

# Technical Report

Department of Computer Science  
and Engineering  
University of Minnesota  
4-192 EECS Building  
200 Union Street SE  
Minneapolis, MN 55455-0159 USA

TR 99-024

Multi-Capacity Bin Packing Algorithms with Applications to Job  
Scheduling Under Multiple Constraints.

William Leinberger, George Karypis, and Vipin Kumar

May 25, 1999



# Multi-Capacity Bin Packing Algorithms with Applications to Job Scheduling under Multiple Constraints \*

William Leinberger, George Karypis, Vipin Kumar  
Department of Computer Science and Engineering, University of Minnesota  
(leinberg, karypis, kumar) @ cs.umn.edu

May 25, 1999

## Abstract

In past massively parallel processing systems, such as the TMC CM-5 and the CRI T3E, the scheduling problem consisted of allocating a single type of resource among the waiting jobs; the processing node. A job was allocated the minimum number of nodes required to meet its largest resource requirement (e.g. memory, CPUs, I/O channels, etc.). Single capacity bin-packing algorithms were applied to solve this problem. Recent systems, such as the SUN E10000 and SGI O2K, are made up of pools of independently allocatable hardware and software resources such as shared memory, large disk farms, distinct I/O channels, and software licenses. In order to make efficient use of all the available system resources, the scheduling algorithm must be able to maintain a job working set which fully utilizes all resources. At the core of this scheduling problem is a  $d$ -capacity bin-packing problem where each system resource is represented by a capacity in the bin and the requirements of each waiting job are represented by the  $d$  capacities of an item in the input list. Previous work in  $d$ -capacity bin-packing algorithms analyzed extensions of single capacity bin-packing. These extended algorithms are oblivious to the additional capacities, however, and do not scale well with increasing  $d$ . We provide new packing algorithms which use the additional capacity information to provide better packing and show how these algorithms might lead to better multi-resource allocation and scheduling solutions.

**Keywords:** multiple capacities, bin packing, multiple constraints, job scheduling

---

\*This work was supported by NASA NCC2-5268, by NSF CCR-9423082, by Army Research Office contract DA/DAAG55-98-1-0441, and by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute. Related papers are available via WWW at URL: <http://www.cs.umn.edu/~karypis>

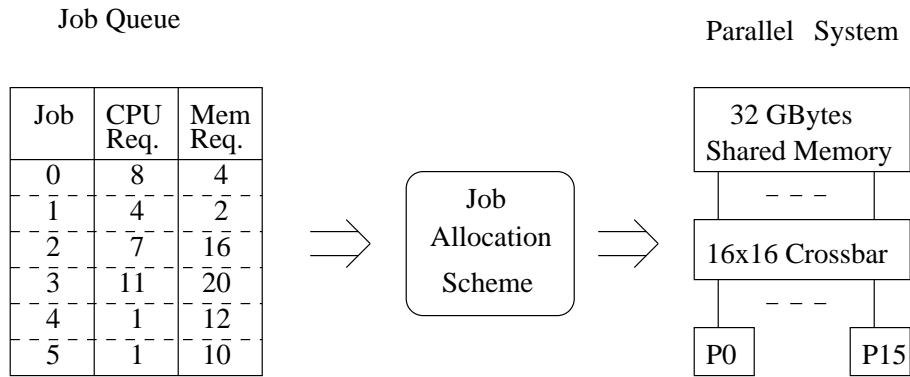
# 1 Introduction

New parallel computing systems, such as the SUN Microsystems E10000, the SRC-6, and the SGI Origin 2000, provide a pool of homogeneous processors, a large shared memory, customizable I/O connectivity, and expandable primary and secondary disk storage support. Each resource in these system architectures may be scaled independently based on cost and user need. A site which typically runs CPU intensive jobs may opt for a configuration which is fully populated with CPUs but has a reduced memory to keep the overall system cost low. Alternatively, if the expected job mix contains a large percentage of I/O and memory intensive jobs, a large memory configuration may be purchased with high I/O connectivity to network or storage devices. Finally, a mixed job set may be best serviced by a balanced system configuration. Therefore, given an expected job mix, a "shared-everything" parallel system can be configured with the minimal set of resources needed to achieve the desired performance. The question, then, is how to schedule jobs from the actual job stream onto a given machine to achieve the expected performance.

In classical job management systems (JMS), a job was submitted along with a set of resource requirements which specify the number of CPUs, amount of memory, disk space, etc., and the expected time to complete. The target systems were primarily distributed memory parallel processors with a single system resource - a processing node consisting of a CPU, memory, and a network connection to I/O devices. Although job allocation research literature is filled with exotic methods of allocating resources to a job stream [9], simple allocation schemes such as *First-Come-First-Serve (FCFS)* or *FCFS with Backfill (FCFS/BF)* were used in practice, providing acceptable levels of performance [8]. These job allocation schemes were limited in part due to the all-or-nothing hardware partitioning of the distributed systems. For example, a memory intensive job must be allocated enough nodes to meet the jobs memory requirements, but may not need all the CPUs which were co-allocated by default. The excess CPUs are not available to other waiting jobs and are essentially wasted. This situation is worse in newer systems where resources may be allocated to a job independently from each other. The greedy FCFS-based job allocation schemes cannot take full advantage of this additional flexibility.

Consider extending the FCFS-based schemes to account for multiple resources in a particular physical system configuration. The pure FCFS job allocation scheme would pack jobs from the job queue into the system, in order of their arrival, until some system resource (CPUs, memory, disk space, etc.,) was exhausted. The weak point of FCFS is that the next job in the queue may require more resources than those left available in the system. In this case, the job allocation scheme is blocked from scheduling further jobs until sufficient resources become available for this large job. This results potentially in large fragments of resources being under-utilized. The FCFS with backfill probabilistically performs better by skipping over jobs which require a large percentage of a single resource and finding *smaller* jobs which can make use of the remaining resources. Still, a single resource becomes exhausted while others remain under-utilized. The FCFS-based algorithms are restricted in selecting jobs based on their general arrival order.

In order for a job allocation scheme to efficiently utilize the independently allocatable resources of a parallel processor, it must be free to select any job based on matching all of



(a)

Scheduling Epoch	FCFS		FCFS/BF		UNC	
	Jobs	P/M	Jobs	P/M	Jobs	P/M
0	0, 1	12/06	0,1,4,5	14/28	0, 2, 4	16/32
1	2	07/16	2	07/16	1, 3, 5	16/32
2	3, 4	12/32	3	11/20		
3	5	01/10				

(b)

Figure 1: Job Allocation Scheme Comparison

the jobs' resource requirements with the available system resources. As an example, consider the JMS state depicted in figure 1 (a). The job allocation scheme must map the six jobs in the job queue to a two-resource system with 16 CPUs and 32 GBytes of memory. The CPU and memory requirements of each job are specified. Assume that the order in the job queue represents the order of arrival and that each job requires the same amount of execution time  $t$ . Under these assumptions, a job allocation scheme would select a set of jobs for execution during scheduling epoch  $e_i$ . The number of epochs required to schedule all jobs in the job queue is used to compare different job allocation schemes. Figure 1 (b) shows the jobs allocated to each scheduling epoch for FCFS, FCFS/BF, and an unconstrained job allocation scheme (UNC). The UNC scheme is free to select any job in the job queue for allocation during the current epoch. Although this is a contrived example, it illustrates the basic flaws of FCFS-based job allocation schemes and the potential of less restrictive job allocation schemes. The FCFS allocation scheme allocates jobs 0 and 1 in the first scheduling epoch but then cannot allocate job 2, due to the total CPU requirement of the three jobs being greater than the system provides ( $8 + 4 + 7 > 16$ ). FCFS/BF overcomes this flaw by skipping job 2 and scheduling jobs 4 and 5 in the first epoch. However, it then must schedule jobs 2 and 3 in separate epochs as there are no other jobs available to backfill in each of these epochs. Finally, the optimal UNC algorithm was smart enough to *not* schedule jobs 0 and 1 in the same epoch. Instead it finds two job subsets which exactly match the machine

configuration. As a result, the unrestricted job allocation scheme requires fewer scheduling epochs to complete all jobs.

