## Applied Research Results

Tuning operating systems helps system administrators improve system performance and efficiency, but it can be time-consuming. These authors propose a mechanism, based on genetic algorithms, to automate this process by running simulations of system performance for various parameter values during the system's idle loop.

# Self-Tuning Systems

**Dror G. Feitelson and Michael Naaman,** The Hebrew University of Jerusalem

Modern operating systems are highly parameterized, meaning that the algorithms and policies used are not completely defined in code. This allows system administrators to modify these parameters to tune system performance. Savvy selection of parameter values can make the difference between fast and slow work flow, a smooth-running system and a balky one, and happy customers and frustrated ones. Perhaps the best-known examples come from file systems, in which various features such as block size and the way in which disk blocks are allocated can be modified, albeit within the general framework dictated by the system design.[1]

However, system tuning is difficult and time-consuming. Typically it is an ad hoc process, possibly with some vague guidelines but nearly always with no direct way to measure the effect of changes to the parameter values. Only by trial and error can system administrators find parameter values that will optimize system performance for their local workload. We suggest a mechanism to automate this process.


0740-7459/99/$10.00 © 1999 IEEE

Our approach is based on the observation that systems typically make detailed records of various aspects of the workload. For example, Unix systems maintain a log of all user sessions and of all processes executed, including detailed resource usage information. Web servers can be configured to maintain a log of all pages served. These log files represent knowledge about the local workload.

We propose using this knowledge to drive simulations of system behavior with different parameter values, and measuring the resulting performance. Genetic algorithms can be used to create new parameter combinations and to conduct a systematic search for optimal parameter values. The simulations are run in place of the system's idle loop to minimize overhead.

## Designing in Optimization

Designing operating system algorithms and policies involves many micro-decisions: the size of a table, the threshold for activating a certain procedure, the order in which a data structure is scanned, and so on. These decisions may have unknown effects on performance. However, exhaustive research of all the alternatives is impossible, both because it is too much work and because the results necessarily depend on the local workload at each installation. Not only is such data not available when the system is being designed, it also differs among installations.

An elegant way out is to parameterize the algorithm or policy in question rather than hard-coding a specific choice. All the parameters then have default values selected by the designers, but each installation's system administrators can change them. For example, the administrator may set table size as part of the system configuration. The threshold value may be set by a special system call, executed by the administrator's user interface.

While this approach shifts the burden from the system designers to its administrators, it raises new problems. True, administrators have more knowledge of the local conditions and workload, which they can use to fine-tune the system. But they may lack detailed knowledge about the system's inner workings and thus not appreciate the finer implications of setting or resetting various parameter values. Also, system administrators are notoriously overworked and busy solving various crises, leaving little if any time for elective chores such as tuning.

### Using genetic algorithms

The alternative we propose, self-tuning systems, lets system designers create the framework that will carry out the optimizations and tuning. The execution of this framework is delayed until the system is deployed in the field and its specific workload can be measured and characterized.

**System administrators are notoriously overworked and have little time for elective chores such as tuning.**

Technically, the framework simply consists of a systematic search of the parameter space. Given that the search space is very large (many parameters can have many different values) and unknown (is there one global optimum? Are there many similar maxima? Do all parameters have the same impact on performance?), an efficient search procedure is required. We chose to use genetic algorithms (see the boxed text, "What Are Genetic Algorithms?" on page 54) as our optimization procedure.

It is easy to represent a set of parameter values as a chromosome: simply concatenate the binary representation of the parameter values. Crossing over then takes one set of values from one parent and the rest from the other. We can also allow the binary representation of a single parameter to be broken in the middle, thus creating two new values. Likewise, mutations can create new values by flipping a single bit.

It is harder to evaluate the fitness of these "chromosomes." First, we must define an appropriate objective function that reflects the performance metric we wish to optimize, such as utilization or response time. Indeed, we could construct a system that can optimize a range of metrics, leaving the choice of metric as the only parameter that the local system administrator has to set.

