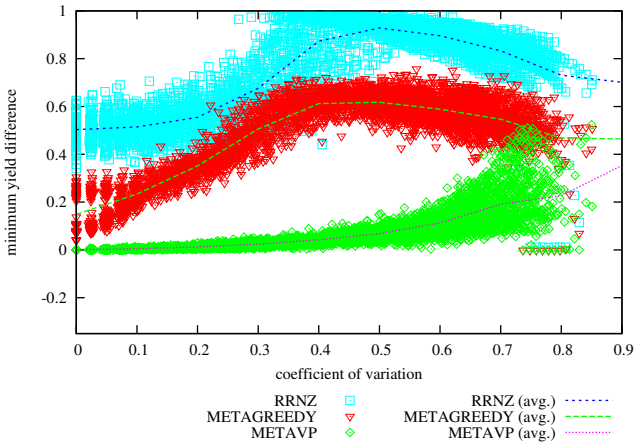

Figure 2: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 500 services, memory slack = 0.3

Table 2 shows algorithm run times averaged over all instances. Expectedly, the RRNZ algorithm has high run times because it solves a linear program (albeit rational). The main observation is that METAHVP leads to higher run time than METAVP by a factor 3 on the average, requiring more than 6 seconds to solve problem instances with 500 tasks. Recall that these results are for "only" 64 nodes. Given that production cloud platforms comprise orders of magnitude more nodes, and thus hosts orders of magnitude more services, using METAHVP in production systems is likely impractical. For example, METAHVP requires an average of 134.52 seconds when run on a 2.27 Ghz Intel Xeon processor with 512 hosts and 2000 services.

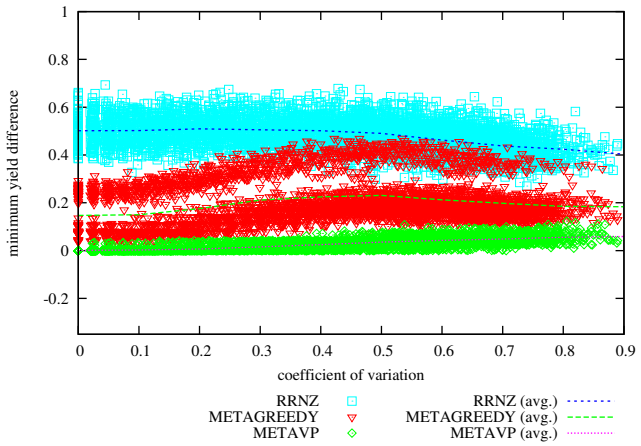

Figure 3: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 500 services, memory slack = 0.3, CPU held homogeneous

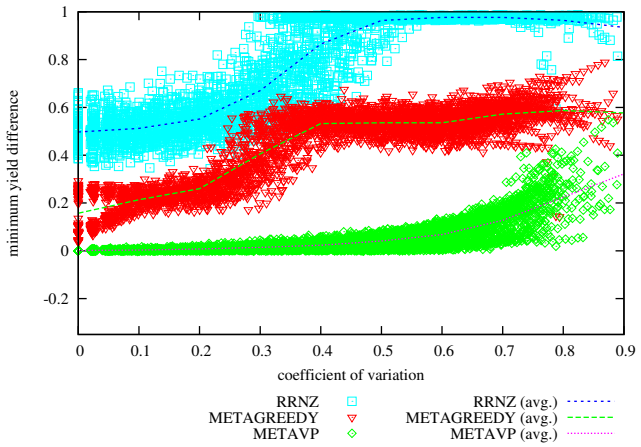

Figure 4: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 500 services, memory slack = 0.3, Memory held homogeneous

Table 2: Algorithm run times in seconds when run on an Intel Xeon 2.27Ghz processor, averaged over all instances.

Algorithm	100 tasks	250 tasks	500 tasks
RRNZ	4.855	45.782	270.245
METAGREEDY	0.014	0.061	0.154
METAVP	0.142	0.564	1.715
METAHVP	0.514	1.943	6.432

5.1 METAHVPLIGHT

Given the high run time of METAHVP, in this section we design a version of METAHVP called METAHVP-LIGHT that includes only a subset of the HVP algorithms.

An exploration of our results shows that no single (small) group of algorithms emerges as a clear winner, and in fact every single algorithm is best on some instances. To filter out the worst performing

algorithms, we sorted the basic HVP algorithms first by success rate, then by average achieved minimum yield. Looking at the top 50 algorithms for each dataset, some trends are clear. 1) The Best-Fit, First-Fit, and Permutation-Pack approaches all perform well when paired with appropriate item and (except in the case of Best-Fit) bin sorting strategies, and each outperforms the others on a small subset of the instances. 2) The descending by MAX, SUM, and MAXDIFFERENCE item sorting strategies are those used by most of the high-performing algorithms, and MAXRATIO can also be a good item sorting strategy for some algorithms. 3) The ascending by LEX (considering CPU needs before memory), MAX, and SUM bin sorting strategies were unsurprisingly among the most successful, but also present were descending by MAX, MAXDIFFERENCE, and MAXRATIO, and the option to not sort bins (NONE), which was somewhat of a surprise.

From these observations we built a simple METAHVPLIGHT algorithm that uses essentially the same procedure as METAHVP, but only considers the descending by MAX, SUM, MAXDIFFERENCE and MAXRATIO item sorting strategies and the ascending by LEX, MAX, SUM, and descending by MAX, MAXDIFFERENCE and MAXRATIO bin sorting strategies, as well as the option to not sort bins. This reduces the number of heterogeneous vector packing heuristics considered at each step of yield optimization from 253 to $4 + 2 \times 4 \times 7 = 60$.

We ran METAHVPLIGHT on all our problem instances and observed the following. Both METAHVP and METAHVPLIGHT successfully solve the same set of 100-service instances (15,377 out of 36,900) and achieve the same average minimum yield of 0.504. METAHVPLIGHT solves 30,737 of the 250-service instances, and METAHVP solves an addition 21 instances that METAHVPLIGHT does not solve. On the instances that both algorithms solve METAHVP achieves an average minimum yield value of 0.817 versus 0.814 for METAHVPLIGHT. Finally, both algorithms solve the exact same 36,844 (out of 36,900) 500-task instances. On these instances both algorithms achieve an average minimum yield of 0.897 .

These results show that METAHVPLIGHT expectedly achieves slightly lower solution quality on a few instances, but the overall performance is quite Importantly, its run time is drastically lower than that of METAHVP. For instance, on the 512-node and 2000-service instance reported in the previous section, METAHVPLIGHT runs in 15.25 seconds as compared to the 134.52 needed by METAHVP.

5.2 Results Conclusion

- Algorithms based on vector packing, known to work well in homogeneous settings, can be adapted to handle heterogeneity and outperform linear programming and greedy approaches.
- These algorithms do much better than other approach for tightly constrained and highly heterogeneous problem instances. No single algorithm, however, emerges as the best. This motivates a "brute-force" approach that runs many of the algorithms and selects the best solution among those produced.
- The brute-force approach leads to unreasonably high times to solution, but one can engineer a "light" algorithm that only runs a smaller subset of the base algorithms. This solution improves the runtime by nearly a factor 10 while leading to solutions of sensibly equivalent quality.

6 The Effects of Error in CPU needs estimates

An important consideration for any scheduling algorithm is how well the resource requirements and needs are known. Although some work has been done with regards to memory consumption [20], it is in general difficult if not impossible to know all of a service's resource requirements before it is run. One option is to rely on benchmark results for services that comprise the workload [21]. But any service that performs data-dependent computations or reacts to a user-dependent workload will necessarily have variations in moment-to-moment requirements each time it runs. CPU needs estimation in particular can be problematic, as the use of this resource tends to be "noisy" and/or "spikey" and prone to varying over time [22].

An interesting question is, given a set of erroneous CPU need estimates, how do errors affect the minimum yield achieved by our algorithms. In particular, how would our algorithms compare to a baseline algorithm that assumes no knowledge of CPU needs? In the total absence of knowledge the best policy is likely to distribute services as evenly as possible across the available nodes (as done in the "scheduling in

the dark" approach [23]) and then use a work-conserving scheduler that makes some effort to distribute available cycles fairly (for some definition of fairness) among processes. It is common for modern virtual machine CPU scheduling systems to offer a work-conserving mode in which processes are given access to the CPU in proportion to administrator-assigned weights. For instance, two CPU intensive competing virtual instances would be initially restricted to using at most 50% of the available CPU cycles, but if one of the instances reduced its CPU consumption, then the other would be allowed to use the unused portion of the CPU resource [6].

We propose an iterative algorithm for determining the CPU consumption of competing services when their needs are not known precisely by the scheduler. First, each service is allocated a portion of the node relative to its weight (e.g., if the weights of all currently active services sum up to 100, and a given service is assigned a weight of 30, then it will initially be able to use up to 30% of the CPU). Any service with actual needs that are less than or equal to its initial allocation is considered "satisfied" and any portions of the CPU that are left unused (because some services have needs strictly less than their initial allocation) are pooled together and redistributed to remaining unsatisfied services again by their weight. This process continues until either all of the services are satisfied or there is no more CPU available. We assume that CPU allocations cannot be smaller than some epsilon (0.0001 in our simulations) in order to avoid potentially infinite recursion. we consider three methods for allocating CPU (or other dynamic) resources to competing tasks once they are assigned to a particular node. The first, ALLOCCAPS, simply assigns limitations on CPU utilization based on the proportion required to maximize the minimum yield on the current node, given known estimates of CPU utilization. The second, ALLOCWEIGHTS, uses the values computed to maximize the minimum yield on the current node as weights, assuming the use of a work conserving scheduler. The third, EQUALWEIGHTS, assumes a work-conserving scheduler as well, but simply assigns equal weights to all competing services.

6.1 A Theoretical Result

It turns out that, for the EQUALWEIGHTS algorithm, one can quantify how far the obtained solution is from the optimal, as stated in the following theorem.

Theorem 1. *In the on-line minimum-yield maximization resource allocation problem for a single dimension,* EQUALWEIGHTS *is* $\frac{2J-1}{J^2}$ *competitive in the worst case, and there is an instance that achieves exactly this performance ratio.*

Proof. Let $\{n_j^a\}_{j=1}^J$ be a one-dimensional set of aggregate service needs (we do not consider elementary needs in this proof). By an abuse of notation, let us assume that the values in this set represent scalar values rather than one-dimensional vectors. As the scheduler is work-conserving, for any resource allocation where $\sum_{j=1}^J n_j^a \leq 1$, no service will have a yield of less than 1, therefore let us assume the opposite, that is, $\sum_{j=1}^J n_j^a > 1$. Since the EQUALWEIGHTS algorithm does not take needs into consideration when allocating resources to unsatisfied services, the services with the minimum yield will necessarily be those with the maximum need. Let us denote $\hat{n^a} = max\{n_j^a\}_{j=1}^J$, and let $\hat{j}$ be an arbitrary service with need $\hat{n^a}$. Clearly $n_{\hat{j}}^a = \hat{n^a} > \frac{1}{J}$.

