A Robust and Flexible Microeconomic Scheduler for Parallel Computers

Ion Stoica* Alex Pothen†

1 Introduction

We have considered the problem of scheduling on-line a set of jobs on a parallel computer with identical processors in earlier work [6], and have described an algorithm for this problem based on microeconomic ideas. In this paper we compare, through simulation experiments, the microeconomic scheduler that we have developed with other scheduling policies. We have tried to design a systematic set of experiments that explore various regions of the parameter space, and thereby to characterize robust scheduling policies. We explore three variants of the microeconomic approach and show how they permit trade-offs between mutually antagonistic goals such as high system utilization and low user response times. The microeconomic approach has the additional advantages of maintaining fairness at the user level and providing each user with control over the performance of his or her jobs.

The work we describe should be useful at many parallel computer installations and workstation clusters in the world, where system administrators face the task of scheduling the jobs submitted on their machines. EASY, a job scheduler for parallel computers developed at Argonne National Labs by Lifka [3], and used at several institutions, uses the first-come-first-served policy with job reservations. This can be shown to be a special case of the microeconomic approach (when "income rates" are set to zero in this policy). Our work shows that the microeconomic approach is capable of modeling other commonly used scheduling policies as well. Furthermore, our results show that the microeconomic scheduler is more robust than the other schedulers included in this study.

The rest of this paper is organized as follows. In Section 2, we briefly describe the microeconomic model of the job scheduling problem and a scheduler based on this approach. We are necessarily brief here due to space considerations, and refer the reader to our earlier

*Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162 (stoica.cs.odu.edu).

†Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162 and ICASE, MS 132C, NASA Langley Research Center, Hampton, VA 23681-0001 (pothen@cs.odu.edu pothen@icase.edu). This author was supported by National Science Foundation grant CCR-9412698, by U. S. Department of Energy grant DE-FG05-94ER25216, and by NASA under Contract NAS1-19480 while in residence at ICASE.

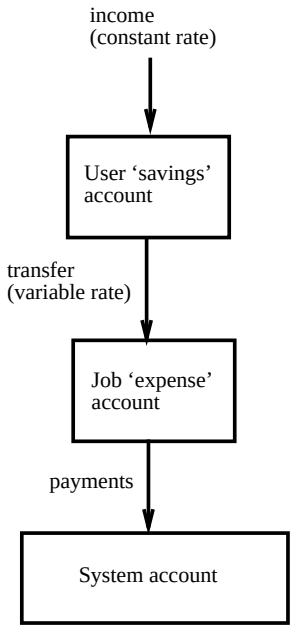

Figure 1: The currency flow.

paper [6] for additional details. A discussion of earlier economic approaches for resource allocation, load balancing, and memory allocation can also be found there. Section 3.1 describes the various policies that we compare, the experimental setting, and the design of our simulations; Section 3.2 describes the results we have obtained. We summarize in Section 4.

2 The Model

We consider a parallel computer consisting of N identical processors interconnected by a general communication network. We assume that the communication parameters for any pair of processors do not depend on their relative position, [1] and therefore the system may be arbitrarily partitioned. Every job specifies, upon its arrival, the number of processors p it needs, and the estimated computation time. Once processors are allocated, they are guaranteed to be exclusively used by the job for the entire duration of its execution. Also, the job is assumed to acquire or release all p processors at the same time.

The computation system is modeled as a microeconomic environment in which different *users* compete for obtaining system *resources* in order to run their *jobs*. To get the requested resources the user has to pay the price asked by the system. As in real life, the buyers (users) and the sellers (system) have antagonistic goals; the users wish to run their jobs as fast as possible with minimum expenses, while the system wants to maximize its income.

The flow of currency in the system is depicted in Figure 1. Every user has a *savings account*

[1] This is a reasonable assumption for many modern multiprocessor architectures (e.g., IBM SP-1/2, Intel Paragon).

in which he receives money at a constant rate, as long as he has less than a specified amount of funds. Whenever a user decides to run a job, he creates an *expense account* for that job to which money from his savings account is transferred. The job uses this account to buy the resources it needs. Once the job is scheduled for execution, all of its money is transferred to the *system account*. In order to maximize the system income, the scheduler applies a simple strategy: it allocates available resources to the job that offers the best price. In a loaded system, it is possible that not all p processors that were requested by a job become available at the same time. In this case, when the job is scheduled it is asked to pay for the wasted resources also to discourage fragmentation. For convenience, we refer to a unit of time as a *minute* and a unit of funds as a *dollar*. More details can be found in [6].