We evaluate the chosen function for a certain set of parameters by simulating the system's behavior based on a record of the local workload. This implements a sampling of the mapping $P \times W \rightarrow Q$, where $P$ is the set of possible parameter value combinations, $W$ is the set of possible local workloads, and $Q$ is the set of possible outcomes in terms of the objective function. Thus we can rank the different

## What Are Genetic Algorithms?

Genetic algorithms, as the name suggests, are based on a biological analogy. It has become common to view the evolution of sexual reproduction—and the genetic mixing that comes with it—as a mechanism for sampling and searching the vast space of possible genomic configurations.[1] In computer science, *genetic algorithms* refers to an optimization procedure that mimics this biological process. The name is actually a bit of a misnomer, as it refers more to a framework than to a specific algorithm.

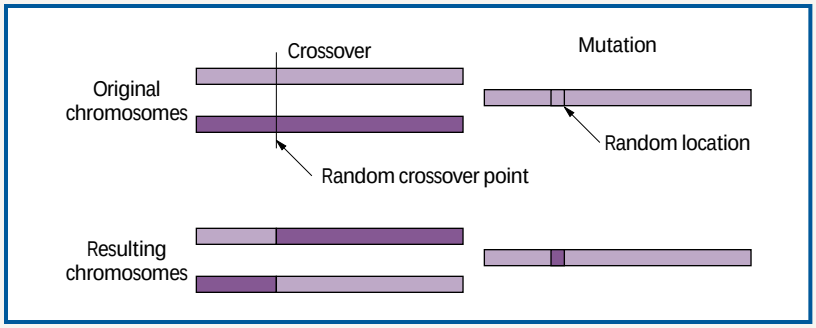



Figure A. The genetic algorithm operations on chromosomes.

### Searching for the Right Mix

In essence, genetic algorithms involve iterative searching in a large configuration space. In each iteration, we evaluate several potential configurations from different parts of the space. We then combine the best ones with each other in random ways, and use them as the starting points for the next iteration. The search terminates when additional iterations do not produce additional improvements, or after a predefined number of iterations.

The biological analogy stems from the theory of evolution and the principle of survival of the fittest. Each iteration is called a *generation*. Each configuration is called an *individual*, and together all the configurations being considered in a certain generation are the *population*. The configurations are represented by *chromosomes*—essentially a list of the parameter values that define the configuration in question.

The evaluation of the different configurations is determined by the goals of the optimization. The result of the evaluation is translated into a single value that reflects the quality of each configuration. In genetic-algorithm terms, this quality index is called the *fitness* of the individual.

The key to the operation of genetic algorithms lies in how the population changes from one generation to the next. First, the members of the population are ranked according to their fitness. Next, they are "mated" with each other to produce offspring (hence the name "genetic algorithms"). The probability of mating is proportional to fitness; individuals with a higher fitness mate more often and produce more offspring, thus passing their high-quality genes (configuration parameters) to the next generation.

### Mating

Mating creates new chromosomes from those of the parents. When two individuals mate, the crossover operator may be applied. This operator chooses a random point in the chromosome and creates two new chromosomes based on the parents: one gets the first part from one parent and the second from the other parent, and the other gets the opposite (as shown in Figure A). This mixes parameter combinations that lead to higher performance in various ways, potentially leading to new combinations with even better performance. If crossover is not applied, the two parent chromosomes are simply inserted into the next generation without change; being selected according to their fitness, the fittest individuals persevere. In addition to crossover, mutations may be inserted into chromosomes to create parameter values that were not present in the original population. This is done by changing a single bit at random (shown in Figure A).

Many variants on this basic scheme are possible, dependent on several decisions:

- How should we represent configurations in chromosomes? Specifically, what alphabet of symbols do we use?
- When a new generation is created, should we take only the offspring, or the best individuals out of the joint pool of parents and offspring?
- How large should each generation be?
- How do we normalize the fitness values and turn them into mating probabilities?
- What are the probabilities of crossover and mutations?

Significant research has been conducted on these and other issues, and on how they affect the convergence properties of the optimization procedure.[2]

### References

1. J.H. Holland, *Adaptation in Natural and Artificial Systems*, MIT Press, Cambridge, Mass., 1992.
2. D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison Wesley Longman, Reading, Mass., 1989.

combinations of parameter values as they relate to the local workload, and quantify them in terms of the chosen performance metric. This quantification is the *fitness value*.