Case 1: All of the services in the system have needs of at least $\frac{1}{J}$. The allocation to the service with need $\hat{n^a}$ is thus $\frac{1}{J}$ and the minimum yield achieved by EQUALWEIGHTS is no less than $\frac{1}{J\hat{n^a}}$. The yield achieved by the optimal algorithm is $\frac{1}{\sum_{j=1}^J n_j^a}$.

We compute the lower bound on the competitive ratio of EQUALWEIGHTS as follows:

$$\frac{\frac{1}{J\hat{n^a}}}{\frac{1}{\sum_{j=1}^J n_j^a}} = \frac{\sum_{j=1}^J n_j^a}{J\hat{n^a}} \geq \frac{\frac{J-1}{J}+\hat{n^a}}{J\hat{n^a}} \geq \frac{\frac{J-1}{J}+1}{J} = \frac{2J-1}{J^2}$$

Case 2: There is at least one service (clearly not $\hat{j}$) with a need of less than $\frac{1}{J}$. Denote the fraction of the resource consumed by services other than $\hat{j}$ as S. As $\hat{j}$ will consume at least $\frac{1}{J}$ of the resource, $S \leq \frac{J-1}{J}$. Since the scheduler is work conserving and the sum of the resource needs is greater than 1, it must be the case that $1 - S \leq \hat{n^a}$. The allocation to the service $\hat{j}$ is thus $1 - S$ and the minimum yield achieved by EQUALWEIGHTS is $\frac{1-S}{\hat{n^a}}$. The lower bound on the ratio to the optimal is is thus:

$$\frac{\frac{1-S}{\hat{n^a}}}{\frac{1}{\sum_{j=1}^{J} n_j^a}} = \frac{(\hat{n^a}+S)(1-S)}{\hat{n^a}} \geq (1+S)(1-S) = 1-S^2 \geq 1-(\frac{J-1}{J})^2 = \frac{2J-1}{J^2}$$

A problem instance that achieves this ratio is $n_1^a = 1$, $n_j^a = \frac{1}{J}$ for $j = 2, ..., J$.

□

6.2 Error Experiments

Starting from the same data set as in Section 5, we selected those problems with slack values of 0.2, 0.4, 0.6, and 0.8 and COV values of 0.0, 0.5 and 1.0. We then specified maximum error values from 0.0 to 0.3 in increments of 0.02. For each of these problems and maximum error values we perturbed the CPU needs by selecting values between the negative and positive maximum value from a uniform random distribution and adding this error to the true total CPU needs (to a minimum of 0.001). Elementary CPU needs were perturbed so as to maintain the same proportion with the aggregate needs are before the error was applied. It should be noted that an average node in our simulations has an aggregate CPU power of 0.5. Services in the 100-service case have a mean CPU need of 0.317, while those in the 250-service case have a mean CPU need of 0.127, and those in the 500-service case have mean CPU need of 0.063. Thus, errors can be large in proportion to average service needs.

These modified values were then used by the METAHVP algorithm to assign nodes and CPU allocations to the services. After the services were mapped to nodes we assessed performance based on the following criteria: 1) the *expected* yield for each, assuming the estimates had been completely correct; 2) the minimum actual yield, if the output of METAHVP was used used to allocate CPU using the ALLOCCAPS algorithm; 3) the minimum actual yield, if the output of METAHVP was used as input to the ALLOCWEIGHTS algorithm; and 4) the minimum actual yield, assuming that after services are assigned to nodes CPU allocations were determined using the EQUALWEIGHTS algorithm.

Unsurprisingly, using the ALLOCCAPS algorithm is a losing proposition in the presence of errors, and we found that it consistently leads to poor performance (worse than the zero-knowledge case) when the error exceeded approximately 30% of the average service need (i.e., 0.08 in the 100-service case, 0.04 in the 250-service case, 0.02 in the 500-service case). Initially, simply switching to the ALLOCWEIGHTS or EQUALWEIGHTS algorithm can help for some instances, but the average-case performance of ALLOCWEIGHTS closely tracks that of ALLOCCAPS, while EQUALWEIGHTS gives initially worse results, but can often remains better than zero-knowledge until the maximum error exceeds the average service need.

A likely reason for poor performance under these circumstances is underestimating the needs of relatively small services. To combat this problem we propose rounding up the estimate of each CPU need to a minimum threshold value. Estimates larger than this threshold are not affected. This allows the algorithm to use some of the information about relative service sizes to make placement decisions, while effectively holding some of the CPU resource in reserve in order to avoid penalizing those services that are the most vulnerable. This strategy necessarily requires the use of ALLOCWEIGHTS or EQUALWEIGHTS rather than ALLOCCAPS to be successful, as it can result in allocating small services more CPU than they need.

Figures 5, 6 and 7 plot the average minimum achieved yield vs. the maximum estimation error for 100-service, 250-service, and 500-services instances, respectively, on moderately heterogeneous platforms (specified coefficient of variation = 0.5). Our results are broadly consistent across all datasets, and we point the interested reader to http://perso.ens-lyon.fr/mark.stillwell/ipdps12/, where the rest of the graphs, as well as problem sets and raw results files, are available.

The main conclusion from these results is that our proposed strategy is effective. Algorithms that must contend with error cannot perform as well as the perfect-knowledge algorithm. But the key observation is that our algorithms perform better than the zero-knowledge algorithm over a wide error range. As the minimum threshold used by the algorithms increases, the sensitivity of the algorithm to increasing error diminishes. That is, the curves becomes flatter, but the performance of the algorithm becomes worse on the average, decreasing toward the zero knowledge case.

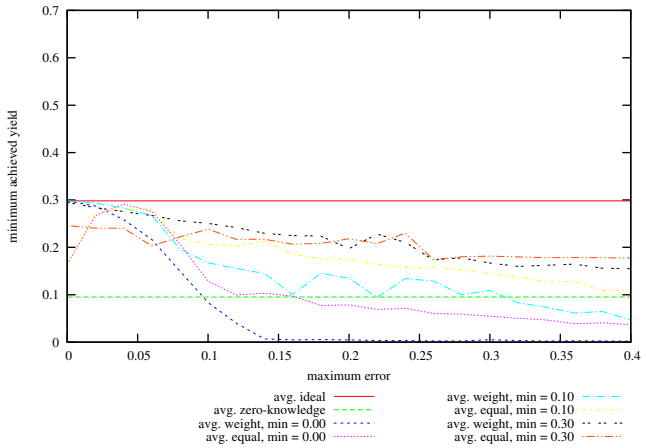

Figure 5: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 100 services, memory slack = 0.4, coefficient of variation = 0.5; values given are averages over successful instances.

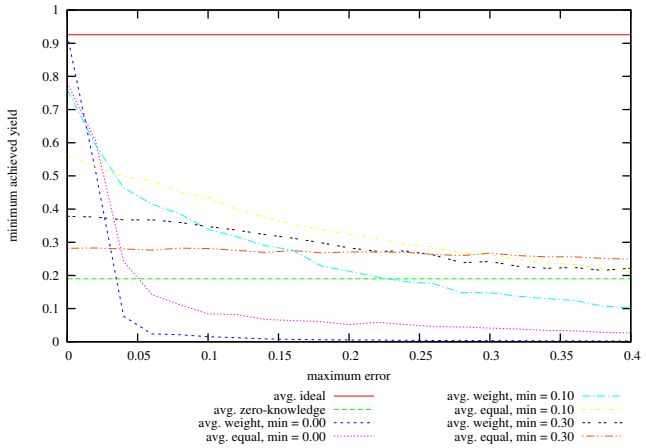

Figure 6: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.4, coefficient of variation = 0.5; values given are averages over successful instances.

7 Related Work

Resource allocation for distributed cluster platforms is currently an active area of research, with application placement [24], load balancing [1, 25], and avoiding QoS constraint violations [26, 27] being primary areas of concern. Some authors have also chosen to focus on optimizing fairness or other utility metrics [28]. Most of this work focuses on homogeneous cluster platforms, i.e., platforms where nodes have identical available resources. Two major research areas that consider heterogeneity are embedded systems and volunteer computing.

In the embeded systems arena, the authors of [29] also employ heterogeneous vector packing algorithms for scheduling, with many of the underlying heuristics being similar to what we propose in this paper. This work is not directly relevant as their solutions are tightly constrained by the data bus, and they consider a narrower range of algorithms. They do show that bin packing by decreasing item size as measured by maximum vector dimension is important.

Most of the existing theoretical research on multi-capacity bin packing has focused on the off-line version of the problem with homogeneous bins. Epstein considers allowing a limited number of different

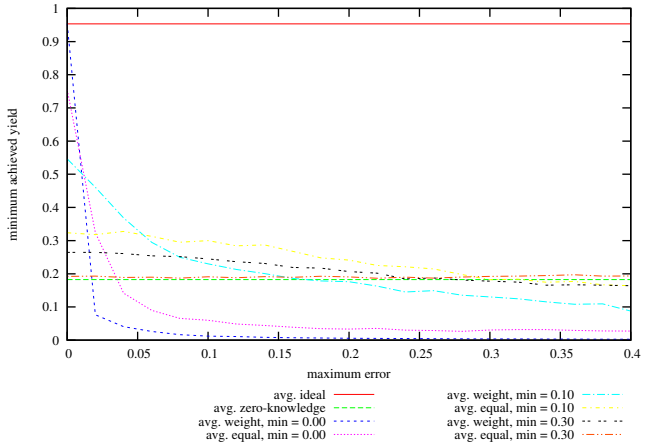

Figure 7: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 500 services, memory slack = 0.4, coefficient of variation = 0.5; values given are averages over successful instances.

classes of bins, though with the objective [30] of minimizing the total cost of the bins, which is a slightly different problem. There are obvious connections to multiple-choice multi-dimensional vector bin packing as explored by Patt-Shamir and Rawitz [31], but our version of the problem is more constrained since we cannot choose the types of the available bins, and must also contend with elementary requirements and needs. The authors of [32] consider a dynamic programming approach for when tasks fall into one of a finite number of types.

As stated previously, the problem of properly modeling resource needs is a challenging one, and it becomes even more challenging with the introduction of error. To date we are not aware of other studies that systematically consider the issues of errors in CPU needs estimates.

8 Conclusion

In this paper we have considered the problem of off-line resource allocation on heterogeneous platforms, with the explicit goal of maximizing the minimum task performance. We have defined the problem to allow for multiple resource dimensions and for discrete resource elements, such as CPUs, that cannot be pooled together arbitrarily. We have provided a MILP that can be solved in a reasonable amount of time for small instances, and heuristic algorithms that perform well in practice for solving larger problem instances. We have also considered the problem of inaccurate CPU needs estimates, and developed an effective strategy for minimizing the worst-case penalty imposed while still taking advantage of some of the improvements in resource allocation offered by aggressive heuristic algorithms.

