Sum-of-Squares Heuristics for Bin Packing and Memory Allocation

MICHAEL A. BENDER
State University of New York at Stony Brook
BRYAN BRADLEY
Hofstra University
GEETHA JAGANNATHAN
Stevens Institute of Technology
and
KRISHNAN PILLAIPAKKAMNATT
Hofstra University

The sum-of-squares algorithm (SS) was introduced by Csirik, Johnson, Kenyon, Shor, and Weber for online bin packing of integral-sized items into integral-sized bins. First, we show the results of experiments from two new variants of the SS algorithm. The first variant, which runs in time $O(n\sqrt{B}\log B)$, appears to have almost identical expected waste as the sum-of-squares algorithm on all the distributions mentioned in the original papers on this topic. The other variant, which runs in $O(n \log B)$ time, performs well on most, but not on all of those distributions.

We also apply SS to the online memory-allocation problem. Our experimental comparisons between SS and Best Fit indicate that neither algorithm is consistently better than the other. If the amount of randomness in item sizes is low, SS appears to have lower waste than Best Fit, whereas, if the amount of randomness is high Best Fit appears to have lower waste than SS. Our experiments suggest that in both real and synthetic traces, SS does not seem to have an asymptotic advantage over Best Fit, in contrast with the bin-packing problem.

Categories and Subject Descriptors: G.2.1 [**Combinatorial Algorithms**]: Distribution and Maintenance—*documentation*

General Terms: Average Case, Experimental

Additional Key Words and Phrases: Bin packing, Sum of squares, memory allocation

Michael A. Bender, Department of Computer Science, State University of New York, Stony Brook, NY 11794-4400, USA. Supported in part by Sandia National Laboratories and NSF grants ACI-032497, CCR-0208670, and EIA-0112849.
Bryan Bradley, Department of Computer Science, Hofstra University, Hempstead, NY 11549 USA.
Geetha Jagannathan, Department of Computer Science, Stevens Institute of Technology, Hoboken, NJ 07030 USA. Part of this work was done at the State University of New York, Stony Brook. Supported in part by NFS grant CCR-0331584.
Krishnan Pillaipakkamnatt, Department of Computer Science, Hofstra University, Hempstead, NY 11549 USA.

1. INTRODUCTION

In classical bin packing the input is a list $L = (a_1, a_2, ..., a_n)$ of n items. Each item a_i has size $s(a_i)$, where $0 < s(a_i) \leq 1$. Given an infinite supply of bins of unit capacity, the objective is to pack the items into a minimum number of bins subject to the constraint that the sum of sizes of the items in each bin is no greater than 1. Bin packing has a wide range of applications including stock cutting, truck packing, commercials assignment to stations breaks in television programming, and memory allocation. Because this problem is NP-hard (Gary and Johnson, 1979), most bin-packing research concentrates on finding polynomial-time approximation algorithms for bin packing. Bin packing is among the earliest problems for which the performance of approximation algorithms was analyzed (Coffman, Garey and Johnson, 1996).

This paper investigates the average-case performance of online algorithms for bin packing, where item sizes are drawn according to some discrete distribution. An algorithm is said to be *online* if the algorithm packs each item as soon as it "arrives" without any knowledge of the items to arrive in the future. That is, the decision to pack item a_i into some particular bin is based only on the knowledge of items $a_1, ..., a_{i-1}$. Moreover, once an item has been assigned to a bin, it cannot be reassigned. Discrete distributions have the property that each item size is an element of some set $\{s_1, s_2, ..., s_J\}$ of integers, where $s_1 < s_2 \ldots < s_J$, and each size has an associated rational probability. The capacity of a bin is a fixed integer $B > s_J$.

We overload notation and write $s(L)$ for the sum of the sizes of the items in the list L. For an algorithm A, we use $A(L)$ to denote the number of bins used by A to pack the items in L. We write OPT to denote an optimal packing algorithm. Let F be a probability distribution over item sizes. Then $L_n(F)$ denotes a list of n items drawn according to distribution F. When F is clear from the context, we use L_n instead of $L_n(F)$. A *packing* is a specific assignment of items to bins. The *size* of a packing P, written $\|P\|$, is the number of bins used by P. For an algorithm A, we use $P_n^A(F)$ to denote a packing resulting from the application of algorithm A to the list $L_n(F)$. Given a packing P of a list L, the *waste* for P, the sum of the unused bin capacities, is defined as $W(P) = B\ \|P\| - s(L)$.

The *expected waste* of an algorithm A on a distribution F is

$$EW_n^A(F) = E[W(P_n^A(F))],$$

where the expectation is taken over the random variable $L_n(F)$.

1.1 Related Work

There are a substantial number of results for the worst-case analysis of algorithms for bin packing (Coffman, Garey and Johnson, 1996). There are also a number of results for the average-case analysis, though fewer than for worst-case analysis.

1.1.1 *Bin Packing.* The average-case analysis of the standard heuristics (Next Fit, Best Fit, and First Fit) has been performed for the discrete uniform distributions $U\{j, k\}$ where the bin capacity is taken as $B = k$, and item sizes are uniformly drawn from $1, 2, ..., j < k$. When $j = k - 1$, the online Best-Fit and First-Fit algorithms have $\Theta(\sqrt{n})$ expected waste (Coffman et al., 1991 and Coffman et al., 1997).

Remarkably, when $1 \leq j \leq k-2$, the expected waste of the optimal is $O(1)$ (Coffman et al., 1991 and Coffman et al., 2000). An algorithm is said to be *stable* under a distribution if the expected waste remains bounded even when the number of items goes to infinity. Coffman et al. (Coffman et al., 1991) proved that Best Fit is stable when $k \geq j(j+3)/2$. Coffman et al. (Coffman et al., 1993) showed that Best Fit is stable under $U\{k-2, k\}$ and is also stable for some specific values of (j, k) with $k \leq 14$. It has been shown experimentally that for most pairs (j, k) the expected waste of Best Fit is $\Theta(n)$ (Kenyon and Mitzenmacher, 2002).

1.1.2 *The Sum-of-Squares algorithm.* Here we describe the Sum-of-Squares algorithm for bin packing in discrete distributions. The *gap* of a bin is the amount of unassigned capacity in the bin. Let $N(g)$ denote the number of bins in the current packing with gap g, $1 \leq g < B$. Initially, $N(g) = 0$ for all g. The sum-of-squares algorithm puts an item a of size $s(a)$ into a bin such that after placing the item a the value of $\sum_{g=1}^{B-1} N(g)^2$ is minimized.

