Self-Tuning Systems

Dror G. Feitelson Michael Naaman
Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
{feit,mnaaman}@cs.huji.ac.il

Abstract

Modern operating systems are highly parameterized, allowing system administrators to tune them in order to achieve optimal performance for the local workload. However, this is a difficult and time consuming process. We propose a mechanism to automate this process, by running simulations of system performance for various parameter values instead of the system's idle loop. The simulations are driven by log files containing information about the local workload, and genetic algorithms are used to search for the optimal parameter values. We evaluated this idea by running such simulations off-line. A case study involving batch scheduling on an iPSC hypercube found parameter values that reduced fragmentation and salvaged a quarter of the computing cycles that were lost when using the default values.

1 Introduction

Modern operating systems are highly parameterized, meaning that the algorithms and policies used are not completely defined. Instead, the policies are controlled by a set of parameters that can be modified by the system administrator in order to tune system performance. Perhaps the best know examples come from file systems, in which various features such as the block size and the way in which disk blocks are allocated can be modified, albeit within the general framework dictated by the system design [11].

Regrettably, the process of tuning the system is typically ad-hoc, possibly with some vague guidelines, but nearly always with no direct way to measure the effect of changes to the parameter values. Thus system administrators are forced to use a trial-and-error approach as they seek parameter values that will optimize system performance for their local workload. We suggest a mechanism to automate this process.

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 the pages that they serve. These log files represent knowledge about the local workload. Our methodology is to use this knowledge to drive simulations of system behavior with different parameter values, and measure the resulting performance. Genetic algorithms are 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, so as not to cause any overhead.

The concept of self-tuning is explained in greater detail in Section 3, and evaluated using a specific case study in Section 4. But first we provide some background on genetic algorithms and their use in optimization.

2 Genetic Algorithms

This section should be a sidebar

Genetic algorithms, as their name suggests, are based on a biological analogy. In fact, it is now common to view the evolution of sexual reproduction — and the genetic mixing that comes with it — mainly as a mechanism for sampling and searching the vast space of possible genomic configurations [7]. 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 to a framework more than to a specific algorithm.

In essence, genetic algorithms involve iterative searching in a large configuration space. In each iteration, several potential configurations from different parts of the space are evaluated. The best ones are then combined with each other in random ways, and used 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 outside the scope of the optimization procedure — it is determined by the goals of the optimization. But 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 is changed 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, and produce offspring (this is the reason for the name "genetic algorithms"). The probability of mating is proportional to the 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.

When two individuals are mated, the cross-over operator may be applied. This operator chooses a random point in the chromosome, and creates two new chromosomes based on the two parents: one gets the first part from one parent and the second from the other parent,

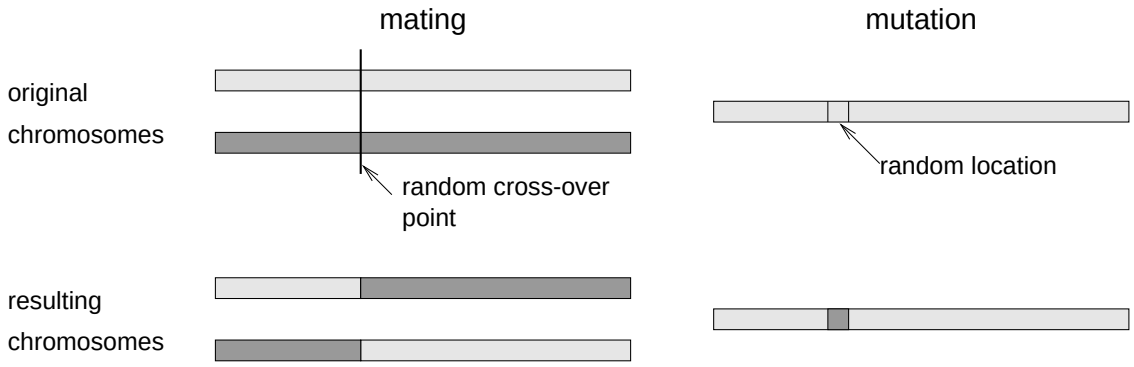

Figure 1: *The genetic algorithm operations on chromosomes.*

and the other gets the opposite (Fig. 1). Thus parameter combinations that lead to higher performance are mixed in various ways, potentially leading to new combinations with even better performance. If cross-over is not applied, the two parent chromosomes are simply inserted into the next generation without change; as they were selected according to their fitness, this means that the fittest individuals persevere.