One next step in this research is to implement our METAHVPLIGHT algorithm in combination with our strategy to mitigate estimate errors as part of the resource management component of an open cloud computing infrastructure. This infrastructure, once deployed on a testbed, will allow for an evaluation of our results with both synthetic and production workloads. One interesting problem will be to develop a method for determining and adapting the threshold used to mitigate estimate errors.

Acknowledgment

Simulations presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] M. Andreolini, S. Casolari, M. Colajanni, and M. Messori, "Dynamic load management of virtual machines in a cloud architectures," in *CLOUDCOMP*, 2009.

[2] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM TAAS*, vol. 3, no. 1, pp. 1–39, 2008.

[3] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource allocation algorithms for virtualized service hosting platforms," *JPDC*, vol. 70, no. 9, pp. 962–974, 2010.

[4] S. Banen, A. I. Bucur, and D. H. Epema, "A measurement-based simulation study of processor co-allocation in multicluster systems," in *JSSPP*, 2003, pp. 184–204.

[5] A. Buttari, J. Kurzak, and J. Dongarra, "Limitations of the PlayStation 3 for high performance cluster computing," U Tenn., Knoxville ICL, Tech. Rep. UT-CS-07-597, Apr. 2007.

[6] D. Gupta, L. Cherkasova, and A. M. Vahdat, "Comparison of the three CPU schedulers in Xen," *ACM SIGMETRICS Perf. Eval. Rev.*, vol. 35, no. 2, pp. 42–51, 2007.

[7] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai, "Translating service level objectives to lower level policies for multi-tier services," *Cluster Computing*, vol. 11, no. 3, pp. 299–311, 2008.

[8] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial, "No justified complaints: On fair sharing of multiple resources," arXiv:1106.2673v1, Jun. 2011.

[9] Y. Becerra, D. Carrera, and E. Ayguadé, "Batch job profiling and adaptive profile enforcement for virtualized environments," in *PDP*, 2009, pp. 414–418.

[10] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan, "Flow and stretch metrics for scheduling continuous job streams," in *ACM-SIAM SODA*, 1998, pp. 270–279.

[11] A. Legrand, A. Su, and F. Vivien, "Minimizing the stretch when scheduling flows of divisible requests," *J. Scheduling*, vol. 11, no. 5, pp. 381–404, 2008.

[12] G. J. Wöginger, "There is no asymptotic PTAS for two-dimensional vector packing," *IPL*, vol. 64, no. 6, pp. 293–297, 1997.

[13] "GLPK." [Online]. Available: http://www.gnu.org/s/glpk/

[14] "CPLEX." [Online]. Available: http://www.ilog.com/products/cplex/

[15] L. Marchal, Y. Yang, H. Casanova, and Y. Robert, "Steady-state scheduling of multiple divisible load applications on wide-area distributed computing platforms," *Intl. J. HPC Apps.*, vol. 20, no. 3, pp. 365–381, 2006.

[16] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "An application of bin-packing to multiprocessor scheduling," *SIAM J. Computing*, vol. 7, pp. 1–17, 1978.

[17] J. Csirik, J. B. G. Frenk, M. Labbe, and S. Zhang, "On multidimensional vector bin packing," *Acta Cybernetica*, vol. 9, no. 4, pp. 361–369, 1990.

[18] W. J. Leinberger, G. Karypis, and V. Kumar, "Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints," in *ICPP*, 1999, pp. 404–412.

[19] "Google cluster data archive." [Online]. Available: http://code.google.com/p/googleclusterdata/

[20] A. Batat and D. G. Feitelson, "Gang scheduling with memory considerations," in *IPDPS*, 2000, pp. 109–114.

[21] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in shared hosting platforms," *ACM SIGOPS OS Rev.*, vol. 36, pp. 239–254, 2002.

[22] B. Abrahao and A. Zhang, "Characterizing application workloads on CPU utilization for utility computing," HP Labs, Tech. Rep. HPL-2004-157, Sep. 2004.

[23] J. Edmonds, "Scheduling in the dark," in *STOC*, 1999, pp. 179–188.

[24] J. Rolia, A. Andrzejak, and M. Arlitt, "Automating enterprise application placement in resource utilities," in *DSOM*, ser. LNCS, 2003, vol. 2867, pp. 118–129.

[25] H. Nguyen Van, F. Dang Tran, and J.-M. Menaud, "Autonomic virtual resource management for service hosting platforms," in *ICSE-CLOUD*, 2009.

[26] M. Hoyer, Schröder, and W. Nebel, "Statistical static capacity management in virtualized data centers supporting fine grained QoS specification," in *E-ENERGY*, 2010, pp. 51–60.

[27] V. Petrucci, O. Loques, and D. Mossé, "A dynamic optimization model for power and performance management of virtualized clusters," in *E-ENERGY*, 2010, pp. 225–233.

[28] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguadé, "Utility-based placement of dynamic web applications with fairness goals," in *NOMS*, 2008, pp. 9–16.

[29] J. E. Beck and D. P. Siewiorek, "Modeling multicomputer task allocation as a vector packing problem," in *ISSS*, 1996, pp. 115–120.

[30] L. Epstein, "On variable-sized vector packing," *Acta Cybernetica*, vol. 16, pp. 47–56, 2003.

[31] B. Patt-Shamir and D. Rawitz, "Vector bin packing with multiple-choice," arXiv:0910.5599, 2009.

[32] S. Baruah and N. Fisher, "A dynamic programming approach to task partitioning upon memory-constrained multiprocessors," in *RTCSA*, 2004.

[33] *E-ENERGY*, 2010.

Additional Graphs

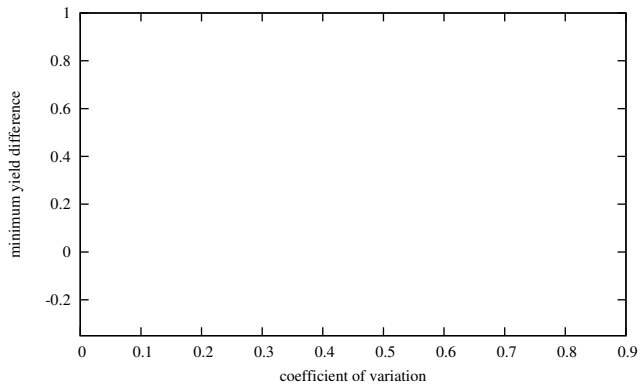

Figure 8: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 100 services, memory slack = 0.1

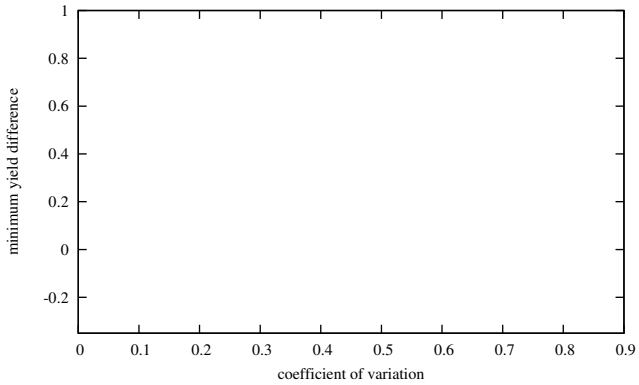

Figure 9: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 100 services, memory slack = 0.2

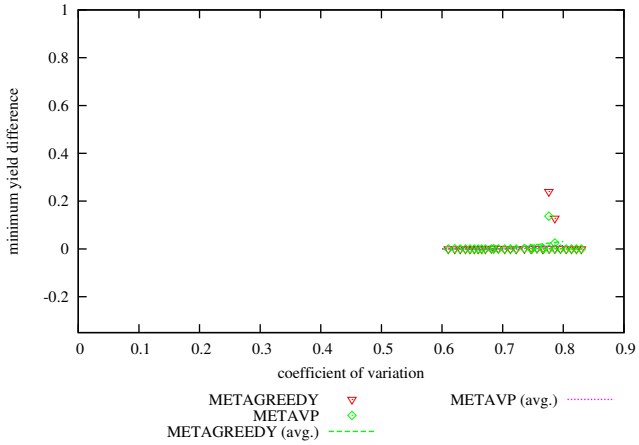

Figure 10: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 100 services, memory slack = 0.3

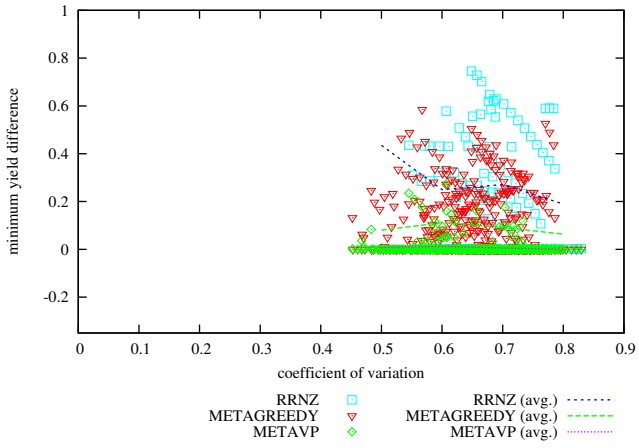

Figure 11: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 100 services, memory slack = 0.4

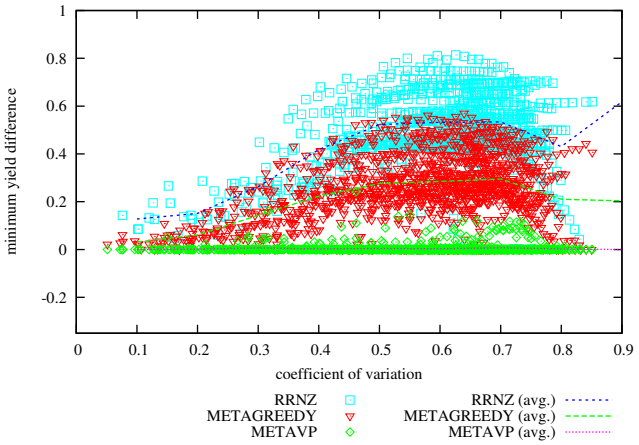

Figure 12: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 100 services, memory slack = 0.5

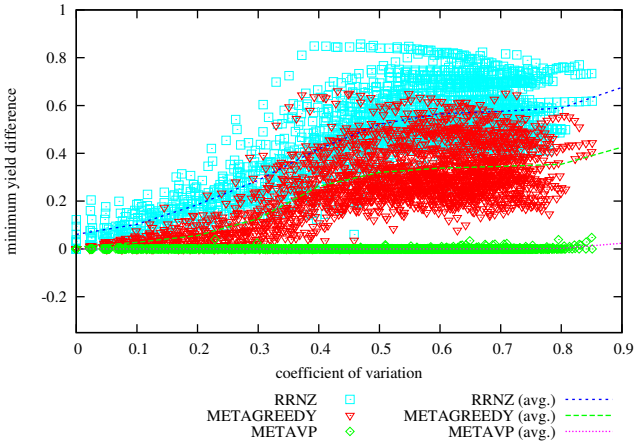

Figure 13: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 100 services, memory slack = 0.6

