# Toward Effective Multi-capacity Resource Allocation in Distributed Real-time and Embedded Systems

Nilabja Roy, John S. Kinnebrew, Nishanth Shankaran,
Gautam Biswas, and Douglas C. Schmidt
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37203, USA

## Abstract

*Effective resource management for distributed real-time embedded (DRE) systems is hard due to their unique characteristics, including (1) constraints in multiple resources and (2) highly fluctuating resource availability and input workload. DRE systems can benefit from a middleware framework that enables adaptive resource management algorithms to ensure application QoS requirements are met. This paper identifies key challenges in designing and extending resource allocation algorithms for DRE systems. We present an empirical study of bin-packing algorithms enhanced to meet these challenges. Our analysis identifies input application patterns that help generate appropriate heuristics for using these algorithms effectively in DRE systems.*

## 1 Introduction

**Emerging trends and challenges.** Open distributed real-time and embedded (DRE) systems form the core of many mission-critical domains. These DRE systems execute in environments where system operational conditions, input workload, and resource availability cannot be fully characterized *a priori*. DRE system characteristics, such as multiple resource constraints and significant fluctuations in resource availability and input workload, make it hard to maintain end-to-end *quality of service* (QoS).

Applications in DRE systems often require multiple resources to execute properly, and need timely allocation of those resources to maintain required QoS. In open DRE systems, system resource utilization is a function of input workload and the required QoS of applications, so runtime utilization may vary significantly from estimated values. Moreover, system resource availability, such as available network bandwidth and battery power, may also be time variant. What is needed are middleware-centric capabilities to allocate and manage DRE system resources at *runtime*.

**Solution approach → A component-based adaptive resource management framework.** To address the needs of DRE systems, we have developed the *Resource Allocation and Control Engine* (RACE) [2]. RACE is an adaptive resource management framework built atop our CIAO QoS-enabled component middleware that supports resource allocation and system adaptation algorithms to manage DRE system resources at runtime. RACE allocates application components to available system resources. Numerous algorithms have been developed, studied, and analyzed for use in resource allocation. For example, Srivastav and Stangier [3] provide a solution to the resource-constrained scheduling problem, which is related to the multi-dimensional bin-packing problem.

In particular, bin-packing algorithms provide a natural solution to many resource allocation problems. The classical bin-packing problem packs a set of $n$ items into $m$ bins each with a maximum capacity $C$, such that the sum of the items in any bin does not exceed $C$. In the context of resource allocation, resources (*e.g.*, processors) form the bins, and items map to tasks (*e.g.*, components) that require a specified amount of resources.

This paper presents an empirical study of widely used bin-packing algorithms, focusing on the applicability of these algorithms in the context of resource allocation in DRE systems. For each algorithm, we study and analyze the following: (1) what extensions are needed to apply it to DRE system resource allocation, (2) how effective the algorithm is in finding a feasible allocation under stringent time limitations, depending on the input application characteristics, and (3) how useful additional computation to find an allocation is for different application characteristics.

## 2 Resource Allocation Challenges in Open DRE Systems

Although adaptive resource management frameworks can help provide greater local autonomy and enhance mission performance for DRE systems, the following resource management challenges remain unresolved.

**Challenge 1: Selection of appropriate multi-resource allocation algorithm(s).** While work has been done on allocation by a single measure, (*e.g.*, CPU usage), few algorithms exist for allocating multiple resources (*e.g.*, CPU and memory usage). Multi-resource allocation is not, in general, a straightforward extension of single-resource allocation. For example, with traditional bin-packing, items are

compared by size, but with multiple resources the appropriate "size" measure is no longer obvious, *e.g.*, it could be the average, sum, larger, or other combination of the multiple resource requirements.

The effectiveness of individual resource allocation algorithms also depends on the characteristics of applications being deployed. In particular, the distribution of component resource usage in an application—or an entire system—may be useful in selecting the most applicable resource allocation algorithm/heuristic. For example, some deployments may consist entirely of components whose resource use is small relative to the resources available at each node. Other deployments may include a mix of relatively large (in terms of resource use) components and relatively small ones. Some algorithms are more effective or efficient than others in finding an allocation for particular distributions of component resource use. Our results in Section 3.1 evaluate performance patterns that characterize the performance of heuristics across various input distributions.