Csirik et al. (Csirik et al., 1999), gave experimental evidence that for discrete uniform distributions $U\{j, k\}$ (with $k = 100$ and $1 \leq j \leq 98$) EW_{SS}^n is $O(1)$. They also showed for $j = 99$, $EW_{SS}^n = O(\sqrt{n})$. Their results indicated that for $j = 97, 98$, and 99 $EW_{SS}^n(U\{j, 100\})$ has values $O(1)$, $O(1)$, $\Theta(\sqrt{n})$ whereas the expected waste of BF has values $\Theta(n)$, $O(1)$, $\Theta(\sqrt{n})$.

In a theoretical analysis of the sum-of-squares algorithm (Csirik et al., 2000) Csirik et al. proved that for any perfectly-packable distribution F, $EW_{SS}^n = O(\sqrt{n})$. They also proved that if F is a bounded waste distribution then EW_{SS}^n is either $O(1)$ or $\Theta(\log n)$. In particular, if F is a discrete uniform distribution $U\{j, k\}$ where $j < k-1$ then $EW_n^{SS}(F) = O(1)$. They also proved that for all lists L,

$$SS(L) \leq 3OPT(L).$$

1.1.3 *Memory Allocation.* Closely related to the bin-packing problem is the memory-allocation problem. Memory is modeled as an infinitely long array of storage locations. An allocator receives requests for blocks of memory of various sizes and requests for their deallocation arrive in some unknown order. Although the size of the request is known to the allocator, the deallocation time is unknown at the time of allocation. The deallocation of a block may leave a "hole" in the memory. The objective of a memory-allocation algorithm is to minimize the total amount of space wasted in these holes.

Although memory allocation has been studied since the early days of computing, only a handful of theoretical results concern the competitive ratios of the standard heuristics for the problem. For the memory-allocation problem, the competitive ratio of an online algorithm is the ratio between the total amount of memory required by the algorithm to satisfy all requests to W, the largest amount of concurrently allocated memory. Luby, Naor, and Orda (Luby, Naor and Orda, 1996) showed that First Fit (the heuristic which assigns a request to the lowest indexed hole that can accommodate the request) has a competitive ratio of $O\{\min(\log W, \log C)\}$, where C is the largest number of concurrent allocations. By Robson's result (Robson, 1974), this bound is the best possible value for any online algorithm. Robson also shows that the worst-case performance of Best Fit (the heuristic that assigns a request to a smallest hole that is large enough to accommodate it) is $O(Wr)$ where r is the ratio of the sizes of the largest and smallest requests.

A large body of literature exists for the empirical evaluations of memory allocation algorithms. A thorough, if slightly dated, review of results for memory allocation can be found in (Wilson et al., 1995). According to the taxonomy of memory allocation algorithms in that review, our extension of the Sum-of-Squares bin packing algorithm to memory allocation can be categorized as a Sequential-Fit algorithm.

Traditionally memory-allocation algorithms were empirically studied using random simulations that created synthetic traces (sequences of allocations and deallocations) using distributions that were presumed to resemble the memory allocation patterns of real programs. The main argument against this way of evaluating allocators is that real programs usually allocate memory in a limited number of sizes, and with very different probabilities for each size (Zorn and Grunwald, 1994 and Wilson et al., 1995). The emphasis lately has been on evaluating memory allocation algorithms using real trace data (Johnstone, 1997), although carefully chosen distributions continue to be used as well (Berger et al., 2000).

Although a number of application-specific memory allocators have been developed, it has been empirically shown that in most cases general purpose allocators perform at least as well as or better than these custom allocators (Berger et al., 2002). Further, it has been observed that for most real-world allocation patterns the standard heuristics have very low “wastage” due to fragmentation (Johnstone and Wilson, 1998). Doug Lea’s algorithm for memory allocation (Lea) is considered to be one of the best algorithms for memory allocation. It uses an approximation of the Best-Fit heuristic.

1.2 Results

In this paper we study the performance of the sum-of-squares algorithm and its variants. We present faster variants of the sum-of-squares algorithm for the online bin-packing problem. We also show the results of applying the sum-of-squares algorithm to memory allocation.

We performed our experiments with the uniform distribution $U\{j, k\}$. We have also run our algorithms on interval distributions and other discrete distributions, especially those that are claimed to be difficult in (Csirik et al., 2006). For the memory-allocation problem we used both real traces from allocation-intensive programs and synthetically generated traces that model memory allocation behavior of real programs.

—In Section 2 we present our variants for the sum-of-squares algorithm. The first is the SSmax($2\sqrt{B}$) variant, which runs in $O(n\sqrt{B}\log B)$ time. Our experiments suggest that the performance of this algorithm is close to the SS algorithm, for all the distributions mentioned in (Csirik et al., 1999, Csirik et al., 2000 and Csirik et al., 2006). The remaining algorithms form a family called the segregated sum-of-squares algorithms (SSS). These algorithms perform well in most of the distributions mentioned in the above papers. But on some distributions they do not have the same expected waste as SS. The best runtime in this family is

$O(n \log B)$.

—In Section 3 we applied SS to the related problem of online memory allocation. Our experimental comparisons between SS and Best Fit (which has been experimentally observed to have among the least amount of fragmentation for real traces) indicate that neither algorithm is consistently better than the other. Smaller allocation durations appear to favor Best Fit, while larger allocation durations favor SS. Also, if the number of possible request sizes is low (as happens in most real-world situations), SS appears to have lower waste than Best Fit, while larger amounts of randomness appear to favor Best Fit.

For interval distributions an interesting phenomenon we observed is that for every possible range of request sizes there exists a lower bound on the time duration from which point onwards the sum-of-squares algorithm has lower waste than Best Fit. In the online memory-allocation problem SS does not seem to have a consistent asymptotic advantage over Best Fit, in contrast to the bin-packing problem.

2. FASTER VARIANTS OF THE SUM-OF-SQUARES ALGORITHM

In this section we present variants of the sum-of-squares algorithm. The SSmax($2\sqrt{B}$) variant of Section 2.2.1 runs in $O(n\sqrt{B}\log B)$ time and appears to have an expected waste remarkably close to that of SS. Experimental results indicate that the segregated sum-of-squares family of algorithms (Section 2.2.2) run faster, but in some cases, have $\Theta(n)$ expected waste when SS has $\Theta(\sqrt{n})$ waste.

2.1 Sum-of-Squares Algorithm

The sum of the sizes of items in a bin is the *level* of the bin. The *gap* of a bin is the amount of its unused capacity. If the level of a bin is ℓ, then its gap is $B - \ell$. Let P be a packing of a list L of items. Let the *gap count of* g, $N(g)$, denote the number of bins in the packing that have a gap of g, $1 \le g < B$. We call N the *profile vector* for the packing. We ignore perfectly packed bins (whose gap is 0) and completely empty bins. The sum-of-squares for a packing P is $ss(P) = \sum_{g=1}^{B-1} N(g)^2$.