3 Experiments

3.1 Experimental setting

We use a simple simulator [6] that models a homogeneous parallel computer with $N = 128$ processors and 10 independent users. We group the jobs into three classes depending on the computation time and the number of processors required. The jobs are taken from a single Poisson source with mean arrival rate λ (measured in jobs/minute). By the decomposition property of a single Poisson process into multiple output streams ([7], Sec. 6.4), we can divide the initial job stream into three independent streams λ_1, λ_2, λ_3, where λ_i represents the aggregate arrival rate of all the jobs in class i. Further, we assume that users generate jobs with equal probability, i.e., the probability that a job belonging to class i is generated by a specific user is $\lambda_i/10$.

In the following experiments we compare three microeconomic scheduling policies and three variable-partitioning (*VP*) policies ([1], Sec. 3.2.3). In a *VP* policy the processors are not partitioned into predetermined subsets dedicated to each class of jobs, as is done in fixed-partitioning policies. Previous work ([5]) has shown that *VP* policies outperform the latter. We consider three *VP* policies:

- *FCFS*—This is the simplest among the policies considered. The jobs are placed in a first-come first-served (FCFS) queue; if there are enough free processors then the first job from the queue is scheduled for execution. If not, the job waits till the requested number of processors becomes free.

- *RES*—This is a modification of *FCFS*. If a sufficient number of processors are not available to run the next job from the queue, the scheduler reserves processors for this job for the earliest time in the future when the required number of processors become available. Further, to make use of idle processors until that time, the scheduler searches the queue and schedules the earliest jobs whose requests can be satisfied before these processors

Class type	*Num. of procs.*	*Exp. 1*		*Exp. 2*		*Exp. 3*		*Exp. 4*		*Exp. 5*		*Coef. of var.*
		Mean service	*Arv. rate*	*Mean service*	*Arv. rate*	*Mean service*	*Arv. rate*	*Mean service*	*Arv. rate*	*Mean service*	*Arv. rate*	
1	1-16	50	0.7λ	**100**	0.7λ	**200**	0.7λ	50	$\mathbf{0.33}\lambda$	50	$\mathbf{0.9}\lambda$	4
2	17-32	100	0.2λ	**100**	0.2λ	**100**	0.2λ	100	$\mathbf{0.33}\lambda$	100	$\mathbf{0.09}\lambda$	2.5
3	33-64	200	0.1λ	**100**	0.1λ	**50**	0.1λ	200	$\mathbf{0.33}\lambda$	200	$\mathbf{0.01}\lambda$	1.8

Table 1: The workload characteristics for five experiments. The last four experiments are derived from Experiment 1 by changing the mean service time and the relative arrival rate for each class (the parameters that are changed are in bold characters). The last column represents the coefficient of variation of the mean service time for each class.

need to be dedicated to the job with the reservation.

- *SCDF*—In the *Smallest-Cumulative-Demand-First* policy, jobs are selected for execution in increasing order of their cumulative computation times (the product of the execution time and the number of processors). This policy could cause starvation like its counterpart in the single processor case, shortest-job-first.

The economic policies we consider differ in how the user distributes the income rate to his/her waiting jobs:

- *ECON_PROP*—In this case, the user distributes his/her income rate among his/her waiting jobs in proportion to their cumulative computation times. We note that this policy reduces to the basic policy we have used in [6].
- *ECON_CONST*—This is the simplest microeconomic policy. The user divides *equally* his/her income rate among his/her waiting jobs, i.e., if user i has n_i jobs in the waiting queue and his/her income rate is r_i, then every job receives money at the rate r_i/n_i.
- *ECON_INV*—Here the user distributes his/her income rate in *inverse* proportion to the jobs' cumulative computation times.

