Extended Abstract - Submitted to SC'99

Job Scheduling in the presence of Multiple Resource Requirements *

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

TR 99-025: Revised May 27, 1999

Abstract

In past massively parallel processing systems, such as the Intel Paragon and the CRI T3E, the scheduling problem consisted of allocating a single type of resource among the waiting jobs; the processing node. A job was allocated the minimum number of nodes required to meet its largest resource requirement (e.g. memory, CPUs, I/O channels, etc.). Recent systems, such as the SUN E10000 and SGI O2K, are made up of pools of independently allocatable hardware and software resources such as shared memory, large disk farms, distinct I/O channels, and software licenses. In order to make efficient use of all the available system resources, the scheduling algorithm must be able to maintain a job working set which fully utilizes all of the resources. Previous work in scheduling multiple resources focused on coordinating the allocation of CPUs and memory, using ad-hoc methods for generating good schedules. We provide new job selection heuristics based on *resource balancing* which support the construction of generalized K-resource scheduling algorithms. We show through simulation that performance gains of up to 50% in average response time are achievable over classical scheduling methods such as First-Come-First-Served with First-Fit backfill.

Keywords: parallel job scheduling, multiple resource constraints, high performance computing

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

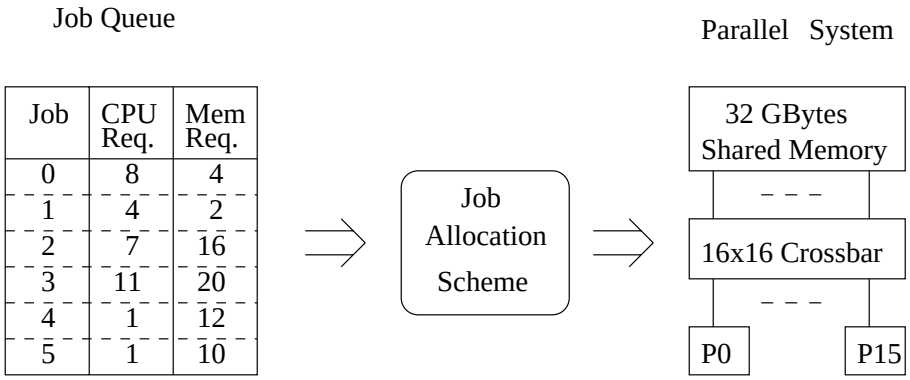
1 Introduction

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

In classical job management systems (JMS), a job was submitted along with a set of resource requirements which specifies the number of CPUs, amount of memory, disk space, etc., and the expected time to complete. The target systems were primarily distributed memory parallel processors with a single system resource - a processing node consisting of a CPU, memory, and a network connection to I/O devices. Although job allocation research literature is filled with exotic methods of allocating resources to a job stream [9],[3], [10], simple allocation schemes such as *First-Come-First-Serve (FCFS)* or *FCFS with First-Fit backfill (FCFS/FF)* were used in practice, providing acceptable levels of performance [8], [4],[14]. These job allocation schemes were limited in part due to the all-or-nothing hardware partitioning of the distributed systems. For example, a memory intensive job must be allocated enough nodes to meet the jobs memory requirements, but may not need all the CPUs which were co-allocated by default. The excess CPUs are not available to other waiting jobs and are essentially wasted. This situation is worse in newer systems where resources may be allocated to a job independently from each other. The greedy FCFS-based job allocation schemes cannot take full advantage of this additional flexibility.

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

The FCFS-based algorithms are restricted in selecting jobs based on their general arrival order. In order for a job allocation scheme to efficiently utilize the independently allocatable resources of the K-resource system, it must be free to select any job based on matching



(a)

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

(b)

Figure 1: Job Allocation Scheme Comparison

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

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

The FCFS/FF-based schemes use a greedy method to backfill a machine by selecting the *next* job which fits, subject to the backfill constraints. In general, these methods do not look at the additional resource requirements of the jobs in the job pool or the current state of the system resource loads. In the K-resource scheduling problem, this approach leads to premature depletion of some resources while others remain under-utilized. In the previous example, selecting job 0 followed by job 1 depletes the available CPU resources *faster* than the available memory resources. Our approach is to provide K-resource *aware* job selection heuristics which use additional job resource requirements and system state to guide the selection of the next job. These heuristics are easily incorporated into the current FCFS/FF-based scheduling algorithms. Our simulation results show that enhancing the FCFS/FF scheduling algorithm with the K-resource aware heuristics provide a substantial performance improvement in average system response time over the previous greedy first-fit heuristics.

