2016 IEEE 2nd International Conference on Big Data Security on Cloud, IEEE International Conference on High Performance and Smart Computing, IEEE International Conference on Intelligent Data and Security

Performance of Partitioned Homogeneous Multiprocessor Real-Time Scheduling Algorithms in Heterogeneous Environments

Andrew Burke
Pace University
Seidenberg School of Computer Science and Information Systems
New York, NY 10038
andrewmburke@gmail.com

***Abstract*—Hard real-time scheduling algorithms are most heavily studied in homogeneous systems. That is, systems that are composed of a single processor specification. As hardware continues to develop, we anticipate that legacy software will be deployed to hardware organizations for which it was not originally intended. We study the performance of standard homogeneous scheduling algorithms in heterogeneous environments via simulation, and conclude that heterogeneous systems do, in fact, offer the opportunity to markedly improve processor availability, though potentially at the cost of throughput.**

***Index Terms*—multiprocessing, scheduling, heterogeneous, homogeneous, real-time.**

I. Introduction

Real-time embedded systems have become commonplace across a wide variety of industries and devices. Such systems are prevalent in both consumer and industrial devices, and new processes and utilizations are continually in development. The typical hardware bottleneck in real-time systems is the central processing unit (CPU) due to the variety of tasks for which it is responsible. One path to improved performance of a real-time system is to increase the performance of the CPU, but the continual advancement of processor technology demands a concomitant development of algorithms. A case in point is the transition to inexpensive multicore CPU packages beginning in the mid-2000s, a development that dramatically reduced the potential for idle CPU time. Various algorithms have been proposed and studied to address the problem of process scheduling in a multicore environment, but these are typically associated with a homogeneous computing system, i.e. one in which all processing units perform identically. With the continual introduction of new hardware paradigms and architectures that support their interconnectivity, we wonder at the possibility of supporting legacy software with hard real- time constraints on a modern, heterogeneous system. In particular, we question whether algorithms designed for homogeneous systems are sufficient in a heterogeneous environment, and hypothesize that homogeneous algorithms will impart an increased miss rate, increased CPU idle time, and fewer task completions when run in a heterogeneous system. Analysis is presented below, and it is observed that heterogeneous processors can actually offer greater performance and CPU availability than homogeneous systems.

II. Background & Definitions

This section provides a primer on selected terminology in real-time multiprocessor scheduling algorithms that will be relevant to this study. A system is referred to as real-time when its correct behavior depends not only on the operations they perform being logically correct, but also on the time at which they are performed [3]. The completion of tasks within a specific span of time is referred to as operating under *hard* real-time constraints, whereas a statistical distribution of response times is acceptable under *soft* real-time constraints [2].

A. Single- vs. Multiprocessor Systems

A *single-*, or *uniprocessor* real-time system is one that features only a single CPU that is responsible for executing all tasks. Historically, these have been the most common types of systems, and were employed heavily in consumer and embedded systems. Beginning in the early to mid 2000s, however, significant interest was paid to increasing the diversity and specialization of processors and computer architectures, making feasible the advent of more powerful, smaller, single-package processors with the ability to execute multiple instructions per cycle [12]. Such strategies have become essential in order to maintain the speedup pace suggested by Moores law as CPU construction approaches the single-micron scale [11]. The result is a *multiprocessor* system, a single computing system with multiple execution cores at its disposal. Such a system can have its processing units distributed across several machines, such as in a distributed environment, or might have several processing cores on a single chip, as has become more common for embedded and consumer-grade systems.

B. Homogenity vs. Heterogenity

In multiprocessor systems, we refer to a system as belonging to one of three classes [3]:

- *Homogeneous*: all processors in a system are identical, and therefore the rate of execution of any task is the same regardless of the processor to which it is assigned.

978-1-5090-2403-2/16 $31.00 © 2016 IEEE
DOI 10.1109/BigDataSecurity-HPSC-IDS.2016.64

- *Heterogeneous*: there are different processors in a system, and therefore the rate of execution of any task is variable depending on the processor to which it is assigned.

Heterogeneous systems may be further delineated as [13]:

- *Uniform*: the only variable among processors is the rate at which a job is executed.
- *Unrelated*: the performance of a processor in executing a task is dependent on the characteristics of the task and of the processor.