In addition to cross over, mutations may be inserted into chromosomes, in order to create parameter values that were not present in the original population. This is done by changing a single bit at random (Fig. 1).

Many variants on this basic scheme are possible. For example, one must decide on the following issues:

- How are configurations represented in chromosomes? Specifically, what alphabet of symbols is used?
- When a new generation is created, does one take only the offspring, or rather the best individuals out of the joint pool of parents and offspring?
- What should the size of each generation be?
- How does one normalize the fitness values and turn them into mating probabilities?
- What are the probabilities of cross-over and mutations?

Significant research has been conducted on these and other issues, and on how they affect the convergence properties of the optimization procedure [6].

3 Self-Tuning Using Genetic Algorithms

When designing operating system algorithms and policies, many micro-decisions have to be made: the size of a table, the threshold used to decide when to activate a certain procedure, the order in which a data structure is scanned, and so on. The problem is that these decisions may have unknown consequences in terms of 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 they can be *changed* by each installation's system administrators. For example, the table size may be decided by the system administrator as part of the system configuration. The threshold value may be set by a special system call, executed by an operator's user interface.

While this approach shifts the burden from the system designers to its operators, it does not always solve the problem. True, the operators should have more knowledge about the local conditions and workload, and should be able to use this knowledge in order to fine-tune the system. But the operators may lack detailed knowledge about the inner workings of the system, and thus not appreciate the finer implications of setting various parameter values. Also, system administrators are notoriously overworked and busy solving various crisis situations, leaving little if any time for elective chores such as tuning.

The alternative that we propose is self-tuning systems. In this approach, the system designers create the framework that will carry out the optimizations and tuning. However, the execution of this framework is delayed until the system is deployed in the field, and can measure its specific workload. Technically, the framework simply consists of a systematic search of the parameter space.

Given that the search space is very large (many parameters that 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 as our optimization procedure. The rest of this section explains the mapping of the system tuning problem into genetic algorithm structures.

The easy part is the representation of a set of parameter values as a chromosome. This can be done by simply concatenating the binary representation of the parameter values. Crossing over will then take one set of values from one parent, and the rest from the other. It is also possible to 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.

The harder part is evaluating the fitness of these chromosomes. First, on must define an appropriate objective function. This objective function reflects the performance metric that one wishes to optimize, such as utilization or response time. Indeed, it is possible to construct a system that can optimize any of a set of metrics, and leave the choice of metric as the only parameter that has to be set by the local system administrator.

The evaluation of the chosen function for a certain set of parameters is done by simulating the behavior of the system based on a record of the local workload. This implements a sampling of the mapping $P \times W \mapsto Q$, 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 are able to rank the different combinations of

parameter values as they relate to the local workload, and quantify their quality 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 whole procedure. Operating systems are complex things, and a detailed simulation may be needed, involving high overhead and extensive logs. Luckily, 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 there are harder cases. Consider the optimization of 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, which is not maintained normally. A possible solution is to use sampling and collect the required information only for a short duration rather than all day long.

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 sets of parameter values, the process of iteratively evaluating these sets using simulation and then combining the best-performing sets together 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 — there is no need to wait 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 essentially at no cost, by running the optimization procedure in the background in place of the idle loop. Thus the system devotes cycles to optimization only if there are no user applications that can use them. In particular, idle time at night can be used to optimize a system that is heavily utilized by day.

4 Case Study

The iPSC/860 hypercube has a well defined, non-trivial, and highly parameterized batch scheduling algorithm [9]. In addition, a trace of a production workload on such a system is available [5]. This therefore makes a good case study, even if batch scheduling and the iPSC/860 are not of much interest in themselves.

4.1 The iPSC/860 System

The iPSC/860 is a parallel supercomputer produced by Intel in the late '80s. The architecture is based on nodes containing an Intel i860 RISC processor and some local memory, which are 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

time	*number of nodes*			
limit	16	32	64	128
20 minutes	q16s	q32s	q64s	q128s
1 hour	q16m	q32m	q64m	q128m
3 hours	q16l	q32l	q64l	q128l

Table 1: *Batch queues used on the 128-node iPSC/860 at NASA Ames.*

jobs on subcubes, i.e. on embedded hypercubes of a lower degree. The operating system imposes a limit of 9 on the degree of multiprogramming.