The remainder of this document is outlined below. Section 2 provides a summary of past research in multi-resource job scheduling algorithms and presents our new K-resource aware job selection heuristics. Experimental results and conclusions are provided in Section 3.

2 The Balanced Resource Heuristics

2.1 Related Research

Job scheduling in parallel processing systems has been extensively researched in the past. Typically this research has focused on allocating a single resource type (e.g. CPUs) to jobs in the ready queue. A limited effort has extended this work to scheduling jobs with two resource requirements; CPUs and memory. In [12], memory requirements were used as a lower-bound constraint on the number of CPUs to allocate to a job in a distributed memory system. In [13], a branch-and-bound approach was used to select jobs which fit in the available memory in a shared memory system then allocated CPUs to those jobs based on the number available and the current system load. The goal was to allocate the minimum CPU and memory resources to a job such that neither resource allocation cause severe performance degradation. Both of these efforts recognized that the performance of a job is dependent upon more than the number of CPUs it receives. This shows that future job scheduling systems must be more cognizant of *all* resource requirements for a given job when attempting to allocate resources to that job. However, these efforts do not provide a clear path for generalized scheduling under K-resource requirements.

Current job scheduling practices typically support variable resource allocation to a job, and run-to-completion scheduling. Scheduling policies are also heavily based on First-Come-First-Served (FCFS) policies which allocate resources to jobs in the order that they arrive to preserve fairness. The FCFS scheduling algorithm breaks down when the next job in the

ready queue requires a large amount of of any single resource. This job essentially blocks any job which arrived after it, even though there may be sufficient resources to execute it in the current scheduling epoch. This deficiency led to the addition of *backfilling*. Backfilling selects jobs from farther down the ready queue for immediate execution subject to interference constraints on jobs near the head of the ready queue. The EASY backfill scheduler selects any job from the queue which does not interfere with the expected start time of the first blocked job in the ready queue [8]. This results in a lower average response time for smaller or shorter jobs, and guarantees a level of progress to larger jobs. Conversely, conservative backfill limits job selection to only those jobs which will not interfere with any job ahead of them in the queue [4]. A generalized backfill scheme protects the top N jobs in the ready queue. This supports trading determinicity for performance in average response time. Other variants of backfill have also been studied [15], [16].

FCFS with First-Fit backfilling (FCFS/FF) performs well when allocating a single resource such as a computational node. However it is still subject to resource depletion in a K-resource scheduling system. This is because these methods typically use a greedy first-fit criteria when selecting jobs to backfill. This can lead to scenarios where consecutively selected jobs may all have high resource requirement in a common resource type, which depletes this system resource type leaving other resource types idle. Thus the basis for our new backfill heuristics is to avoid premature resource depletion by maintaining *equal utilization of all resource types in the system*. Essentially, the relative usage of all system resources is *balanced*. The goal is to allocate each of the K machine resources to the same level so that no single resource becomes prematurely depleted. Backfill jobs are selected which best move the system towards a balanced state. Heuristically, this increases the probability that the additional jobs will fit during the current scheduling epoch. Two such balancing heuristics are described in the next section.

2.2 The BL and BB Heuristics

The First-Fit (FF) backfill job selection heuristic requires only that the job fits into the system. The K-resource balancing heuristics select jobs which attempt to correct a *resource imbalance* in the system. A resource imbalance is defined as the condition where $K_i < K_j, 1 \leq i, j \leq K$ in the current scheduling epoch. Essentially, at least one resource is more heavily used than the other resources. The general notion is that if the resource usages are all kept balanced, then more jobs will likely fit into the system. The first simple heuristic algorithm follows from this notion. Consider a system state in which K_j is the resource which is currently at a lower utilization than all the other resources. A *lowest-utilization aware* selection algorithm searches the job queue looking for a job which fits in the system *and* which has K_j as its largest resource requirement. Adding this job to system heuristically lessens the capacity imbalance at resource K_j. The lowest-resource aware packing algorithm can be generalized to the case where the algorithm looks at the $w, 0 \leq w \leq K-1$ lowest resource utilization and searches for an item which has the same w corresponding largest resource requirements. The parameter w is a *window* into the current system state. This is the general *windowed lowest-resource job selection heuristic*. Similar