For the purpose of this study, we will consider only homogeneous, unformly heterogeneous, and unrelated heterogeneous systems.

C. Core Problems

The basic task of any real-time multiprocessor scheduling algorithm is to address two core problems [3]:

- *Allocation*: deciding which processor will receive a task.
- *Prioritization*: deciding when each task should execute, respecting the demands of other tasks.

Allocation can be further described by the term *migration*, referring to the propensity for the assignment of a task to a specific processor. In a no-migration, or *partitioned*, algorithm, tasks are only permitted to execute on a specific processor. The counterpart is a *global* algorithm, in which tasks may be allocated to any processor. Allocation in heterogeneous systems is strongly NP-hard [10], so for efficiencys sake the algorithms in question are sufficient rather than necessary. For the purposes of this study, we will consider partitioned algorithms.

We also consider whether a system is *preemptive*, indicating whether a task currently executing on a CPU may be suspended in favor of a task with higher priority. For the purposes of this study, we will choose preemptive scheduling algorithms, as these cover a broad range of real-world applications. This is to acknowledge that there exist events crucial to the successful operation of a computer, such as interrupts.

D. Measurements, Calculations, & Abstractions

- *Task*(τ_i): A *taskset* is the set of programming tasks that compose an application. Taskset T is composed of the static set $\{\tau_1, \tau_2, ..., \tau_i\}$.
- *Deadline*(D_i): the maximum allowable time for the completion of task i as defined by the real-time functional requirements of the application.
- *Period*: the minimum time interval since the arrival of the previous job of the same task.
- *Utilization*: the utilization of task i is the ratio given by tasks worst-case execution time divided by the tasks period:

$$u_i = \frac{c_i}{\tau_i} \tag{1}$$

- *Approximation Ratio*($\Re_A$) [3]: a way of comparing the performance of a scheduling algorithm A with that of an optimal algorithm. Given taskset τ, let the number of processors required according to an optimal algorithm be $M_O(\tau)$ and the number required according to algorithm A be $M_A(\tau)$. Then:

$$\Re_A = \lim_{M_O(\tau)\to\infty} (\max_{\forall\tau}(\frac{M_A(\tau)}{M_O(\tau)})) \tag{2}$$

Note that a smaller value of $\Re_A$ indicates a more efficient scheduling algorithm. An algorithm with $\Re_A = 1$ is an optimal algorithm.

TABLE I
SELECTED PARTITIONED ALGORITHMS SUMMARY

Partition Strategy	Allocation Strategy	Approximation Ratio ($\Re_A$)	Source
EDF	FF	1.7	[9]
RM	GT	$7/4$	[6]
RM	ST	$\frac{1}{1-u_max}$[1]	[6]
RM	BF	2.33	[4]
RM	NF	2.67	[1]
RM	FF	2.33	[1]

III. ALGORITHMS IN QUESTION

The algorithms listed in Table I comprise a subset of the many extant scheduling algorithms. The selected algorithms are intended to be a representative group, incorporating the broad types of allocation strategies and of prioritization strategies.

Each of the prioritization and allocation strategies employed are further described below.

A. Prioritization Strategies for Partitioned Algorithms

- *Rate-Monotonic (RM)*: priority assignment is solely based on the frequency that a task is requested [2].
- *Earliest Deadline First (EDF)*: priority assignment is fixed and solely dependent on the most urgent tasks [9].

B. Allocation Strategies for Partitioned Algorithms

Given a particular taskset τ, allocation strategies decide which processor will receive its individual tasks. One processor may receive several tasks:

- *Next-Fit (NF)*: mark a processor as the "active" processor. Assign to this processor, if feasible. otherwise assign to the next processor, which may or may not be idle, and mark this processor as the active one [14].
- *Best-Fit (BF)*: assign to the processor for which the task maximizes utilization [4, 14].
- *First-Fit (FF)*: assign to the minimally indexed processor (i.e. the first processor encountered in the sequence of processors) that can complete the task [4, 14].
- *Small Tasks (ST)*: intended for tasksets where the maximum utilization of all tasks is much smaller than the processing speed of each processor, we assign tasks to processors in such a manner as to ensure a relatively even distribution across all processors [6].
- *General Tasks (GT)*: allocation is handled bimodally. Tasks are grouped into two subsets, A and B, such that