The sum-of-squares algorithm is an online algorithm that works as follows. Let a be the next item to pack. It is packed into a legal bin (whose gap is at least $s(a)$) such that for the resulting packing P' the value of $ss(P')$ is minimum over all possible packings of a.

When an item of size s arrives, there are three possible ways of packing the item

(1) Open a new bin: Here the sum of squares increases by $1 + 2N(B - s)$.

(2) Perfectly fill an old bin: Here the sum of squares decreases by $2N(s) - 1$.

(3) Put the item into a bin of gap g where $g > s$: Here the sum of squares increases by $2 \cdot (N(g - s) - N(g) + 1)$. This step requires finding a g which maximizes the value of $N(g) - N(g - s)$.

Each time an item arrives, the algorithm performs an exhaustive search to evaluate the change in $ss(P)$ and finds an appropriate bin in $O(B)$ time. In (Csirik et al., 2006) the authors discuss variants of the original SS algorithm. They present $O(n \log B)$ and $O(n)$ variants that approximate the calculation of the sum-of-squares. The authors prove that these variants have the same asymptotic growth rate for expected waste as SS, but with larger multiplicative constants. For example, they considered the distribution $U\{400, 1000\}$. For $n = 100,000$ items, they state that the number of bins used by the variant is 400% more than optimal, whereas Best Fit uses only 0.3% more bins, and SS uses 0.25% more bins than necessary. The situation does improve in favor of their variant for larger values of n. For $n = 10^7$, the variant uses roughly 9.8% more bins than optimal, but SS uses 0.0025% more bins than necessary. The authors claim their fast variants of SS are "primarily of theoretical interest" and that they are unlikely to be competitive with Best Fit except for large values of n.

They also looked at other generalizations of the sum-of-squares algorithm. Instead of considering the squares of the elements of the profile vector, they examined the *rth* power, for $r \geq 1$. These generalizations yield various SrS algorithms. For all finite $r > 1$, SrS performed qualitatively the same as SS. For the limit case of $r = 1$ (S1S), the resulting algorithm is one that satisfies the *any-fit* constraint. That is, the algorithm does not open a new bin unless none of the already open bins can accommodate the new item to be packed. Best Fit and First Fit are examples of algorithms that satisfy the any-fit constraint. The S∞S algorithm chooses to minimize the largest value in the profile vector. This variant has been empirically observed to have an optimal waste for the uniform distributions $U\{j, k\}$. However, the authors provide examples of distributions for which S∞S has linear waste while the optimal and SS have sublinear waste.

2.2 Our Variants

Our variants try to approximate the behavior of the sum-of-squares algorithm in order to decrease the runtime while retaining the performance in qualitative terms. By attempting to minimize the value of $ss(P')$, SS effectively tries to ensure that it has an approximately equal number of bins of each gap g, where $1 \leq g < B$. Intuitively, the value of $ss(P')$ can usually be reduced by lowering the largest gap count. However, minimizing the largest value in the profile vector *alone* is insufficient, as the S∞S experiments indicate (Csirik et al., 2006). Nevertheless, this insight plays a key role in our variants for the sum-of-squares algorithm. Instead of examining all possible legal gaps for the best possible gap (as SS does), we examine only the k largest gap counts, for some value of k.

2.2.1 *Parameterized SSmax Algorithm.* The SSmax(k) parameterized algorithm is based on the S∞S variant mentioned above. The right choice for k is discussed later in this section. To pack an item a, SSmax(k) considers a limited set of options:

—place a in an empty bin,

—place it in a non-empty bin with gap $s(a)$ if one exists, thus perfectly packing the bin, or

—place a in a bin whose gap is one of the k highest gap counts (among those gaps whose sizes are at least as big as $s(a)$).

It chooses that option which minimizes the resulting value of $ss(P)$. (Note that when we set $k = B$, we get the original SS algorithm. Also note that SSmax(1)

Distribution	**Gaps ranked $\leq 2\sqrt{B}$ (%)**	**New bins included (%)**
$U\{10, 100\}$	94.49	100
$U\{30, 100\}$	84.50	100
$U\{60, 100\}$	69.50	100
$U\{90, 100\}$	51.82	97.33
$U\{97, 100\}$	46.68	95.74
$U\{98, 100\}$	48.25	97.83
$U\{99, 100\}$	49.34	99.44
$U\{18..27, 100\}$	45.20	67.70
$U\{11..13, 15..18, 51\}$	71.41	100

Table I. Fraction of gaps ranked $\leq 2\sqrt{B}$ selected by SS when packing a new item.

is not the same as S∞S, given that the latter does not consider the gap with size $s(a)$ unless that gap has the highest count, nor does it start a new bin unless this is unavoidable.) Using a tournament tree (Horowitz, Mehta and Sahni, 2006), we can find the k largest gap counts in the profile vector in $O(k \log B)$.

Experimental Results. When given a new item to pack, SS performs an exhaustive search to find a gap which minimizes $ss(P)$. This takes $O(B)$ time per item. Our goal was to create an algorithm that approximates the performance of SS by computing the change in $ss(P)$ at some $k = o(B)$ gaps. Although the S∞S algorithm fails to perform as well as SS on some distributions, it does suggest that it might be worthwhile to examine the largest gap counts. To this end, we performed the following experiment: We dynamically ordered gaps by decreasing gap counts and tracked the ranks of the gaps chosen by SS when it packed items. We observed in almost all the distributions we examined that when a new item arrives SS almost always choses either to:

(1) use a gap that had a rank $\leq 2\sqrt{B}$, or
(2) open a new bin.

(See Table I for some uniform and interval distributions.) Further, for the uniform distributions $U\{j, B\}$, and especially when $j << B$, SS chose the highest ranked gap to accommodate new items. This explains why S∞S works as well as SS for uniform distributions. We ran most of our experiments on SSmax with k set to $2\sqrt{B}$.

We selected a number of distributions on which to test the performance of SSmax($2\sqrt{B}$). Our first category of distributions included the uniform and interval distributions used by Csirik et al. (Csirik et al., 1999) to compare SS against Best Fit. The second category included all distributions mentioned in (Csirik et al., 2006) in which S∞S had linear waste but SS has sublinear waste.

Of particular interest are the interval distributions $U\{18..j, 100\}$, for $18 \leq j \leq 99$. In (Csirik et al., 1999) the authors present a figure that shows the optimal waste for these distributions. As j increases from 18 to 99, the optimal waste undergoes a large number of transitions. We wanted to verify that SSmax($2\sqrt{B}$) has the same order of waste as SS though all the transitions.