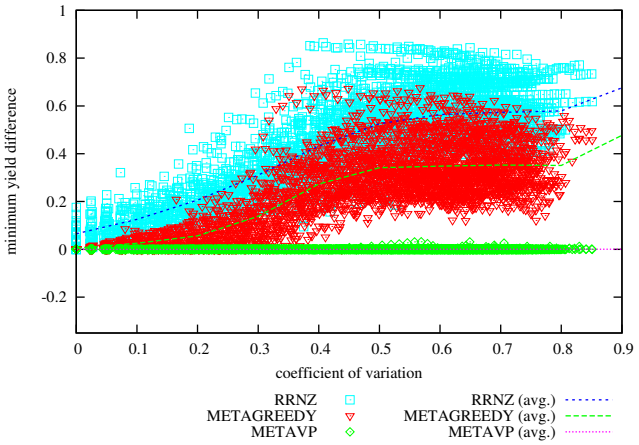

Figure 14: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 100 services, memory slack = 0.7

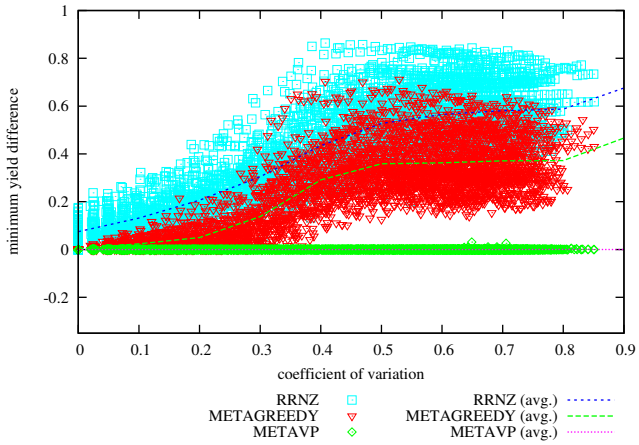

Figure 15: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 100 services, memory slack = 0.8

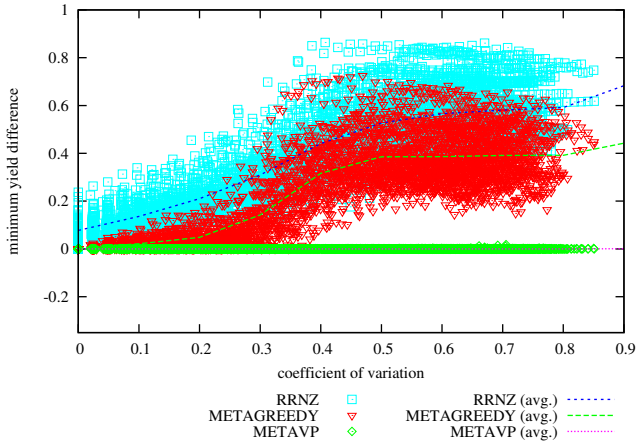

Figure 16: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 100 services, memory slack = 0.9

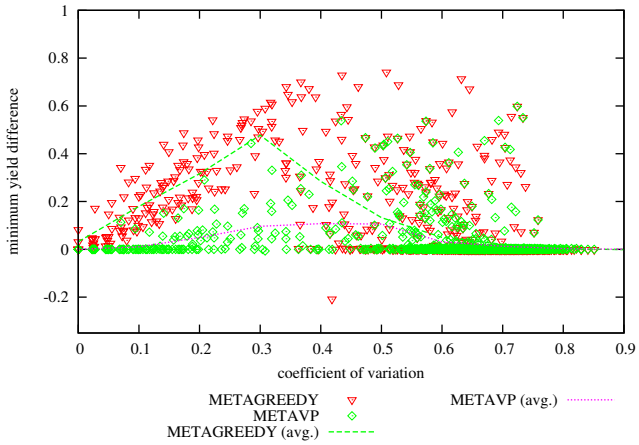

Figure 17: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 250 services, memory slack = 0.1

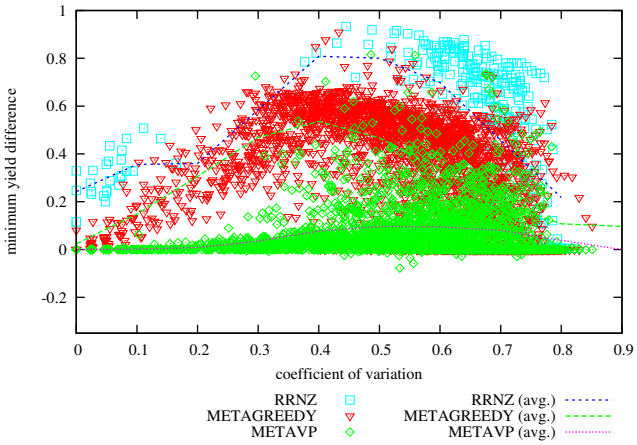

Figure 18: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 250 services, memory slack = 0.2

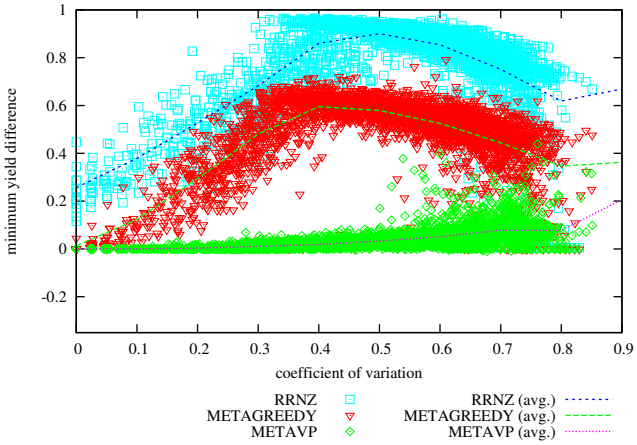

Figure 19: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 250 services, memory slack = 0.3

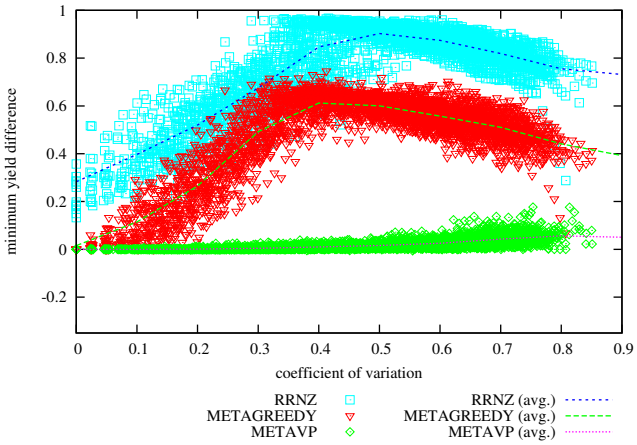

Figure 20: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 250 services, memory slack = 0.4

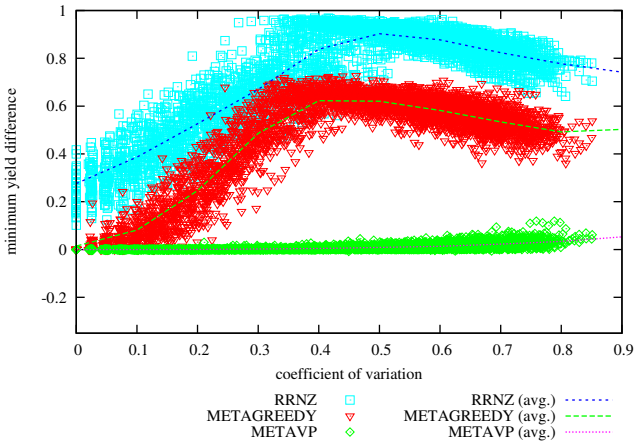

Figure 21: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 250 services, memory slack = 0.5

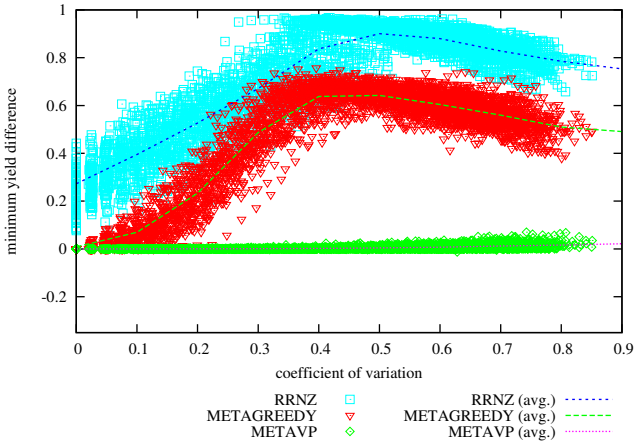

Figure 22: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 250 services, memory slack = 0.6

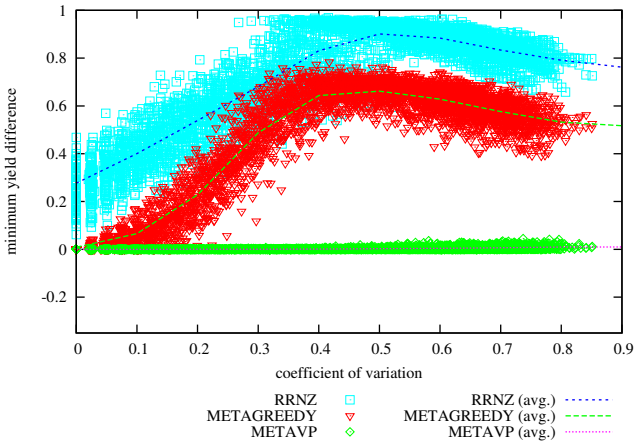

Figure 23: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 250 services, memory slack = 0.7

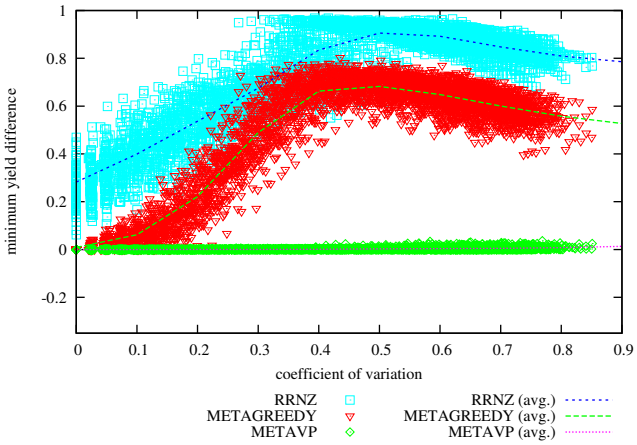

Figure 24: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 250 services, memory slack = 0.8