The UNC allocation scheme tries to select a subset of jobs whose total resource requirements match the physical configuration of the target parallel system. This can be generalized to solving a *multi-capacity bin-packing* problem. The parallel system is represented by a bin with  $d$  capacities corresponding to the multiple resources available in the system. The job wait queue is represented by an item list where each item is described by a  $d$ -capacity requirements vector. A scheduling epoch consists of packing jobs from the job queue into the currently available resources in the parallel system. The information available in the additional capacity requirements for each job is used to guide the scheduling process.

Our contribution is to provide multi-capacity *aware* bin-packing algorithms which make use of the information in the additional capacities to guide item selection in the packing process. Past research in multi-capacity bin-packing has focused on extending the single capacity bin-packing to deal with the multiple capacities, and on providing performance bounds on these simple algorithms. In general, these naive algorithms did not use the additional capacity information to guide them so do not scale well with increasing capacity counts. Our simulation results show that the multi-capacity aware algorithms provide a consistent performance improvement over the previous naive algorithms. Further simulation results shows that the multi-capacity aware algorithms can produce a better packing from a small input list, which supports its use in online job scheduling. The complete bridge between bin-packing and job scheduling under multiple constraints is the subject of our current work in progress.

The remainder of this document is outlined below. Section 2 provides a summary of past research in multi-capacity bin-packing algorithms and discusses some of the limitations of these algorithms. Our new multi-capacity aware bin-packing algorithms are presented in Section 3, with experimental results and conclusions provided in Section 4.

## 2 Related Research

A variety of past research has dealt with single and  $d$ -capacity bin-packing problem formulations and their connection to the generalized scheduling problem [1], [2], [3], [5]. A brief summary of this work is provided below. In general, the  $d$ -capacity bin-packing algorithms are extensions of the single capacity bin-packing algorithms. However, they do not take advantage of the information in the additional capacities, and therefore do not scale well with increasing  $d$ .

The classical single capacity bin-packing problem may be stated as follows. We are given a positive bin capacity  $C$  and a set (or list) of scalar items  $L = \{x_1, x_2, \dots, x_i, \dots, x_n\}$  with each item  $x_i$  having an size  $s(x_i)$  satisfying  $0 \leq s(x_i) \leq C$ . What is the smallest  $m$  such that there is a partition  $L = B_1 \cup B_2 \cup \dots \cup B_m$  satisfying  $\sum_{x_i \in B_j} s(x_i) \leq C, 1 \leq j \leq m$ ?  $B_i$  is interpreted as the contents of a bin of capacity  $C$  and the goal is to pack the items of  $L$  into as few bins as possible.

The single capacity bin-packing problem formulation has been generalized to support

$d$ -capacities as follows [10], [7]. The capacity of a container is represented by a  $d$ -capacity vector,  $\vec{C} = (C_1, C_2, \dots, C_j, \dots, C_d)$ , where  $C_j, 0 \leq C_j$ , represents the  $k$ th component capacity. An item is also represented by a  $d$ -capacity vector,  $\vec{X}_i = (X_{i1}, X_{i2}, \dots, X_{ij}, \dots, X_{id})$ , where  $X_{ij}, 0 \leq X_{ij} \leq C_j$ , denotes the  $j$ th component requirement of the  $i$ th item. Trivially,  $\sum_{j=1}^d C_j > 0$  and  $\sum_{j=1}^d X_{ij} > 0 \forall 1 \leq i < n$ . An item  $\vec{X}_i$  can be *packed* (or fit) into a bin  $\vec{B}_k$ , if  $\vec{B}_k + \vec{X}_i \leq \vec{C}$ , or  $B_{kj} + X_{ij} \leq C_j \forall 1 \leq j \leq d$ . The items are obtained from an initial list  $L$ , and the total number of items to be packed is denoted by  $n$ . Again, the goal is to partition the list  $L$  into as few bins  $\vec{B}_k$  as possible.

The approach to solving the  $d$ -capacity bin-packing problem has mainly been to extend the single capacity bin-packing algorithms to deal with the  $d$ -capacity items and bins. The Next-Fit (NF) algorithm takes the next  $d$ -capacity item  $\vec{X}_i$  and attempts to place it in the current bin  $\vec{B}_k$ . If it does not fit (ie, if  $X_{ij} + B_{kj} > C_j$  for some  $j$ ) then a new bin,  $\vec{B}_{k+1}$ , is started. Note that no bin  $\vec{B}_l, 1 \leq l < k$  is considered as a candidate for item  $\vec{X}_i$ . The First-Fit (FF) algorithm removes this restriction by allowing the next item  $\vec{X}_i$  to be placed into any of the  $k$  currently non-empty bins. If  $\vec{X}_i$  will not fit into any of the current  $k$  bins, then a new bin  $\vec{B}_{k+1}$  is created and accepts the item. The Best-Fit adds a further bin selection heuristic to the First-Fit algorithm. Best-Fit places the next item into the bin in which it leaves the least empty space. Other variations of these simple algorithms have also been extended to support the  $d$ -capacity formulation.

Orthogonal to the item-to-bin placement rules described above is the method for pre-processing the item list before packing. For the single capacity bin-packing problem, sorting the scalar item list in non-increasing order with respect to the item weights generally improves the performance of the packing. First-Fit Decreasing (FFD) first sorts the list  $L$  in non-increasing order and then applies the First-Fit packing algorithm. Next-Fit and Best-Fit may be extended in a similar manner. The impact of pre-sorting the item list may be thought of as follows. Consider the First-Fit packing algorithm. When the input list is pre-sorted, the largest items are placed into the lower-numbered bins. Each successive item considers each currently defined bin in the order of their creation until it finds a bin into which it will fit. The result of this process is that the large items placed in the earlier bins are usually paired with the smaller items placed last. This avoids cases where the many small items may be *wasted* by filling common bins with other small or medium items, leaving no small items to be paired with the larger items. Sorting in the  $d$ -capacity formulation has also been explored with similar success as in the single capacity case. In the  $d$ -capacity formulation, however, the items are sorted based on a *scalar* representation of the  $d$  components. A simple extension to the single capacity case is to sort the items based on the sum of their  $d$  components (Maximum Sum). Other methods include sorting on the maximum component, sum of squares of components, product of components, etc.. The goal is to somehow capture the relative *size* of each  $d$ -capacity item.

The performance bounds for  $d$ -capacity bin-packing have also been studied [4]. If  $A$  is an algorithm and  $A(L)$  gives the number of bins used by that algorithm on the item list  $L$ , then define  $R_A \equiv A(L)/OPT(L)$  as the *performance ratio* of algorithm  $A$ , where  $OPT(L)$  gives the optimal number of bins for the given list. It has been shown that  $R_A \leq d + 1$  for any *reasonable* algorithm. Reasonable implies that no two bins may be combined into a single

bin. Note that the Next-Fit algorithm is not reasonable whereas the First-Fit and Best-Fit are reasonable. While this bound may seem a bit dismal, simulation studies have shown that the simple algorithms described above perform fairly well over a wide range of input. However, even though these algorithms perform better, on average, than the worst-case performance bound might suggest, there is still room for improvement.

Consider the First-Fit algorithm. When selecting a bin for placing the next item, First-Fit essentially ignores the current component weights of the item and the current component capacities of the bins. Its only criteria for placing an item in a bin is that the item fits. As a result, a single capacity in a bin may fill up much sooner than the other capacities, resulting in a lower overall utilization. This suggests that an improvement may be made selecting items to pack into a bin based on the current relative weights or *rankings* of its  $d$  capacities. For example, if  $B_{kj}$  currently has the lowest available capacity, then search for an item  $\vec{X}_i$  which fits into  $\vec{B}_k$  but which also has  $X_{ij}$  as its *smallest* component weight. This reduces the pressure on  $B_{kj}$ , which may allow additional items to be added to bin  $\vec{B}_k$ . This multi-capacity aware approach is the basis for the new algorithm designs presented in Section 3.