**Challenge 2: Point of diminishing returns for running resource allocation algorithms.** DRE systems often operate on strict time constraints and it may be necessary to terminate the search for an allocation if it takes excessive computation/time. In that case, a mission planning application may be able to provide a new/modified application that has less stringent resource requirements (while potentially having lower quality or utility).

As with algorithm selection, determining whether additional computation may be successful in finding an allocation can depend on application characteristics (such as percentage of components with large resource requirements or number of components to be allocated), as well as the particular allocation algorithm(s) used. Our analysis of experimental results in Section 3.2 identifies cases in which additional computation is unlikely to improve performance.

## 3 Solution Approach: Integration of RACE and Resource Allocation Heuristics Based on Performance Patterns

This section empirically evaluates bin-packing algorithms that RACE uses to make the initial and subsequent resource allocations. Since complete bin-packing algorithms can be computationally expensive, we study different heuristic schemes in a multi-capacity bin-packing framework to simplify the allocation task. Our goal is to determine resource allocation heuristic *performance patterns*, *i.e.*, the likelihood of a heuristic finding an allocation for different classes of input. RACE uses these performance patterns to (1) select appropriate resource allocation algorithms based on the input data set at runtime and (2) determine how much computation to expend on each.

### 3.1 Resolving Challenge 1: Selection of Appropriate Multi-resource Allocation Algorithm(s)

**Problem.** RACE allocates available resources to components based on allocation algorithms/heuristics. If the relative performance of these allocation algorithms is known for particular distributions of component resource usage, RACE can choose the one(s) most effective at allocating applications in the system. Moreover, dynamically choosing the set of applicable algorithms/heuristics can make certain DRE systems even more efficient and effective in finding allocations. The patterns of performance for the available heuristics must therefore be identified across a variety of input distributions.

**Solution → Empirical comparison of heuristic performance.** To determine the performance of multi-capacity extensions to common bin-packing heuristics, we ran a series of experiments with problems drawn from various input distributions. These experiments used two-capacity bins, applicable to the case of system nodes with two resource attributes, such as CPU and memory. The performance metrics considered are "number of successes" in a fixed number of runs. The extension to additional resources for these heuristics is straightforward from the two-capacity implementation. We set the size of each of the two bin capacities to 100, representing 100% of the resource. In analyzing the results, we consider three orthogonal dimensions to the cases being tested, as described below.

- **Heuristic** is the performance of each algorithm/heuristic on the generated problems. We evaluate the extensions to multi-capacity bin-packing of the popular best-fit, first-fit, and worst-fit heuristics.

- **Sorting method** is the method used to sort the items before applying the above heuristics. Sorting items by decreasing size before packing the bins often provides better performance in traditional bin-packing. In our experiments, items are also sorted in decreasing order of size, but because the problems are multi-capacity ones, the sorting criteria is non-trivial. Several definitions for a scalar size value (combining or comparing the multiple dimensions) could be used, including sum, product, sum of squares, and maximum component. Our current experiments focus on two scalar definitions of size: sum and maximum component.

- **Item distribution** is the characterization of the distribution from which item sizes are drawn. Different solution methods and heuristics may be more/less applicable to particular item size distributions. One goal of our experiments is to determine if/when these heuristics are more effective based on characterization of input item sizes. For these experiments we used uniform distributions with various mean values. We also compared across the total amount of slack between the capacities of the bins and the sizes of the items. For example, a problem with 10 bins of capacity (100,100) and a slack of exactly 10 percent in each dimension, would

have a set of items whose sizes sum to (900,900).

### 3.1.1 Problem Generation

Three input parameters characterize the problems generated for a given set of test runs: (1) number of (100,100) capacity bins, (2) range of item sizes, and (3) percentage slack allowed (as a range) in the generated problems. These experiments use two-capacity bins/items, with the item's size in each dimension independently drawn from uniform distributions. For example, with 10 bins, 0-70 for item sizes, and 5-10 as the allowable percentage of slack, the set of problems generated would have items with an average size of $\sim$35 in each resource and total size for the sum of items would be between 900 and 950 in each resource.

To generate the problems, we used *rejection-sampling* [1], which samples from an arbitrary distribution $f(x)$ using some standard distribution $g(x)$ that is easy to sample. We generated items with the constraint that the sum of the item sizes should be less than the sum of all bin capacities by an amount within the range of allowable slack. Items were generated from the specified distribution until their sum was within the allowable range of slack or was greater than the maximum value, in which case that set of items was rejected. The generated set of items thus meets global bin capacity constraints, but a valid allocation is not guaranteed, *i.e.*, the problem may or may not be solvable.