In addition, we assume that in all the microeconomic policies every user has the same income rate equal to 100 dollars/minute.

To study how these policies perform under various loads we perform five experiments. The basic parameters used in each experiment are given in Table 1. In all the experiments the job service time is assumed to satisfy a biphase hyperexponential distribution [4]. The parameters in Experiment 1 are derived from the observed workload on an Intel iPSC/860 hypercube at NASA Ames, reported by Feitelson and Nitzberg [2]. [2] These parameters include the relative

[2]Since we consider a more general architecture than an iPSC/860 hypercube, we assume that the number

values for the mean service time and coefficient of variation for each class (see Table 1). The parameters for the other four experiments are derived from those of Experiment 1: in Experiments 2 and 3, we change the relative arrival rates associated with each class, while in Experiments 4 and 5, we change the mean computation times associated with each class. Note that in each experiment we change only one parameter: either the class arrival rates, or the mean computation times. This design of the experiments isolates the effects of each parameter on the system performance.

Let w_i be the time that job i waits in the ready list before being scheduled, and let T_i be the execution time for job i. (We use both service time and computation time interchangeably with execution time.) Further, let $s_i = w_i + T_i$ be the system response time of job i, and let u_i denote the user response time, where $u_i = s_i/T_i$. Intuitively, u_i measures for how long a user should wait for executing a job in terms of the job's service time. We note that according to this metric small jobs are much more sensitive to larger delays. Also, observe that u_i is always larger than 1. In analyzing the scheduling policies, following Naik, Setia and Squillante [5], we use two performance metrics: the *mean system response time* $\overline{S}$, and the *mean user response time* $\overline{U}$:

$$\overline{S} = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} s_i; \quad \overline{U} = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} u_i. \tag{1}$$

Note that $\overline{S}$ measures the performance from the system's point of view, while $\overline{U}$ measures the performance from the user's point of view [5]. We note that the mean system waiting time can be obtained by subtracting the mean service time from the mean system response time $\overline{S}$. Hence we will not include results for the mean waiting time in this paper. Instead, in Figure 4 we give the results for the *maximum waiting time*, i.e., the maximum waiting time incurred by *any* job during an entire experiment.

We define the *system load* as the ratio of the total demand received by the system in one time unit, and the available computation time per time unit. Similarly we define the *system utilization* as the ratio of the total computation time allocated during one time unit, and the available computation time per time unit. Note that the system utilization is never greater than the system load, since it is not possible to allocate more time than the service time requested by the incoming jobs. In each of the following experiments, we generate a system load between 0.1 and 0.9 in steps of 0.1, by suitably varying λ. To attain steady state we run each experiment (for every value of the system load) for 500,000 time-units. In all our plots the system utilization is represented on the x-axis. It is important to note that whenever a point's x-coordinate is not a multiple of 0.1, as in Figures 2, 3, and 4, the system utilization

of processors that a job requests is uniformly distributed. For example, a job that takes 64 processors on a hypercube is assumed to request any number of processors between 32 and 64, with equal probability. Also, we have not used the *absolute* values for service-times as given in [2]; instead we have chosen values that approximate the ratios between the service-times of different classes.

is *less* than the system load. For example, in Figure 2(d) the system utilization for *SCDF* is approximately 0.84 when the system load is 0.9. The system utilization is lower than the system load when the waiting jobs cannot use available processors either due to requirement constraints (i.e., there are not enough free processors to accommodate a job's request), or due to scheduling policy constraints (e.g., in *FCFS* jobs are strictly scheduled in order of their arrival).

3.2 Experimental Results

The *FCFS* policy performs the worst among these policies in every category. This policy tends to heavily penalize small jobs when the system load increases. Specifically, when the first job in the queue requests a large number of processors and its request cannot be satisfied, then every subsequent job has to wait, even if there are enough free processors to satisfy its needs. We note that this behavior was also mentioned in other previous studies [2, 3, 5].