TABLE II
TESTING TASKSETS WITH MILLISECONDS REQUIRED PER TASK

Task Index	$\mathbf{T}_1$	$\mathbf{T}_2$	$\mathbf{T}_3$	$\mathbf{T}_4$	$\mathbf{T}_5$
0	602	351	276	669	417
1	274	406	537	693	626
2	481	282	337	523	646
3	701	252	276	704	264
4	561	316	636	428	597
5	680	729	497	413	712
6	670	408	713	391	552
7	677	688	684	749	533
8	544	727	311	474	318
9	706	382	514	619	386
10	364	362	371	749	289
11	475	747	649	441	579
12	320	493	265	622	310
13	476	254	524	445	634
14	659	422	254	595	618
15	657	608	268	338	690
16	385	426	382	408	480
17	722	727	502	547	476
18	503	654	629	491	701
19	386	513	514	653	349

$\forall \tau_i \in A, u_i \leq 1/3$. Tasks in A are assigned to processors via the ST scheme, while tasks in B are assigned via FF [6].

C. Additional Relevant Variables

- n: the total number of tasks to be executed.
- m: the number of processors available.
- t: time.
- *miss*: a miss occurs when a task arrives at the CPUs for execution, but there is no combination of CPUs immediately available for scheduling that will meet the tasks deadline. The task is forced to wait until sufficient CPU resources are available per the algorithm employed.

IV. METHODOLOGY

We devise a simulation in order to test the performance of each algorithm. The simulation was run for a set of four CPUs $(m = 4)$ with both homogeneous and heterogeneous performance speeds. The simulation was performed over identical tasksets across twelve separate trials.

A. Taskset Creation

Before running the simulation, five tasksets were created, each with 20 tasks of various execution times. Table II illustrates the tasksets, with each tasks execution time measured in milliseconds. In addition, each taskset was assigned a period, as shown in Table III.

TABLE III
TASKSET PERIODS IN MILLISECONDS

	$\mathbf{T}_1$	$\mathbf{T}_2$	$\mathbf{T}_3$	$\mathbf{T}_4$	$\mathbf{T}_5$
T_i(ms)	10000	12000	14000	15000	8000

Finally, we calculate a deadline for each taskset. The deadline D_i for taskset T_i composed of n_i number of tasks is defined as:

$$D_i = 1.35 \left(\sum_{j=0}^{n_i} cost(\tau_j) \right) \tag{3}$$

This yields the following deadlines for the tasksets in the experiment:

TABLE IV
TASKSET DEADLINES (MS AFTER RAISING AN INTERRUPT)

	$\mathbf{T}_1$	$\mathbf{T}_2$	$\mathbf{T}_3$	$\mathbf{T}_4$	$\mathbf{T}_5$
T_i (ms)	14638	13158	12337	14785	13738

B. Trials

With our tasksets established, we then run twelve sets of trials, five with heterogeneous processor speeds and seven with homogeneous processor speeds. For the purposes of the simulation, processor speed is abstracted to the time the processor needs to complete a single cycle and is measured in milliseconds. When a processor receives a task, it marks the time needed to complete the task, then "ticks" until the time requirement is met. The processor then serves its next task. It should be noted that under this scheme, a lower speed implies a faster processor. The values for speeds selected were normalized to 100; that is to say, we assume that a processor with speed 70 is 30% faster than one of speed 100.

Table V illustrates the five trials conducted, where the four processors are identified as P_0, P_1, P_2, P_3 and trials are assigned a unique String identifier. These identifiers correspond with "Homogeneous 40 ms speed," "Homogeneous 45 ms speed," "Homogeneous 50 ms speed," "Homogeneous 55 ms speed," "Homogeneous 60 ms speed," "Homogeneous 65 ms speed," "Homogeneous 70 ms speed," "Heterogeneous (slowest)," "Heterogeneous (slow)," "Heterogeneous (medium)," "Heterogeneous (fast)," and "Heterogeneous (fastest)" speeds. These trials are designed to test algorithmic performance at varying speeds, including in heterogeneous systems that may have slower processors than might be available in an extant homogeneous system.

We run each trial for 10 minutes. The number of completed tasks (i.e., the throughput) and idle time for each processor is recorded every 200 ms and written to a spreadsheet at the conclusion of each test. The total number of misses is also included in this spreadsheet. A "perfect" execution for these tasksets and periods, i.e., one in which no task is left incomplete or in progress after the 10 minutes have elapsed, is 5360 total tasks completed, which would result in an average of 1340 tasks completed by each processor.