We generated 10,000 instances of the problems that were run with the different heuristics. The number of bins used was 4 because using more bins made the running time of the complete algorithm too large to generate results in a reasonable amount of time. The running time was primarily a problem for distributions with smaller average item size because there were more items in total. For the other distributions, we also performed the experiments with 10 bins and obtained results that followed the same patterns identified in the 4 bin experiments presented here.

Figure 1 shows a representative frequency distribution of the item sizes in one of the two dimensions for a problem with 0-100 item distribution and slack between 0% and 5%. The problem sets for the other distributions and slack values also closely match their specified uniform distributions and are not included.

### 3.1.2 Analysis of Results

Figure 2a shows the relative performance of the different heuristics with a uniform distribution of item sizes between 0 and 100 for each dimension. This result shows that the best-fit and first-fit heuristics outperform the worst-fit ones overall (for this distribution). Moreover, the choice of sorting criteria does not make a large difference, as shown by the similar results for both sum and maximum component sorting when used with each heuristic.
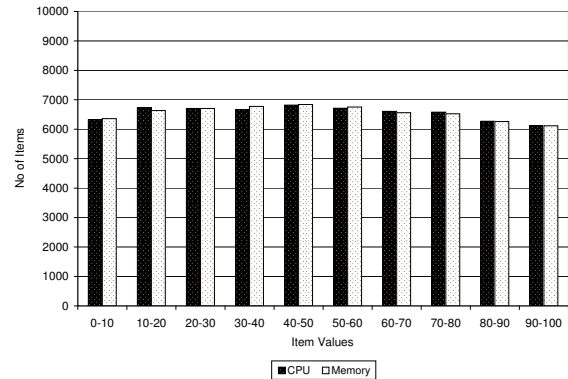


Figure 1: Distribution of Items (0-100 range)

The results for the items with ranges 0-70 and 0-30 show the same patterns, and are presented in the Figures 2b and 2d. The results from the 0-50 distribution exhibit a slightly different behavior as shown in Figure 2c. For these problems, there is a significant difference in performance between sorting criteria, with the sum sorting criteria outperforming the maximum component sorting criteria.

The results for these experiments clearly show a performance pattern for each heuristic across different input distributions, which RACE exploits during runtime resource allocation. For example, if the input distribution is roughly uniform with a mean item size of 25, these results would direct RACE to first employ the best-fit heuristic with sorting based on sum of the items, before expending computation on the other heuristics. Conversely, for uniform distributions with a mean item size near 50, RACE would prefer to first try the first-fit heuristic with sorting based on maximum component size.

## 3.2 Resolving Challenge 2: Point of Diminishing Returns for Running Resource Allocation Algorithms

**Problem.** The set of multi-capacity bin-packing heuristics tested in these experiments make a single attempt at finding an allocation. Extending these heuristics (*e.g.*, with backtracking or local search), however, is likely to produce better results in many cases at the cost of additional computation. Still, this is not consistently the case across all input distributions tested. RACE must therefore determine whether additional computation would significantly enhance its chance of finding a solution.