The *RES* policy attempts to eliminate this problem by providing future reservations for a large job that cannot run immediately, and utilizing idle processors to run jobs that will complete before the reservation time. Notice that the *RES* policy is a special case of the *ECON* policy in which the income rate of every user is zero, assuming that the scheduler selects the job that arrives first among jobs that offer the same price. The EASY scheduler developed by Lifka uses this policy [3].

As shown in Figures 2 and 3, in comparison to *FCFS*, the *RES* policy dramatically improves both the system response time and the user response time. However, when compared with other policies *RES* performs consistently worse. This is because this policy fails to take into account significant job characteristics other than the job's arrival time. Both the number of processors requested by a job and its computation time heavily impact $\overline{S}$ and $\overline{U}$. On the other hand, the *RES* policy is consistently the best in terms of the maximum delay incurred by a job during the entire simulation. This is to be expected since, whenever possible, this policy tries to serve all the jobs in the order of their arrival. However, these results should be considered together with the mean waiting times, which can be obtained by a translation along the y-axis in the plots in Figure 2. Recall that the mean waiting times can be expressed as the difference between the system response time $\overline{S}$ and the mean service time, and that the mean service time is constant by construction for each experiment (see Table 1).

While in the *FCFS* and the *RES* policies the incoming jobs are scheduled according to their arrival times, in the *SCDF* policy they are scheduled in increasing order of their cumulative computation times. As noted in [1], the intuition behind this scheme is to approximate the shortest-job-first policy which is optimal in the one processor case. As expected, this policy performs well (see Figures 2 and 3) when the number of processors requested by a job is correlated with its mean computation time, since in this case a job that belongs to a lower numbered class is likely to request both fewer processors and a shorter computation time. This trend is

clear from Experiments 1, 4, and 5, where both the number of jobs and the mean computation time increase monotonically with class number. On the other hand, in Experiments 2 and 3, where the mean computation times and the number of processors requested by the jobs are not correlated, *SCDF* does not perform as well.

Figures 2 and 3 also show that except for the *FCFS* policy, *SCDF* performs the worst in terms of system utilization at high loads. For example, in Experiment 1 when the system load is 0.9, the system utilization for *SCDF* is 0.87, while for the *RES* and microeconomic policies it approaches 0.9 (see also the other experiments). In the *SCDF* policy, when the job with the smallest cumulative computation time cannot be scheduled because the number of processors it requests is not available, it and all other jobs have to wait until a sufficient number of processors become free. We tried a reservation policy similar to the one in *RES*, but this performed even worse.

Finally, Figure 4 shows that the *SCDF* policy has the largest waiting times. This result reflects the fact that the *SCDF* is the only policy considered here that is susceptible to starvation; hence large jobs may wait indefinitely in the ready queue.

Unlike the previous policies, the microeconomic policies take into account both the arrival time and the cumulative computation time of each job. The arrival time is implicitly taken into account since a job that arrives earlier will receive more money in its expense account, and consequently it can offer a better price per computation time than another job with the same cumulative computation time that arrived later. On the other hand, the cumulative computation time is used in computing the price offered per computation time-unit; in both *ECON_PROP* and *ECON_INV* policies, the cumulative computation time is also used to compute the rate at which money is transferred from the user's saving account into the job's expense account. We notice that when compared to the *ECON_PROP* policy, both *ECON_CONST* and *ECON_INV* favor small jobs over large jobs. As an illustration, consider an example in which a user has two ready jobs, the first requesting 500 minutes, and the second requesting 1500 minutes. Then in the *ECON_PROP* policy the transfer rates are 25 dollars/minute for the first job and 75 dollars/minute for the second job; for the *ECON_CONST* policy both transfer rates are equal to 50 dollars/minute; and for the *ECON_INV* policy these rates are inversely proportional to cumulative computation times, i.e., 75 dollars/minute for the first job and 25 dollars/minute for the second one.

As a general trend we note that in all five experiments, when the response times $\overline{S}$ and $\overline{U}$ are considered, the *ECON_INV* policy performs better than the *ECON_CONST* policy, which, in turn, performs better than the *ECON_PROP* policy. Hence if we want to improve $\overline{S}$ and $\overline{U}$ then small jobs should be favored over large jobs. We note that in the limit this is similar to the *SCDF* policy, in which smaller jobs are *always* scheduled before larger ones. This is why *ECON_INV* is the closest economic policy to *SCDF*.