Doing the simulation correctly is perhaps the most challenging aspect of the procedure. Operating systems are complex and may require a detailed simulation, involving high overhead and extensive logs. Yet this need not always be the case: some aspects of the operating system can be evaluated in isolation with little information, such as the batch-scheduling algorithm used in our case study. But harder cases do exist, such as optimizing the scheduling parameters that govern the priority boost given to processes that complete an I/O operation. Simulating this requires detailed information about individual CPU bursts and I/O operations, not normally maintained. A possible solution is to use sampling and collect the required information for only a short period.

**Our approach permits using new and improved parameter values as they are found.**

Given the definition of chromosomes and the procedure to evaluate fitness, the genetic-algorithm machinery can be put into motion. Starting with the default system parameter values and some other randomly chosen parameter value sets, iteratively evaluating these sets using simulation and then combining the best-performing sets will lead to the generation of new and better combinations.

A nice feature of this design is that new and improved parameter values can be used immediately as they are found, without waiting for a separate optimization procedure to complete. Moreover, by continuously using this procedure with the latest system logs, the parameter values will track changes in the workload as they occur. All this can be achieved at essentially no cost, by running the optimization procedure in the background in place of the idle loop. The system devotes cycles to optimization only if no user applications can use them. In particular, idle time at night can be used to optimize a system that is heavily utilized by day.

## A Case Study

The iPSC/860 is a late-1980s Intel parallel supercomputer whose hypercube has a well-defined, nontrivial, highly parameterized batch-scheduling algorithm.[2] In addition, a trace of a production workload on such a system is available.[3] This therefore makes a good case study, even if batch scheduling and the iPSC/860 are not of much interest in themselves.

The workload trace used in this study covers the fourth quarter of 1993 and comes from an iPSC/860 installed at NASA Ames.[3] At the time, this machine was the workhorse for computations at the Numerical Aerospace Simulation facility. The log includes a total of 1,044 batch jobs.

The iPSC/860 architecture is based on nodes containing an Intel i860 RISC processor and some local memory, connected to each other in a hypercube topology. The topology implies that the number of nodes in the system has to be a power of two. Our workload data comes from a 128-node machine, which is a hypercube of dimension 7. Multiprogramming is possible by running jobs on subcubes, that is, embedded hypercubes of a lower degree. The operating system imposes a limit of 9 on the degree of multiprogramming.

### The batch scheduling algorithm and its parameters

The iPSC scheduling algorithm works on two types of jobs: interactive and batch. Interactive jobs require immediate running, while batch jobs are submitted to some queue and await their turn. The system divides the day into two: the prime shift during the day and the nonprime shift at night. During prime time, some nodes are allocated to the batch partition; the rest are reserved for interactive work. During nonprime time, all nodes are in the batch partition. Batch jobs may only run on nodes from the batch partition, while interactive jobs can run on any available nodes. Jobs run to completion (or until a time limit is exceeded); preemption is not used.

We chose to focus only on the scheduling of batch jobs, as this would suffice to demonstrate how self-tuning works. Thus we only handle the optimization of those parameter values unique to batch scheduling. The following description is based on the Intel MACS (Multiuser Accounting, Control, and Scheduling) manual.[2]

The batch-scheduling algorithm is based on two main concepts:

