Proceedings of the 2000 Winter Simulation Conference
J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds.

A SIMULATION MODEL OF BACKFILLING AND I/O SCHEDULING IN A PARTITIONABLE PARALLEL SYSTEM

Helen D. Karatza

Department of Informatics
Box 114
Aristotle University of Thessaloniki
54006 Thessaloniki, GREECE

ABSTRACT

A special type of scheduling called backfilling is presented using a parallel system upon which multiple jobs can be executed simultaneously. Jobs consist of parallel tasks scheduled to execute concurrently on processor partitions, where each task starts at the same time and computes at the same pace. The impact of I/O scheduling on system performance is also examined. The goal is to achieve high system performance and maintain fairness in terms of individual job execution. The performance of different backfilling schemes and different I/O scheduling strategies is compared over various processor service time coefficients of variation and for various degrees of multiprogramming. Simulation results demonstrate that backfilling improves system performance while preserving job sequencing. Also, the results show that when there is contention for the disk resources, trends in system can differ from those appearing in the research literature if I/O behavior is negligible or it is not explicitly considered.

1 INTRODUCTION

The efficient scheduling of multiprogrammed parallel systems has been a major research and development goal for many years. On one hand, users expect their individual jobs to achieve excellent performance. On the other hand, system resources must be judiciously allocated to satisfy the demands of jobs and produce the best overall performance. These objectives raise a number of scheduling policy issues with respect to large parallel computing environments (Dowdy et al. 1999, Feitelson 1994, Feitelson and Rudolph 1995, Karatza 1998).

Our work considers a shared memory system with 128 processors. We use a scalable, coherent shared address space (SAS) multiprocessing since it has been the focus of research in many other studies. Over the last decade, a number of hardware cache-coherent, non-uniform memory access architectures (so-called hardware-DSM or CC-NUMA machines) have been built and shown to perform well at the moderate scale of about 32 processors. In fact, such machines are fast becoming the dominant forms of tightly coupled multiprocessors built by commercial vendors. An open question is how scalable these architecture configurations are to larger processor counts. Jiang and Singh (1998) studied the performance of a wide range of SAS parallel applications on a 128-processor hardware cache-coherent machine (the SGI Origin2000). They showed that scalable performance is indeed be achieved with this programming model over a wide range of applications, including the challenging of kernels like FFT.

Most research in this area has focused only on the scheduling of processors. However, improvements in processor speed and main memory size have exposed I/O subsystems as a significant bottleneck that keeps applications from achieving full system utilization. This problem is more serious in multiprocessing systems where multiple processors must share the I/O subsystem. Therefore, I/O scheduling should be examined along with parallel job scheduling.

This study considers a partitionable parallel processing system where the partitions are subsystems allocated to independent jobs. Jobs consist of parallel tasks that are scheduled to execute concurrently on a set of processors. The parallel tasks need to start at essentially the same time, co-ordinate their execution, and compute at the same pace. This type of resource management is called “coscheduling” or “gang scheduling” and has been extensively studied in the literature of distributed and shared memory systems (Feitelson and Jette 1997, Feitelson and Rudolph 1995, Karatza 1999a, Karatza 1999b, Setia 1997, Wang, Papaefthymiou, and Squillante 1997).

Jobs start to execute only if enough idle processors are available to handle them. However, a scheduling policy is needed to determine which parallel program is to be mapped to the available processors. Job sequencing needs to be preserved as much as possible in order to achieve fairness in job execution.

In multiprogrammed parallel systems, processor partitions are usually allocated on a FCFS basis. This approach can result in severe fragmentation, because processors that cannot fulfill demands of the next job in the queue remain idle until the needed resources are freed. To avoid fragmentation, a non-FCFS policy for queuing waiting jobs on a partitionable system should be used.

When several small jobs running and a job that requires the entire system is at the head of the scheduling queue, a FCFS scheduler will reserve freed processors until the entire system is available, keeping processors idle until the final running job has terminated. A non-FCFS scheduler can execute small jobs from the back of the queue until the last job finishes, thus improving the total utilization of the system. Such an approach is called backfilling or aggressive backfilling. Idle processors are assigned small jobs at the back of the queue on the condition they do not delay the large job in the first queue position. Clearly, such a system should also be cautiously designed avoid starving large jobs. In reality, backfilling is just a natural extension of FCFS scheduling.

Research on backfilling includes Feitelson and Weil (1998), Talby and Feitelson (1999). Feitelson and Weil (1998) show that a conservative approach, where small jobs move ahead only if they do not delay any job in the queue, produces the same utilization as aggressive backfilling, but it has the advantage that it does not normally cause unbounded queuing time. However, the conservative approach involves extra overhead in that it examines all jobs in the queue for possible delay.