The workload trace used in this study comes from the iPSC/860 installed at NASA Ames, and covers the fourth quarter of 1993 [5]. At the time, this machine was the workhorse for computations at the Numerical Aerospace Simulation facility. The log includes a total of 1044 batch jobs. The batch queues that were in effect at the time are summarized in Table 1. The use of these queues is explained below.

4.2 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 non-prime shift at night. During prime time, some of the nodes are allocated to the batch partition, and the rest are reserved for interactive work. During non-prime 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 nodes that are available. 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 was sufficient in order to demonstrate the workings of self-tuning. Thus we only handle the optimization of those parameter values that are unique to batch scheduling. We do not optimize other parameters, such as the one that controls the size of the interactive partition during the prime shift. the following description is based on the Intel MACS (Multiuser Accounting, Control, and Scheduling) manual [9].

The batch scheduling algorithm is based on two main concepts. The first is *prioritizing* the jobs to decide which job will be scheduled next. The second is the use of reservations in order to accumulate processors for large jobs, which is called *leveling*.

The scheduler has a set of queues, to which jobs are submitted. Each queue is characterized by several attributes. For example, queue attributes include limits on the number of requested nodes and on the requested run time (the actual values in effect in the traced system are given in Table 1). Another attribute is whether the queue is active only during non-prime time, or also during prime time. The most important attribute for our work is the queue's priority, which has a direct impact on the priority of jobs submitted to it. In

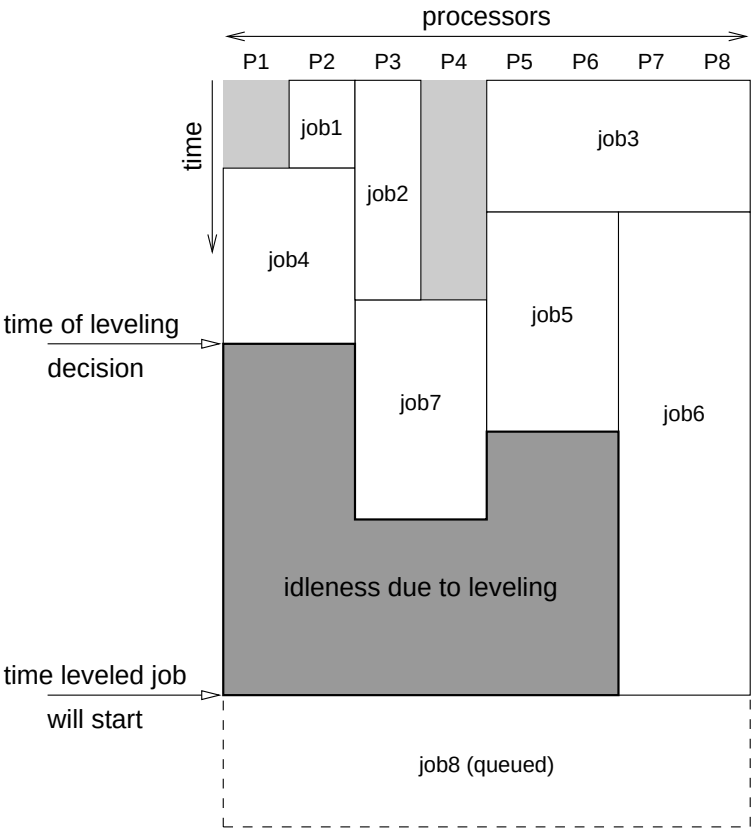

Figure 2: *Leveling is done if the induced idleness is smaller than* $A\_HOLE\_SIZE + B\_HOLE\_SIZE \times req\_nodes$*; otherwise it is considered too expensive. Gray shading represents idle nodes.*

addition, there is a global system parameter called `A_TIME_PRI`, which 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 basic priority of the queue, and $wait\_time$ is the time the job is already 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 then there are free, the scheduler considers leveling it. Leveling means not scheduling any more jobs until there are enough nodes to run the waiting job. Obviously such leveling requires the scheduler to 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 how much resources will be lost, and compares this amount with a tolerance that is 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.