### 3 The Windowed Multi-Capacity Aware Bin-Packing

#### Algorithms

The  $d$ -capacity First-Fit(FF) bin-packing algorithm presented in section 2 looks at each item  $\vec{X}_i$  in the list  $L$  in order and attempts to place the item in any of the currently existing bins  $\vec{B}_1 \dots \vec{B}_k$ . If the item will not fit in any of the existing bins, a new bin  $\vec{B}_{k+1}$  is created and the item is placed there. An alternate algorithm which achieves an identical packing to FF is as follows. Initially, bin  $\vec{B}_1$  is created and the first item in the list  $L$ ,  $\vec{X}_1$ , is placed into this bin. Next, the list  $L$  is *scanned* from beginning to end searching for the next element  $\vec{X}_i$  which will fit into bin  $\vec{B}_1$ . Place each successive  $\vec{X}_i$  which fits into bin  $\vec{B}_1$ . When no element is found which will fit, then bin  $\vec{B}_2$  is created. Place the first of the remaining elements of  $L$  into  $\vec{B}_2$ . The process is repeated until the list  $L$  is empty. The primary difference is that each bin is filled completely before moving on to the next bin. With respect to job scheduling, this is analogous to packing jobs into a machine until no more will fit during a single scheduling epoch. At the start of the scheduling epoch, a bin is created in which each component is initialized to reflect the amount of the corresponding machine resource which is currently available. Jobs are then selected from the job wait queue and packed into the machine until there are not sufficient quantities of resources to fill the needs of any of the remaining jobs.

The list scanning process provides the basic algorithm structure for our new multi-capacity aware bin-packing algorithms. The key differences between the new algorithms and the FF algorithm is the criteria used to select the next item to be packed into the current bin. Whereas FF requires only that the item fits into the current bin, the multi-capacity aware algorithms will use heuristics to select items which attempt to correct a *capacity imbalance*



in the current bin. A capacity imbalance is defined as the condition  $B_{ki} < B_{kj}, 1 \leq i, j \leq d$  in the current bin  $\vec{B}_k$ . Essentially, at least one capacity is fuller than the other capacities. The general notion is that if the capacities are all kept balanced, then more items will likely fit into the bin. A simple heuristic algorithm follows from this notion. Consider a bin capacity vector in which  $B_{kj}$  is the component which is currently filled to a lower capacity than all other components. A *lowest-capacity aware* packing algorithm searches the list  $L$  looking for an item which fits in bin  $\vec{B}_k$  and in which  $X_{ij}$  is the largest resource requirement in  $\vec{X}_i$ . Adding item  $\vec{X}_i$  to bin  $\vec{B}_k$  heuristically lessens the capacity imbalance at component  $B_{kj}$ . The lowest-capacity aware packing algorithm can be generalized to the case where the algorithm looks at the  $w, 0 \leq w \leq d - 1$  lowest capacities and searches for an item which has the same  $w$  corresponding largest component requirements. The parameter  $w$  is a *window* into the current bin state. This is the general *windowed multi-capacity bin-packing heuristic*. Similar heuristics have been successfully applied to the multi-constraint graph partitioning problem [6]. Two variants of this general heuristic applicable to the  $d$ -capacity bin-packing problem are presented below.

**Permutation Pack.** Permutation Pack (PP) attempts to find items in which the largest  $w$  components are *exactly ordered* with respect to the ordering of the corresponding smallest elements in the current bin. For example, consider the case where  $d = 5$  and the capacities of the current bin  $\vec{B}_k$  are ordered as follows:

$$B_{k1} \leq B_{k3} \leq B_{k4} \leq B_{k2} \leq B_{k5}$$

The limiting case is when  $w = d - 1$ . In this instance, the algorithm would first search the list  $L$  for an item in which the components were ranked as follows:

$$X_{i1} \geq X_{i3} \geq X_{i4} \geq X_{i2} \geq X_{i5}$$

which is exactly opposite the current bin state. Adding  $\vec{X}_i$  to  $\vec{B}_k$  has the effect of increasing the capacity levels of the smaller components ( $B_{k1}, B_{k3} \dots$ ) *more* than it increases the capacity levels of the larger components ( $B_{k2}, B_{k5}, \dots$ ). If no items were found with this relative ranking between their components, then the algorithm searches the list again, relaxing the orderings of the smallest components first, and working up to the largest components. For example, the next two item rankings that would be searched for are:

$$X_{i1} \geq X_{i3} \geq X_{i4} \geq X_{i5} \geq X_{i2}$$

and

$$X_{i1} \geq X_{i3} \geq X_{i2} \geq X_{i4} \geq X_{i5}$$

... and finally,

$$X_{i5} \geq X_{i2} \geq X_{i4} \geq X_{i3} \geq X_{i1}$$

In the limiting case, the input list is essentially partitioned into  $d!$  logical sublists. The algorithm searches each logical sublist in an attempt to find an item which fits into the current bin. If no item is found in the current logical sublist, then the sublist with the

next best ranking match is searched, and so on, until all lists have been searched. When all lists are exhausted, a new bin is created and the algorithm repeats. The drawback is that the search has a time complexity of  $O(d!)$ . A simple relaxation to this heuristic is to consider only  $w$  of the  $d$  components of the bin. In this case, the input list is partitioned into  $d!/(d-w)!$  sublists. Each sublist contains the items with a common permutation of the largest  $w$  elements in the current bin state. Continuing the previous example, if  $w = 2$ , then the first list to be searched would contain items which have a ranking of the following form:

$$X_{i1} \geq X_{i3} \geq X_{i4}, X_{i2}, X_{i5}$$

The logic behind this relaxation is that the contribution to adjusting the capacity imbalance is dominated by the highest relative item components and decreases with the smaller components. Therefore, ignoring the relative rankings of the smaller components induces a low penalty. The algorithm time complexity is reduced by  $O(d-w)!$ , to approximately  $O(d_w)$ . The simulation results provided in Section 4 show that substantial performance gains are achieved for even small values of  $w \leq 2$ , making this a tractable option.

**Choose Pack.** The Choose Pack (CP) algorithm is a further relaxation of the PP algorithm. CP also attempts to match the  $w$  smallest bin capacities with items in which the corresponding  $w$  components are the largest. The key difference is that CP does not enforce an ordering between these  $w$  components. As an example, consider the case where  $w = 2$  and the same bin state exists as in the previous example. CP would search for an item in which

$$X_{i1}, X_{i3} \geq X_{i4}, X_{i2}, X_{i5}$$

but would not enforce any particular ordering between  $X_{i1}$  and  $X_{i3}$ . This heuristic partitions the input list into  $d!/w!(d-w)!$  logical sublists thus reducing the time complexity by  $w!$  over PP.

An example is provided in Tables 1 and 2 which further illustrates the differences between the FF, PP, and CP algorithms. Table 1 provides an item list for  $d = 5$ . Associated with each item is an *item rank* which indicates the relative rank of a component with respect to the other components in the same item. Item components are ranked according to the *maximum* so the largest component is ranked 0, the second largest is ranked 1, etc.. Table 2 shows the items selected by the FF, PP, and CP algorithms in packing the first bin, given  $w = 2$ . All algorithms initially select the first item,  $\vec{X}_1$ . The *bin rank* is analogous to the item rank in that it ranks the relative sizes of each component capacity. However, the bin rank uses a minimum ranking so the smallest component is ranked 0, the next smallest is ranked 1, and so forth. For each algorithm, Table 2 shows the item selection sequence, the resultant cumulative bin capacities, and the resultant bin ranking. The bin ranking is used by the PP and CP algorithms to filter the input list while searching for the next item. The FF algorithm ignores the current bin ranking.