**Solution → Empirical study of heuristics performance on solvable problems.** Determining where additional computation can and cannot yield better performance requires further analysis of our experimental results. By identifying which input distributions yielded relatively low rates of success relative to total solvable problems, we can find
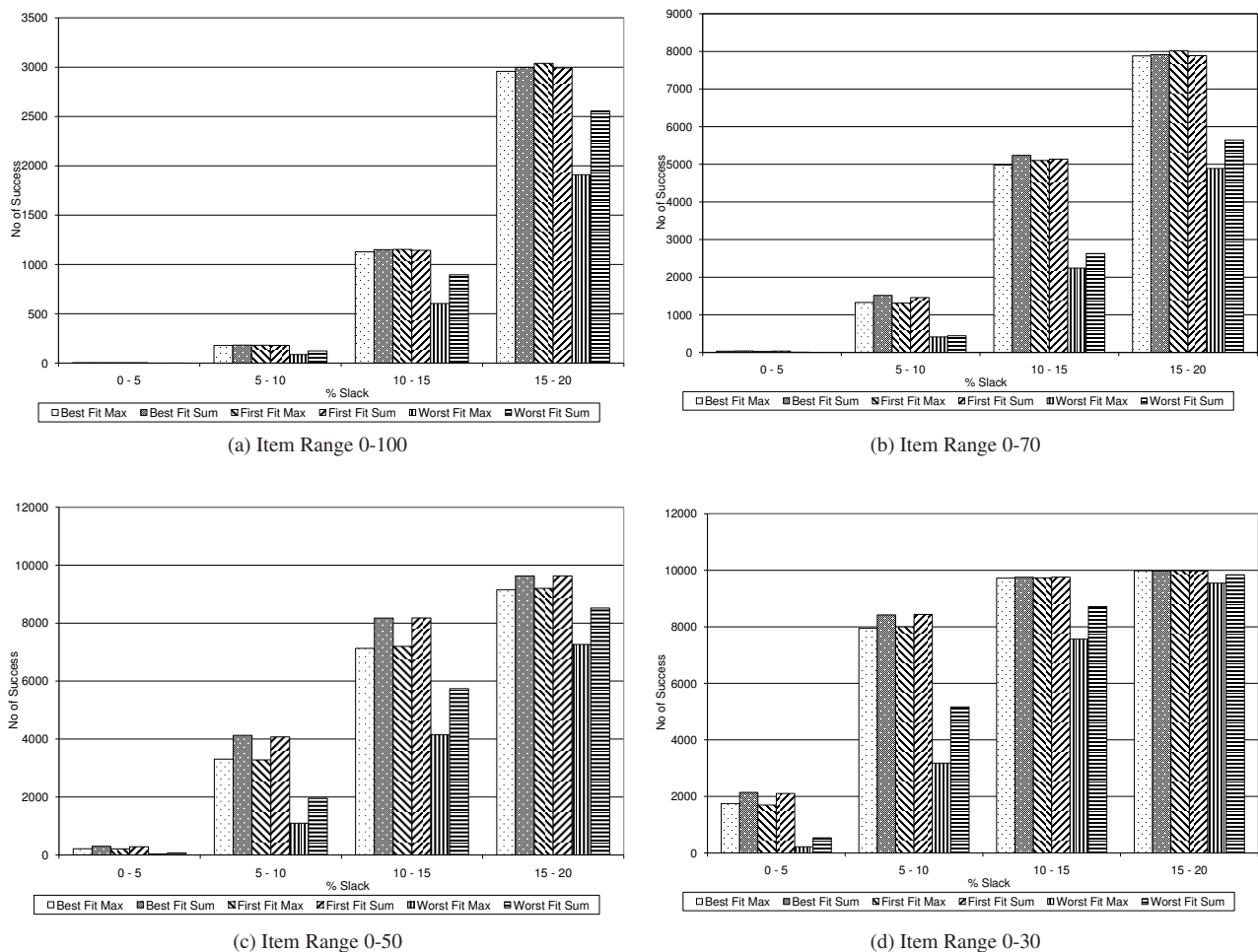
(a) Item Range 0-100



(b) Item Range 0-70



(c) Item Range 0-50



(d) Item Range 0-30

Figure 2: Performance Comparison of Different Heuristics

| Item Size | % Slack | | | |
|---|---|---|---|---|
| | 0 - 5 | 5 - 10 | 10 - 15 | 15 - 20 |
| 0 - 30 | 34.84 | 97.96 | 99.97 | 100 |
| 0 - 50 | 10.57 | 65.49 | 96.14 | 99.67 |
| 0 - 70 | 26.44 | 65.68 | 93.02 | 99.14 |
| 0 - 100 | 100 | 94.93 | 99.34 | 99.64 |

Table 1: Success Rate of Heuristics on Solvable Problems

the cases in which additional computation would likely benefit heuristic performance. Table 1 summarizes the combined performance of the heuristics on the *solvable* problems from each distribution. To determine whether a solution existed for each problem, we implemented a complete algorithm that searched all combinations for packing the items. Due to the running time of this algorithm, we limited the number of bins in the experiment to 4.

The success rate of the heuristics (computed for the sub-set of problems for which there existed a valid allocation) varies with the distribution ranges as well as with the slack in the bins. It is clear from the results that the heuristics perform quite well when the slack is relatively large (*i.e.*, greater than 10%). When there is very little slack (*i.e.*, 0-5% slack), however, the heuristics mostly perform poorly, with performance increasing with greater slack.