An example is given in Fig. 2. When job 4 terminates, the scheduler has two nodes available, and the queued job with the highest priority (job 8) requires all 8 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 the loss of resources, the scheduler does attempt to schedule small jobs on the idle nodes, provided their time limit indicates that they will end before the time of leveling. However, this is only done after the decision to level.

4.3 Formulation for Genetic Algorithms

The iPSC batch scheduling algorithm on the NASA system has 15 parameters. But what parameter values will lead to the best performance? In order to use the self-tuning framework to find optimal parameter values, it is necessary to encode the algorithm for optimization using genetic algorithms.

4.3.1 Representation in Chromosomes

The different parameters to be optimized have different ranges of values:

- A_HOLE_SIZE — in the range of 0–255
- B_HOLE_SIZE — in the range of 0–5
- A_TIME_PRI — in the range of 0–5
- 12 queue priorities — in the range of 0–255

We represented all 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.

4.3.2 Fitness Function

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

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

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

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

(where $total\_time$ is the wall clock time from the start of the first job to the completion of the last job.)

This ratio gives a non-normalized 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 reasons of simplicity, and because it matches the goals of a batch scheduler. The results may be different if another definition of fitness is used.

4.3.3 Evaluating the Fitness

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

- Only batch mode — all 128 nodes of the machine are in the batch partition, and we are only optimizing the batch scheduling algorithm.
- All jobs were submitted before scheduling begins. This assumption reflects a model where batch jobs are submitted during the day, but not scheduled, because most batch queues are disabled. Then, when the prime-time shift ends, all nodes are allocated to the batch partition, and all queues are enabled, allowing the jobs that were accumulated during the day to be scheduled.
- All queues can be scheduled. Again, this is the situation in the non-prime-time shift.

As noted above, the workload used to drive the simulation is based on a detailed log of everything that ran on the iPSC/860 at NASA Ames during the fourth quarter of 1993 [5]. 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. Instead, we sometimes unified groups of several consecutive days of real workload into a single day of simulated workload, thus artificially increasing the load during each simulated day. The job characteristics (number of processors and runtime) remain the same as in the original workload. The criteria for unifying days was the desire to achieve either of the following: a load of around

Population size	120
Chromosome length	120 bits
Probability of mutation	0.1
Probability of cross-over	0.5
Run length	150 generations

Table 2: *Parameters used in genetic algorithm implementation.*

20–25 jobs, or 1000–1300 node hours (corresponding to 8–10 hours of using 128 nodes). After these unifications, the duration of the log was reduced from three months to 70 days.

The simulation handles each day individually, and then reports the average utilization for all the days. It is an event driven simulation of the scheduler, where the events are the terminations of running jobs (it is assumed that there are no additional arrivals during the non-prime shift). The simulation then schedules the next jobs to be run, according to the algorithm described above. If the next job cannot be scheduled, it tries to level it. If the job can be leveled, smaller jobs are scheduled as possible to reduce the idleness. Otherwise, this job scheduling is deferred, and the next job will be considered. This is done until there are no free nodes, or no jobs to schedule. Then, simulation time is advanced to the finish time of the next finishing job. Its resources are freed, and scheduling runs again.

4.3.4 Genetic Algorithm Dynamics

The implementation of the genetic algorithms framework was done with the sga-c package, an implementation of Goldberg's Simple Genetic Algorithms [6, chap. 3]. This implementation gives a very basic set of tools to implement genetic algorithms, and was sufficient for our needs.

The parameters used in our experiments are summarized in Table 2. 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 cross-over during mating was 50%, and the probability of mutation was 10%.

4.4 Experimental Results

The following results were obtained by executing the genetic algorithm as described above. This is a retrospective experiment, using old logs, rather than performing a run within a live system. However, note that this is not a simulation of the self-tuning idea, but rather these are exactly the same simulations 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, and those for raw values are shown.

Using a Pentium Pro 200 running BSDI, the time to complete one generation was about 1 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 between 2 and 3 minutes, and executing all 100 runs with different initial populations took several hours. While this is a significant amount of time, it should be noted that we only needed it in order to evaluate the approach. A real implementation only needs to perform a single run, which in our case takes a couple of 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 were found 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.