Talby and Feitelson (1999) apply a technique called slack-based backfilling where the scheduler gives each waiting job a slack (time delay) that determines how long it will wait before running: 'Important' or 'lengthy' jobs that have little slack in comparison to others. When a new job is submitted, all possible schedules are priced out according to utilization and priority considerations and so long as no job is delayed beyond its slack, the cheapest schedule is selected.

This paper examines the aggressive backfilling method, and proposes a version of it where a job in the queue is scheduled if it will not delay the first job for more than a small time interval. We also examine the strict FCFS policy for comparison purposes.

Generally, other papers found in the literature study processor scheduling only. They do not explicitly model the I/O processing, even though it can significantly influence the overall system performance. However, scheduling is not an isolated issue. It is only a single service provided by the operating system. Any solution to the scheduling problem must be integrated with other problem solutions, e.g. I/O management. Different parts of the system must work together to create a cohesive whole in such a way that it makes sense. The Rosti et al. (1998) study of large-scale parallel computer systems suggests that the overlapping of I/O demands of some jobs with the computation demands of other jobs offer a potential improvement in performance.

I/O scheduling is examined in Kwong and Majumdar (1999), Seltzer, Chen, and Ousterhout (1990), Worthington, Ganger, and Patt (1994). However these papers do not consider processor scheduling. Our previous papers Karatza (1998, 1999a, 1999b) examine processor scheduling, but only the FCFS scheduling policy is applied at the I/O unit. We employ three I/O scheduling algorithms – FCFS, Shortest Time First, and Weighted Shortest Time First. The last algorithm uses the standard Shortest Time First technique, but also applies an aging function to the times computed. The performance of these algorithms has been studied in conjunction with gang scheduling in a distributed system in Karatza (2000). In that paper, each processor is equipped with its own queue, each parallel task of a job is assigned to a different processor queue, and the scheduling method at the processor queues is different from each of the methods that we used in this research.

The design choices considered in this paper include different ways to schedule jobs for service on the system processors and on the I/O subsystem. The performance of the different scheduling policies is compared for various coefficients of variation of the processor service times and for different degrees of multiprogramming. The author has not found a combined analysis of backfilling scheduling and I/O scheduling anywhere else in the research literature.

This paper is theoretical in that the results are obtained from simulation studies instead of from the measurements of real systems. Nevertheless, the results presented are of practical value. All of the algorithms are practical in that they can be implemented. Although we do not derive absolute performance values for specific systems and workloads, we do study the relative performance of the different algorithms across a broad range of workloads and analyze how changes in the workload can affect performance.

Some simple idealized systems can be mathematically analyzed using techniques such as queuing theory to determine performance measures. In addition to exponential distribution for job processing times, our system includes Branching Erlang. It also applies scheduling policies with different complexities. For complex systems, analytical modeling typically requires additional simplifying assumptions, and those assumptions frequently have unforeseeable influences on the results. Therefore, research efforts have been devoted to finding approximate analysis, developing tractable models for special cases, and conducting simulations. We chose simulations because it is possible to simulate the system under study in a direct manner, thus lending credibility to the results. Detailed simulation models help determine performance bottlenecks in architecture and assist in refining the system configuration.

The structure of this paper is as follows. Section 2.1 specifies system and workload models, sections 2.2 and 2.3 describe the processor and the I/O scheduling strategies, and section 2.4 presents the metrics employed while assessing the performance of the scheduling policies. Model implementation and input parameters are described in section 3.1 while the results of the simulation experiments are presented and analyzed in section 3.2. Section 4 is the conclusion and provides suggestions for further research, and the last section is references.

2 MODEL AND METHODOLOGY

2.1 System and Workload Models

A closed queuing network model is considered that consists of $P = 128$ parallel homogeneous processors and a multiserver disk center. This allows files to be stripped across a variable number of disks, and a natural way to capture the effects of disk striping is via a fork-join system. We considered that each I/O request forks sub-requests that can be served by the parallel disk servers.

All processors share a single queue (memory). The effects of the memory requirements and the communication latencies are not represented explicitly in the system model. Instead, they appear implicitly at job execution time. By covering several different types of job execution behaviors, we expect that various architectural characteristics will be captured, as well.

Since we are interested in a system with balanced program flow, we considered an I/O subsystem with the same service capacity as the processing unit. The model is considered closed since the degree of multiprogramming N is constant. The configuration of the model is shown in Figure 1.