Another observation from Experiments 2 and 3 is that these policies are more sensitive to

the variations in the transfer rate when the number of large jobs increases. In this situation, for the *ECON_CONST* and *ECON_PROP* policies, when many large jobs reserve processors for future times, it becomes harder to find small jobs to fit into the residual idle processor slots.

When the maximum waiting time is considered, *ECON_PROP* performs the best, while *ECON_INV* performs the worst. This is expected because when small jobs are favored over large jobs, the waiting time of larger jobs increase. Finally, we note that among the microeconomic policies the *ECON_PROP* policy attains the highest system utilization while *ECON_INV* attains the lowest system utilization.

4 Conclusions

In this paper we use simulation to compare three microeconomic policies with three variable partitioning policies: first-come first-served with and without reservation (*FCFS* and *RES*), and smallest cumulative demand first (*SCDF*). The microeconomic policies differ in the way in which the user distributes the income rate among waiting jobs. Specifically we consider the cases when the income rate is distributed directly in proportion to (*ECON_PROP*), inversely in proportion to (*ECON_INV*), and independently of (*ECON_CONST*) the jobs' cumulative computation times. In order to isolate the effects of various parameters, we designed experiments that vary one parameter at a time.

When user and system response times are considered, the only policy that outperforms the microeconomic policies for some experiments is the *SCDF* policy. However, we note that this policy is the only one in this group susceptible to starvation, and it is also consistently the worst in terms of system utilization and maximum waiting time. When the maximum waiting times are considered, the *RES* policy performs the best in all cases. However, excepting *FCFS*, this policy has the worst system and user response times. The microeconomic policies perform the best overall when *all* of these mutually antagonistic criteria are considered. This should not be surprising, since both *RES* and *SCDF* policies are extreme limiting cases of the microeconomic approach.

By being able to gradually change the "degree" to which small jobs are favored over large jobs (by considering different transfer rates in the microeconomic policies) we learn several important lessons. First, by giving a higher priority to smaller jobs over larger jobs one may hope to increase both the system and user response times at the expense of system utilization and the maximum waiting time. Second, when the fraction of large jobs in the system increases, both the user and the system response times become more sensitive to the variations in the income transfer rates.

We conclude that microeconomic scheduling policies exhibit *robust performance* across a broad range of parameters. Changing the income-distribution rule among the jobs of a user

is an effective way to trade between user and system response times on one hand, and system utilization and maximum waiting time on the other hand. This makes it easier to achieve *flexibility* in the microeconomic scheduler relative to the other schedulers.

References

[1] D. G. Feitelson, "A Survey of Scheduling in Multiprogrammed Parallel Systems", *Research Report RC 19790*, IBM T.J. Watson Research Center, 1994.

[2] D. G. Feitelson and B. Nitzberg, "Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860", D. G. Feitelson and L. Rudolph (eds.), *Lecture Notes in Computer Science*, Vol. 949, Springer-Verlag, 1995.

[3] D. A. Lifka, "The ANL/IBM SP Scheduling System", *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing, IPPS'95*, Santa-Barbara, April. 1995, pp. 187-191.

[4] S. Majumdar, D. L. Eager, and R. B. Bunt, "Scheduling in Multiprogrammed Parallel Systems", *Proceedings of the 1988 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pp. 104–113.

[5] V. K. Naik, S. K. Setia and M. S. Squillante, "Performance Analysis of Job Scheduling in Parallel Supercomputing Environments", *Research Report RC 19138*, IBM T.J. Watson Research Center, 1993.

[6] I. Stoica, H. Abdel-Wahab and A. Pothen, "A Microeconomic Scheduler for Parallel Computers", in *Load Balancing and Job Scheduling for Parallel Computers*, D. G. Feitelson and L. Rudolph (eds.), Lecture Notes in Computer Science, Vol. 949, Springer Verlag, pp. 200–218, 1995.
Available on the Web at `URL: http://www.cs.odu.edu/~pothen/papers.html`

[7] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, Prentice-Hall, 1982.