heuristics have been successfully applied to the multi-constraint graph partitioning problem [6] and a generalized multi-capacity bin-packing problem [7]. The connection between multi-constraint bin-packing and multi-resource scheduling has also been studied [11], [1], [5]. The additional selection granularity provide by the window, w, provides a benefit when there are sufficient items from which to choose. However, in the typical job scheduling system, the number of jobs in the queue can be quite small. Therefore, a version of this heuristic which look only at the single lowest utilized resource ($w = 1$) is pursued here and is called Backfill Lowest (BL).

The BL heuristic may also suffer from a resource size granularity problem. In the previous bin-packing effort, the performance of the windowed lowest-resource degraded as the average resource requirement of an item grew with respect to the size of the bin. This reduced the number of items that would fit into a bin, which reduced the opportunity of the heuristic to correct an imbalance. In the typical job scheduling system the size of the jobs relative to the *empty space* in the machine is often quite large. When one job leaves the system, often only a single job can be selected to replace it. Therefore, the BL heuristic cannot guarantee to actually balance the resource utilization as it will only attempt to balance one of the K resources with the single replacement job. For this reason, we created a second balancing heuristic which selects a job based on its overall ability to balance the resource utilization, termed Backfill Balanced (BB). BB uses a *balance measure* to score each potential backfill job and then selects the job which results in achieving the best resource utilization balance based on this measure. For our experiments, we used the following balance measure:

$$\frac{\max_i(U_{Si} + R_{Ji})}{\frac{\sum_{ik=1}^{K} U_{Si}+R_{Ji}}{K}}$$

where U_{Si} represents the current utilization of resource i by the system S, and R_{Ji} represents the requirement for resource i by job J. This is essentially a $max/average$ balance measure where a lower value indicates a better balance.

Finally, both BL and BB, as defined, guide the job selection towards balancing the resource utilization, but not *increasing* the utilization levels. Therefore, a *fullness* modifier is applied to both schemes which relaxes the balance criteria as the system becomes full. This allows a larger job which achieves a worse balance to be selected over a smaller job which achieves a better balance when the larger job nearly fills the machine. This fullness modifier is simply $1.0 - \frac{\sum_{ik=1}^{K} U_{Si}+R_{Ji}}{K}$, or the average resource availablilty. Note that as the resource availability decreases, the BL and BB schemes approach FF functionally.

3 Experimental Results

The following subsections describe a parametric simulation study to evaluate the BL and BB heuristics. The FCFS algorithm with First-Fit EASY backfill (FCFS/FF-EASY) is used as the baseline for performance comparison purposes. This algorithm was enhanced with the BL and BB heuristics to for FCFS/BL-EASY and FCFS/BB-EASY. Additionally, a FCFS with conservative backfill, FCFS/FF-CONS was also simulated for completeness.

3.1 Workload Model

The baseline workload model was the Feitelson workload model [2]. This workload model is based on a detailed analysis of six production parallel system work logs and is generally accepted as capturing the major characteristics of a typical production site. A more detailed description and the code for generating the workload is available from: www.cs.huji.ac.il/labs/parallel/workload/wlmodels.html . This code generator creates a list of jobs, with each job described by size (number of processes, or parallelism), runtime (weakly correlated to the size), and a repeat count. The repeat count captures a secondary effect in the fact that a user will submit a job, wait for results, and then submit the *same* job again, slightly altered. We used the job size and runtime characteristics for this study.

The Feitelson workload model was extended with the following three parameters:

1. K: This study involves the scheduling and allocation of a K-resource system. For each job in the baseline Feitelson model, the job size was used to generate a vector of K resource requirements for that job. For these simulations, values of K were taken from $\{2, 4, 8\}$.

2. Relationship of the K_is: The relationship between the K_is of a single job was explored using two different distributions: Uniform and Exponential. For the uniform distribution, a set of K random numbers was drawn from a uniform random variable and then scaled with the job size from the Feitelson model. A similar method was used for the exponential distribution.