In general the results we achieved showed a significant improvement in utilization. Running the simulation with the default Intel parameters resulted in 88.4% utilization for our workload. In all our runs we achieved utilizations of between 91.03% and 91.25%. This is a 3.2% increase in utilization in absolute terms, and a 24.6% reduction in wasted processing capacity because of fragmentation. These results also testify to the efficiency of genetic algorithms as a search procedure: each run included specific checks of only $120 \times 150 = 18000$ parameter combinations out of the 2^{120} possible combinations, yet they all achieved essentially the same results.

Though we found an improvement in utilization, there is no straightforward behavior of the parameters. It seems as though the surface of the fitness function is 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 lead to about the same utilization. This implies that there are no prameter combinations that can achieve better performance than those we found, at least for this workload.

The combinations of parameters we found are presented graphically in Fig. 3. There is a graph for each optimized parameter, showing the value of this parameter in the best set from each of the 100 runs. It can be easily seen that the values of all parameters do not converge into some specific value, but are rather scattered. However, it is clear that 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 first pack the long jobs, and then pack the short ones in the space that is left [2]. We note in passing that the Intel manual suggests that queues for long jobs be given a priority of 40 rather than 15 as for short jobs [9], but our results indicate that this value is still much too low. The priority of the 128-long queue is especially interesting as it had a bi-modal distribution: in some cases it was low, and in others high. This may be interpreted as meaning that the