We also considered uniform distributions, but with large bin sizes to see ensure that the choice of $k = 2\sqrt{B}$ was not merely applicable to small values of B. Note that for very small values of B, the value of $2\sqrt{B}$ is fairly close to that of B. As one

Item sizes: 11-13,15-18, Bin size: 51	10^5	10^6	10^7	10^8	10^9
SS	9538	31272	97650	300121	516184
SSmax($2\sqrt{B}$)	9559	31314	97665	300172	516388
SSmax(1)	316247	3161114	3.159×10^7		
SSS(sqrt)	19259	144570	1287001	12281857	1.237×10^8
18-27, Bin size: 100	10^5	10^6	10^7	10^8	10^9
SS	8639	25454	76259	229869	622198
SSmax($2\sqrt{B}$)	12333	36082	108789	327609	943798
SSmax(1)	939548	9409415	9.399×10^7		
SSS(sqrt)	146468	1447068	1.444×10^7	1.443×10^8	
1,11-13,15-18, Bin size: 51	10^5	10^6	10^7	10^8	10^9
SS	585	595	635	414	628
SSmax($2\sqrt{B}$)	525	537	594	465	577
SSmax(1)	26422	2639009	2.64×10^7		
SSS(sqrt)	1130	1236	1278	1026	1648

Table II. Comparison of waste for SS, SSmax($2\sqrt{B}$), SSmax(1), and SSS(sqrt) for various discrete distributions with the number of items $n \in \{10^5, 10^6, 10^7, 10^8, 10^9\}$. The waste of SS and SSmax($2\sqrt{B}$) are of the same order of magnitude. SSmax(1) has linear waste for all three distributions. SSS(sqrt) has linear waste for the first two distributions while SS has sublinear waste.

would expect in such cases, the performance of SSmax($2\sqrt{B}$) is virtually identical to that of SS.

Finally, we ran experiments to see if SSmax(k) for k much smaller than $2\sqrt{B}$ would perform just as well as SSmax($2\sqrt{B}$). In these experiments k took values in $\{1, \log B, \sqrt{B}\}$.

For each of the distributions mentioned above, we observed that the performance of SSmax($2\sqrt{B}$) matched that of SS to within a factor of 1.4. In many cases the factor is as low as 1.02. (See Table II and Figure 1 for results from some of the distributions.)

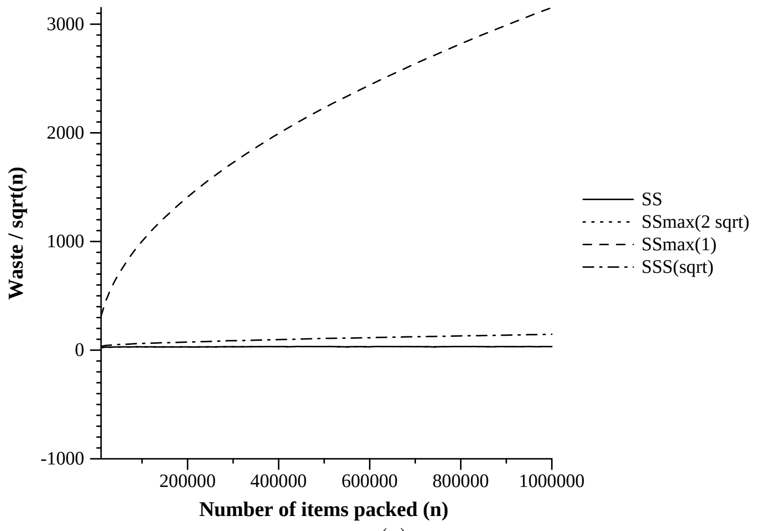
Note that in Table II and other tables when it becomes clear that the growth rate of an algorithm is linear in n, we did not run experiments for $n = 10^8$ and $n = 10^9$. In all other cases, we used 100 samples for $n \in \{10^5, 10^6\}$, 25 samples for $n \in \{10^7, 10^8\}$, and 3 samples for $n = 10^9$.

Table III shows that for the interval distributions $U\{18..j, 100\}$, for $18 \leq j \leq 99$, SSmax($2\sqrt{B}$) does indeed track the performance of SS. Setting k to $\sqrt{B}$, $\log B$ or 1 results in SSmax(k) having suboptimal waste for one or more distributions.

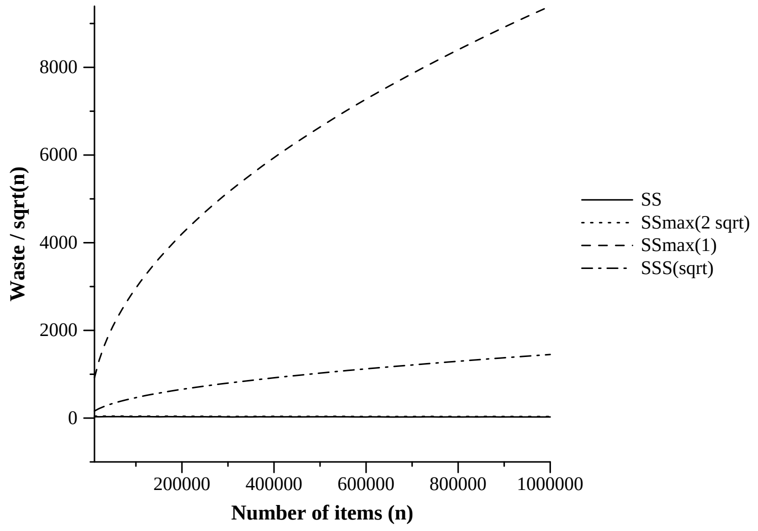
We also compared the waste of this algorithm with SS for the distributions $U\{j, 100\}$, $U\{j, 200\}$ and $U\{j, 1000\}$. As our results in Table IV, and Figure 2 show, the performance of the SSmax($2\sqrt{B}$) algorithm is remarkably similar to that of the SS algorithm.

Our experiments thus indicate that the sum-of-squares minimization is sufficiently robust that it is not necessary to compute it precisely to receive its benefits.