After the selection of item  $\vec{X}_1$ , the FF algorithm searches the list for the next item which will fit. It finds that item  $\vec{X}_2$  fits and selects it next. The next item,  $\vec{X}_3$  will not fit as the capacity  $B_{k5}$  would be exceeded. Therefore,  $\vec{X}_3$  is skipped as is  $\vec{X}_4$  and  $\vec{X}_5$ . Item  $\vec{X}_6$  fits into the bin and completely exhausts  $B_{k1}$  and  $B_{k5}$  so the algorithm creates a new bin and selects item  $\vec{X}_3$  as the first item.

Table 1: Example Item Input List with Item Rankings;  $d = 5$ ;

Item#	Capacities					Item Rank (Max)				
	$X_{i1}$	$X_{i2}$	$X_{i3}$	$X_{i4}$	$X_{i5}$	$r_1$	$r_2$	$r_3$	$r_4$	$r_5$
1	0.1	0.3	0.2	0.5	0.4	4	2	3	0	1
2	0.4	0.1	0.1	0.2	0.5	1	3	4	2	0
3	0.1	0.3	0.2	0.1	0.4	3	1	2	4	0
4	0.4	0.1	0.6	0.2	0.3	1	4	0	3	2
5	0.5	0.1	0.3	0.2	0.3	0	4	1	3	2
6	0.5	0.3	0.2	0.1	0.1	0	1	2	3	4
7	0.1	0.5	0.3	0.1	0.2	3	0	1	4	2
8	0.4	0.1	0.2	0.1	0.1	0	2	1	3	4
9	0.3	0.1	0.1	0.2	0.1	0	2	3	1	4
10	0.1	0.2	0.3	0.5	0.4	4	3	2	0	1

Table 2: Example Item Selection for FF, BP( $w = 2$ ) and CP( $w = 2$ );  $d = 5$ ;

Algo.	Item#	Item Capacities					Cum. Bin Capacities					Bin Rank (Min)				
		$X_{i1}$	$X_{i2}$	$X_{i3}$	$X_{i4}$	$X_{i5}$	$B_{k1}$	$B_{k2}$	$B_{k3}$	$B_{k4}$	$B_{k5}$	$r_1$	$r_2$	$r_3$	$r_4$	$r_5$
FF	1	0.1	0.3	0.2	0.5	0.4	0.1	0.3	0.2	0.5	0.4	0	2	1	4	3
	2	0.4	0.1	0.1	0.2	0.5	0.5	0.4	0.3	0.7	0.9	2	1	0	3	4
	6	0.5	0.3	0.2	0.1	0.1	1.0	0.7	0.5	0.8	1.0	3	1	0	2	4
Total Bin Weight = 4.00																
PP	1	0.1	0.3	0.2	0.5	0.4	0.1	0.3	0.2	0.5	0.4	0	*	1	*	*
	5	0.5	0.1	0.3	0.2	0.3	0.6	0.4	0.5	0.7	0.7	*	0	1	*	*
	7	0.1	0.5	0.3	0.1	0.2	0.7	0.9	0.8	0.8	0.9	0	*	1	*	*
	9	0.3	0.1	0.1	0.2	0.1	1.0	1.0	0.9	1.0	1.0	1	*	0	*	*
Total Bin Weight = 4.90																
CP	1	0.1	0.3	0.2	0.5	0.4	0.1	0.3	0.2	0.5	0.4	0	*	0	*	*
	4	0.4	0.1	0.6	0.2	0.3	0.5	0.4	0.8	0.7	0.7	0	0	*	*	*
	6	0.5	0.3	0.2	0.1	0.1	1.0	0.7	1.0	0.8	0.8	*	0	*	0	*
Total Bin Weight = 4.30																

The PP algorithm revises the bin rank as each item is selected and uses it to guide the selection of the next item. After the selection of item  $\vec{X}_1$ , the bin rank is  $(0, *, 1, *, *)$  indicating that the smallest capacity is  $B_{k1}$  and the next smallest capacity is  $B_{k3}$ . The  $*$ 's represent don't cares to the PP algorithm (remember that on the  $w$  largest component capacities are of interest. PP then attempts to find an item in which the  $X_{i1}$  is the largest component and  $X_{i3}$  is the next largest. This item will have a ranking identical to the current bin ranking, due to the fact that item ranks are based on the maximum and bin ranks are based on the minimum components. Therefore, PP skips all items in the input list which are not ranked the same as the bin ranking for the first  $w$  components. Item  $\vec{X}_5$  matches the bin ranking and fits into the bin so it is selected next. After the addition of item  $\vec{X}_5$ , the new bin ranking is  $(*, 0, 1, *, *)$ . Item  $\vec{X}_7$  is the first item in the list which matches this ranking and fits within the space remaining in the bin, so it is selected next. This results in a bin ranking of  $(0, *, 1, *, *)$ . Item  $\vec{X}_8$  matches this ranking but does not fit in the bin as it would exceed capacity  $B_{k1}$ . No other item which matches this bin ranking will fit either so PP searches for items which match the next best bin ranking of  $(0, *, *, 1, *)$ . Item  $\vec{X}_9$  matches this ranking and fits, so it is selected and results in filling all capacities in  $\vec{B}_k$  except  $B_{k3}$ , so PP creates a new bin and continues by selecting item  $\vec{X}_2$  as the first item.

The CP algorithm works much the same way as the PP algorithm except that the  $w$  smallest bin items are all ranked equally. When comparing a bin rank to an item rank, the  $w$  largest item components are all treated equally as well. After the selection of item  $\vec{X}_1$ , CP searches for an item in which  $X_{i1}$  and  $X_{i3}$ . To reiterate, the ordering between  $X_{i1}$  and  $X_{i3}$  is not considered. Therefore, CP selects item  $\vec{X}_4$  and adds it to the bin. Note that this item was skipped by PP because it did not have the exact ordering of  $X_{i1} \leq X_{i3}$ . However, since CP has relaxed this requirement,  $\vec{X}_3$  is an acceptable item candidate. Next, CP selects item  $\vec{X}_6$  which succeeds in filling bin capacities  $B_{k1}$  and  $B_{k3}$ . CP creates a new bin and selects item  $\vec{X}_2$  as the first item.

Note that other multi-capacity aware heuristics may be employed which essentially look at the *relative state or ordering* of the individual bin capacities and search for items which exhibit compatible relative state which could be used to correct a capacity load imbalance. For example, suppose that  $B_{kl}$  and  $B_{km}$  represent the largest and smallest component capacities, respectively, in the current bin. One heuristic would be to search for an item in which  $X_{il}$  and  $X_{im}$  are the smallest and largest component (and that  $\vec{X}_i$  fits, naturally). A more relaxed heuristic may only require that  $X_{il} < X_{im}$ .

## 4 Experimental Results

The following subsections present simulation results for the Permutation Pack (PP) and Choose Pack (CP) bin-packing algorithms. The performance measure of interest is the number of bins required to pack all the items in the list. The results are reported as normalized to the First-Fit (FF) algorithm. Note that  $PP(w = 0)$  and  $CP(w = 0)$  are identical to the FF algorithm where  $w$  is the number (or window) of capacity components used to guide the packing process. In the following discussion, FF refers to PP or CP with  $w = 0$  and PP and

CP imply that  $w \geq 1$ .

Both the PP and CP algorithms were tested with  $d$ , the capacity count, ranging from 1 to 32 and  $w$ , the capacity window, ranging from 0 to 4. Results are reported for  $d = 8$  and the full range of  $w$  tested. The results for the other test cases had similar characteristics as those provided below so are omitted here for the sake of brevity.

The input list of  $n = 32768$  items is generated as follows. For item  $\vec{X}_k$   $1 \leq k \leq n$ , the  $l$ th capacity component,  $X_{kl}$   $1 \leq l \leq d$ , was drawn from the  $l$ th independent random number generator. The  $d$  independent random number generators each followed an exponential distribution with a mean of  $\bar{X}$ . The mean weight,  $\bar{X}$ , was varied from 0.05 to 0.35 which provides substantial range of test input cases for the packing algorithms. The most profound effect of the average weight is the resultant average number of items which can be packed into a bin. At the low end of 0.05, the packing algorithms pack between 15 and 25 items per bin, with an average of approximately  $1.0/(\bar{X})$ , or 20. As the average weight increases, the average number of items packed drops due to the items being larger but also due to there being fewer *small* items to fill in the gaps in the bins. At an average weight of 0.35, only 2 or 3 items can be packed into a bin on the average no matter which packing algorithm is used. Above this average weight, we found the results to be approximately the same with all the algorithms so they are omitted here for the sake of brevity.