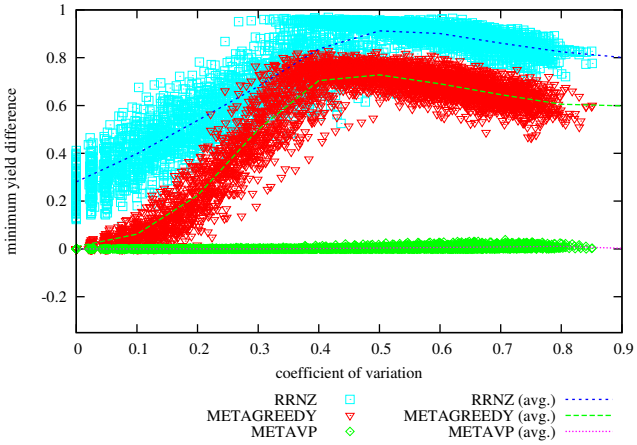

Figure 25: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 250 services, memory slack = 0.9

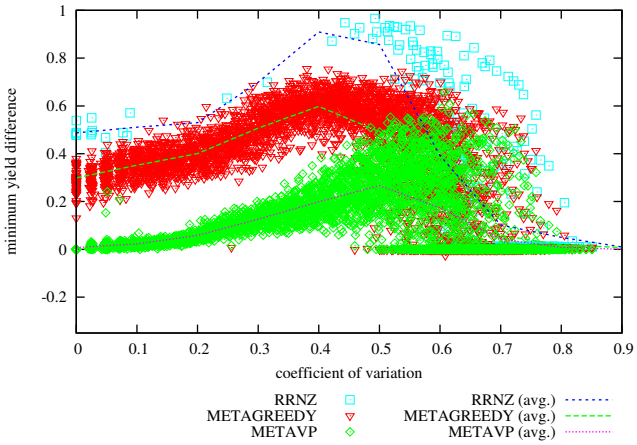

Figure 26: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 500 services, memory slack = 0.1

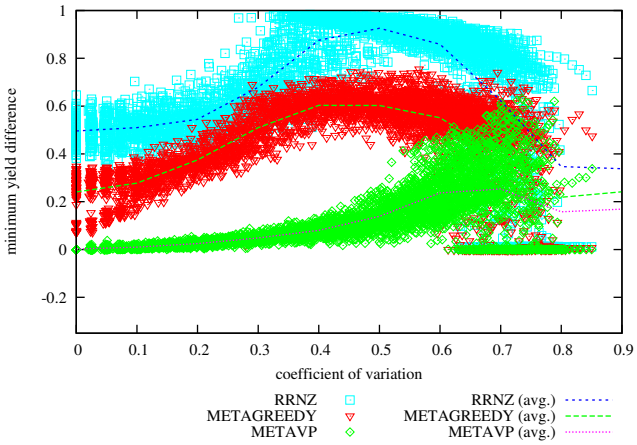

Figure 27: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 500 services, memory slack = 0.2

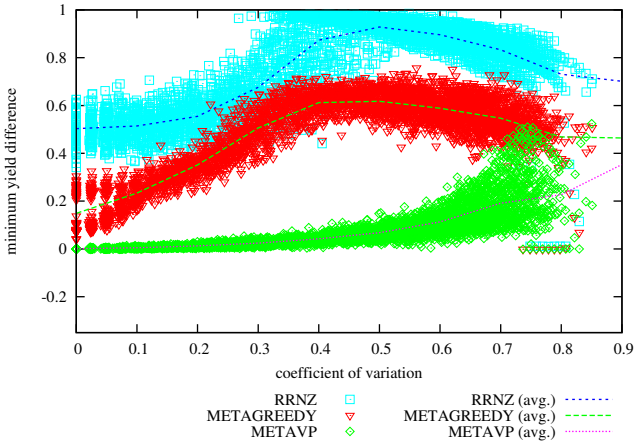

Figure 28: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 500 services, memory slack = 0.3

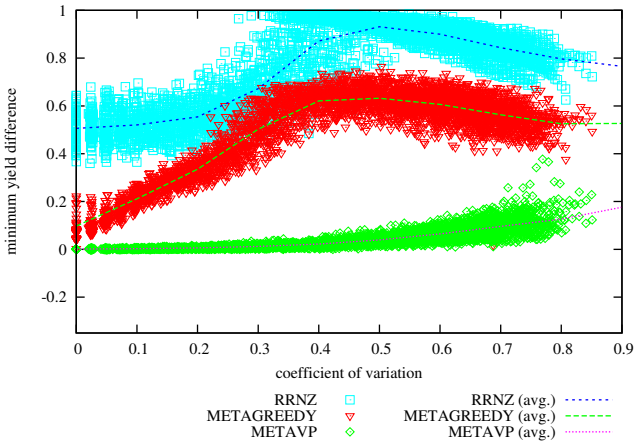

Figure 29: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 500 services, memory slack = 0.4

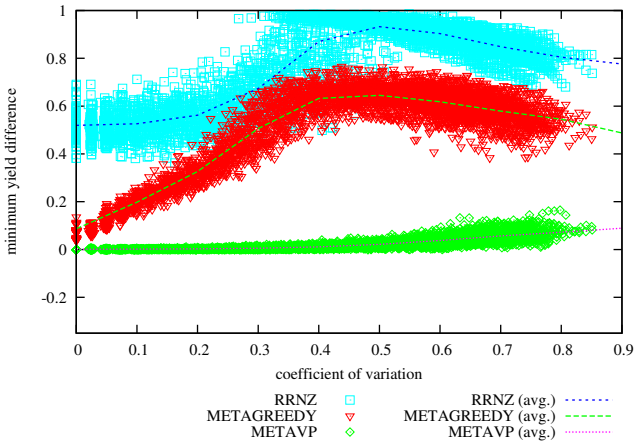

Figure 30: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 500 services, memory slack = 0.5

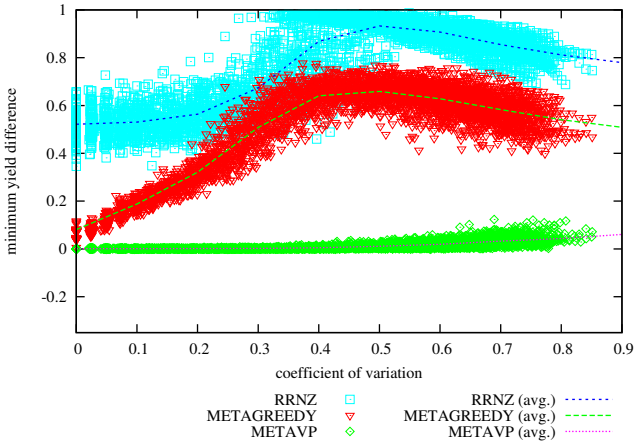

Figure 31: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 500 services, memory slack = 0.6

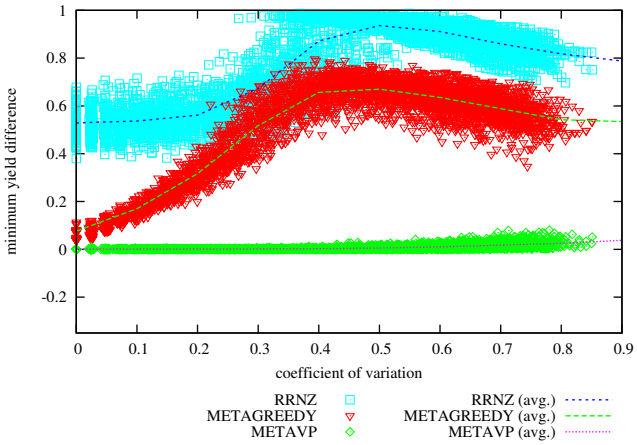

Figure 32: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 500 services, memory slack = 0.7

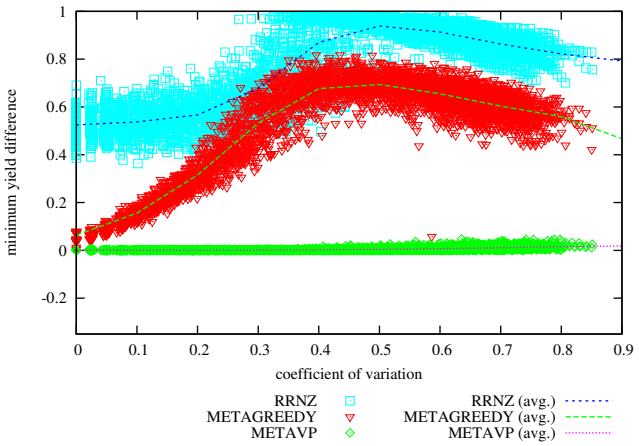

Figure 33: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 500 services, memory slack = 0.8

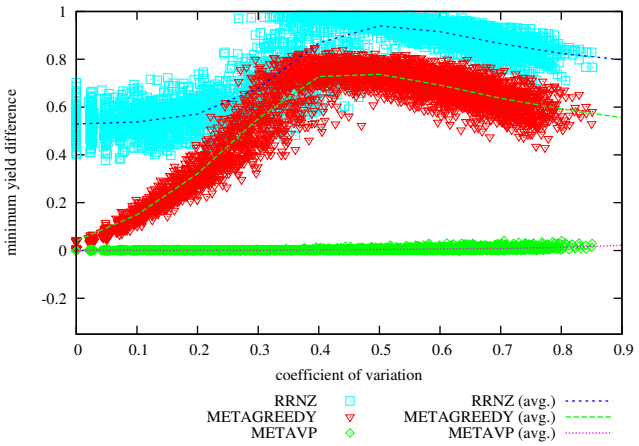

Figure 34: Difference in Achieved Minimum Yield from METAHVP vs Coefficient of Variation on problem instances with 64 hosts, 500 services, memory slack = 0.9

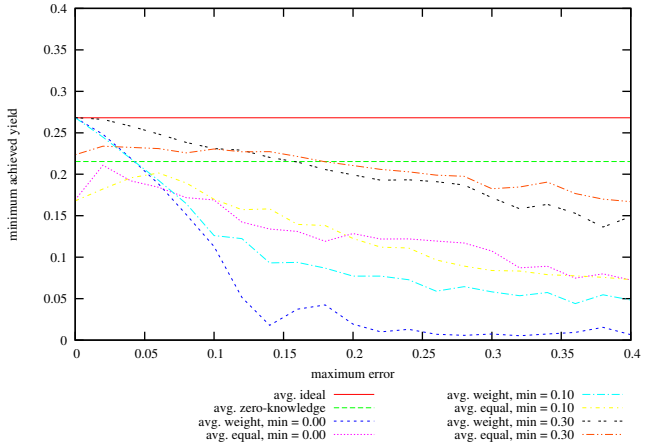

Figure 35: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 100 services, memory slack = 0.6, coefficient of variation = 0.0; values given are averages over successful instances.