2.2.2 *Segregated Sum of Squares Variants.* All Segregated Sum-of-Squares algorithms (SSS) divide the profile vector N into some number t of approximately equal-sized contiguous sections of the vector. Each section thus has at most $\lceil B/t \rceil$ gaps. For each section the algorithm maintains the gaps in a heap, whose top el-

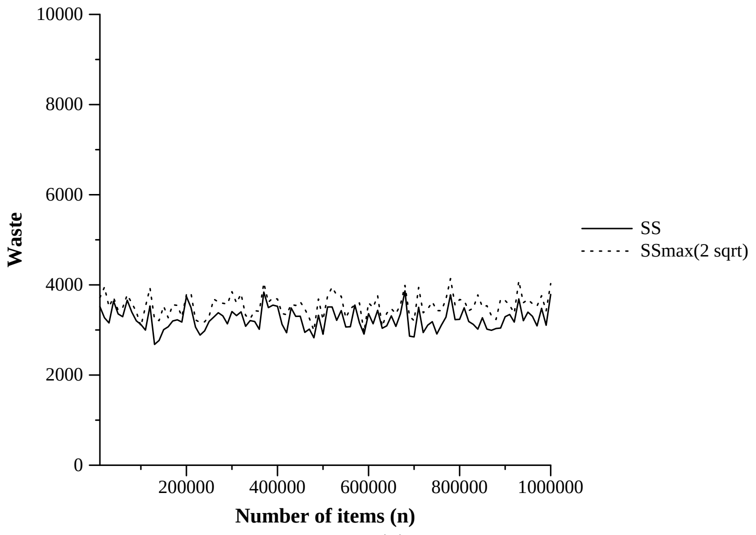


(a)

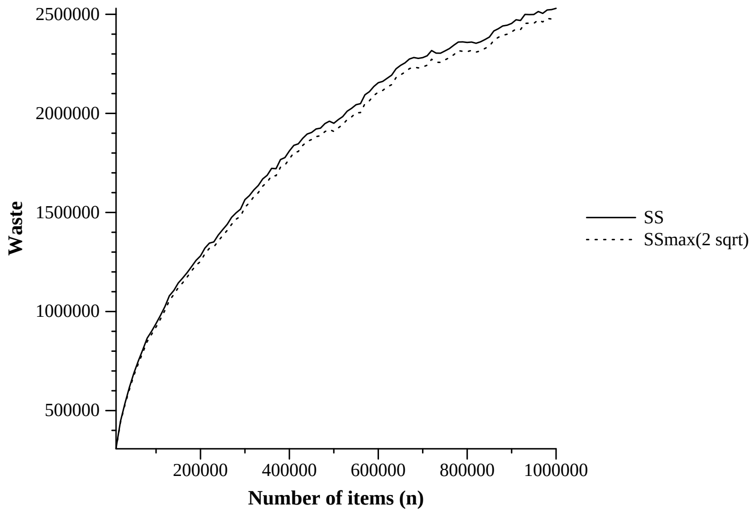


(b)

Fig. 1. Comparison of waste for SS, SSmax($2\sqrt{B}$), SSmax(1), and SSS(sqrt). (a) Item sizes uniformly drawn in the range 11-13,15-18 with bin size = 51 (block 1 of Table II). (b) Item sizes uniformly drawn in the range 18–27 with bin size = 100 (block 2 of Table II). In both figures, the curves represent plots for the number of items packed vs waste/sqrt(number of items). In both (a) and in (b), SS and SSmax($2\sqrt{B}$) are horizontal lines and are indistinguishable, and hence they are $\Theta(\sqrt{n})$ waste. Other algorithms in these graphs have linear waste.



(a)



(b)

Fig. 2. Comparison of waste for SS and SSmax($2\sqrt{B}$). In both figures, the curves represent plots for the number of items packed vs waste. (a) Item sizes are uniformly drawn in the range 1–150 with bin size = 200 (block 2 of Table IV). Both SS and SSmax($2\sqrt{B}$) appear to have constant waste. (b) Item sizes are uniformly drawn in the range 1–990 with bin size = 1000 (block 3 of Table IV). The waste of SS and SSmax($2\sqrt{B}$) are of the same order of magnitude.

Distrib. $U\{18..j, 100\}$	**SS**	**SSmax**			
		$2\sqrt{B}$	$\sqrt{B}$	$\log B$	1
$j = 18$ to $j = 21$	n	n	n	n	n
$j = 22$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	n
$j = 23$ to $j = 26$	n	n	n	n	n
$j = 27$	$\sqrt{n}$	$\sqrt{n}$	n	n	n
$j = 28$ to $j = 29$	$\log n$	$\log n$	$\log n$	n	n
$j = 30$ to $j = 31$	$\log n$	$\log n$	$\log n$	$\log n$	n
$j = 32$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	n
$j = 33$	n	n	n	n	n
$j = 34$ to $j = 35$	$\log n$	$\log n$	$\log n$	n	n
$j = 36$	$\log n$	$\log n$	$\log n$	1	n
$j = 37$	$\log n$	$\log n$	$\log n$	$\log n$	1
$j = 38$	1	1	1	1	1
$j = 39$ to $j = 48$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
$j = 49$ to $j = 50$	n	n	n	n	n
$j = 51$	$\log n$	$\log n$	$\log n$	1	$\log n$
$j = 52$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
$j = 53$	$\log n$	$\log n$	$\log n$	1	$\log n$
$j = 54$	1	1	1	$\log n$	1
$j = 55$ to $j = 61$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
$j = 62$ to $j = 64$	$\log n$	$\log n$	$\log n$	$\log n$	n
$j = 65$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	n	n
$j = 66$	n	n	n	n	n
$j = 67$ to $j = 69$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$
$j = 70$ to $j = 72$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	n
$j = 73$ to $j = 76$	n	n	n	n	n
$j = 77$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$
$j = 78$ to $j = 81$	n	n	n	n	n
$j = 82$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$
$j = 83$ to $j = 99$	n	n	n	n	n

Table III. Orders of magnitude of wastes measured under distributions $U\{18..j, 100\}$.

ement is a gap with the largest count. To pack an item a of size s, the algorithm considers the following options:

—place a in an empty bin,

—place it in a non-empty bin with gap s if one exists, thus perfectly packing the bin,

—place a in a bin with gap in the set $\{s+1, s+2, \ldots, s+t\}$, or

—place a in a bin whose gap is the top element in a section that contains gaps larger than s.

It chooses that option which minimizes the resulting value of $ss(P)$. Thus, the best gap size can be selected in $\Theta(t)$ steps. The choices of t differentiate the various members of the SSS family. The two values of t we experimented with were $\sqrt{B}$ and $\log B$.