The exception to this trend is the 0-100 range items where there is a preponderance of medium-to-large size items (relative to bin capacity). In that case, the heuristics perform extremely well, with the unexpected result that they perform best with very little slack. These results suggest that when there are a significant number of large items and very little slack, the problem may only be solvable in one way (or a small number of ways) that is immediately found by the heuristics as they attempt to pack the bins without exceeding their capacity. The success rate diminishes a little with greater slack but is still quite high (99%). This may

mean that there are only a few ways to allocate the large items to bins, but a number of different ways to attempt to pack the remaining smaller items. Due to these additional possibilities, the heuristics may be less likely to find a valid allocation with their single attempt. Moreover, the hardest problems are those where the item sizes range from small to medium or medium-large (*e.g.*, 0-50 and 0-70 in these experiments). The 0-30 range is easier because there are many small items, allowing many valid allocations.

RACE employs the analysis above to determine when to terminate a particular algorithm. For example, when the components tend toward medium-to-large resource requirements, and the heuristics fail to find an allocation, RACE assumes that the components most likely cannot be allocated, so it does not bother expending additional computation searching for a solution. Conversely, if the component resource requirements are all between 0% and 50% with little slack (0-10%), then RACE runs the heuristics with backtracking for a longer duration before terminating the execution.

## 4 Concluding Remarks

The work presented in this paper provides an empirical evaluation of several multi-capacity bin-packing heuristics for resource allocation to identify performance patterns associated with these heuristics. These patterns provide a basis for our RACE adaptive resource management framework to select an appropriate suite of resource allocation methods based on the resource requirement characteristics of application components. This selection can be done at design time or runtime.

The lessons learned from our work can be summarized as follows:

**Use a suite of heuristics.** Analysis of the heuristics presented in Section 3 shows that the performance of a given heuristic depends on (1) the sorting method used to order the items and (2) the distribution of the item sizes and slack (difference between the bin capacities and the sum of item sizes). Moreover, no heuristic consistently out-performs all others. To increase the likelihood of successful runtime resource allocation in DRE systems, an adaptive resource management framework, such as RACE, should employ a suite of algorithms/heuristics that execute in parallel.

**Spend time wisely in searching for an allocation.** In addition to using each of these heuristics as a single-shot attempt to find an allocation, further computation may be fruitful in certain cases where the heuristics do not immediately find a solution. For example, our results suggest there is little benefit to using additional computation when the input contains a preponderance of medium and large components relative to node capacity (*e.g.*, with the 0-100 distribution the heuristics found a solution almost every time one existed).

Similarly, when there is a great deal of slack between component resource requirements and total system resources, the heuristics were likely to find a solution, if one existed, and further computation would not improve performance. Moreover, when the heuristics are extended to perform multiple attempts at finding an allocation (*e.g.*, through backtracking or local search), our results suggest that the most efficient solution will be to provide some of the heuristics more computational resources/time than others.

**Classify input to dynamically create weighted heuristic suite.** Based on our experiment results, it appears that an effective and efficient process for allocating system resources to application components involves (1) inspecting and analyzing component resource requirements to classify the input item distribution and (2) weighted selection of a suite of allocation algorithms/heuristics that are most likely to find a valid allocation of system resources. While the relative weight given to the heuristics could be set based on system-wide availability of components (*e.g.*, at design time), a more flexible solution is to dynamically adjust the weights at runtime as applications are provided for allocation. RACE can characterize the input application based on component resource requirements and adjust the weights dynamically to efficiently find a valid allocation.

In future work, we will test the performance of multi-capacity bin-packing heuristics on a wider range of input distributions, including normal distributions with a variety of means and variances. We will also classify input patterns for which particular heuristics are likely/unlikely to succeed with additional backtracking or local search computation. This classification will help RACE support a wider range of systems and applications by improving the efficiency and effectiveness of dynamic resource allocation.

The implementations of RACE and the resource allocation heuristics evaluated in this paper are available as open-source software from `deuce.doc.wustl.edu/Download.html` and `www.dre.vanderbilt.edu/~nilabjar/Allocation`, respectively.

## References

[1] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[2] N. Shankaran, D. C. Schmidt, Y. Chen, X. Koutsoukous, and C. Lu. The Design and Performance of Configurable Component Middleware for End-to-End Adaptation of Distributed Real-time Embedded Systems. In *Proc. of the 10th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2007)*, Santorini Island, Greece, May 2007.

[3] A. Srivastav and P. Stangier. Tight approximations for resource constrained scheduling and bin packing. In *Proceedings of the 4th Twente Workshop on Graphs and Combinatorial Optimization*, pages 223–245, New York, NY, USA, 1997. Elsevier North-Holland, Inc.