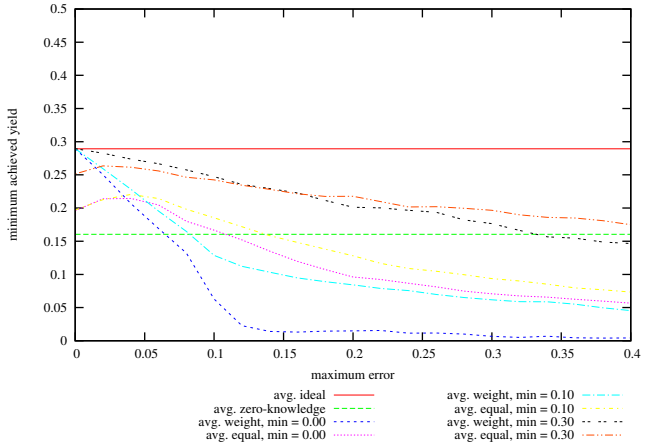

Figure 36: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 100 services, memory slack = 0.8, coefficient of variation = 0.0; values given are averages over successful instances.

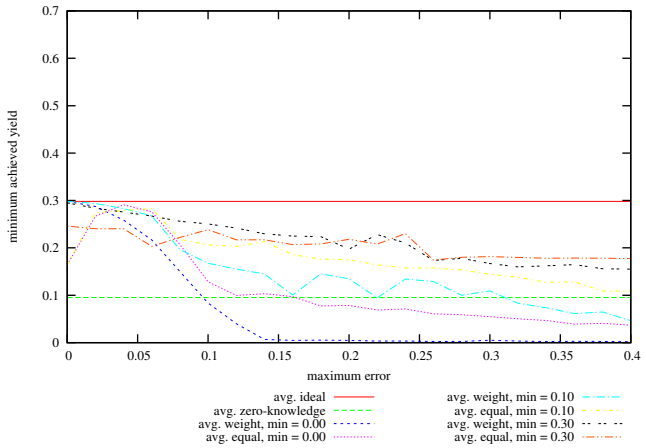

Figure 37: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 100 services, memory slack = 0.4, coefficient of variation = 0.5; values given are averages over successful instances.

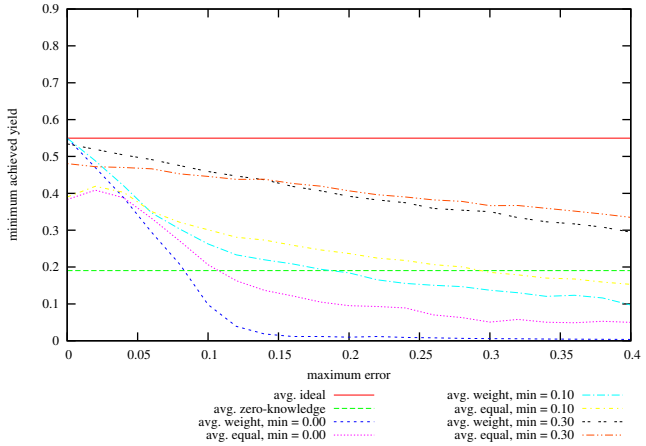

Figure 38: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 100 services, memory slack = 0.6, coefficient of variation = 0.5; values given are averages over successful instances.

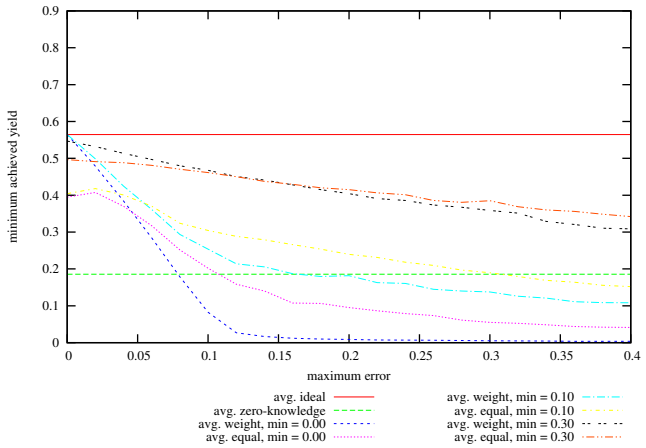

Figure 39: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 100 services, memory slack = 0.8, coefficient of variation = 0.5; values given are averages over successful instances.

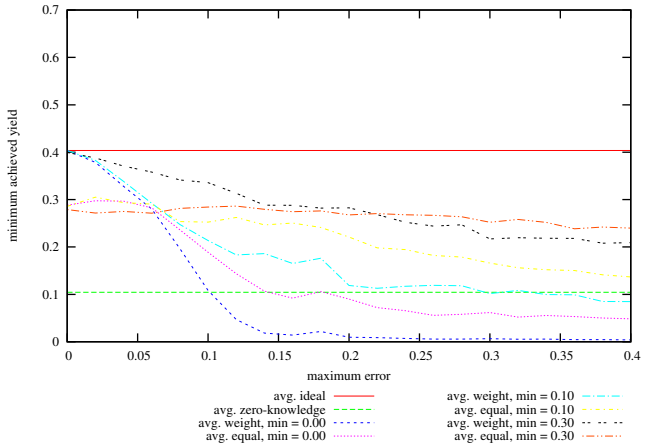

Figure 40: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 100 services, memory slack = 0.4, coefficient of variation = 1.0; values given are averages over successful instances.

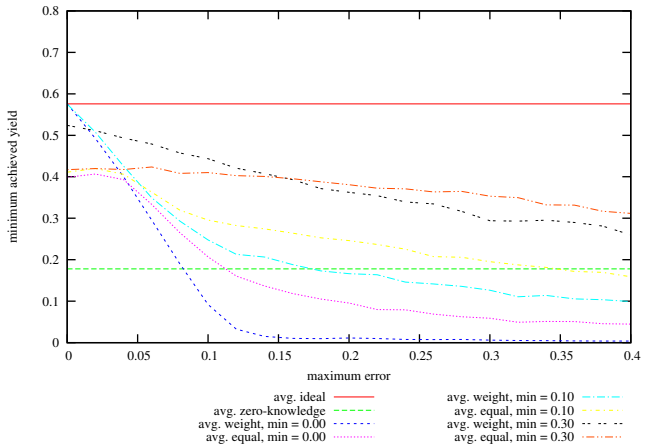

Figure 41: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 100 services, memory slack = 0.6, coefficient of variation = 1.0; values given are averages over successful instances.

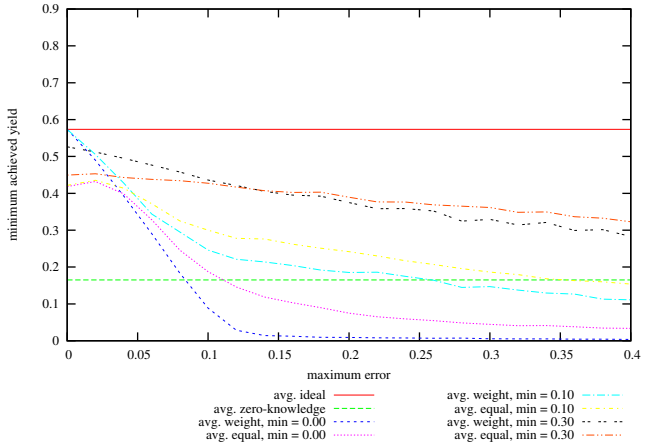

Figure 42: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 100 services, memory slack = 0.8, coefficient of variation = 1.0; values given are averages over successful instances.

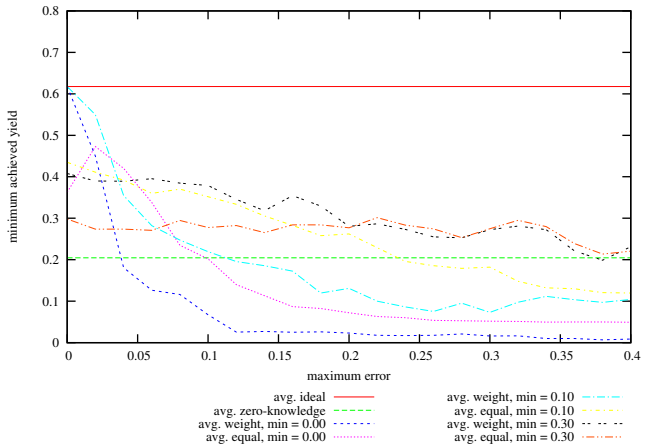

Figure 43: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.2, coefficient of variation = 0.0; values given are averages over successful instances.

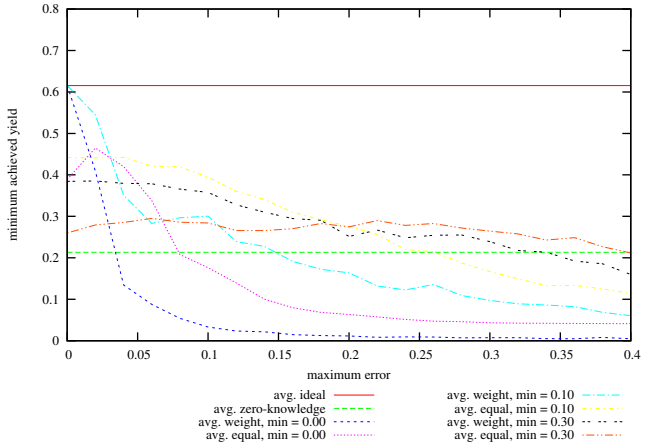

Figure 44: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.4, coefficient of variation = 0.0; values given are averages over successful instances.

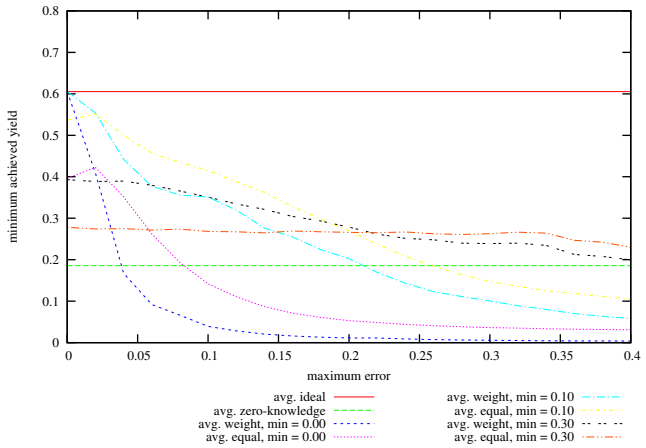

Figure 45: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.6, coefficient of variation = 0.0; values given are averages over successful instances.