TABLE V
SUMMARY OF PROCESSOR SPEEDS BY TRIAL (MS, LOWER VALUES REPRESENT FASTER PROCESSORS)

Trial Identifier	P0	P1	P2	P3
homog40	40	40	40	40
homog45	45	45	45	45
homog50	50	50	50	50
homog55	55	55	55	55
homog60	60	60	60	60
homog65	65	65	65	65
homog70	70	70	70	70
heteroSlowest	90	80	70	60
heteroSlow	80	70	60	50
heteroMedium	70	60	50	40
heteroFast	60	50	40	30
heteroFastest	50	40	30	20

V. RESULTS & ANALYSIS

During each trial, we record results for number of completed tasks and total time spent idle for each algorithm. Summary averages of each trial are presented in Tables VI and VII. Graphed summaries of the results are available on request.

Given the large number of variables and values in this experiment, it is prudent to normalize our data. To that end, the averages of idle time, tasks completed, misses, and average time per task per CPU were calculated and plotted from the results obtained. These graphs are presented in the Appendix below as Figures 1 through 4. With these figures as reference, I address comments to each algorithm in turn.

A. EDF-FF

The EDF-FF algorithm showed consistent performance across all of the tests, with middle-of-the road processor availability and task completion. It achieved perfect execution in two trials, homog40 and homog45. The miss rate was generally consistent between the homogeneous and heterogeneous trials.

One interesting note is in the heterogeneous tests, EDF-FF showed among the highest idle processor times, but was not a top performer in throughput.

B. RM-ST

The RM-ST algorithm performed very well in the homogeneous environments. Indeed, it was little affected by alterations in the speed of the processors, surprisingly even showing more tasks completed on slower hardware. The low miss rate and low average time per task indicate a very efficient distribution of load across the available process.

RM-ST was, however, very negatively affected by the heterogeneous environments. The number of tasks completed dropped significantly, with the heteroSlow trial performing two standard deviations worse than in homog40, its worst performance in a homogeneous environment.

Interestingly, RM-ST showed low idle time throughout all tests. This indicates that it had very high usage of available resources, even though it did not produce an efficient distribution of tasks. The high usage is counterbalanced by the low task throughput, where RM-ST competed with RM-NF for fewest tasks completed.

C. RM-GT

Even though the RM-GT algorithm uses RM-ST for scheduling certain tasks, it is clearly an improvement. Its metrics beat out RM-ST in nearly every category. Additionally, it had the lowest miss rate in every test. Across each graph, its performance is in a nearly linear relationship with processor speed.

It should be noted that RM-GT is an enormous outlier in the average time per task calculations. As the above tables indicate, particularly Table VI, RM-GT assigned fewer tasks to the later processors in the array. This results in a skewed usage, where some processors were left idle for very long periods of time. This would appear to be a factor in the RM-GTs throughput, which was lower than some other algorithms.

Neither RM-GT nor RM-ST were able to achieve a perfect execution in any trial.

D. RM-NF

RM-NF was, clearly, the worst performing algorithm. It placed at or near the bottom of each metric. Though it jockeyed with RM-ST for worst throughput in the homogeneous tests, it was clearly inadequate for the heterogeneous tests. Indeed, it was, in the heteroSlow trial, nearly four standard deviations worse than the top performer in throughput.

It is the miss rate, however, that truly demonstrates RM-NFs inefficiency. RM-NF typically posted miss rates over 1000, well beyond the other algorithms. This is despite the fact that time per task and idle time were consistent with other algorithms, though unimpressive. This indicates an extremely inefficient scheduling scheme, as processors were routinely loaded up with other tasks.

RM-NF failed to achieve a perfect execution in any trial.

E. RM-FF

The RM-FF algorithm was a strong contender in most of the trials. It posted consistently high throughput, regardless of processor speed, though it would appear to be approaching a breaking point in the heteroSlowest and homog70 trials, as its throughput sloped sharply negatively. This assertion is reinforced by the extremely low idle time in these two trials, indicating that the algorithm was heavily taxed.

Additionally, we can note that the miss rate appears to be parabolic in trend, while the idle rate appears to be linear. Fitting trendlines to these points would likely show a crossover point very near the upper speeds in these trials, a point at which we could conclude that RM-FF would be incapable of achieving success.