## 4.1 Performance of the Permutation and Choose Pack Algorithms on Unsorted Lists

The PP and CP algorithms were implemented and simulated on the synthetic test cases as described above. Figure 2 shows the results for the PP algorithm with similar results provided for CP in figure 3. These figures plot the bin requirement for the PP and CP algorithms, respectively, normalized to the FF algorithm versus the average capacity weight,  $\bar{X}$ . The data represents the ratio of the FF bin requirement to the PP or CP bin requirement. Therefore, a value greater than 1.0 represents a performance *gain*.

Consider the results for the PP algorithm shown in figure 2. For the case where  $w \geq 1$  and the average weight  $\bar{X}$  is low, the PP algorithm provides approximately a 10% improvement over the classical FF algorithm. The performance difference diminishes as the average weight  $\bar{X}$  grows, due to granularity issues. The larger component weights result in a less efficient packing of any single capacity in a bin, and is independent of  $d$ . Basically there are not enough *small* items to pair with the many *large* items. As  $w$  increases above 1 the additional performance gains also diminish. This is due to three different effects. First, the influence of the largest weight is most important in achieving a balanced capacity state. As  $w$  increases, the impact to the capacity balancing by the lesser weighted components is also smaller. The second reason for the diminishing performance at higher  $w$  is a reflection of the static and finite population of the input list. Essentially, there are a fixed and limited number of *small* items in the input list. An item,  $\vec{X}_k$  is considered small if the individual components are generally much smaller than the average item weight,  $\bar{X}$ . Small items are valuable for filling

in the cracks of a bin which has already has several items. Initially, there is a large sample of items to select from, and PP has a lot of success in packing the first few bins. In doing so, however, PP essentially *depletes* the input list of small items. Simulations have shown that as PP progresses, the average item size in the input list increases more rapidly than with FF. Additionally, as the bin number grows, the average number of items packed into a bin decreases more rapidly than with FF. The overall result is that, for large  $w$ , the performance gains from intelligent item selection are offset by the performance losses due to depleting the supply of small items early in the packing process. In fact, for large average weights, FF performs slightly *better* than PP as this effect is amplified by the larger average item sizes initially in the input list. This is evident in figure 2 for  $w = 4$  and  $\bar{X} > 0.25$ . This situation is exacerbated by pre-sorting the input item list and will be explored further in Section 4.2. Note that this situation is primarily due to the finite population of the input list used for bin-packing experiments. When PP is applied to job scheduling, the input stream is constantly re-newed so the impact of small item depletion does not become a global issue. This will be explored further in Section 4.4. The third reason for a diminished performance with increasing  $w$  has to do with the way PP splits up the input list into logical sublists. Recall that PP filters the input list into logical sublists, searching for an item with a specific ranking among its component weights. If it does not find an item with this specific ranking, it then adjusts its search to the next best ranking and repeats its search on that logical sublist. As  $w$  gets larger, the number of logical lists grows as  $d!/(d-w)!$ . Note that each list represents a specific permutation of the  $w$  capacity rankings. The windowed multi-capacity aware heuristic is successful only if it is able to find an item with the proper component rankings among the  $d!/(d-w)!$  lists. For this to be true,  $d!/(d-w)!$  must be small with respect to  $n$ . As PP packs the first few bins, this relationship is true (for our experiments). However, as items are removed from the input list,  $n$  is effectively reduced and the probability that the PP algorithm will find the properly ranked item diminishes. The net effect is that the first few bins are packed very well but the *average* improvement over all the bins is less.

Now consider the performance of the CP algorithm depicted in figure 3. The first thing to note is that the general performance of CP is nearly as good or better than PP even though it uses a relaxed selection method. The CP method is not as strict as the PP method in selecting the next item for packing, therefore, it does not achieve the high efficiency bin-packing on the first few bins as does the PP method. However, it does not suffer as bad from the small item depletion syndrome seen in the PP algorithm at the higher  $w$  values. This is seen by comparing the performance results between figures 2 and 3 for the case  $w = 4$  and  $\bar{X} > 0.25$ . Whereas the performance of PP gets worse than FF in figure 2, CP maintains a performance advantage over FF as shown in figure 3.

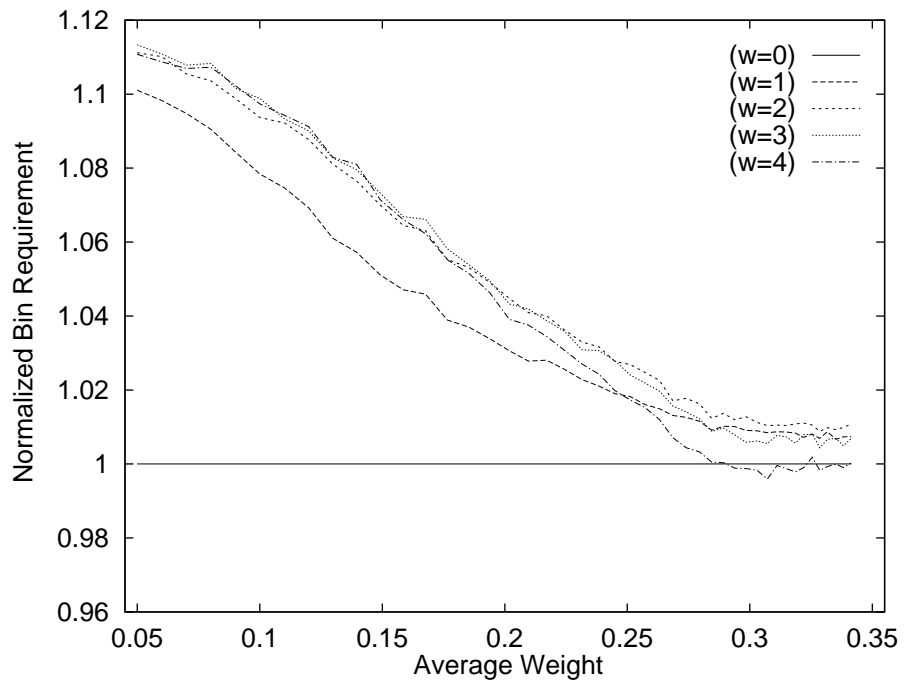


Figure 2: Performance Gains for Permutation Pack (d=8; No Pre-sorting)

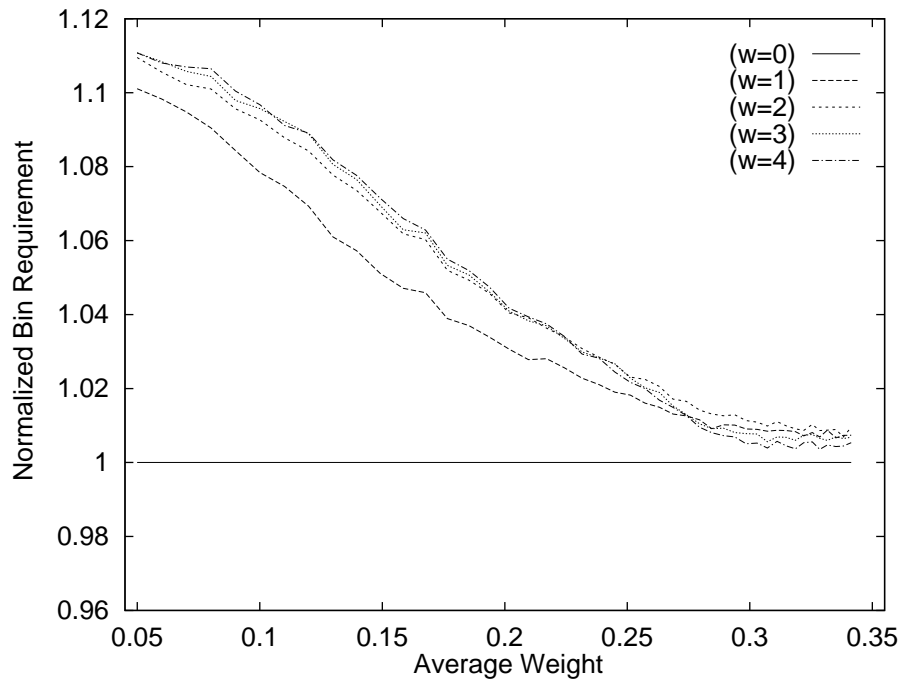


Figure 3: Performance Gains for Choose Pack (d=8; No Pre-sorting)