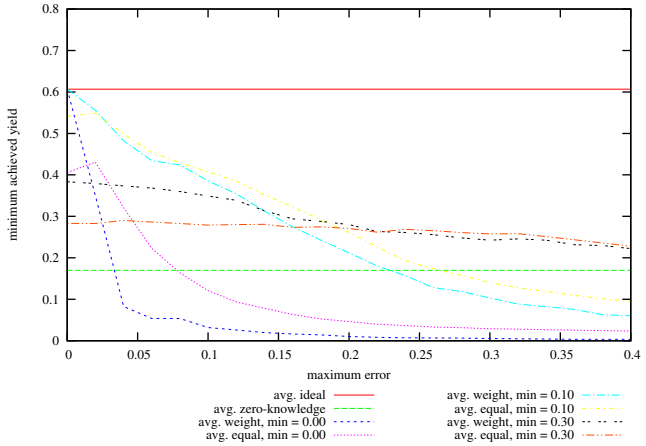

Figure 46: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.8, coefficient of variation = 0.0; values given are averages over successful instances.

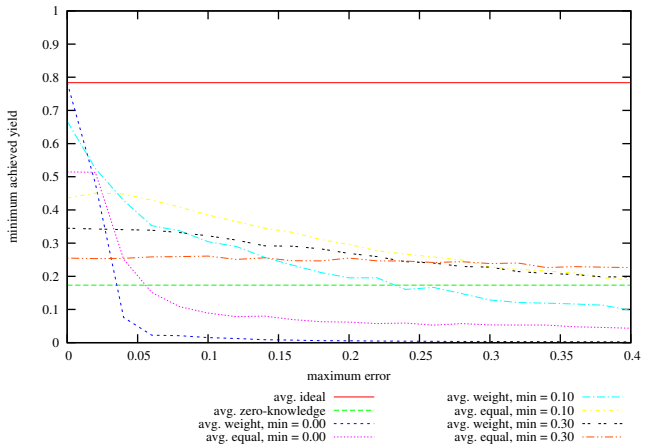

Figure 47: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.2, coefficient of variation = 0.5; values given are averages over successful instances.

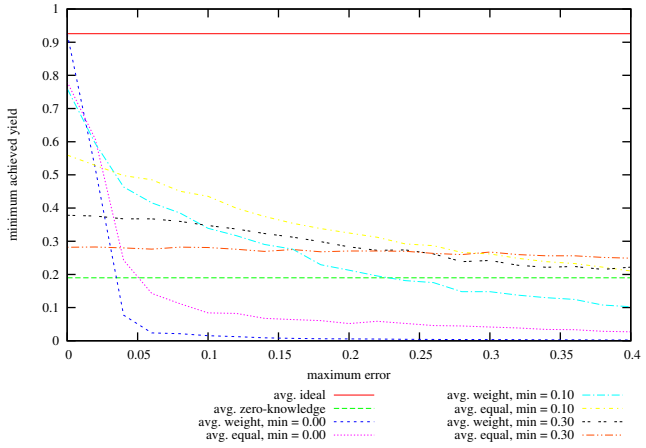

Figure 48: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.4, coefficient of variation = 0.5; values given are averages over successful instances.

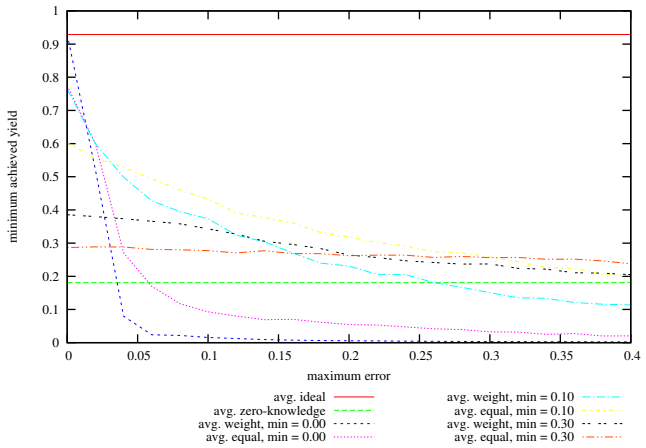

Figure 49: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.6, coefficient of variation = 0.5; values given are averages over successful instances.

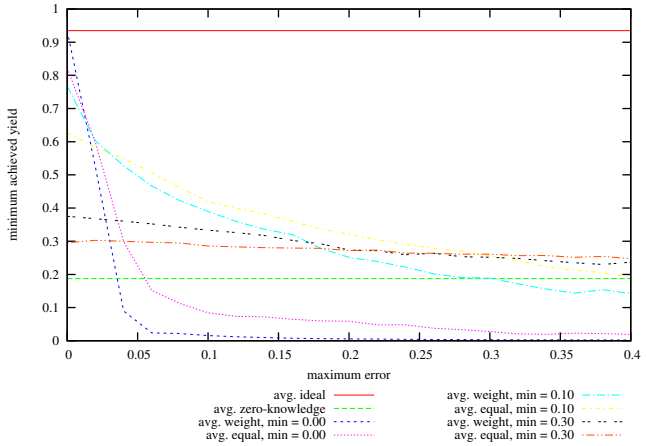

Figure 50: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.8, coefficient of variation = 0.5; values given are averages over successful instances.

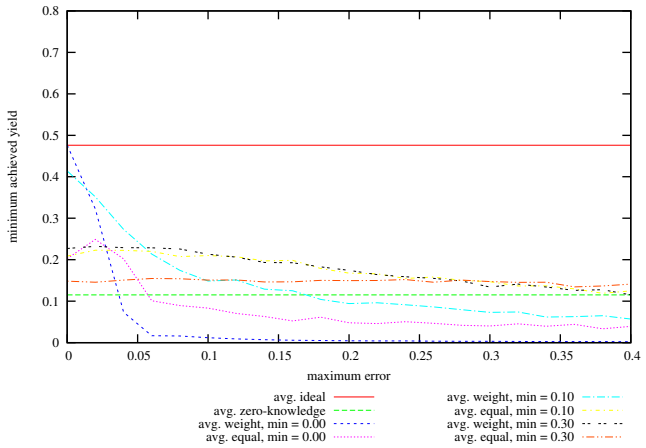

Figure 51: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.2, coefficient of variation = 1.0; values given are averages over successful instances.

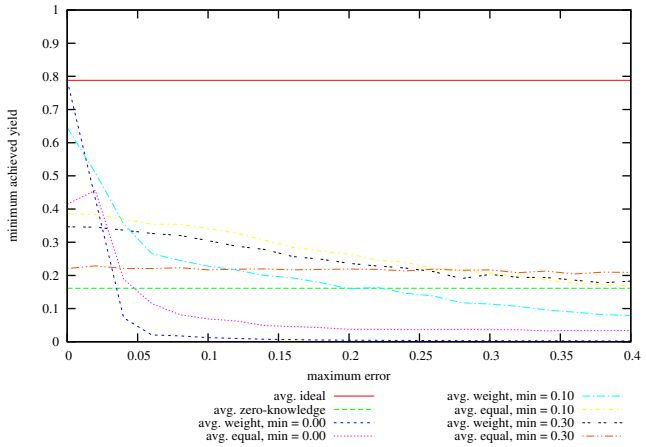

Figure 52: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.4, coefficient of variation = 1.0; values given are averages over successful instances.

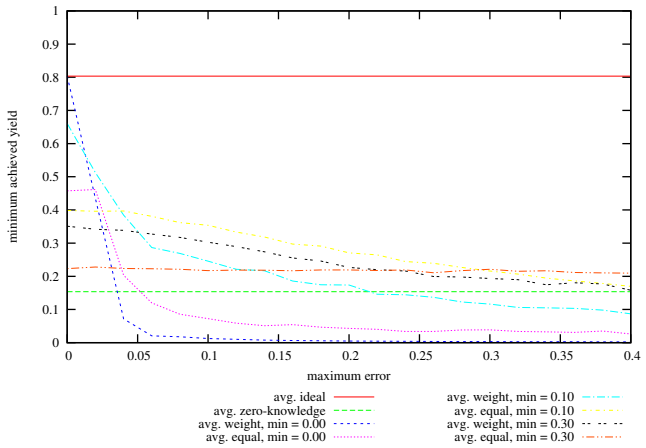

Figure 53: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.6, coefficient of variation = 1.0; values given are averages over successful instances.

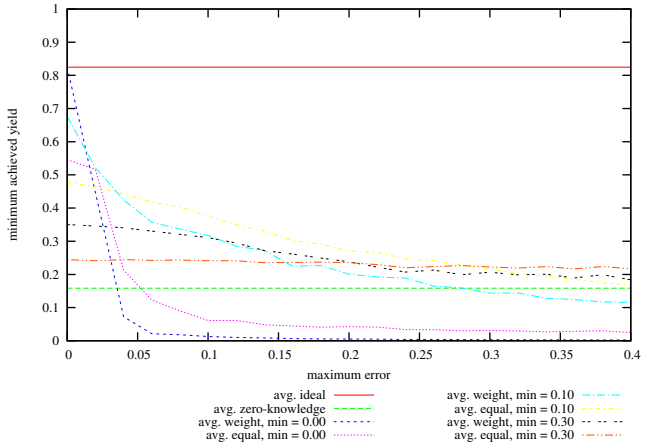

Figure 54: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.8, coefficient of variation = 1.0; values given are averages over successful instances.

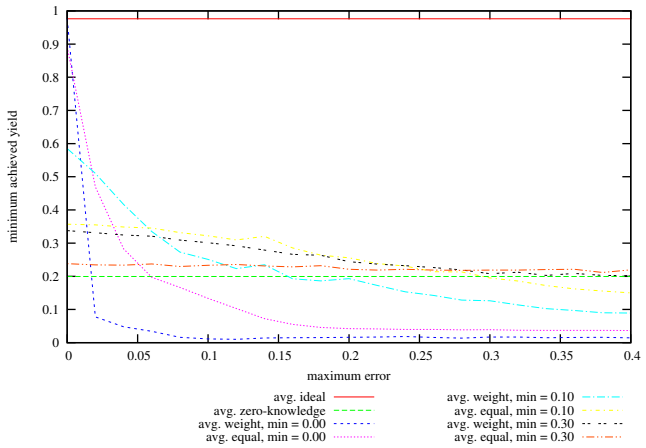

Figure 55: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 500 services, memory slack = 0.2, coefficient of variation = 0.0; values given are averages over successful instances.

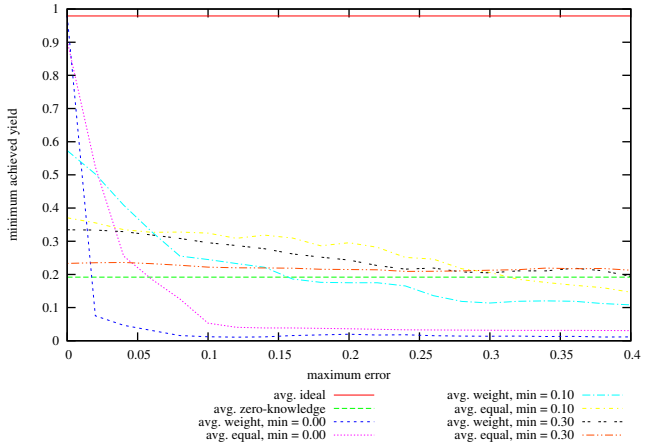

Figure 56: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 500 services, memory slack = 0.4, coefficient of variation = 0.0; values given are averages over successful instances.