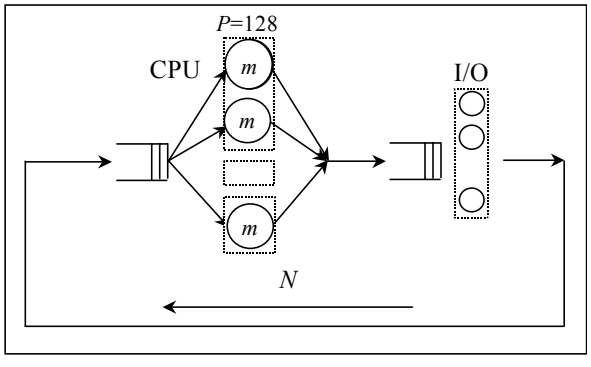

Figure 1: The Queuing Network Model

A technique used to evaluate the performance of the scheduling disciplines is experimentation using a synthetic workload simulation. In studies like this, one is usually required to use synthetic workloads because real workloads cannot be simulated efficiency enough and real systems with actual workloads are not available for experimentation. Also, useful analytic models are difficult to derive because the subtleties between various disciplines are difficult to model and because the workload model is quite complex.

A partitionable parallel processing system is used which dynamically allocates jobs to processor subsystems. Jobs consist of a set of $n \geq 1$ tasks where each task executes on one processor. Processors are allocated at job initiation and once they are committed to a job they cannot be reallocated until the job terminates. The number of processors required by job x is represented as $p(x)$, and is called the "size" of job x. A job is said to be "small" (or "large") if it requires a small (or large) number of processors. The $p(x)$ processors must be allocated simultaneously to job x, and once they are allocated, they are held by job x until its completion. Jobs $x_1, x_2, \ldots x_l$ can be executed simultaneously if and only if the following relation holds:

$$p(x_1)+p(x_2)+\ldots+p(x_l) \leq P.$$

Each job begins execution only when enough idle processors are available to meet its needs. When a job terminates execution, all processors assigned to it are reclaimed. Each time a job returns from I/O service to parallel processors, it needs a different number of processors for execution, that is its degree of parallelism is not constant during a job's lifetime in the system. The sequence that jobs in the queue are served depends on the scheduling policy. Fairness is required across competing jobs.

The workload considered here is characterized by the following four parameters: distribution of job sizes, distribution of processor service times, distribution of I/O service times, and the degree of multiprogramming. We assume that there is no correlation between job size and processor service demand. For example, a small job may have a long processor service time. We also assume that job sizes are uniformly distributed over the range [1..128]. The number of jobs that can be processed in parallel, depends on job sizes and on the scheduling policy applied.

Numerous scheduling disciplines have been proposed for multiprocessor systems, the evaluation of which, for the most part, has been conducted on workloads with a relatively small variability in job processing requirements. However, high performance computer centers report that their service time coefficient of variation can in fact be greater than one.

We investigate the impact of varying processor service times on system performance. A high variability in job service demand implies that there is proportionately a high number of service demands that are very small in comparison to the mean processor service time, and a comparatively low number of service demands that are very large. When a job with a long service demand enters

the processor system and begins execution, it will occupy a number of processors for a long time and depending on the scheduling policy applied, it may introduce inordinate queuing delays for the other jobs waiting for service.

The parameter, which represents the variability in job execution times, is the coefficient of variation of processor service time (C). This is the ratio of the standard deviation of processor service times to its mean. We examine the following cases:

- Processor service times are independent and identically distributed (IID) exponential random variables with a mean m.
- Processor service times have a Branching Erlang distribution (Bolch et al. 1998) with two stages and are IID. The coefficient of variation is C, where $C > 1$ and the mean is m.

After processor service, a job requests service from the I/O subsystem.

- The I/O service times are exponentially distributed with a mean k and are IID.

Next we describe the scheduling strategies employed in this work. As with most studies we assume that scheduling overhead is negligible.

2.2 Processor Scheduling Policies

We assume that the scheduler has perfect information when making decisions, i.e. it knows:

- The exact number of processors required by all jobs in the queue.
- Job service demands.

We analyze performance of the following policies:

- *First Come First Served* (*FCFS*). When a job leaves the system, the first job in the ready queue is examined. If the job does not fit into any of the available processors, it is not scheduled and furthermore, no other jobs are scheduled. If the first job does fit, it is scheduled, and while there are more processors available, remaining jobs in the ready queue are scheduled in the order they arrived.

  Unfortunately, with the FCFS policy, jobs may be retained in the ready queue even when there are a sufficient number of idle processors to handle them. The following two methods can eliminate this problem.