3. System Load: The Feitelson model does not provide an interarrival time between jobs, as this is dependent upon the repeat execution characteristic. However, the arrival characteristic is generally believed to be Poisson distributed. For this simulation, we chose to use the average ready queue length achieved by the FCFS/FF-EASY scheduling algorithm as a baseline. Basically, a series of experiments were run using FCFS/FF-EASY to determine an average arrival rate from a Poisson process which would produce a desired average ready queue length. The same arrival rate was then used to simulate the other scheduling algorithms. Ready queue lengths were taken from $\{64, 128, 256\}$.

3.2 Performance Metrics

Two basic metrics were collected to compare the performance of these algorithms:

1. Average Response Time: This the sum of the wait time plus the execution time of a job, averaged over all jobs. This metric captures the ability of the scheduling algorithm to service the smaller, shorter jobs. Since the job stream contains a large number of smaller short jobs, the metric provides a good measure of the quality of the scheduling algorithm.

2. Weighted Average Response Time: The weight of a job is defined as the product of its resource requirement and execution time. The weighted average response time is then

the product of the job weight and the job response time, averaged over all jobs. This is basically a measure of how well the scheduling system provides progress to the large jobs. It is also a measure of overall resource utilization over time.

3.3 Simulation Results

Figures 2 (a)-(f) and 3 (a)-(f) summarize our simulation results. Each graph in the figures shows the results for one value of K, one K_i distribution, and the three system load values. Recall that the load value is referenced by the average job queue length achieved by the FCFS/FF-EASY scheduling algorithm. In each figure, the performance gains (or losses) of FCFS/BL-EASY, FCFS/BB-EASY, and FCFS/FF-CONS are normalized to the baseline performance of FCFS/FF-EASY. A positive performance gain indicates a performance increase while a negative performance gain indicates a poorer performance.

In general, the new FCFS/BB-EASY out performs all other algorithms for both performance metrics by up to 50%. The FCFS/FF-EASY generally out performs both FCFS/BL-EASY and FCFS/FF-CONS. FCFS/FF-CONS consistently performs much worse than the other algorithms due primarily to a diminished number of jobs which meet its stronger backfill requirements. The effects of each of the three parameters on the average response time metric is described below, followed by a comparison between the average and weighted average response time results.

3.3.1 Effect of K

As K increases, the BB heuristic maintains significant performance gain over FF-EASY. However, the performance gains achieved by the BL balancing heuristics decreases. This is primarily due to the inability of the heuristics to correctly *score* a match between the current system state and a job candidate in the ready queue. The error rate of the lowest-utilization scheme used by BL increases with K. Also note that the BL heuristic effectively partitions the set of candidate jobs into K subsets and selects a job from subset containing jobs in which the K_i resource requirement is largest. Therefore, in order for BL to be effective, it must have K times as many samples as the BB heuristic.

3.3.2 Effect of Relationship between K_is

The performance gains seen by the BB heuristic is approximately 10% higher when the K_is of a job are drawn from a uniform distribution as compared to an exponential distribution. This is due to the fact that the uniform distribution results in more of the K_is being *large* and harder to pack by the FF heuristics. However, the balancing heuristics were designed to deal with these variances. The exponential distribution creates K_is which are mostly *small* and therefore easier for the FF heuristics to pack so the performance gains are less.

3.3.3 Effect of System Load

As the system load increases, the performance gains achieved by the BB heuristic increase. This is due to the increased number of jobs in which the BB heuristic can draw from to improve the system resource utilization imbalance. This same increase in job numbers in the ready queue is actually the down fall of the BL heuristic which performs poorer at high loads. While it improves the sample size for BL, it also increases the probability that BL will heuristically skip over jobs selected by FF-EASY and BB-EASY in search of a job with a specific resource distribution, but which arrived later. This has the effect of increasing the waiting time of jobs which were skipped. So while the packing efficiency might improve (from the larger sample size), the average response time suffers. The net result is that the BL heuristic performs poorer than FF-EASY for large K.