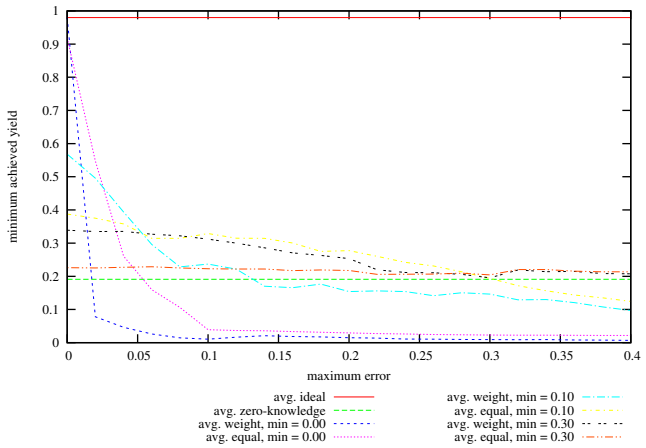

Figure 57: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 500 services, memory slack = 0.6, coefficient of variation = 0.0; values given are averages over successful instances.

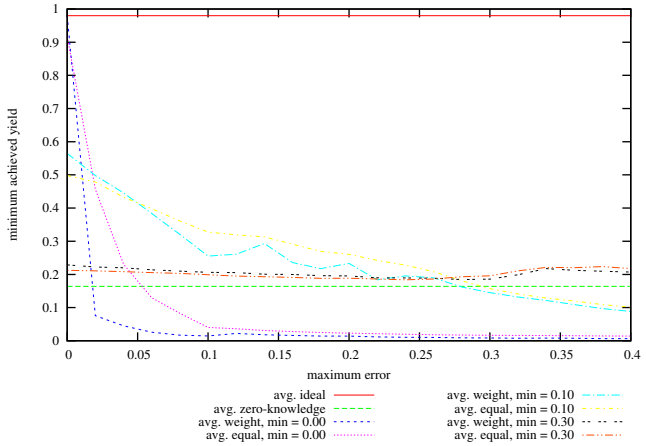

Figure 58: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 500 services, memory slack = 0.8, coefficient of variation = 0.0; values given are averages over successful instances.

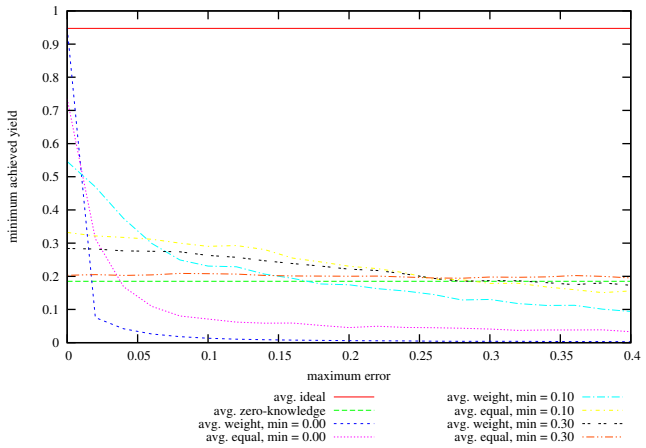

Figure 59: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 500 services, memory slack = 0.2, coefficient of variation = 0.5; values given are averages over successful instances.

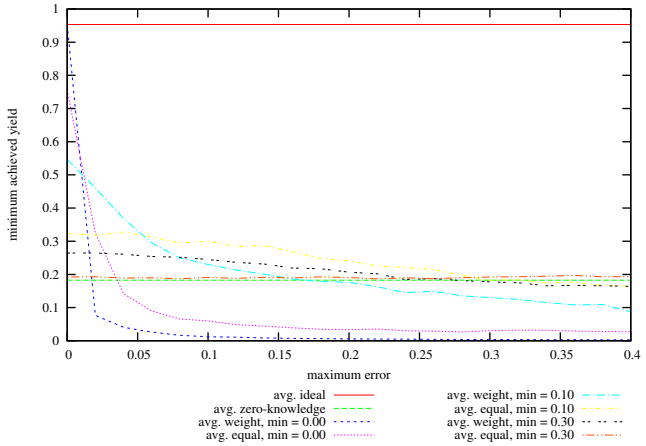

Figure 60: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 500 services, memory slack = 0.4, coefficient of variation = 0.5; values given are averages over successful instances.

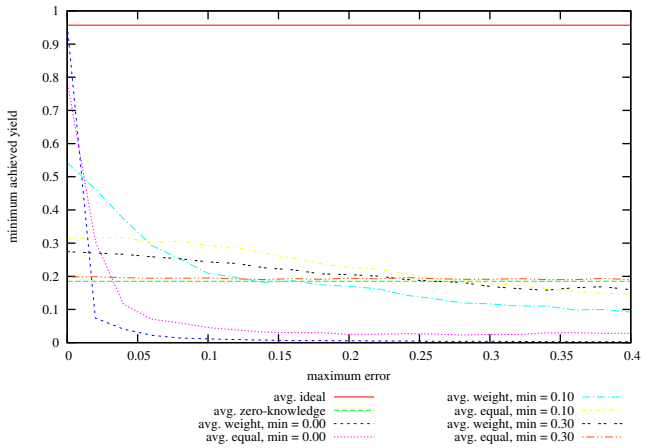

Figure 61: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 500 services, memory slack = 0.6, coefficient of variation = 0.5; values given are averages over successful instances.

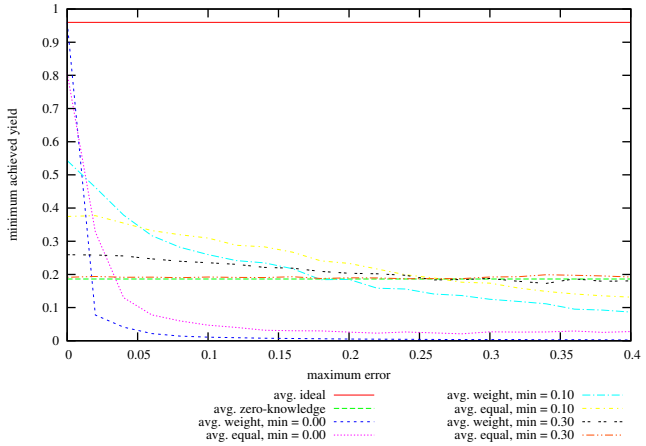

Figure 62: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 500 services, memory slack = 0.8, coefficient of variation = 0.5; values given are averages over successful instances.

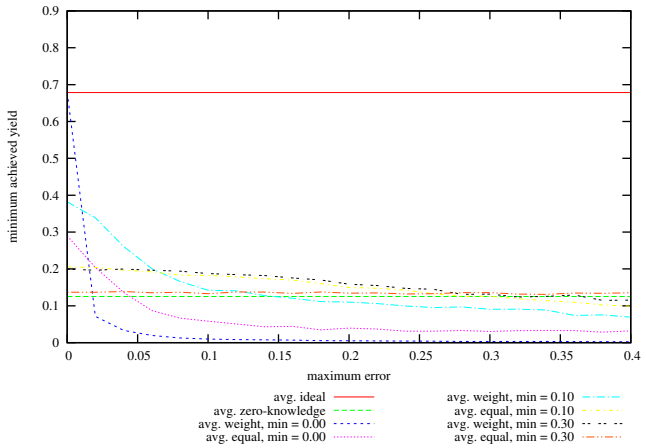

Figure 63: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 500 services, memory slack = 0.2, coefficient of variation = 1.0; values given are averages over successful instances.

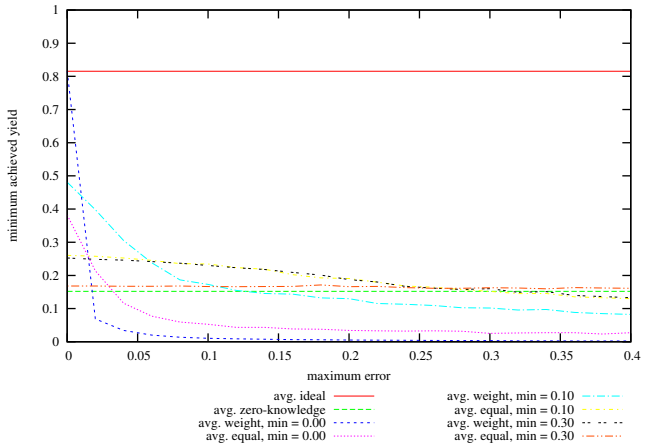

Figure 64: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 500 services, memory slack = 0.4, coefficient of variation = 1.0; values given are averages over successful instances.

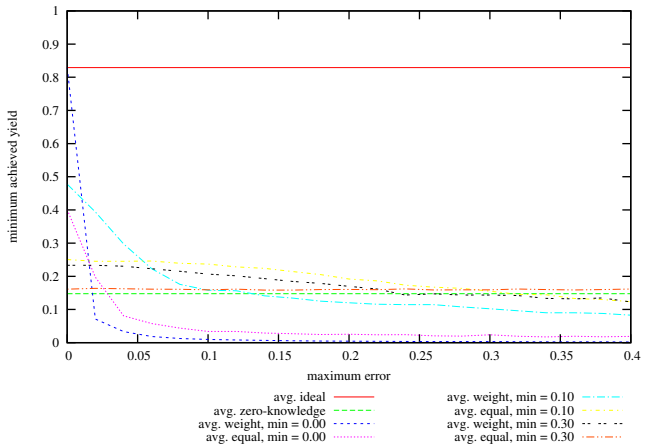

Figure 65: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 500 services, memory slack = 0.6, coefficient of variation = 1.0; values given are averages over successful instances.

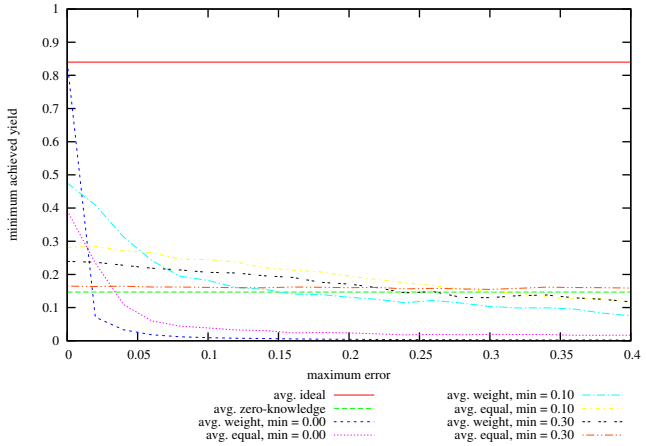

Figure 66: Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 500 services, memory slack = 0.8, coefficient of variation = 1.0; values given are averages over successful instances.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399