Experimental Results. We ran our experiments on the $U\{j,k\}$ distributions. For $B = k = 100$, for each value of j from 1 to 99, and for each of $n \in \{10^5, 10^6, 10^7, 10^8, 10^9\}$, we computed the average waste of SS, BF, SSS(sqrt), and SSS(log). We

U{j,100} $k = 2\sqrt{B}$

j=12	SS	SSmax(k)	j=75	SS	SSmax(k)	j=99	SS	SSmax(k)
10^5	40	42		900	974		32302	32287
10^6	47	57		916	982		108251	108176
10^7	52	52		680	776		308919	308223
10^8	45	45		923	1014		876747	876354

U{j,200} $k = 2\sqrt{B}$

j=24	SS	SSmax(k)	j=150	SS	SSmax(k)	j=198	SS	SSmax(k)
10^5	117	117		3094	3574		82875	83680
10^6	91	91		2691	2895		193144	193332
10^7	63	63		3920	3723		541186	538786
10^8	78	78		3639	3684		1908245	1908852

U{j,1000} $k = 2\sqrt{B}$

j=120	SS	SSmax(k)	j=400	SS	SSmax(k)	j=990	SS	SSmax(k)
10^5	253	253		4432	8062		977566	959566
10^6	876	876		5139	8328		2552676	2498732
10^7	580	580		5020	8031		5201380	5086175
10^8	672	672		4882	8081			

Table IV. Comparison of waste for SS and SSmax($2\sqrt{B}$) for various uniform distributions $U\{j,k\}$. Here the item sizes are drawn at random between $1 \ldots j$ and k denotes the bin size. The number of items $n \in \{10^5, 10^6, 10^7, 10^8\}$. The waste of SS and SSmax($2\sqrt{B}$) are of the same order of magnitude. For $j = 99$, both SS and SSmax($2\sqrt{B}$) have $\Theta(\sqrt{n})$ waste. In all the other cases both algorithms have constant waste.

averaged the waste over 100 runs for $n \in \{10^5, 10^6\}$, over 25 runs for $n \in \{10^7, 10^8\}$, and over 3 runs for $n = 10^9$. We know from (Coffman et al., 2000) that $EW^n_{OPT} = O(1)$ for $j \leq k - 2 = 98$, and $EW^n_{OPT} = \Theta(\sqrt{n})$ when $j = k - 1 = 99$. We have summarized the results in Table V, showing the waste for j at 12, 25, 75, 97, 98 and 99.

The experiments show all versions of the SSS algorithms have the constant-waste property for $j \leq 98$, and appear to have qualitatively the same waste as SS for $j = 99$. Both SSS algorithms appear to be within 0.80% of SS in terms of waste.

The SSS variants, however, do not fare as well on some of the interval distributions. Table II suggests for some of the distributions, SS has sublinear waste, while the SSS($\sqrt{B}$) algorithm appears to exhibit linear waste.

The variants we have presented are uncomplicated algorithms that are easily implemented.

3. MEMORY ALLOCATION

Researchers have long observed the similarity between bin packing and memory allocation (Dyckhoff, 1990, Coffman and Leung, 1977, and Coffman and Leighton, 1986). A number of heuristics that apply to bin packing translate directly to memory allocation and vice-versa. Since the sum-of-squares algorithm is effective for bin packing (Csirik et al., 2006), we also study the algorithm in the context of memory allocation.

j=12	10^5	10^6	10^7	10^8	10^9
SS	46	48	53	50	18
SSS(sqrt)	46	49	53	50	18
SSS(log)	46	48	53	50	18
BF	46	49	53	50	18

j=25	10^5	10^6	10^7	10^8	10^9
SS	60	62	58	53	68
SSS(sqrt)	61	63	58	53	68
SSS(log)	61	63	58	53	68
BF	177	834	7322	71893	728368

j=75	10^5	10^6	10^7	10^8	10^9
SS	920	902	917	873	1041
SSS(sqrt)	934	906	927	902	1025
SSS(log)	1003	953	973	933	1008
BF	43448	425606	4243029	42460163	424241908

j=97	10^5	10^6	10^7	10^8	10^9
SS	22546	44786	70274	69163	75352
SSS(sqrt)	22456	44721	70604	69663	75932
SSS(log)	22171	44391	70244	69493	75452
BF	23176	53006	100244	1791953	1.76610^7

j=98	10^5	10^6	10^7	10^8	10^9
SS	25863	69799	153116	231056	254947
SSS(sqrt)	25523	69639	153076	232996	256707
SSS(log)	25453	69224	152036	231396	255127
BF	24068	78129	178346	254016	214507

j=99	10^5	10^6	10^7	10^8	10^9
SS	33996	107707	354053	1089447	2848891
SSS(sqrt)	33772	107207	354003	1079132	2484005
SSS(log)	33651	106743	352245	1076027	2832558
BF	26330	88169	302489	950047	27843191

Table V. Comparison of SSS algorithms with SS and BF for various uniform distributions, where items sizes are drawn in the ranges $1 \dots j$ for $j = 12, 25, 75, 97, 98, 99$ and bin size = 100, and the number of items $n \in \{10^5, 10^6, 10^7, 10^8, 10^9\}$. All variants of SS have waste of the same order of magnitude as SS. BF has linear waste when $j \in \{25, 75, 97, 98\}$, while SS has sublinear waste for those values of j.

3.1 Memory Allocation

Closely related to the bin-packing problem is the *memory-allocation* problem in paged-memory operating systems. In this online problem, memory is represented as an infinitely long array of storage locations. Requests for blocks of memory of various sizes, and requests for their deallocation arrive over time. Although the size of the request is known to the allocator, the deallocation time is unknown at the time of allocation. When an allocation request arrives, it must be satisfied by assigning a contiguous set of free memory locations. Once an allocation has been made, the block cannot be moved until it is deallocated. The deallocation of a block may leave a "hole" in the memory. The objective of a memory allocation-algorithm is to minimize the total amount of space wasted in these holes.

3.2 Sum-of-Squares Algorithm for Memory Allocation

The algorithm maintains a list L of free blocks of memory. It also maintains a profile vector P, which lists the number of free blocks of each size. (The block sizes that have a zero count are not explicitly stored in the vector.) When a new request for allocation arrives, the algorithm computes, for each block size that is sufficiently large to satisfy the request and has nonzero count, the sum-of-squares that would result of a block that size were used to satisfy the request. The algorithm selects a block that has the minimum sum-of-squares value. When a free block is selected the requested memory is allocated from the beginning of the block. The selected block is deleted from the list L and a new block that represents the left-over portion of the block is inserted into L. If a request does not fit into any of the free blocks, then we "enlarge" memory to accommodate the new request. When a deallocation request arrives, we obtain a free block and it is added into L. If the newly deallocated block is contiguous with a previously deallocated block (or blocks) in L, these blocks are merged to form one larger free block. Free blocks at the end of memory are discarded and the memory "shrinks" to end at the last allocated block. In all cases, the profile vector P is updated appropriately.

3.3 Experimental Results

We compared the performance of the SS algorithm for memory allocation against Best Fit, which is among the best-performing algorithms for memory allocation (Berger et al., 2002, and Johnstone and Wilson 1998). Two of our experiments involved real traces from allocation-intensive programs, namely espresso, a program for logic circuit optimization, and cfrac, a program for factoring integers based on the method of continued fractions. For espresso we used the largest sample input file provided with the software. For cfrac we used as input a 52 digit integer. The rest of the experiments were based on synthetic traces. The parameters for our experiments in memory allocation were

—the distribution of request sizes, and

—the distribution of request durations.