Despite bordering failure at times, RM-FF was able to achieve a perfect execution in the homog40 and heteroFastest trials.

TABLE VI
SIMULATION RESULTS

	EDFFF	RMST	RMNF	RMGT	RMFF	RMBF
homog40	1340	1067	1072	1251	1340	1340
homog45	1340	1074	1057	1243	1337	1340
homog50	1335	1118	1075	1231	1336	1340
homog55	1325	1148	1121	1220	1331	1340
homog60	1325	1162	1173	1209	1315	1340
homog65	1205	1148	1167	1192	1303	1340
homog70	1276	1138	1146	1180	1268	1299
heteroSlowest	1139	762	743	1152	1207	1240
heteroSlow	1295	741	676	1178	1286	1340
heteroMed	1322	825	774	1195	1325	1340
heteroFast	1333	873	821	1219	1334	1340
heteroFastest	1333	933	913	1231	1340	1268
	μ		1186.05			
	σ		177.38			

(a) Summary of Tasks Completed per CPU

	EDFFF	RMST	RMNF	RMGT	RMFF	RMBF
homog40	41.20	2.70	1.79	43.78	39.59	41.23
homog45	33.59	0.74	1.29	37.67	33.25	33.88
homog50	26.50	2.74	0.75	32.00	25.84	26.34
homog55	19.77	3.07	0.25	25.96	19.08	18.98
homog60	12.46	5.42	1.89	20.44	13.01	11.63
homog65	13.22	4.61	0.33	15.36	6.75	4.27
homog70	1.75	0.97	0.27	9.74	2.29	0.00
heteroSlowest	6.90	0.04	0.15	5.18	0.89	0.04
heteroSlow	8.15	0.04	0.02	12.53	6.81	6.38
heteroMed	18.69	0.06	0.14	21.21	15.29	20.46
heteroFast	30.20	0.16	0.35	28.86	24.41	34.74
heteroFastest	42.86	0.52	0.12	31.04	29.93	52.02
	μ		14.37%			
	σ		14.55%			

(b) Summary of average time spent idle per CPU (%).

TABLE VII
NUMBER OF MISSES

	EDFFF	RMST	RMNF	RMGT	RMFF	RMBF
homog40	39	97	1386	0	37	3
homog45	29	103	1532	0	45	7
homog50	65	64	1472	0	52	6
homog55	107	30	1415	2	126	6
homog60	165	19	1201	0	171	6
homog65	149	34	53	3	269	6
homog70	505	36	1374	7	528	290
heteroSlowest	549	396	2126	20	948	303
heteroSlow	343	403	2250	5	351	6
heteroMed	130	335	2064	2	153	9
heteroFast	51	279	988	0	64	4
heteroFastest	23	222	1727	1	30	3
	μ		350.33			
	σ		577.90			

F. RM-BF

The RM-BF algorithm showed the best and most consistent performance across the trials. With very low miss rate and low average time per task, RM-BF clearly made excellent use of the available hardware. Throughput was similarly excellent, posting the highest scores on many of the trials, and showing greater resilience than other algorithms as processors became slower.

Like RM-FF, however, the homog70 and heteroSlowest trials were clearly borderline breaking points for RM-BF. The miss rate is highly demonstrative here, as it was typically in the single digits except for these two trials. Again, the linear nature of the idle time shows that RM-BF would likely have faced trouble on any hardware slower than that employed in these trials.

RM-BF was able to achieve a perfect execution in ten of the trials, failing only in the homog70 and heteroSlowest trials.

VI. CONCLUSION

One clear conclusion, and one that should come as no surprise, is that the data demonstrate that faster CPUs result in greater processor availability and throughput. Though this is a trivial and obvious observation, I feel that the data amply illustrate these beneficial effects, and I would be remiss to ignore this fact.

The rate-monotonic algorithms were the most affected by CPU speed, particularly RM-FF and RM-BF. This is logical; the rate-monotonic strategy is based on the fact that there is a predictable time at which tasks will arrive and need to be scheduled. If you need more time to prepare for the next arrival, then performance will necessarily suffer.

RM-GT and RM-ST were less affected by processor speed. Indeed, performance remained broadly the same across all trials, though certainly with some degradation on slower hardware. RM-GT in particular proved to be highly reliable regardless of its environment.