1. prioritizing the jobs to decide which will be scheduled next, and

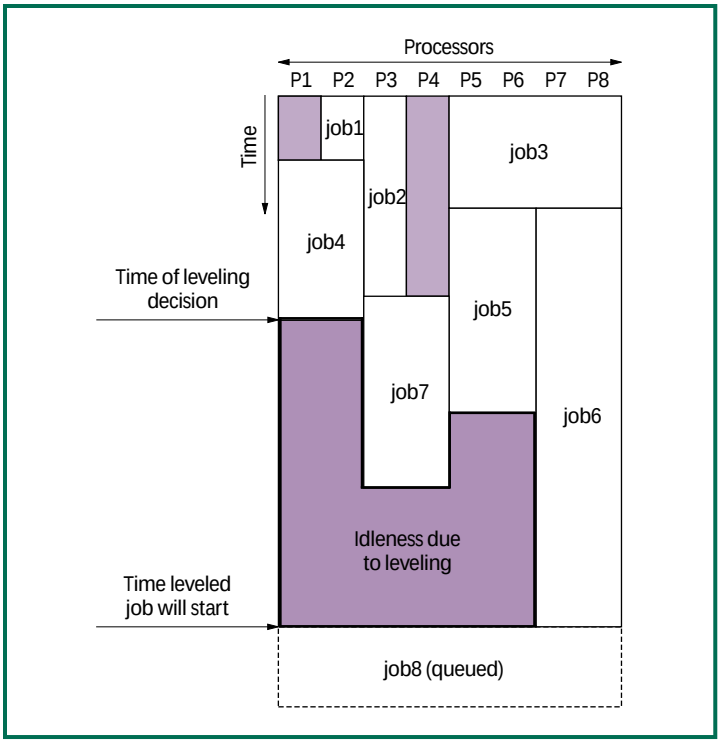



Figure 1. Leveling is done if the induced idleness is smaller than A_HOLE_SIZE + B_HOLE_SIZE × req_nodes; otherwise it is considered too expensive. Gray shading represents idle nodes.

2. using reservations to accumulate processors for large jobs (leveling).

The scheduler has a set of queues, to which jobs are submitted. Each queue is characterized by several attributes, such as limits on the number of requested nodes and on requested runtime; or whether the queue is active only during nonprime time or also during prime time. The most important attribute for our work is the queue's priority, which directly affects the priority of jobs submitted to it. In addition, a global system parameter called *A_TIME_PRI* determines the weight of waiting time in the queue. The formula for computing a job's priority is

$$pri = q\_pri + wait\_time \times A\_TIME\_PRI$$

where *q_pri* is the queue's basic priority, and *wait_time* is the time the job has been waiting on the queue. This formula is used to sort the jobs and decide which job to run next.

If the next job to run requires more nodes than are free, the scheduler considers leveling it. Leveling means not scheduling any more jobs until enough nodes exist to run the waiting job. Obviously such leveling requires that the scheduler leave nodes idle, so processing resources are lost. It is therefore necessary to carefully weigh when to engage in leveling. The algorithm tries to estimate resource loss, and compares this amount to a tolerance determined by these parameters:

- *A_HOLE_SIZE*: node-hour idleness tolerated forthwith
- *B_HOLE_SIZE*: additional node-hour idleness tolerated per requested node (allowing more tolerance for large jobs)

The maximum idleness to be tolerated is then calculated by

$$A\_HOLE\_SIZE + B\_HOLE\_SIZE \times req\_nodes$$

where *req_nodes* is the number of nodes required by the job.

Figure 1 shows an example. When job 4 terminates, the scheduler has two nodes available, and the queued job with the highest priority (job 8) requires all eight nodes. Using the runtime bounds on the currently running jobs (jobs 5, 6, and 7), the scheduler can estimate when all the nodes will be freed and job 8 will be able to run. However, such leveling will cause nodes 1 through 6 to remain idle for various durations. The scheduler sums up these idle node-hours (represented by the dark gray area with the heavy border in the figure) and compares it with the value of $A\_HOLE\_SIZE + B\_HOLE\_SIZE \times 8$. If the wasted area is not bigger than this value, the job will be leveled. If it is bigger, the job will remain in the queue and other smaller jobs will be scheduled.

To reduce resource loss, the scheduler attempts to schedule small jobs on the idle nodes, provided their time limit indicates they will end before the time of leveling. This is done only after the decision to level.

## Formulation for genetic algorithms

The iPSC batch-scheduling algorithm on the NASA system has 15 parameters. Which parameter values will lead to the best performance? To use the self-tuning framework to find optimal parameter values, we encoded the algorithm for optimization using genetic algorithms.