Traditionally, synthetic traces for memory allocation have used exponential distributions for request sizes and distributions. However, according to (Zorn and Grunwald, 1994), interval distributions (as represented by their MEAN model) are almost as accurate as more complex distributions in modeling the memory allocation behavior of real programs. Further, according to (Wilson et al., 1995), exponential distributions are unrealistic in modeling request durations. Therefore, we restricted our experiments to interval distributions.

For our experiments with real traces we compared the maximum amount of live data in a run against the maximum amount of memory used by the allocator in that run. This is similar to one of the measures of fragmentation in (Johnstone and Wilson, 1998). Note that these events may not occur at the same time in a run.

Table VI shows the results of our experiments. We observed very similar performance for both algorithms. CFRAC does not deallocate most of the blocks it allocates—this explains the very low fragmentation.

Program	**Maximum live memory**	**Max. usage (SS)**	**Max. usage (BF)**	**Frag. (SS)**	**Frag. (BF)**
Espresso	158187	168624	168396	6.59%	6.45 %
Cfrac	2413594	2413597	2413597	0.00012%	0.00012%

Table VI. Comparison of SS and Best Fit for real traces.

For the synthetic traces we measured the waste of the algorithm (as the sum of the blocks on the free list) after each request was assigned a block of memory. The waste of the algorithm for a run is the maximum waste taken over all the allocation requests. While this might seem like an unconventional measure of performance for memory allocation algorithms, it is indeed a reasonable measure because, after a while, allocation and deallocation requests result in a "stable state" where the amount of new blocks being created is almost exactly matched by existing blocks being deleted.

The value of the request size, B, ranged from 1 to 2000. The values of the request duration, t, ranged from 1 to 21,000.

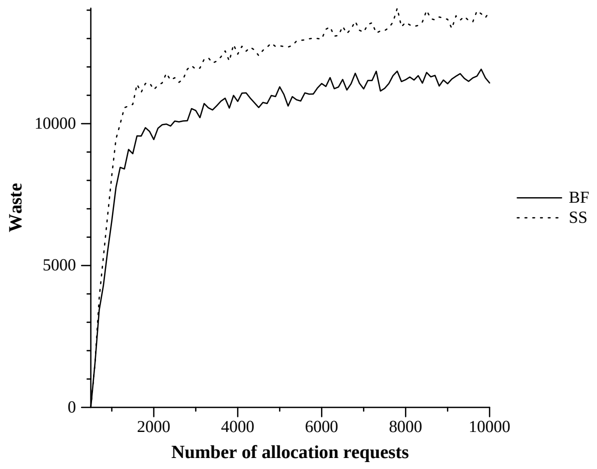
The experiments indicate that neither algorithm is consistently of lower waste than the other for these distributions. Even when an algorithm has a lower waste than the other, both algorithms seem to have waste of the same order of magnitude. Figure 3(a) is a typical example of the scenario where BF has lower waste than SS. Our experiments suggest that BF appears to have an advantage over SS when the duration of the requests are small. Figure 3(b) is a typical example of the scenario where SS has lower waste than BF. Figure 4(a) shows the difference in waste between the two algorithms as the request size increases. The difference between the waste of BF and that of SS also increases as the longevity of the allocation requests increases (Figure 4(b)). Our experiments also suggest that for every possible range of request sizes there exists a lower bound on the time duration from which point onwards the sum-of-squares algorithm has lower waste than Best Fit (Figure 5).

Our experiments suggest that in the online memory-allocation problem SS does not seem to have an asymptotic advantage over Best Fit for the uniform and interval distributions, in contrast with the bin-packing problem. In bin packing SS performs well since it tries to maintain an equal number of bins for each gap size. So when an item arrives, the allocator finds a suitable gap size and is able to pack a bin perfectly.

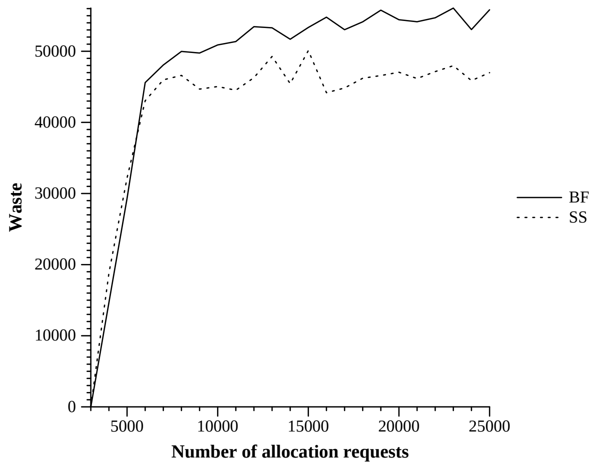
A possible explanation for the behavior of the algorithms indicated by our experiments is as follows. In our experiments with synthetic traces the request size and the duration are uncorrelated (they are independent random variables). So when the system reaches a steady state where the rate of allocation and deallocation are the same, most of the time the Best-Fit allocator is able to find a hole of an appropriate size for any request. Deallocation provides Best Fit the same advantage that helps SS perform well in bin packing.

Acknowledgment

We wish to thank the anonymous referee for his or her valuable comments.

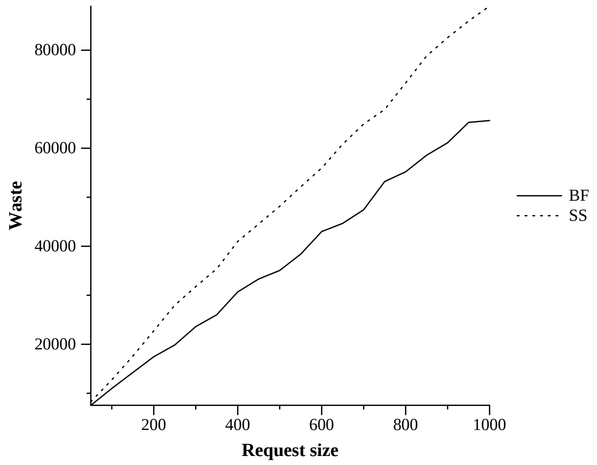


(a)

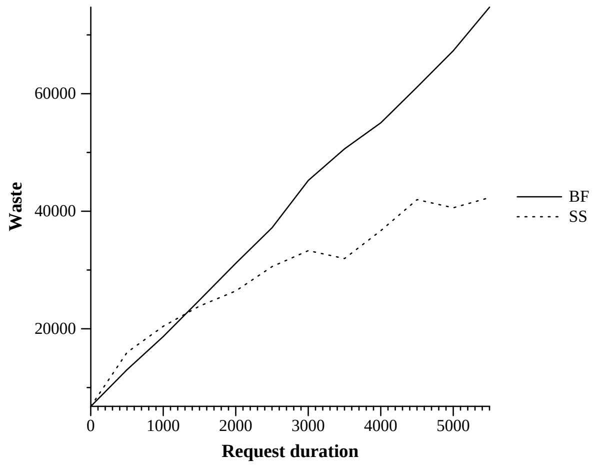


(b)

Fig. 3. Comparison of the waste of Best Fit against Sum of Squares for memory allocation. These are plots of the number of allocation requests vs. waste. (a) Duration range is 500-599, and size range is 100-199. The size range and the duration range are fixed. Best Fit has lower waste, but the waste appears to be of the same order as for Sum of Squares. (b) Duration range is 3000-3099, size range is 150-249. The size range and the duration range are fixed. Best Fit has higher waste, but the waste appears to be of the same order as for Sum of Squares.