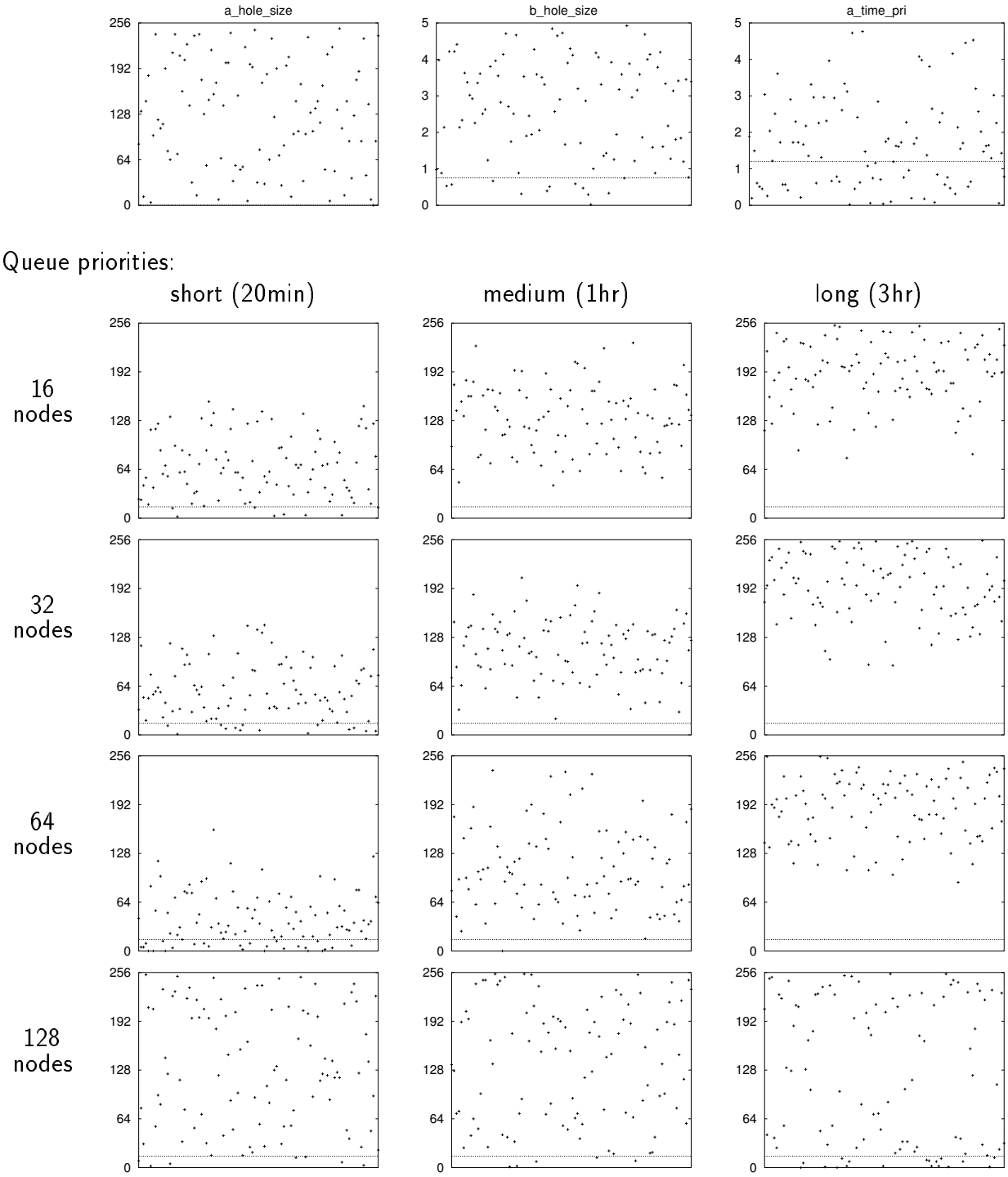

Figure 3: *Parameter values that produced the best results in different runs (run number is from 1 to 100 along the x axis). The dashed lines are the default values; for* `A_HOLE_SIZE` *it is zero.*

128-node jobs should either be scheduled first, or last, but not mixed with other sizes, so as to reduce the waste of multiple leveling actions.

It can also be seen that the parameters that control leveling decisions (A_HOLE_SIZE and B_HOLE_SIZE) are 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 indicate that the decision was to level in about 84% of the cases when it was considered. The values for A_TIME_PRI are generally low, as the queueing time of batch jobs is indeed not an important consideration when optimizing for utilization. Had response time been included in the fitness function, we expect that this parameter would have been more important.

5 Related Work

This section should be a sidebar, with separate references

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 that is built into a specific algorithm.

Interestingly, the concept of tuning system behavior to the workload has been rather popular in the field of parallel job scheduling. Indeed, the whole area of scheduling with adaptive or dynamic partitioning is based on systems that change the allocation they make as a function of load conditions. Sevcik has proposed adaptive policies that decide on partition sizes based on the load and information about characteristics of the applications [15]. McCann et al. have proposed a dynamic policy that changes the allocation at runtime to reflect changes in the load and requirements [10]. Severance et al. propose a scheme that is less dependent on explicit information, in which the system measures the performance of a barrier synchronization to decide if the current number of threads is appropriate [16]. The closest scheme to ours was proposed by Nguyen et al., who measure the efficiency of a parallel job on several partition sizes and then decide on the allocation [12]. However, these schemes involve learning about a specific application at run time, 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.

Scheduling is not the only area where the system may learn something about its workload. Another area where significant research has been performed concerns memory management and page placement. The question is where to map a memory page, and when to move it to another processor, in order to reduce communication; this has to be done subject to dampening rules that avoid ping-pong situations. For example, Cox and Fowler describe a system in which pages are replicated and migrated according to their usage, but pages that migrate too often are frozen in place [4]. The Millipede system uses a more sophisticated algorithm to detect ping-pong conditions, based on the access history of each thread [13]. Moreover, it combines page migration with thread migration in order to ease such situations. An on-line competitive algorithm for page placement was suggested by Black et al. [1]: the page is moved when the cumulative cost of remote accesses matches the cost of moving it.

Again, these schemes learn about a specific job, at the expense of that job; the collected information cannot be used to benefit the whole workload.

There has also been some work that is directly related to our case study, in that it uses genetic algorithms to solve scheduling problems. However, this is typically done in the context of off-line algorithms that search for a specific near-optimal schedule [8], whereas our work is about finding good parameters for an on-line policy. Interestingly, it has also been suggested that the genetic algorithms themselves be parallelized [14].

Finally, it should be noted that other search techniques are also possible, in place of our use of genetic algorithms. For example, simulated annealing has been used in the context of task scheduling [3].

6 Conclusions

We have introduced a general framework for the optimization and performance tuning of operating systems: using the idle loop to run genetic algorithms that search for optimal parameter values based on data about the local workload. With this approach, the trial-and-error methodology often employed by system administrators is replaced by a scheme that at once removes load from human system administrators, and uses real data and measurements for a more methodological search for optimal solutions.