### *Representation in chromosomes*

The different parameters to be optimized have different value ranges:

- *A_HOLE_SIZE*: 0–255

- *B_HOLE_SIZE*: 0–5
- *A_TIME_PRI*: 0–5
- 12 queue priorities: 0–255

We represented these parameters as a string of bits, each parameter occupying 8 bits, for a total chromosome length of 120 bits. Thus the resolution of values in the queue priorities and *A_HOLE_SIZE* was 1, and in *B_HOLE_SIZE* and *A_TIME_PRI* it was approximately 0.02. When running the simulation, the string was first transformed to a `struct` holding the parameters. This `struct` was then passed to the simulation function, which used it to run the simulation and compute the fitness.

### Fitness function

We used the average machine utilization as a fitness parameter. For each day's simulation, we calculated the utilization as the ratio between the resources (measured in node-seconds) that the jobs actually used:

$$\sum_i run\_time_i \times nodes_i$$

and the total resources available for the duration of all jobs:

$$total\_time \times batch\_partition\_size$$

where *total_time* is the clock time from the start of the first job to the completion of the last job.

This ratio gives a nonnormalized fitness function: the maximum utilization is 1, but the sum of all fitnesses is bigger than 1. We used utilization as a fitness function for simplicity and because it matches the goals of a batch scheduler. The results may differ when using another definition of fitness.

### Evaluating the fitness

For each set of parameter values (represented by a chromosome in the current population) we simulated the scheduler's behavior to evaluate its performance. The simulation assumed the following:

- All 128 nodes of the machine were in the batch partition, and we only optimized the batch-scheduling algorithm.
- All jobs were submitted before scheduling begins, that is, during the day, when they were not scheduled because most batch queues were disabled. When the prime-time shift ended, all nodes were allocated to the batch partition and all queues were enabled, allowing the accumulated jobs to be scheduled.
- All queues could be scheduled in the non-prime-time shift.

**Table 1**
**Parameters Used in Genetic Algorithm Implementation**

| | |
|---|---|
| Population size | 120 |
| Chromosome length | 120 bits |
| Probability of mutation | 0.1 |
| Probability of crossover | 0.5 |
| Run length | 150 generations |

As noted above, the workload used to drive the simulation was based on a detailed log of everything that ran on the iPSC/860 at NASA Ames during the fourth quarter of 1993.[3] However, we did not use the recorded workload directly because the batch load on that system was generally too low to exercise the scheduling algorithm. We instead unified several consecutive days of real workload into a single day of simulated workload, artificially increasing the load during each simulated day. The job characteristics (number of processors and runtime) remained the same as in the original workload. The criteria for unifying days was the desire to achieve either a load of 20 to 25 jobs or 1,000 to 1,300 node hours (corresponding to 8 to 10 hours of using 128 nodes). This compression reduced the duration of the log from three months to 70 days.

The simulation handled each "day" individually and then reported the average utilization for all the days. It was an event-driven simulation of the scheduler, where the events were the terminations of running jobs (we assumed no additional jobs arrived during the non-prime shift). The simulation then scheduled the next jobs to be run, according to the algorithm described earlier. If the next job could not be scheduled, the simulation tried to level it. If the job could be leveled, smaller jobs were scheduled as possible to reduce the idleness. Otherwise, this job's scheduling was deferred and the next job was considered. This occurred until there were no free nodes or jobs to schedule. Simulation time was then advanced to the finish time of the next finishing job. Its resources were freed, and scheduling ran again.

### Genetic algorithm dynamics

The implementation of the genetic algorithms framework was done with the sga-c package, an implementation of Goldberg's Simple Genetic