- *Backfilling* (*BF*). Backfilling is the process of allowing small jobs to run, on the condition they do not delay a large job that is waiting at the head of the scheduling queue. This policy assumes that a priori knowledge about a job is available in form of a service demand. When such knowledge is available, the time when a job will terminate execution and release the processors needed by the large job can be predicted. It should be noted, however, that a priori information is not often available and only an estimate of task execution time is possible. In this study, the estimated job execution time is uniformly distributed within ± E% of the exact value.
- *Loose Backfilling* (*LBF*). A job in the scheduling queue starts execution if its predicted service time assures that the job will not delay the first job in the queue for a time interval that is larger than some value d. For d=0 this method is the same as BF.

2.3 I/O Scheduling Policies

- *First Come First Served* (*FCFS*). This disk scheduling policy often results in suboptimal performance. Many scheduling algorithms have been proposed that achieve higher performance by taking into account information about individual requests. Frequently, this policy is used as a performance yardstick against which the other policies are compared to determine whether the job based I/O scheduling procedures produce any performance benefits.
- *Shortest Time First* (*STF*). This policy chooses the request which yields the shortest I/O time, and includes both seek time and the rotational latency. STF should yield the best throughput since the fastest I/O service is always selected. The algorithm scans the entire queue calculating the time each request will consume. It then selects that request with the shortest expected service time. This policy is denoted as Shortest Positioning Time First (SPTF) in Worthington, Ganger, and Patt (1994). It is obvious that this method is not fair to a job in the I/O queue with very large service demand since it may be scheduled only after a very long wait in the queue. In this paper, however, we do not consider highly variable I/O service times (C=1). For this reason we do not encounter very large I/O service times that can cause unbounded job delays in the I/O queue.
- *Weighted Shortest Time First* (*WSTF*). This method is a version of the STF strategy and is based on algorithms proposed by Seltzer, Chen,

and Ousterhout (1990), Worthington, Ganger, and Patt (1994). Priority is assigned to requests that have been in the pending queue for excessive periods of time. The priority may slowly increase as the request ages, or a time limit may be set after which requests are served on a FCFS basis. This algorithm uses the standard shortest time first technique, but it applies an aging function to the times computed as follows:

1. First, we assume that the FCFS policy is applied to jobs that are waiting in the queue for a time interval greater than 10 * k, where k is the mean I/O subsystem service time. This means that these jobs are given the highest priority, and that the expected number of other jobs, which have bypassed them in the I/O subsystem, cannot be greater than 10.
2. For each STF calculation, the actual I/O time is multiplied by a weighting value W. W is computed by calculating how much time is left before this request will exceed the time interval 10 * k. Thus, the weighted time can be calculated in the following way.

Let:

T_w be the Weighted Time,
T_{real} be the actual I/O Time,
M be equal to 10 * k,
T_E be the Elapsed Time since this request arrived.

Then:

$$T_w = T_{real} * (M - T_E)/M.$$

Implementation Issues: Although processor scheduling policies that based on explicit knowledge of job characteristics offer excellent performance potential, their implementation has been difficult on general purpose systems because it is hard to acquire such a priori knowledge. In comparison to processor scheduling, policies based on job characteristics are easier to implement in the context of I/O scheduling. It is possible for the operating system to keep track of the age of a job and estimate I/O demand associated with a request. Consequently, it is quite possible to implement policies such as STF or WSTF on a real system. In this study, the I/O estimated service time is uniformly distributed within ± E% of the exact value.

When we use priorities and a tie occurs, FCFS is used to break the tie. Next, when we use a notation of the form: "policy a – policy b" we mean that "policy a" is a processor scheduling policy, while "policy b" is an I/O scheduling policy. For example, BF-STF means that we use BF processor scheduling and STF I/O scheduling.

2.4 Performance Metrics

Consider the following definitions:

Response time of a job is the time interval from the arrival of that job at the processor queue to the service completion time for that job (i.e., time spent in the processor queue plus job service time).

Cycle time of a job is the elapsed time between two successive service requests for a job on the processors. This includes processor queuing and service times, plus I/O queuing and service times.

Parameters used in simulations computations (presented later) are shown in Table 1.

Table 1: Notations

RT	Mean response time
K	Mean cycle time
R	System throughput
U_{proc}	Mean processor utilization
$U_{I/O}$	Mean I/O unit utilization
N	Degree of multiprogramming
m	Mean processor service time
k	Mean I/O service time
E	Estimation error in service time

System throughput (system performance) and mean cycle time (program performance) determine the overall performance of the model.

3 SIMULATION RESULTS AND DISCUSSION

3.1 Model Implementation and Input Parameters

The queuing network model was simulated with discrete event simulation models (Law and Kelton 1991) using the independent replication method. For every mean value, a 95% confidence interval was computed. All confidence intervals were within 5% of the mean values.