3.3.4 Comparison of Average vs Weighted Average Response Time Results

In comparing figures 2 and 3, we see that the performance gains achieved by the BB heuristic in average response time are higher than the performance gains achieved in weighted average response time. While the BB achieved performance gains of $10 - 50\%$ in average response time, it achieves only gains of $10 - 30\%$ in weighted average response time. This implies that the BB heuristic is efficient at moving small jobs ahead of large jobs with minimal but positive impact to the waiting time of the large jobs. In many production sites, this is actually a desirable effect during peak daytime hours as it improves the response time of the small (and likely interactive) jobs.

4 Summary

In this report, we defined a new K-resource scheduling problem and provided two heuristics which may be used to extend current scheduling methods to work in this environment. The FCFS/BB-EASY backfill heuristic provides up to a 50% performance gain in average system response time and 40% performance in weighted average response time. The success of the FCFS/BB-EASY scheduling algorithm shows that balancing resource usage can be effective in increasing system performance in terms of the average and weighted average response time metrics. This success can be attributed to the fact that FCFS/BB-EASY can perform a better job packing of the small (common case) jobs while maintaining sufficient progress on the large jobs. Future work will address further support for large job scheduling for both online and offline scheduling of K-resource systems.

A second heuristic was adapted from two other domains: multi-constraint graph partitioning and multi-capacity bin-packing. The FCFS/BL-EASY scheduling algorithm failed for two reasons. First, the job size relative to the available system resources at any single scheduling epoch provides little opportunity for the heuristic to pick a single job which improves the system resource usage balance. This is due to the fact that the heuristic looks at a single (maximal) resource imbalance at a time which does not accurately capture the state