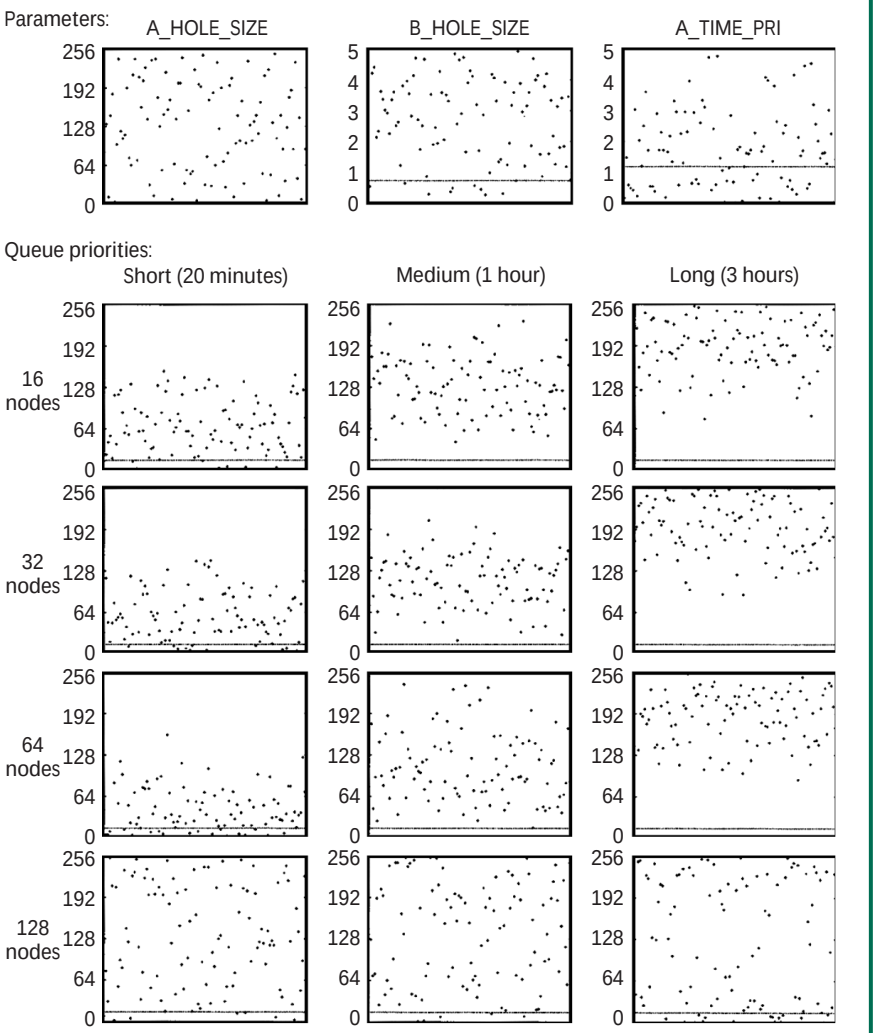



Figure 2. Parameter values that produced the best results in different runs (run number is from 1 to 100 along the x axis). The dashed lines are the default values; for A_HOLE_SIZE it is zero.

Algorithms.[4] This implementation gives a very basic set of tools to implement genetic algorithms, which sufficed for our needs.

Table 1 summarizes the parameters used in our experiments. The population consisted of 120 individuals. The first generation started with randomly generated chromosomes; we also checked starting with the Intel defaults as one of the chromosomes, but this did not affect the results. Experiments continued for a total of 150 generations. At each generation, the probability of crossover during mating was 50 percent, and the probability of mutation was 10 percent.

## Experimental results

This was a retrospective experiment, using old logs, rather than a run performed within a live system. However, it is not simply an experimental illustration of the self-tuning idea but rather the exact kind of simulation that would be used in a real implementation.

Starting with random sets of parameters, we tracked the best set of parameters found in a run, that is, over all 150 generations. We performed 100 such runs to see if the genetic algorithm would converge to a single set of optimal parameter values. We tried two methods for selection: one using the "raw" fitness values and one using normalized fitness (which amplifies the differences between individual fitness values). The results were similar; here we show those for raw values.

Using a Pentium Pro 200 running BSDI Unix from Berkeley Software Design Inc., the time to complete one generation was about one second. This includes the simulation of scheduling about 20 jobs in each of 70 days under each of 120 different sets of parameters. A run involving 150 generations therefore took two to three minutes, and executing all 100 runs with different initial populations took several hours. While this is a significant amount of time, we only needed it to evaluate the approach. A real implementation only needs to perform a single run, which in our case takes a few minutes, to find a set of good parameter values. The overhead for such a procedure is negligible.

Moreover, it is not always necessary to perform a complete run. In some of our runs, the optimal parameter values emerged as early as the fourth generation. A real implementation can also use good parameter values as soon as they are found, and continue the search for even better values only as time permits.