A balanced system with m=1.0 and k = 0.504 was considered. The value k=0.504 was chosen for balanced program flow because the processors average 64.5 tasks per job. When all processors are busy, an average of 1.9845 jobs are served each unit of time. This implies that I/O mean service time must be equal to 1/1.9845 = 0.504 if the I/O unit is to have the same service capacity.

The system was examined in cases of job execution times with exponential distribution (C = 1), and Branching Erlang for C = 2, 4. The degree of multiprogramming N was 8, 12, 16, 20, and 24. The reason for examining

different degrees of multiprogramming is that it is a critical parameter in determining system load.

In the LBF case *d* was taken as 0.1, 0.2, 0.3. This is a reasonable assumption taken into account that the mean processor service time is equal to 1. In the cases where estimation of service time is required we have also examined estimation errors ±10% and ±30%.

3.2 Performance Analysis

Due to space limitations only the following results are presented:

- Tables 2-3. Performance parameters for the FCFS-STF, and BF-STF cases for *C*=1.
- Figures 2-7. FCFS, BF and LBF processor scheduling policies are combined with the FCFS, and the STF I/O scheduling methods.
- Figures 8-13. The performance of the FCFS and BF processor scheduling policies are shown when they are combined with each one of the three I/O scheduling policies.
- Figure 14. Throughput in the BF-STF, *C*=4 case for *E*=0%, 10%, 20%, 30%.

Table 2: *C*=1, FCFS-STF Case

N	U_{proc}	$U_{I/O}$	*RT*	*K*	*R*
8	0.69	0.69	4.55	5.81	1.38
12	0.70	0.70	7.32	8.65	1.39
16	0.70	0.70	10.19	11.53	1.39
20	0.70	0.70	13.07	14.41	1.39
24	0.70	0.70	15.94	17.30	1.39

Table 3: *C*=1, BF-STF Case

N	U_{proc}	$U_{I/O}$	*RT*	*K*	*R*
8	0.73	0.73	4.22	5.51	1.45
12	0.74	0.74	6.76	8.14	1.47
16	0.74	0.74	9.39	10.79	1.48
20	0.75	0.75	11.87	13.36	1.50
24	0.75	0.75	14.63	16.07	1.49