## 4.2 Effects of Pre-sorting the Input List on the Performance of

### PP and CP

Pre-sorting the input list in a non-increasing order of item size has been used to improve the performance of the single capacity bin-packing algorithms. Our simulations show that this general trend continues for the  $d$ -capacity aware algorithms. For this experiment, the input list was sorted using a maximum sum method to assign a scalar key ( $\bar{X}_i(key) = \sum_{j=1}^d X_{ij}$ ) to an item. The PP and CP algorithms were then applied to the sorted list. The results for the PP and CP algorithms are depicted in figures 4 and 5 respectively.

The results depicted for PP in figure 4 show approximately an 8% performance gain at low average component weights for  $w \geq 1$  as compared to the FF applied to the same pre-sorted list. Note that this performance gain is less than the approximately 10% seen for the case when the input list is unsorted as depicted in figure 2. The reasons for this diminished return are twofold. First, since the PP algorithm is *more selective* in picking the next item to pack into a bin, it searches deeper into the list to find an item to adjust the capacity imbalance. Alternatively, FF finds the next item which fits. Since the list is pre-sorted, the item found by PP is *no greater than* the item found by FF. After the initial item selection, PP tends to fill the current bin with smaller items resulting in depleting the small items in the finite list population. This contributes to a overall diminished performance as the larger items are left for the last bins, with no smaller items to pair with them. This effect was also noted for PP on the unsorted list for  $w = 4$  and average weight  $\bar{X} > 0.25$ . Pre-sorting the list merely amplifies this phenomena. The second reason for a diminished performance with increasing  $w$  has to do with the way PP splits up the input list into logical sublists. The globally sorted input list is fragmented into  $d!/(d-w!)$  locally sorted sublists which are searched in an order which is dependent on the capacity ranking of the current bin. The net result is that as  $w$  increases (with respect to constant input list size), the *actual search order* of the items in the input list becomes globally random so the performance gain due to pre-sorting is nullified.

The CP algorithm relaxes its search criteria with respect to PP. As shown in figure 5, this results in slightly higher performance gains over FF as compared to gains achieved by PP for  $w > 1$ . Specifically, CP maintains a performance advantage at high average component weights and higher  $w$ . This is due to the fact that CP partitions its input list into  $d!/(w!(d-w!))$  logical sublists (a factor of  $w!$  fewer than PP) so the effects of fragmentation are reduced. As a result, CP realizes a higher benefit from the pre-sorting than does PP.

### 4.3 Adaptive Pack: An Adaptive Multi-Capacity Aware Packing

#### Algorithm

The results presented in Sections 4.1 and 4.2 may be generalized as follows. For lower average component weights, the PP and CP algorithms perform better with a higher  $w$ . At higher average weights, they perform better with a lower  $w$ . This is a reflection of the ability of the PP and CP algorithms to aggressively pack the first few bins with the smaller items in the finite population list, leaving the larger grained items for packing last. The high packing efficiency on the first bins is offset by the lower efficiency on the later bins. In view of these results, an *adaptive* packing algorithm could be devised which modifies the window,  $w$ , based on the probability of finding smaller items among those remaining in the input list. As this probability gets higher, a more aggressive  $w$  (larger) could be used to pack the abundant smaller items into bins to a higher capacity. Conversely, as the probability gets lower, a less aggressive  $w$  (smaller) could be used to pack the larger items greedily as done by FF. Adaptive Pack (AP) adjusts  $w$  based on the average component weight of the items remaining in the input list after each bin is packed. The performance results for AP are shown in Figures 6 and 7 for unsorted and pre-sorted input lists for a range of capacity counts  $2 \leq d \leq 32$ .

In general, the AP performs as good or better than the PP and CP algorithms over the range of input simulated. Specifically, the degradation seen in the PP and CP algorithms at high average weights,  $\bar{X}$ , and high windows,  $w$ , is avoided by the AP algorithm. Also, the performance gains for each  $d$  value are as good or better than the PP or CP algorithms using any single  $w$  value. This may be seen by comparing the data for  $d = 8$  in Figures 6 and 7 with the data in Figures 2 and 4. In Figure 6, for  $d = 8$  and  $\bar{X} = 0.15$ , the performance gain of AP over FF is approximately 8% while for the same case in Figure 2, the gain is approximately 7%. A similar comparison between Figures 7 and 4 shows that AP maintains the performance seen by PP( $w = 4$ ).

### 4.4 A First Step Towards Job Scheduling under Multiple Constraints

Bin-packing is basically an abstraction of a restricted batch processing scenario in which all the jobs arrive before processing begins and all jobs have the same execution time. The goal is to process the jobs as fast as possible. Define  $A_i$  as the arrival time and  $T_i$  as the expected execution time of job  $i$ . In the batch processing scenario,  $A_i = 0$ , and  $T_i = T$  for some constant  $T$ . The results of Sections 4.1 and 4.2 suggest that the windowed multi-capacity aware bin-packing algorithms may be used as the basis for a scheduling algorithm. Basically, each bin corresponds to a scheduling epoch on the system resources, and the scheduling algorithm must pack jobs onto the system in an order such that it all jobs are scheduled using the fewest epochs. The Adaptive Pack algorithm with a pre-sorted job queue should

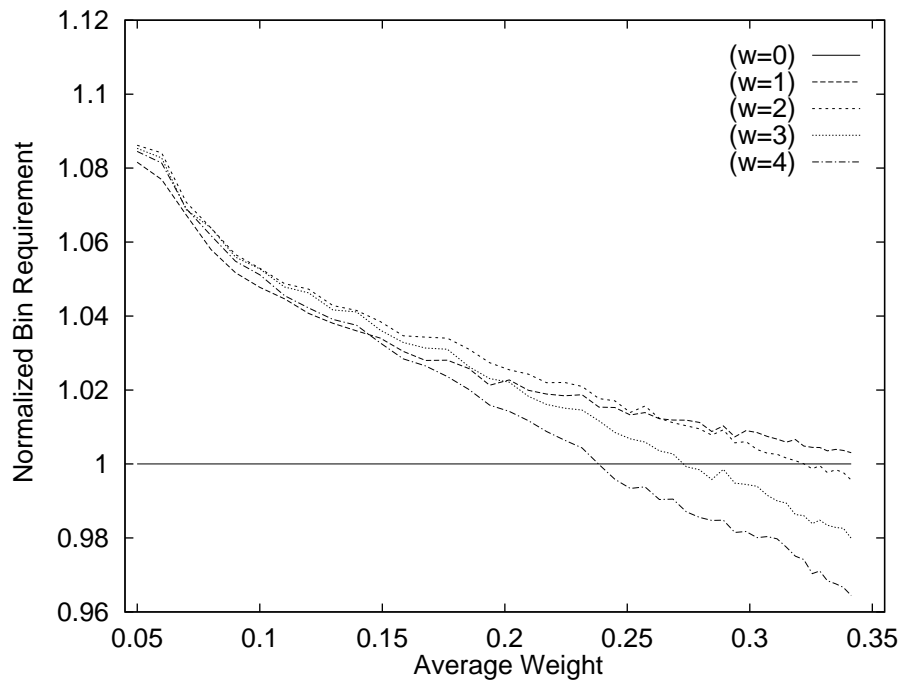


Figure 4: Performance Gains for Permutation Pack ( $d=8$ ; Maximum Sum Pre-Sorting)