A case study involving the batch scheduling algorithm from the iPSC/860 hypercube was conducted to validate this approach. We carried out the proposed optimization scheme off-line, simulating the scheduling of multiple jobs under various scheduler parameter values. The results were very promising. Specifically, the search procedure always found parameters that lead to about 91% utilization for the workload we used, as opposed to only 88% utilization for the default parameters. While this is only a difference of 3 percentage points, it represents a reduction of one quarter of the resources that are lost to fragmentation. In retrospect, it turns out that the parameter values that were found by self tuning cause long jobs to be scheduled first, which is indeed known to lead to better packing.

While we are confident that the proposed approach has merit, much remains to be done. Our main goal is to implement self tuning in a real system setting, and test its performance in such a context. This will enable us to also consider self-tuning at the price of additional overhead. The question is whether the potential improvement in performance is worth running the optimization procedure at the expense of user applications, rather than only instead of the idle loop, and also whether the overhead for additional logging of information (beyond that normally collected by the system) is worth while.

Acknowledgements

Thanks to Bill Nitzberg for providing the NASA Ames iPSC workload log, and to 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 paper.

References

[1] D. L. Black, A. Gupta, and W-D. Weber, "*Competitive management of distributed shared memory*". In 34th *IEEE Comput. Soc. Intl. Conf. (COMPCON)*, pp. 184–190, Spring 1989.

[2] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "*Approximation algorithms for bin-packing — an updated survey*". In *Algorithm Design for Computer Systems Design*, G. Ausiello, M. Lucertini, and P. Serafini (eds.), pp. 49–106, Springer-Verlag, 1984.

[3] C. Coroyer and Z. Liu, "*Effectiveness of heuristics and simulated annealing for the scheduling of concurrent tasks — an empirical comparison*". In 5th *Parallel Arch. & Lang. Europe*, pp. 452–463, Springer-Verlag, Jun 1993. Lect. Notes Comput. Sci. vol. 694.

[4] A. L. Cox and R. J. Fowler, "*The implementation of a coherent memory abstraction on a NUMA multiprocessor: experiences with* PLATINUM". In 12th *Symp. Operating Systems Principles*, pp. 32–44, Dec 1989.

[5] D. G. Feitelson and B. Nitzberg, "*Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 337–360, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[6] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison Wesley, 1989.

[7] J. H. Holland, *Adaptation in Natural and Artificial Systems.* MIT Press, 1992.

[8] E. S. H. Hou, N. Ansari, and H. Ren, "*A genetic algorithm for multiprocessor scheduling*". *IEEE Trans. Parallel & Distributed Syst.* **5(2)**, pp. 113–120, Feb 1994.

[9] Intel Corp., *iPSC/860 Multi-User Accounting, Control, and Scheduling Utilities Manual.* Order number 312261-002, May 1992.

[10] C. McCann, R. Vaswani, and J. Zahorjan, "*A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors*". *ACM Trans. Comput. Syst.* **11(2)**, pp. 146–178, May 1993.

[11] M. McKusick, W. Joy, S. Leffler, and R. Fabry, "*A fast file system for UNIX*". *ACM Trans. Comput. Syst.* **2(3)**, pp. 181–197, Aug 1984.

[12] T. D. Nguyen, R. Vaswani, and J. Zahorjan, "*Maximizing speedup through self-tuning of processor allocation*". In 10th *Intl. Parallel Processing Symp.*, pp. 463–468, Apr 1996.

[13] A. Schuster and L. Shalev, *Access Histories: How to Use the Principle of Locality in Distributed Shared Memory Systems.* Technical Report LPCR-9701, Computer Science Dept., The Technion, Jan 1997.

[14] M. Schwehm and T. Walter, "*Mapping and scheduling by genetic algorithms*". In *Parallel Processing: CONPAR 94 – VAPP VI*, pp. 832–841, Springer-Verlag, Sep 1994. Lect. Notes Comput. Sci. vol. 854.

[15] K. C. Sevcik, "*Characterization of parallelism in applications and their use in scheduling*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 171–180, May 1989.

[16] C. Severance, R. Enbody, and P. Petersen, "*Managing the overall balance of operating system threads on a multiprocessor using automatic self-allocating threads (ASAT)*". *J. Parallel & Distributed Comput.* **37(1)**, pp. 106–112, Aug 1996.