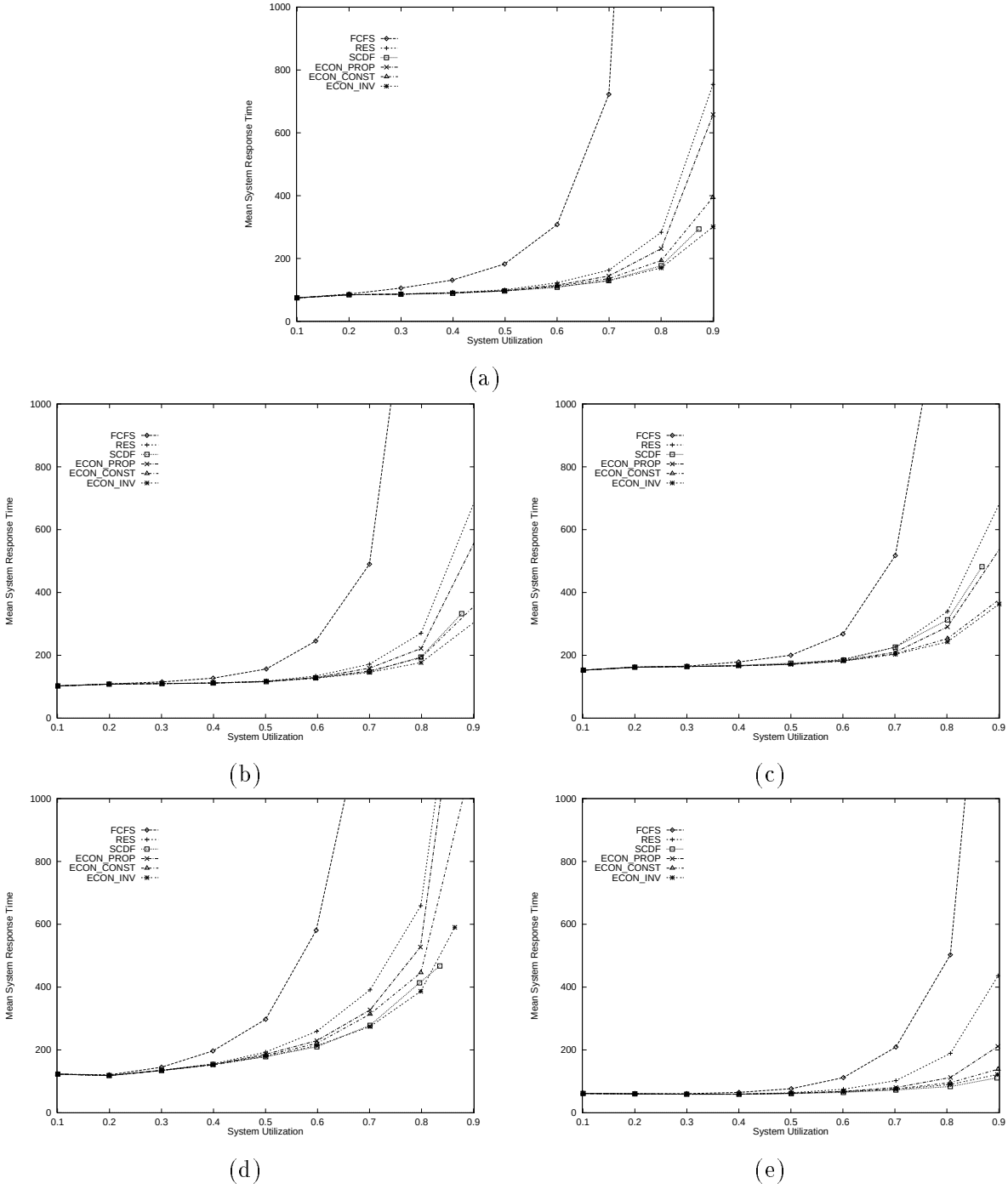

Figure 2: The mean system response time $\overline{S}$ for: (a) experiment 1, (b) experiment 2, (c) experiment 3, (d) experiment 4, and (e) experiment 5.