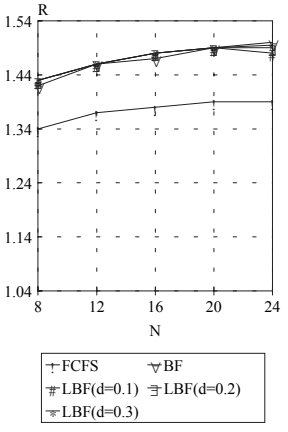

Figure 2: *R* versus *N*, *C*=1, FCFS I/O Scheduling

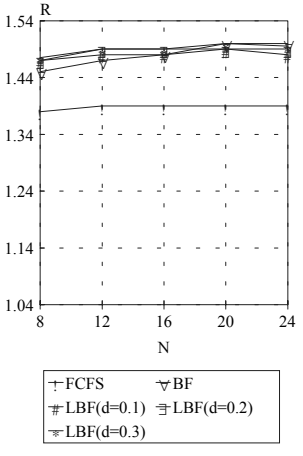

Figure 3: *R* versus *N*, *C*=1, STF I/O Scheduling

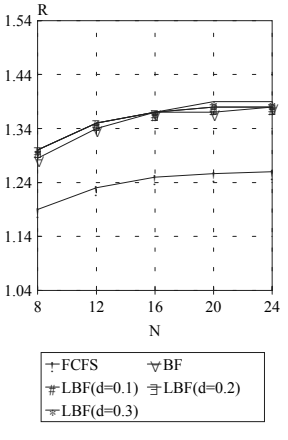

Figure 4: *R* versus *N*, *C*=2, FCFS I/O Scheduling

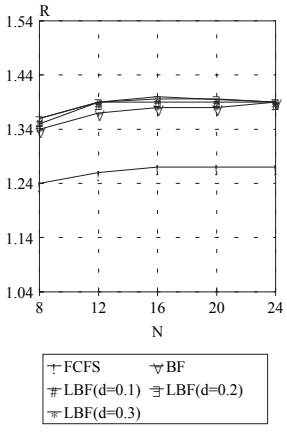

Figure 5: *R* versus *N*, *C*=2, STF I/O Scheduling

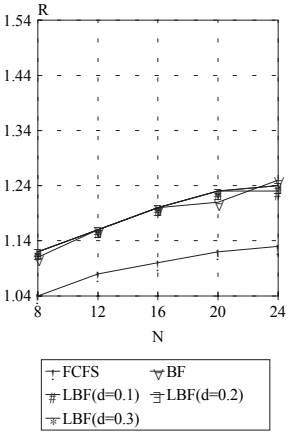

Figure 6: *R* versus *N*, *C*=4, FCFS I/O Scheduling

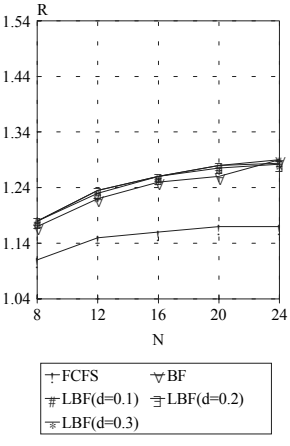

Figure 7: *R* versus *N*, *C*=4, STF I/O Scheduling

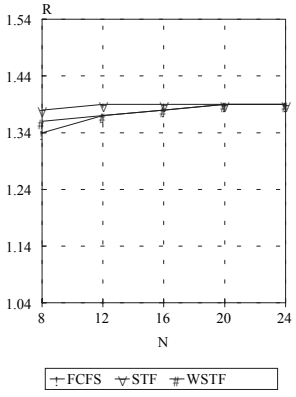

Figure 8: *R* versus *N*, *C*=1, FCFS Processor Scheduling

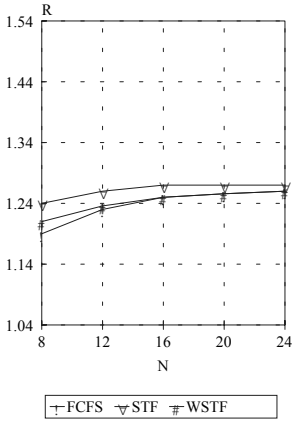

Figure 9: *R* versus *N*, *C*=2, FCFS Processor Scheduling

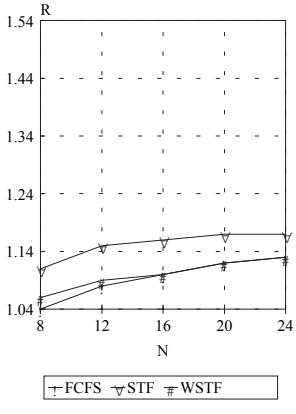

Figure 10: *R* versus *N*, *C*=4, FCFS Processor Scheduling

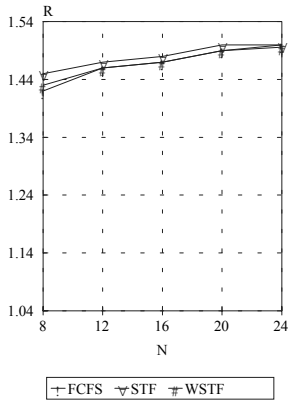

Figure 11: *R* versus *N*, *C*=1, BF Processor Scheduling

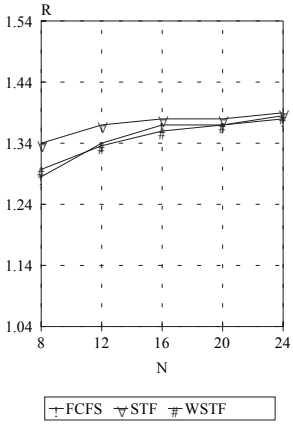

Figure 12: *R* versus *N*, *C*=2, BF Processor Scheduling

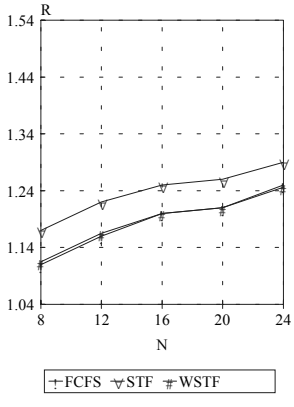

Figure 13: *R* versus *N*, *C*=4, BF Processor Scheduling

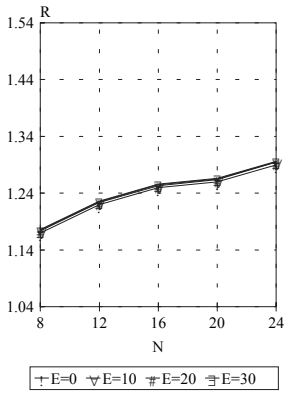

Figure 14: *R* versus *N*, *C*=4, BF-STF (*E*: Estimation Error)

The results demonstrate the following:

As far as overall performance is concerned, the LBF method performs almost the same as BF for all *d*. Actually, in most cases LBF performed only slightly better than BF.