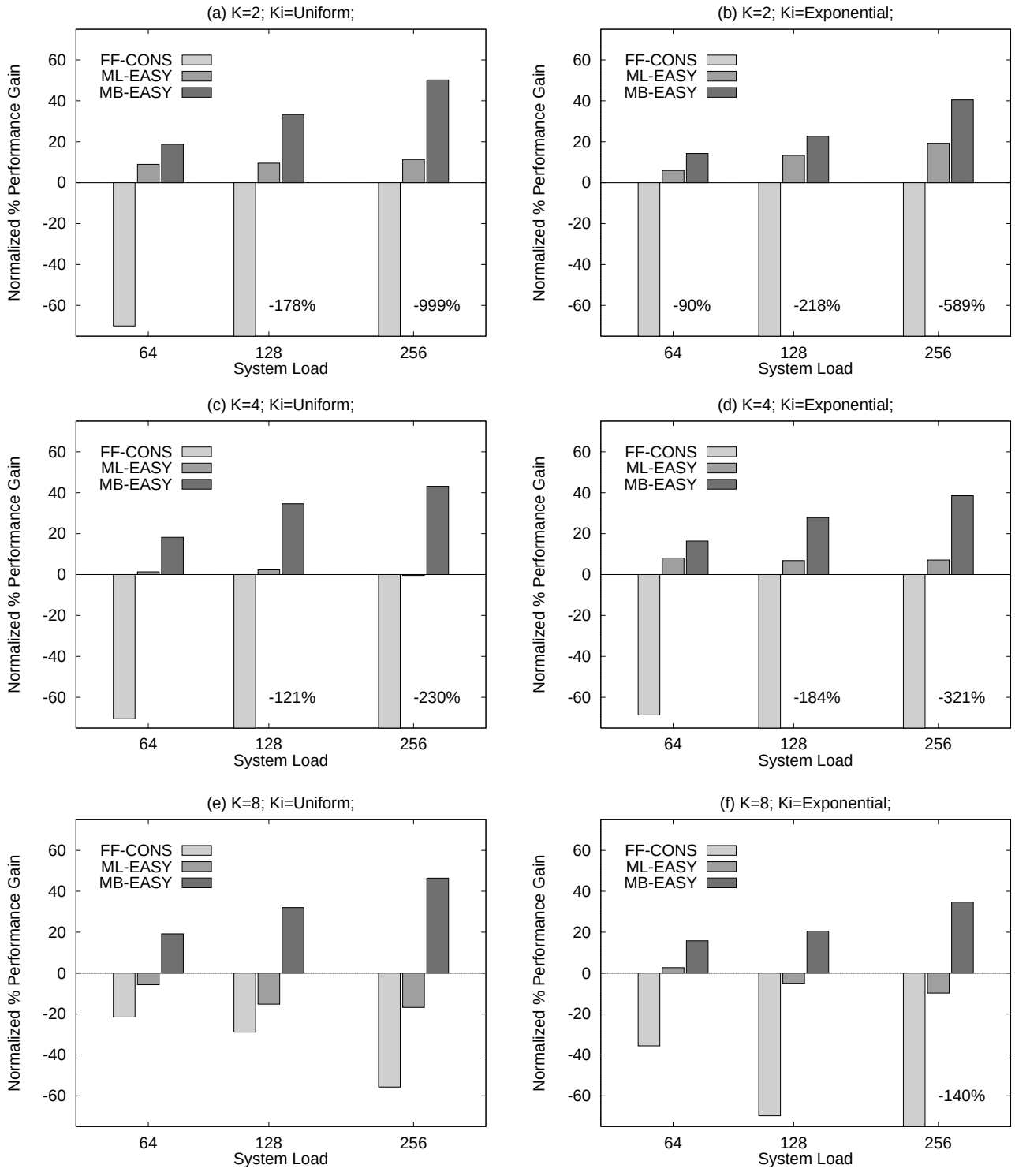


Figure 2: Average Response Time

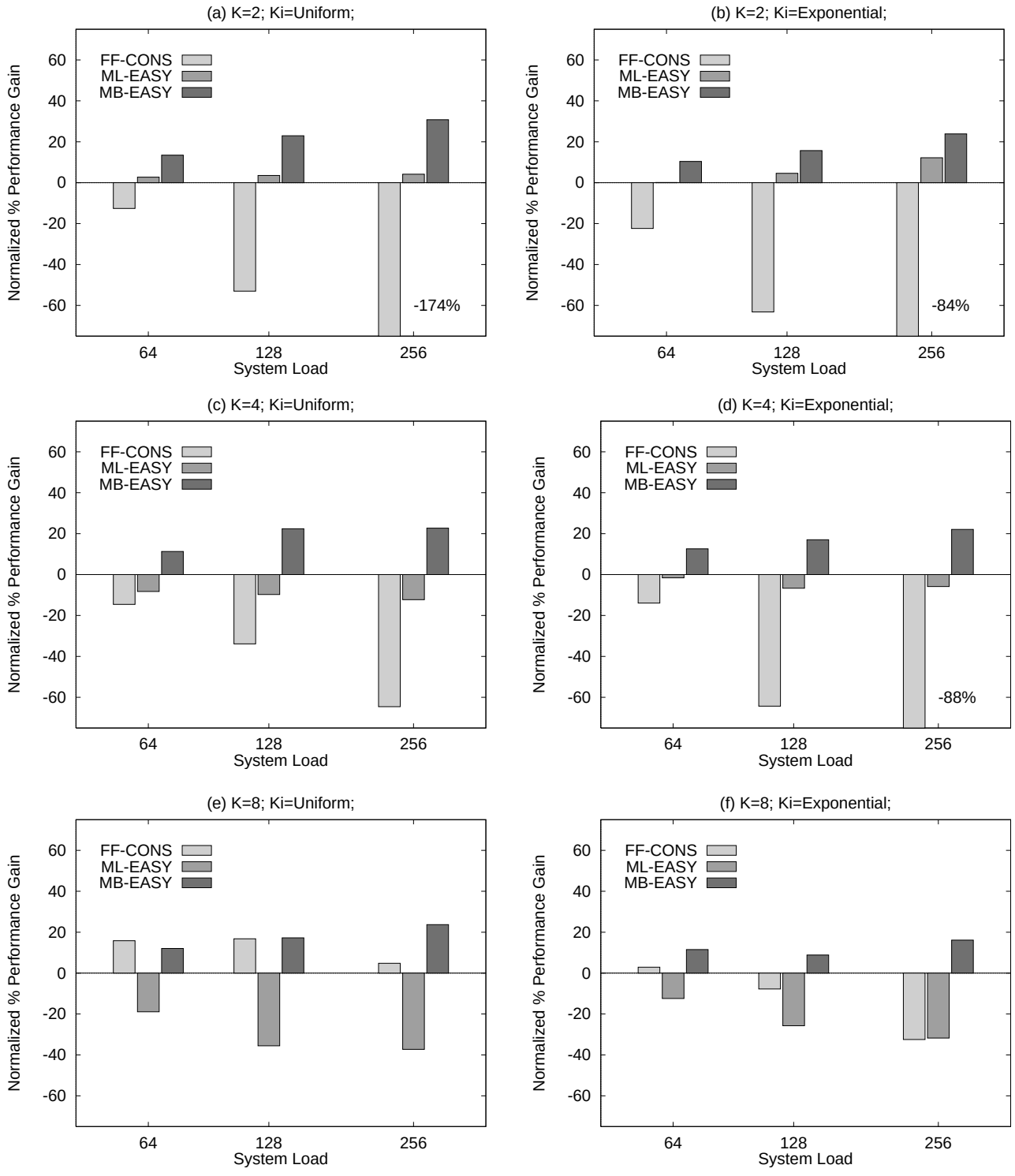


Figure 3: Weighted Average Response Time

of the system. Second, the small sample size in the job ready queue is further decreased (actually reordered) due to the implicit partitioning performed by the heuristic. Therefore, this heuristic, while appropriate for the original problem domains, is probably not appropriate for online job scheduling.

Additional work is required to further characterize the exact relationship between the resource requirements of a job in a production job stream.

References

[1] Jr. E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin-packing - an updated survey. In G. Ausiello, M. Lucertini, and P. Serafini, editors, *Algorithm Design for Computer System Design*, pages 49–99. Springer-Verlag, New York, 1984.

[2] D.G. Feitelson. Packing schemes for gang scheduling. In D.G. Feitelson and L. Rudolph, editors, *IPPS '96 Workshop: Job Scheduling Strategies for Parallel Processing*, volume 1162, pages 65–88. Springer-Verlag, New York, 1996. Lect. Notes in Computer Science.

[3] D.G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. In D.G. Feitelson and L. Rudolph, editors, *IPPS '95 Workshop: Job Scheduling Strategies for Parallel Processing*, volume 949, pages 1–18. Springer-Verlag, New York, 1995. Lect. Notes in Computer Science.

[4] D.G. Feitelson and A.M. Weil. Utilization and predictability in scheduling the ibm sp2 with backfilling. In *Proceedings of IPPS/SPDP 1998*, pages 542–546. IEEE Computer Society, 1998.

[5] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal of Computing*, 4(2):187–201, June 1975.

[6] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. Technical Report 98-019, University of Minnesota, Department of Computer Science, Army HPC Research Center, 1998.