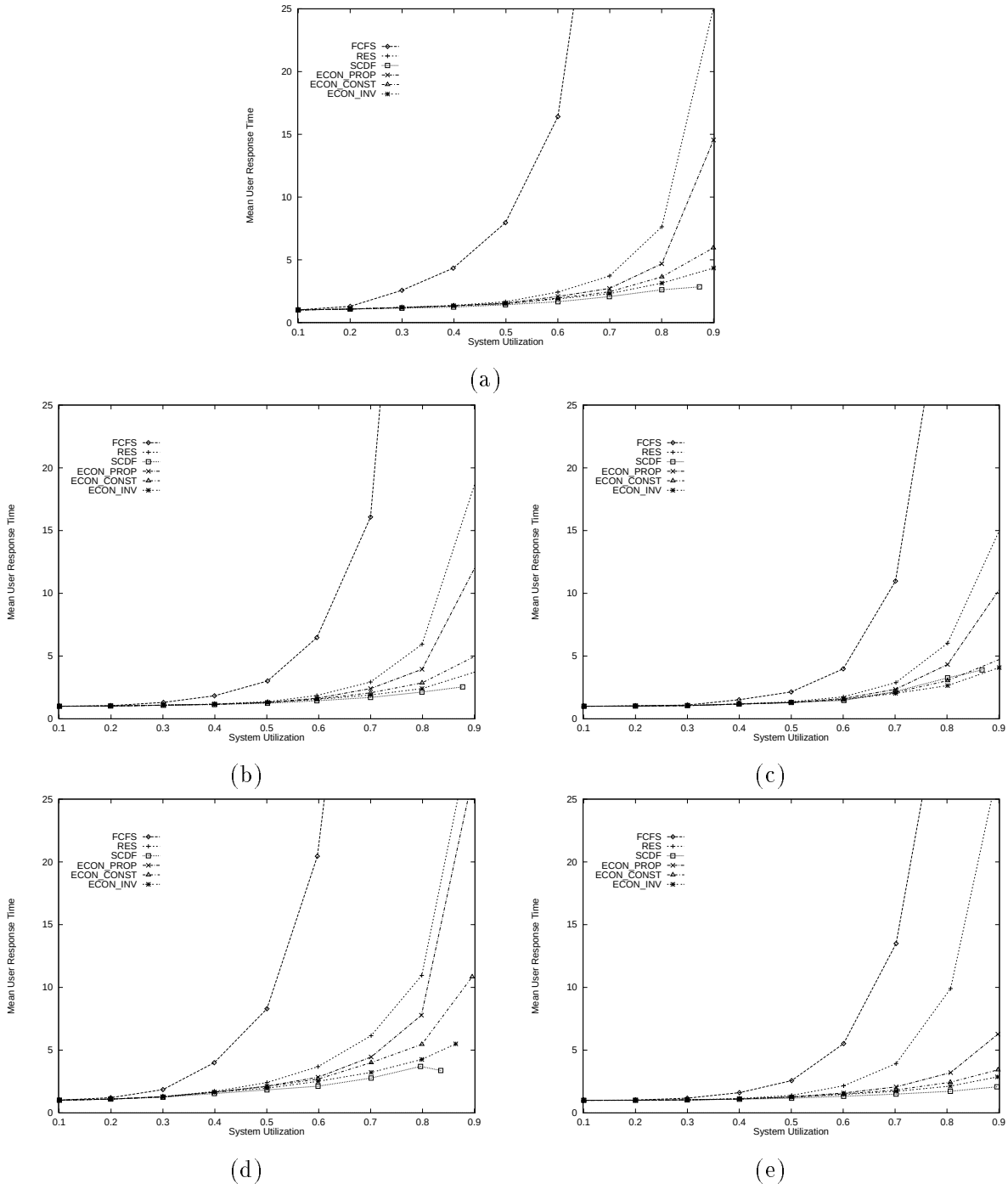

Figure 3: The mean user response time $\overline{U}$ for: (a) experiment 1, (b) experiment 2, (c) experiment 3, (d) experiment 4, and (e) experiment 5.