The worst system performance is encountered with the FCFS method. This is because FCFS yields higher *RT* than the other methods, resulting in higher mean cycle time and lower system throughput. Utilization is also lower in the FCFS case than it is in BF and LBF cases.

For all *N* and *C* the difference in performance between the FCFS processor scheduling policy and each one of BF, and LBF policies varies between 6.5% to 10.5% (relative increase). This result held true for any of the I/O scheduling policies that we employed.

Regarding I/O scheduling, the WSTF I/O method performed almost the same as the FCFS I/O scheduling. Therefore, considering the FCFS and the WSTF methods at the I/O subsystem, the FCFS method is preferable since it is easier to implement it in practice, incurs less overhead, and is the fairest of the other I/O methods.

The STF method performs better than the other two I/O policies. This I/O method demonstrates higher superiority at low degrees of multiprogramming. This is due to the fact that all three processor-scheduling methods examined are conservative in job scheduling, as they seek to preserve job sequence. Jobs tend to be kept in processor queues when they might be scheduled if a different scheduling method were applied. The blocking of jobs is higher at high *N*. Therefore, no matter which scheduling method is applied at the I/O unit, the delay of jobs in the processor queues is a factor that restricts performance.

The level of the superiority of STF over FCFS I/O scheduling was different for the various cases examined. For example, when STF is combined with the FCFS processor scheduling policy (FCFS-STF case), the largest superiority of it over FCFS-FCFS was 3%, 4.2% and 6.7% for the *C*=1, 2, 4 cases respectively. When STF is combined with the BF method the largest superiority of BF-STF over BF-FCFS was 2%, 4%, and 5.4% for *C*=1, 2, and 4 respectively.

Generally, simulation results reveal that the effect of I/O scheduling on system performance is more significant at C=4 than at C<4. This is due to the fact that at C>1 job service times present high variability. In this case, the majority of jobs have small service times and are served very fast by the processors, but then they have to wait at the I/O subsystem. This is the reason that the STF I/O method potential is better exploited at C=4.

Additional simulation experiments were conducted to assess the impact of service time estimation error on the performance of the scheduling methods that require a priori knowledge of service times. Figure 14 shows the effect of service time estimation error on system throughput for the BF-STF, C=4 case. The estimation error in this figure is set at ±0%, ±10%, ±20%, and ±30%. The graph shows that the estimation error in processor and I/O service times marginally affect system performance. Therefore, no profit can be gained from the a priori knowledge of exact service times.

All policies have their merit:

Regarding processor scheduling policies, FCFS is easier to implement and results in less overhead than backfilling. BF, and LBF methods assume a priori knowledge of an approximate job execution time, but these methods can perform better than the FCFS strategy. BF performs almost the same as LBF and is fairer than LBF because it preserves job sequence.

Regarding I/O scheduling, STF needs advance information about I/O service time. This method achieves performance improvement only in certain cases. On the other hand WSTF performs very close to FCFS and results in more overhead than the STF method.

The above observations indicate that the BF-STF is best for C=4. For C<4, the BF-STF method should be used only for low N, while for high N the BF-FCFS method is more appropriate.

4 CONCLUSIONS AND FURTHER RESEARCH

This research studies backfilling in conjunction with I/O scheduling in a partitionable parallel processing system. We use simulation as the means of generating results used to compare different configurations.

Three processor scheduling policies were considered (FCFS and two backfilling methods – BF and LBF) as well as three I/O scheduling methods (FCFS, STF, and WSTF). Their performance was simulated and then compared for various degrees of multiprogramming N and coefficients of variation C of processor service times.

The simulation results reveal the following:

- The BF policy as compared with the other methods exhibits good system performance and fairness.
- Among the three I/O scheduling methods, for C = 4 the STF method is recommended. For C < 4 STF should be used for low N and FCFS should be used for high N since it is easier to implement, fair, and it performs close to STF.
- Backfilling and I/O scheduling can tolerate estimation errors in job service time.

This work is a case study. It can be extended as follows:

- Different job size distributions could be considered.
- Different distributions of I/O service times could be examined.
- The overhead involved with backfilling could be more accurately taken into account.

REFERENCES

Bolch, G., S. Greiner, H. de Meer, and K. S. Trivedi. 1998. *Queuing networks and Markov chains*. New York: J. Wiley & Sons, Inc.

Dowdy, L.W., E. Rosti, G. Serazzi, and E. Smirni. 1999. Scheduling issues in high-performance computing. *Performance Evaluation Review* 26 (4): 60-69.