[7] W. Leinberger, G. Karypis, and V. Kumar. Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints. In *submitted to ICPP 99*, 1999.

[8] D. Lifka. The anl/ibm sp scheduling system. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1995.

[9] R. K. Mansharamani and M. K. Vernon. Comparison of processor allocation policies for parallel systems. Technical report, Computer Sciences Department, University of Wisconsin, December 1993.

[10] R.K. Mansharamani and M.K. Vernon. Properties of the eqs parallel processor allocation policy. Technical Report 1192, Computer Science Department, University of Wisconsin-Madison, November 1993.

[11] K. Maruyama, S. K. Chang, and D. T. Tang. A general packing algorithm for multidimensional resource requirements. *International Journal of Computer and Information Sciences*, 6(2):131–149, May 1976.

[12] C. McCann and J. Zahorjan. Scheduling memory constrained jobs on distributed memory computers. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modelling of Computer Systems*, pages 208–219, 1996.

[13] E. W. Parsons and K. C. Sevcik. Coordinated allocation of memory and processors in multiprocessors. Technical report, Computer Systems Research Institute, University of Toronto, October 1995.

[14] E.W. Parsons and K. C. Sevcik. Implementing multiprocessor scheduling disciplines. Technical Report 356, Computer Systems Research Institute, University of Toronto, Canada, 1998.

[15] J. Skovira, W. Chan, H. Zhou, and D.Lifka. The easy-loadleveler api project. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162, pages 41–47. Springer-Verlag, New York, 1996. Lect. Notes in Computer Science.

[16] D. Talby and D. G. Feitelson. Supporting priorities and improving utilization of the ibm sp2 scheduler using slack-based backfilling. Technical report, Institute of Computer Science, The Hebrew University of Jerusalem, 1999.