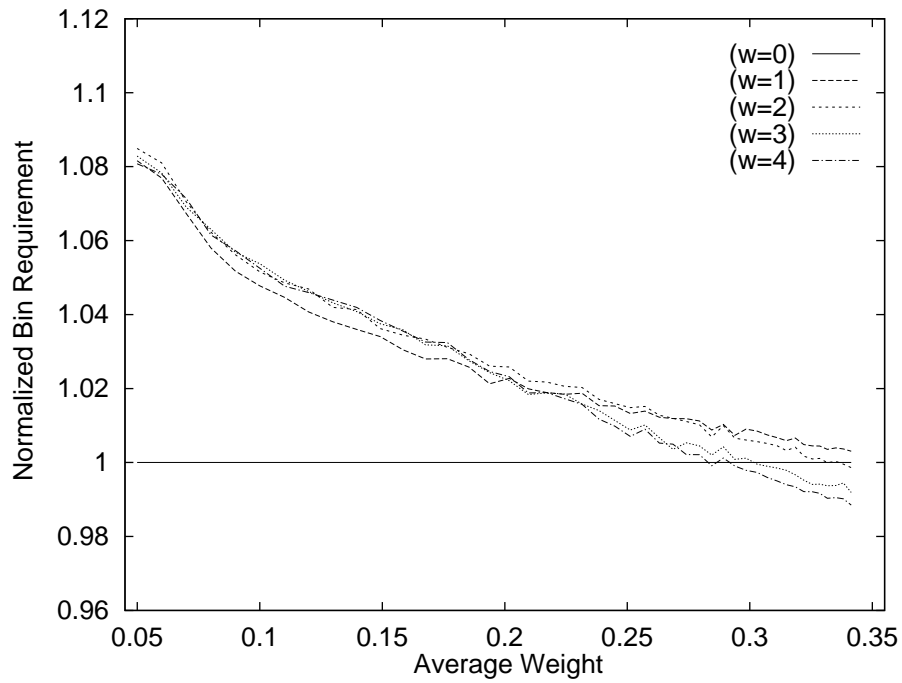


Figure 5: Performance Gains for Choose Pack ( $d=8$ ; Maximum Sum Pre-Sorting)

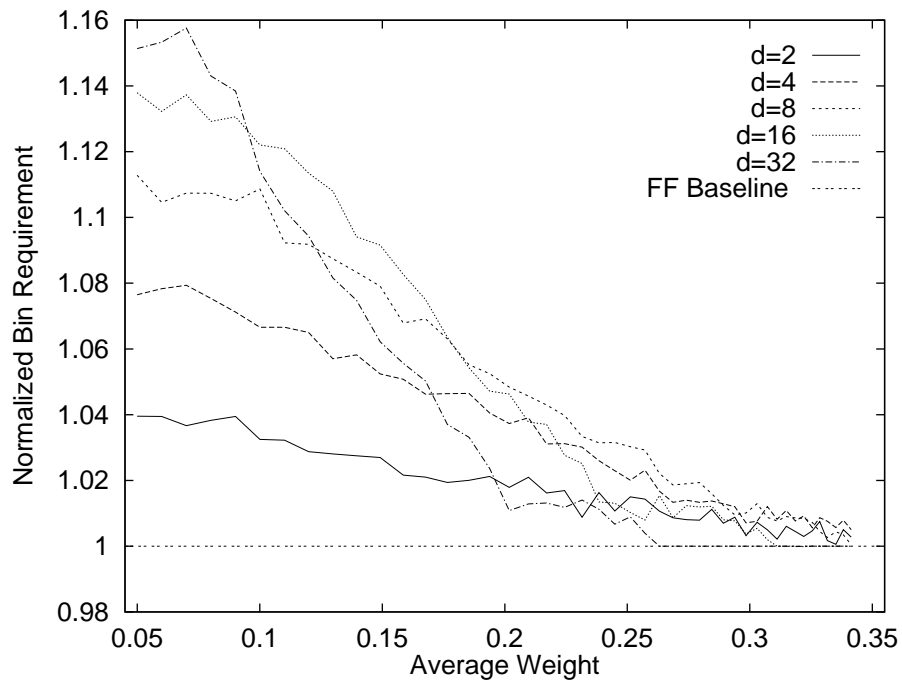


Figure 6: Performance Gains for Adaptive Pack (No Pre-Sorting)

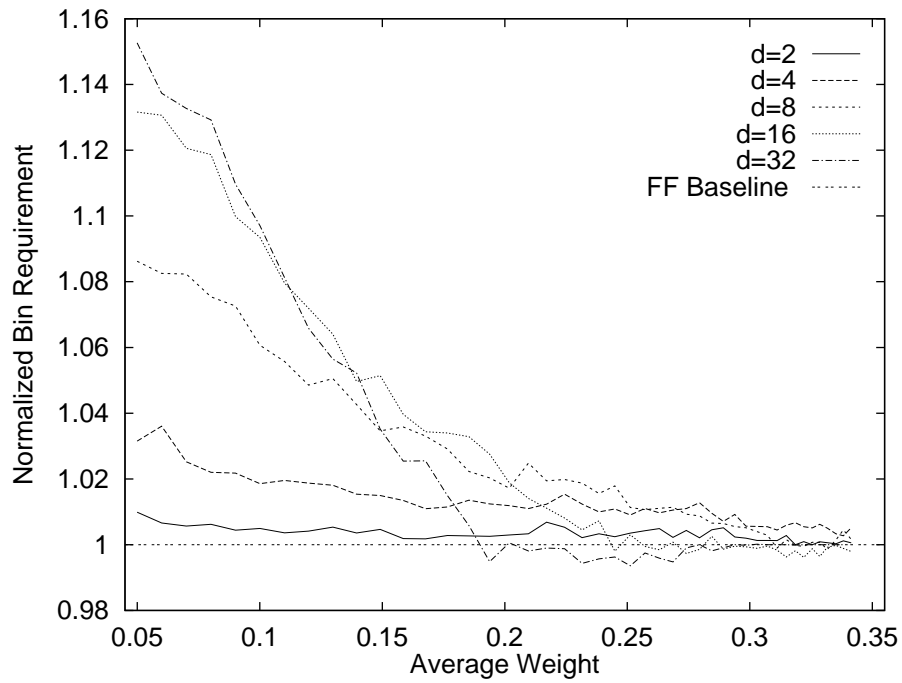


Figure 7: Performance Gains for Adaptive Pack (Maximum Sum Pre-Sorting)

give good results as it provides the best performance in achieving the lowest number of bins.