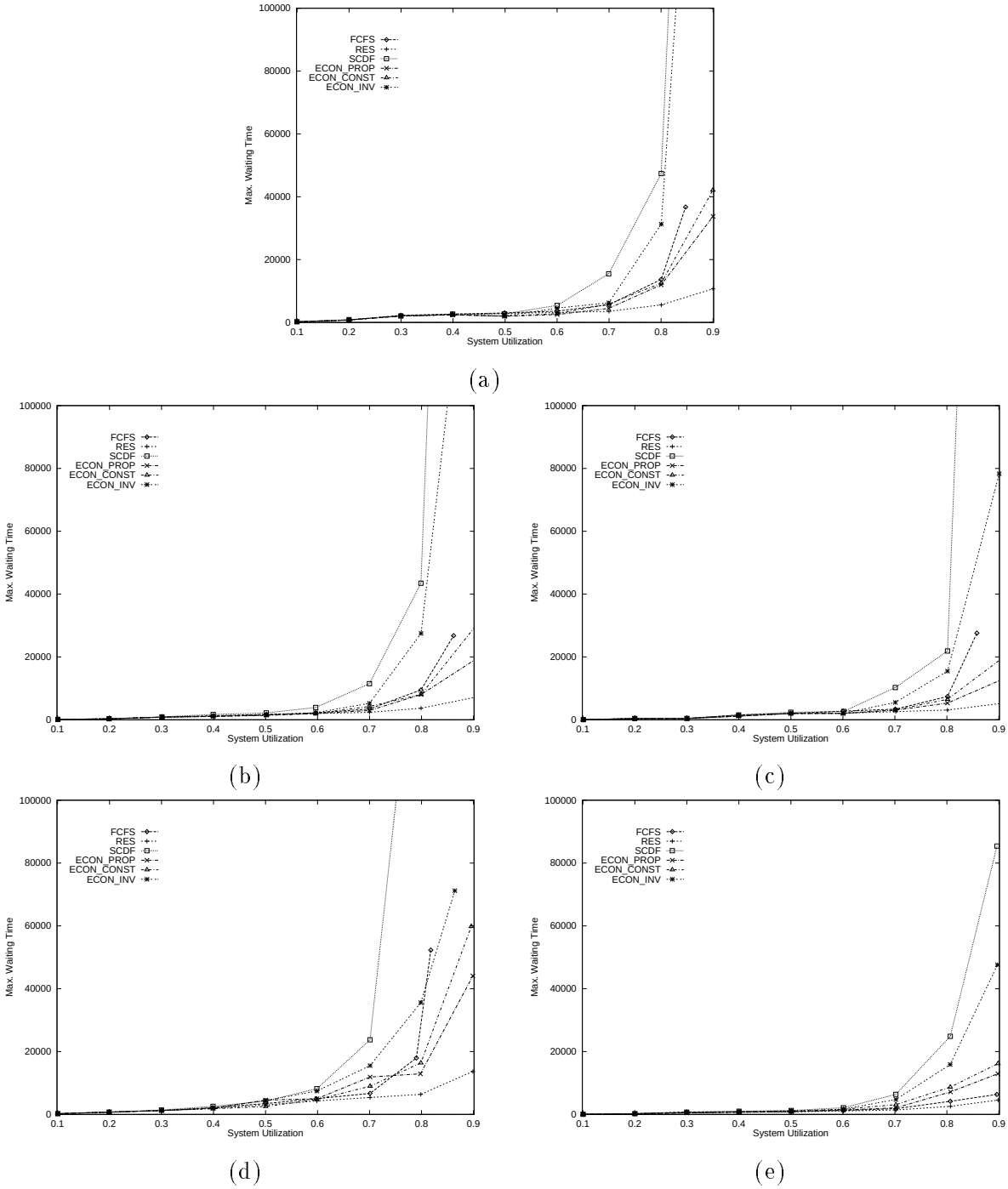

Figure 4: The maximum waiting time for: (a) experiment 1, (b) experiment 2, (c) experiment 3, (d) experiment 4, and (e) experiment 5.