(a)



(b)

Fig. 4. Comparison of waste of Best Fit and Sum of Squares for memory allocation. (a) The is a plot of request size vs. waste. Duration range is 500-599. The number of allocation requests is 5000. The duration range and the number of allocation requests are fixed. The difference in waste increases as the block size increases. (b) This is a plot of request duration vs. waste. Block size range is 101-200. The number of allocation requests is 20,000. The difference between the waste of BF and SS increases as item duration increases.

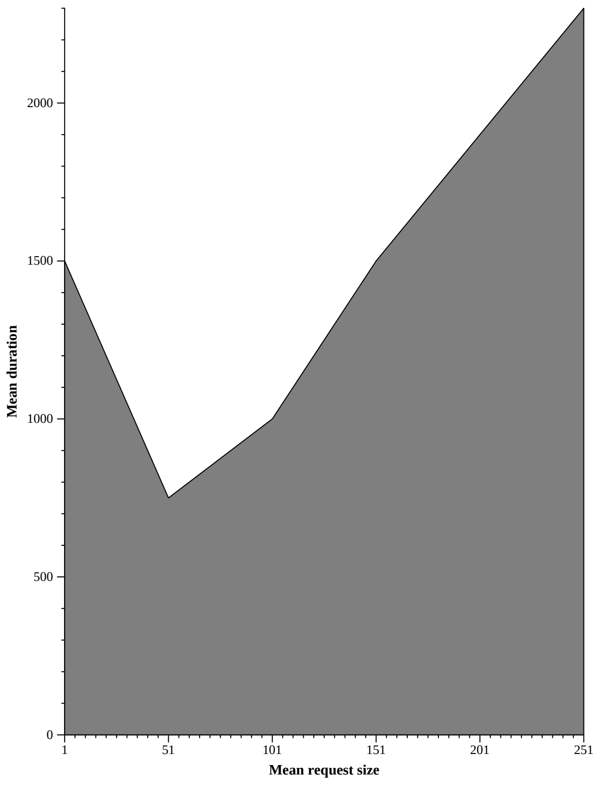


Fig. 5. Comparison of SS and BF over a range of parameters of request sizes and durations. For all points above the curve SS has lower waste than BF. For all points below the curve, BF has lower waste than SS.

REFERENCES

Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, 2000.

Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–12, New York, NY, USA, 2002. ACM Press.

E.G. Coffman, C. Courcoubetis, M. R. Garey, D. S. Johnson, L. A. McGeogh, P. W. Shor R. R. Weber, and M. Yannakakis. Fundamental discrepancies between average-case analyses under discrete and continuous distributions. In *Proceedings of the 23rd Annual Symposium on Theory of Computing*, pages 230–240, 1991.

E. G. Coffman, C. Courcoubetis, M. R. Garey, D. S. Johnson, P. W. Shor R. R. Weber, and M. Yannakakis. Bin packing with discrete item sizes, part I: Perfect packing theorems and average case behavior of optimal packings. *SIAM Journal on Discrete Math.*, 13:384–402, 2000.

E. G. Coffman, M. R. Garey, and D. S. Johnson. *Approximation Algorithm for NP-Hard Problems*, chapter Approximation Algorithms for Bin Packing: A Survey, pages 46–93. 1996.

J. Csirik, D. S. Johnson, C. Kenyon, P. W. Shor, and R. R. Weber. A self organizing bin packing heuristic. In *Proceedings of ALENEX workshop*, pages 246–265, 1999.

J. Csirik, D. S. Johnson, C. Kenyon, J. Orlin, P. W. Shor, and R. R. Weber. On the sum-of-squares algorithm for bin packing. In *Proceedings of the 32nd Annual ACM Symp. on the Theory of Computing*, pages 208–217, 2000.

Janos Csirik, David S. Johnson, Claire Kenyon, James B. Orlin, Peter W. Shor, and Richard R. Weber. On the sum-of-squares algorithm for bin packing. *J. ACM*, 53(1):1–65, 2006.

E. G. Coffman, Jr. and J. Y. Leung. Combinatorial analysis of an efficient algorithm for processor and storage allocation. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 214–221, 1977.

E. G. Coffman, Jr. and F. T. Leighton. A provably efficient algorithm for dynamic storage allocation. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 77–90, 1986.

H. Dyckhoff. A typology of cutting and packing problem sacking. *European Journal of Operational Research*, 44:145–159, 1990.

Jr. E. G. Coffman, D. S. Johnson, P. W. Shor, and R. R. Weber. Markov chains, computer proofs, and average-case analysis of best fit bin packing. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 412–421, New York, NY, USA, 1993. ACM Press.

M. R. Garey and D. S. Johnson. *Computers and Intractability A Guide to the Theory of NP-Completeness.* 1979.

Ellis Horowitz, Dinesh Mehta, and Sartaj Sahni. *Fundamentals of Data Structures in C++*. W. H. Freeman & Co., New York, NY, USA, 2006.

E. G. Coffman Jr., D. S. Johnson, P. W. Shor, and R. R. Weber. Bin packing with discrete item sizes, part ii: Tight bounds on first fit. *Random Structures and Algorithms*, 10(1–2):69–101, 1997.

Mark Stuart Johnstone. *Non-compacting memory allocation and real-time garbage collection.* PhD thesis, 1997. Supervisor-Paul R. Wilson.

Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: solved? In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 26–36, New York, NY, USA, 1998. ACM Press.

C. Kenyon and M. Mitzenmacher. Linear waste of best fit bin packing on skewed distributions. *Random Struct. Algorithms*, 20(3):441–464, 2002.

D. Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html.

M. G. Luby, J. Naor, and A. Orda. Tight bounds for dynamic storage allocation. *SIAM Journal on Discrete Math.*, 9:155–166, 1996.

J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM.*, 12:491–499, 1974.

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

Benjamin Zorn and Dirk Grunwald. Evaluating models of memory allocation. *ACM Trans. Model. Comput. Simul.*, 4(1):107–131, 1994.

Received Month Year; revised Month Year; accepted Month Year