The next level of complexity is to remove the restriction on  $A_i$  to allow the continuous arrival of new jobs. Now the performance of the scheduling algorithm depends on the packing efficiency of only the *first* bin or epoch from a much *smaller* item list or job queue. The PP and CP algorithms work even better under this dynamic item list population scenario. In the static input list scenario, the PP and CP are able to pack a lot of smaller items into the first few bins. However, this depleted the supply of small items on the earlier bins resulting in a less efficient packing of the remaining items due to their large granularity. In the dynamic item list scenario, each bin is packed from essentially a new list as items are replaced as soon as items are packed. Also, since the PP and CP algorithms select the first element of the input list before initiating capacity balancing, the waiting time of any item is bounded by the number of items ahead of it in the queue. Figure 8 shows the performance of the PP algorithm on first-bin packing efficiency for  $d = 8$ . In this simulation, the number of items in the item list is initialized to 4 times the expected number of items which would optimally fit into a bin. Specifically,  $n = 4.0 * \lceil 1.0/\bar{X} \rceil$ . Note that this  $n$  is much smaller than the  $n$  used for the bin-packing experiments. This reflect the smaller size of job wait queues expected to be seen by the scheduler. The simulation loops between packing an empty bin and replacing the items drawn from the list. In this manner, the *number* of items that a packing algorithm starts with is always the same. The  $d$ -capacity items are generated as in previous simulations. As shown in figure 8, for small average weights  $\bar{X}$  and  $w > 0$ , the PP algorithm achieves a 13% to 15% performance gain over the FF algorithm. Compare this performance gain to the 11% gain seen by the AP algorithm in Figure 6. The multi-capacity aware algorithms can pack any *single* bin much better than the naive FF algorithm, when starting from the same input list. For higher  $w$  and  $\bar{X}$ , maintains its performance gain over FF. The results for other  $d$  were simulated and showed similar trends. In general, the packing efficiency of the PP algorithm increases with increasing  $w$ . The diminished increases in performance for higher  $w$ , while positive and finite, are due primarily to the lower impact of considering the smaller item components when performing capacity balancing. Additionally, for higher  $w$ , the probability of finding an item which best matches the current bin capacity imbalance is decreased due to the relatively small population from which to choose ( $n \ll d/(d-w)!$ ). Recall that the search performed by PP( $w = i$ ) is a refinement of the search used by PP( $w = j$ ) for  $i > j$ . If PP( $w = i$ ) cannot find the exact item it is looking for, then it should heuristically find the item that PP( $w = j$ ) would have found. Therefore, increasing  $w$  should heuristically do no worse than for lower  $w$  at the cost of higher time complexity. Essentially, when  $(d!/(d-w)! \gg n)$ , PP( $w = i$ ) collapses to PP( $w = j$ ).

As the average weight increases, the item granularity issues diminish the packing efficiency for any packing algorithm. The relaxed selection criteria used by the CP method results in little performance gains for  $w > 1$  so those results are omitted from the graph for the sake of clarity. However, the CP algorithm still has a much lower time complexity for higher  $w$  than does the PP algorithm, so a trade is available.

The final level of complexity in bridging the gap from bin-packing to job scheduling is to remove the execution time restriction and allow each item to have a different execution time. This is the subject of our current work in progress.

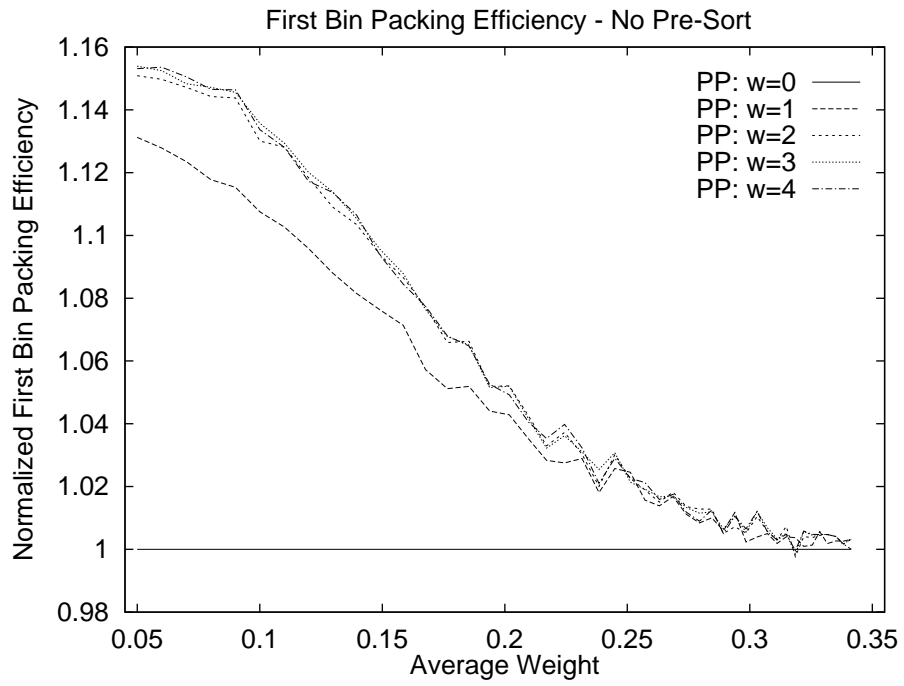


Figure 8: Performance Gains in First-Bin Packing Efficiency for Permutation Pack ( $d=8$ ; No Pre-Sorting)

## 4.5 Summary of Experimental Results

The experimental results for the PP and CP algorithms are summarized below.

1. The windowed multi-capacity aware bin-packing algorithms, PP and CP, provide a consistent performance increase over the classical FF algorithm for items with smaller average weights and comparable performance for items with higher average weights in an unsorted list.
2. A large percentage of the performance gains are achieved by a small window  $w \leq 2$  which reduces the time complexity of the general windowed heuristic.
3. Pre-sorting the input list provides performance gains for all the tested bin-packing algorithms but the gains are less for the multi-capacity aware algorithms using high window values on lists with high average component weights due to list fragmentation and small item depletion.
4. An adaptive algorithm, AP, was devised which maximizes the performance gains by adapting the capacity window according to the average weight of the items remaining in the input list. AP performs as good or better than the PP and CP algorithms and does not suffer from the same degradation seen by PP and CP at high average component weights.
5. The first-bin packing efficiency of the PP algorithms provides substantial performance over the FF algorithm which provides a proof-of-concept that the windowed multi-capacity aware heuristic may be applied to the generalized online multi-constraint job scheduling problem.

In general the experimental results show that the multi-capacity aware heuristics, which strive to correct local capacity imbalances, provides consistent performance gains over the classical capacity oblivious bin-packing algorithms. Additionally, a large percentage of the performance gains come from small  $w$  which greatly reduces the running time complexity of the algorithms. Further, the heuristic produces superior first-bin packing efficiencies from a small population list which shows the applicability of the heuristic to job scheduling under multiple constraints.

As a final note, the simulation results presented here are in some respect an artifact of the synthetic input data. Specifically, the *relationship* between the  $d$  components in a given item was uncorrelated as they were drawn from *independent* random number streams. In a job scheduling scenario, the relationships between the components of a jobs requirement vector may be quite correlated. In one case, if the items are proportional, (e.g. large memory implies large CPU requirements), then the dimension of the packing problem is effectively reduced from a 2-capacity to a 1-capacity. In this case the multi-capacity aware algorithms would provide a smaller performance gain with respect to the naive packing algorithms. If, however, the requirement components are *inversely related*, (e.g. Large memory requirement with small CPU and medium I/O requirements), then the performance gains seen by the multi-capacity aware algorithms should be substantial. Job stream characterization is part of our on-going work in applying the multi-capacity aware heuristics to the general problem of job scheduling under multiple constraints.

## References

- [1] Jr. E. G. Coffman, M. R. Garey, and D. S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal of Computing*, 7(1):1–17, February 1978.
- [2] Jr. E. G. Coffman, M. R. Garey, and D. S. Johnson. Dynamic bin packing. *SIAM Journal of Computing*, 12(2):226–258, May 1983.
- [3] Jr. E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin-packing - an updated survey. In G. Ausiello, M. Lucertini, and P. Serafini, editors, *Algorithm Design for Computer System Design*, pages 49–99. Springer-Verlag, New York, 1984.
- [4] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal of Computing*, 4(2):187–201, June 1975.
- [5] M. R. Garey, R. L. Graham, D. S. Johnson, and Andrew Chi-Chih Yao. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory*, pages 257–298, 1976.
- [6] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. Technical Report 98-019, University of Minnesota, Department of Computer Science, Army HPC Research Center, 1998.
- [7] L. T. Kou and G. Markowsky. Multidimensional bin packing algorithms. *IBM Journal of Research and Development*, 21(5):443–448, September 1977.
- [8] D. Lifka. The anl/ibm sp scheduling system. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1995.
- [9] R. K. Mansharamani and M. K. Vernon. Comparison of processor allocation policies for parallel systems. Technical report, Computer Sciences Department, University of Wisconsin, December 1993.
- [10] K. Maruyama, S. K. Chang, and D. T. Tang. A general packing algorithm for multidimensional resource requirements. *International Journal of Computer and Information Sciences*, 6(2):131–149, May 1976.