Feitelson, D.G. 1994. A Survey of scheduling in multiprogrammed parallel systems. Research Report RC 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, New York.

Feitelson, D.G., and M.A. Jette. 1997. Improved utilization and responsiveness with gang scheduling. In *Job Scheduling Strategies for Parallel Processing*, *Lecture Notes in Computer Science*, ed. D.G. Feitelson, and L. Rudolph, 1291: 238-261. Berlin: Springer-Verlang.

Feitelson, D.G., and L. Rudolph. 1995. Parallel job scheduling: issues and approaches. In *Job Scheduling Strategies for Parallel Processing*, *Lecture Notes in Computer Science*, ed. D.G. Feitelson, and L. Rudolph, 949: 1-18. Berlin: Springer-Verlang.

Feitelson, D.G., and L. Rudolph. 1995. Coscheduling based on runtime identification of activity working sets. *International Journal of Parallel Programming* 23 (2): 135-160.

Feitelson, D.G., and A.M. Weil. 1998. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *Proceedings of the 12th International Parallel Processing Symposium*, 542-546. IEEE Computer Society, Los Alamitos, California.

Jiang, D., and J. Pal Singh. 1998. Scaling application performance on cache-coherent multiprocessors. *Performance Evaluation Review* 26 (1): 171-181.

Karatza, H.D. 1998. Eager scheduling versus lazy scheduling with resequencing. In *Proceedings of 1998*

Symposium on Performance Evaluation of Computer and Telecommunication Systems, ed. M.S. Obaidat, and H. Khalid, 261-267. The Society for Computer Simulation International, San Diego, California.

Karatza, H.D. 1999a. A Simulation-based performance analysis of gang scheduling in a distributed system. In *Proceedings of 32nd Annual Simulation Symposium*, 26-33. IEEE Computer Society, Los Alamitos, California.

Karatza, H.D. 1999b. Coscheduling in a partitionable parallel processing system. In *Proceedings of 7th Hellenic Conference on Informatics,* ed. D.I. Fotiadis, and S.D. Nikolopoulos, IV: 29-37. University of Ioannina Press, Ioannina, Greece.

Karatza, H.D. 2000. Gang scheduling and I/O scheduling in a multiprocessor system. In *Proceedings of 2000 Symposium on Performance Evaluation of Computer and Telecommunication Systems*, ed. M.S. Obaidat, F. Davoli, and M.A. Marsan, 245-252. The Society for Computer Simulation International, San Diego, California.

Kwong, P., and S. Majumdar. 1999. Scheduling of I/O in multiprogrammed parallel systems. *Informatica* 23: 67-76.

Law, A., and D. Kelton. 1991. *Simulation modeling and analysis*. 2d ed. New York: McGraw-Hill, Inc.

Rosti, E., G. Serazzi, E. Smirni, and M. Squillante. 1998. The impact of I/O on program behavior and parallel scheduling. *Performance Evaluation Review* 26 (1): 56-65.

Seltzer, M., P. Chen, and J. Ousterhout. 1990. Disk scheduling revisited. In *Proceedings of the 1990 Winter Usenix*, 313-324. Usenix Association, Berkeley, California.

Setia, S.K. 1997. Trace-driven analysis of migration-based gang scheduling policies for parallel computers. In *Proceedings of the International Conference on Parallel Processing*, 489-492. IEEE Computer Society, Los Alamitos, California.

Talby, D., and D.G. Feitelson. 1999. Supporting priorities and improved utilization of the IBM SP2 scheduler using slack-based backfilling. In *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*. 513-517. IEEE Computer Society, Los Alamitos, California.

Wang, F., M. Papaefthymiou, and M.S. Squillante. 1997. Performance evaluation of gang scheduling for parallel and distributed systems. In *Job Scheduling Strategies for Parallel Processing*, *Lecture Notes in Computer Science*, ed. D.G. Feitelson, and L. Rudolph, 1291: 184-195. Berlin: Springer-Verlang.

Worthington, B.L., G.R. Ganger, and Y.N. Patt. 1994. Scheduling algorithms for modern disk drives. In *Proceedings of the ACM Sigmetrics Conference*, 241-251. The Association for Computing Machinery, New York.

AUTHOR BIOGRAPHY

HELEN D. KARATZA is an Assistant Professor at the Department of Informatics at the Aristotle University of Thessaloniki, Greece. Her research interests mainly include Performance Evaluation of Parallel and Distributed Systems, Multiprocessor Scheduling and Simulation. Her email and web address are <karatza@csd.auth.gr> and <www.csd.auth.gr/~karatza>.