## Related Work and Further Resources

Our tuning algorithm is related to the concept of systems that learn about their environment. However, to the best of our knowledge, this is the first general methodology for creating self-tuning systems. Previous work has only dealt with self-tuning built into a specific algorithm.

The concept of tuning system behavior to the workload has been popular in the field of parallel job scheduling. Scheduling with adaptive or dynamic partitioning is based on systems that change the allocation they make as a function of load conditions. Some have proposed adaptive policies that decide on partition sizes based on the load and application characteristics.[1] Others have proposed a dynamic policy that changes the allocation at runtime to reflect changes in the load and requirements.[2] A scheme proposed by Charles Severance et al. is less dependent on explicit information; the system measures the performance of a barrier synchronization to decide if the current number of threads is appropriate.[3] The scheme closest to ours was proposed by Thu Nguyen et al., who measured the efficiency of a parallel job on several partition sizes and then decided on the allocation.[4] However, these schemes all involve learning about a specific application at runtime and are irrelevant for other jobs. They do not learn about the workload in general, and therefore cannot make a persistent change in the system parameters.

Work exists that directly relates to our case study in that it uses genetic algorithms to solve scheduling problems. However, this is typically done in the context of offline algorithms that search for a specific near-optimal schedule,[5] whereas our work is about finding good parameters for an online policy. Some have also suggested that the genetic algorithms themselves be parallelized.[6]

The Genetic Algorithm Archives at www.aic.nrl.navy.mil/galist/ contain a calendar of events, an archive of source code, and links to research sites. The CMU Artificial Intelligence Repository page on genetic algorithms also contains an archive and code, as well as documentation and papers. It is at www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/genetic/ga/0.html.

### References


1. K.C. Sevcik, "Characterization of Parallelism in Applications and Their Use in Scheduling," *Proc. SIGMETRICS Conf. Measurement and Modeling of Computing Systems*, ACM Press, New York, 1989, pp. 171-180.
2. C. McCann, R. Vaswani, and J. Zahorjan, "A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors," *ACM Trans. Computing Systems*, Vol. 11, No. 2, May 1993, pp. 146-178.
3. C. Severance, R. Enbody, and P. Petersen, "Managing the Overall Balance of Operating System Threads on a Multiprocessor Using Automatic Self-Allocating Threads," *J. Parallel and Distributed Computing*, Vol. 37, No. 1, Aug. 1996, pp. 106-112.
4. T.D. Nguyen, R. Vaswani, and J. Zahorjan, "Maximizing Speedup through Self-Tuning of Processor Allocation," *Proc. 10th Int'l Parallel Processing Symp.*, IEEE Computer Society Press, Los Alamitos, Calif., 1996, pp. 463-468.
5. E.S.H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Trans. Parallel and Distributed Computing*, Vol. 5, No. 2, Feb. 1994, pp. 113-120.
6. M. Schwehm and T. Walter, "Mapping and Scheduling by Genetic Algorithms," *Parallel Processing: Proc. CONPAR 94/VAPP VI*, Springer-Verlag, Heidelberg, Germany, 1994, pp. 832-841.


In general, our results showed a significant improvement in utilization. Running the simulation with the default Intel parameters resulted in 88.4 percent utilization for our workload. In all our runs, however, we achieved utilization of between 91.03 percent and 91.25 percent.

This is a 3.2 percent increase in utilization in absolute terms, and a 24.6 percent reduction of wasted processing capacity because of fragmentation (that is, resources not usable because of packing difficulties). These results also testify to the efficiency of genetic algorithms as a search procedure: each run included specific checks of only $120 \times 150 = 18{,}000$ parameter combinations out of the $2^{120}$ possible combinations, yet all achieved essentially the same results.

Though we found an improvement in utilization, the parameters showed no straightforward behavior. The surface of the fitness function seemed relatively flat, with many local peaks and valleys but no one outstanding peak. Thus the different runs produced widely different sets of parameter values that all led to about the same utilization. This implies that no parameter combinations exist that can achieve better performance than those we found, at least for this workload.