I will single out the rate-monotonic RM-NF as completely inadequate in all cases. With the highest $\Re_A$ value of the algorithms in use, it was certainly estimated to be a lackluster performer. In the context of these tasksets and hardware profiles, however, it was hopeless.

EDF-FF was a middle-of-the-road performer throughout, showing good performance and even a pair of perfect executions. The disparity between idle time and throughput in comparison to other algorithms indicate that it was not making the most efficient use of resources.

We can conclude that two of the algorithms would represent the best choices for a real-time system; RM-BF and RM-GT.

RM-BF is the better choice for transferring legacy software to a newer hardware platform, as it demonstrates significant speedup when given faster processors. It would be appropriate to evaluate its performance prior to deployment, however, as it could potentially be presented with a taskset that is beyond its capability to schedule on the hardware. A potentially user would do well to find this crossover point.

RM-GT is the better choice for its predictability. With extremely low miss rates, good throughput, and high processor availability, RM-GT demonstrated that it is likely the most efficient scheduling algorithm on display here. Though it did not post the lowest $\Re_A$ value in the preliminary research, its consistency and relative immunity to the hardware environment in testing show an invaluable reliability. In an overall conclusion, we can note that four of the algorithms were not significantly affected by the change of environment. These were EDF-FF, RM-GT, RM-BF, and RM-FF. The graphs show that their performance was similar across trials with the same average processor speed, and their performance trends were predictable and generally linear. For these algorithms, a heterogeneous environment proved no barrier to task completion.

Conversely, RM-ST and RM-NF were badly affected by the environment change, with each metric typically worse by a half of a standard deviation or more on heterogeneous hardware. Combined with their poor performance in comparison to the other algorithms in this experiment, we conclude that RM-ST and RM-NF are inappropriate choices for a modern real-time system.

ACKNOWLEDGMENT

Thanks to Pace University, for the opportunity to engage in graduate-level research, and to Dr. Meikang Qiu for guidance on the conduct and process of computing research.

REFERENCES

[1] S. Dhall and C. Liu, 'On a Real-Time Scheduling Problem', Operations Research, vol. 26, no. 1, pp. 127-140, 1978.
[2] C. Liu and J. Layland, 'Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment', Journal of the ACM, vol. 20, no. 1, pp. 46-61, 1973.
[3] R. Davis and A. Burns, 'A survey of hard real-time scheduling for multiprocessor systems', CSUR, vol. 43, no. 4, pp. 1-44, 2011.
[4] Y. Oh and S. Son, 'Tight Performance Bounds of Heuristics for a Real-Time Scheduling Problem', Department of Computer Science, University of Virginia, Charlottesville, VA, 1993.
[5] Y. Oh and S. Son, 'Allocating fixed-priority periodic tasks on multiprocessor systems', Real-Time Syst, vol. 9, no. 3, pp. 207-239, 1995.
[6] A. Burchard, J. Liebeherr, Yingfeng Oh and S. Son, 'New strategies for assigning real-time tasks to multiprocessor systems', IEEE Transactions on Computers, vol. 44, no. 12, pp. 1429-1442, 1995.
[7] T. Rothvoss, 'On the Computational Complexity of Periodic Scheduling', Ph.D., cole Polytechnique Fdrale de Lausanne, 2009.
[8] S. Davari and S. Dhall, 'On a periodic real-time task allocation problem', in Annual International Conference on System Sciences, 1986.
[9] M. Garey and D. Johnson, Computers and intractability. San Francisco: W.H. Freeman, 1979.
[10] S. Baruah, 'Partitioning real-time tasks among heterogeneous multiprocessors', 2004, pp. 467–474.
[11] T. Austin, D. Blaauw, S. Mahlke, T. Mudge, C. Chakrabarti and W. Wolf, 'Mobile supercomputers', Computer, vol. 37, no. 5, pp. 81–83, 2004.
[12] W. Wolf, 'How many system architectures?', Computer, vol. 36, no. 3, pp. 93–95, 2003.
[13] S. Baruah and J. Goossens, 'Deadline monotonic scheduling on uniform multiprocessors', pp. 89–104, 2008.
[14] D. Johnson, 'Near-Optimal Bin Packing Algorithms', Ph.D., Massachusetts Institute of Technology, 1973.