Figure 2 shows the combinations of parameters we found. For each optimized parameter, a graph shows the value of this parameter in the best set from each of the 100 runs. Clearly the values of all parameters do not converge into some specific value but are scattered. However, for job sizes of 16, 32, and 64 nodes, the priority of the short queue should be relatively low, whereas the priority of the

long queue should be relatively high. This indicates that the system "invented" the first-fit decreasing bin-packing algorithm: it is more efficient to pack the long jobs first, then pack the short ones in the remaining space.[5]

The Intel manual suggests that queues for long jobs be given a priority of 40,[2] but our results indicate this value is much too low. The priority of the 128-long queue had a bimodal distribution: in some cases it was low, in others high. This may mean that the 128-node jobs should be scheduled either first or last, but not mixed with other sizes, so as to reduce the waste of multiple leveling actions.

The parameters that control leveling decisions (*A_HOLE_SIZE* and *B_HOLE_SIZE*) are also much higher than the defaults, leading to a tendency to allow more leveling and larger idle times than the default parameters. On average, our results show that the decision was to level in about 84 percent of the cases where it was considered. The values for *A_TIME_PRI* are generally low, as the queueing time of batch jobs is not an important consideration when optimizing for utilization. Had response time been included in the fitness function, this parameter would likely have been more important.

While we are confident that our method for tuning operating systems has merit, much remains to be done. Our main goal is to implement self-tuning in a real system setting and test its performance. This will also enable us to consider self-tuning at the price of additional overhead. We may then be able to determine whether the potential improvement in performance is worth running the optimization procedure at the expense of user applications rather than just the idle loop, and also whether the overhead for additional logging of information (beyond that normally collected by the system) is worthwhile. ❖

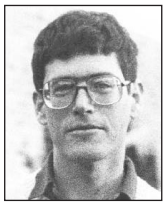
## Acknowledgments

We thank Bill Nitzberg for providing the NASA Ames iPSC workload log, and Reagan Moore of SDSC for introducing us to the iPSC batch-scheduling algorithm. Thanks are also due to the reviewers (especially #4) for their help in improving this article.

## References

1. M. McKusick et al., "A Fast File System for Unix," *ACM Trans. Computing Syst.*, Vol. 2, No. 3, Aug. 1984, pp. 181-197.
2. *iPSC/860 Multi-User Accounting, Control, and Scheduling Utilities Manual*, Intel Corp., Beaverton, Ore., May 1992.
3. D.G. Feitelson and B. Nitzberg, "Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860," *Job Scheduling Strategies for Parallel Processing*, D.G. Feitelson and L. Rudolph, eds., Springer Verlag, Heidelberg, Germany, 1995, pp. 337-360.
4. D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison Wesley Longman, Reading, Mass., 1989.
5. E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson, "Approximation Algorithms for Bin-Packing—An Updated Survey," *Algorithm Design for Computer Systems Design*, G. Ausiello, M. Lucertini, and P. Serafini, eds., Springer Verlag, Heidelberg, Germany, 1984.

## About the Authors



**Dror Feitelson** is on the faculty of the Institute of Computer Science at the Hebrew University of Jerusalem, Israel, where he conducts research on parallel operating systems. His current research focuses on the analysis and characterization of workloads on parallel machines that are in production use, with the goal of creating workload models that can be used to evaluate parallel job scheduling policies.

Feitelson received a BSc in mathematics, physics, and computer science, an MSc in computer science, and a PhD, all from the Hebrew University of Jerusalem. He then spent three years at the IBM T. J. Watson Research Center, where he contributed to the system software of the SP1 and SP2. He is the author of *Optical Computing: A Survey for Computer Scientists* (MIT Press, 1988), selected by the Association of American Publishers as the best new professional and scholarly publishing book in computer science for 1988.

**Michael Naaman** received a BSc from the Hebrew University, Jerusalem, where he is currently an MSc student. His research focuses on computational learning and time series. His research interests also include artificial intelligence and genetic algorithms.

Address questions about this article to Feitelson at the Institute of Computer Science, The Hebrew University of Jerusalem, 91904 Jerusalem, Israel, e-mail feit@cs.huji.ac.il.