Evolving Processes and Evolution Schedulers for Concurrent Scheduling Controls and Parallel Evolutionary Computation

Tzilla Elrad and Jinlong Lin

Computer Science, Illinois Institute of Technology, Chicago IL 60616, USA

Abstract. In this paper we present concepts of evolution scheduling, which will provide us with a better scheduling mechanism for competing processes in the ready queue, the blocked queue, the PC (Priority Control) queue, and the CS (Client & Server) queue. The evolution scheduling is acceptable for the soft real-time system and useful for its performance to achieve fair and effective scheduling. In the comparison of queue dispatching and process dispatching of the evolution scheduler with other existing schedulers we can confidently conclude that where we use round robin, priority, and even FIFO scheduling, we can implement evolution scheduling to substitute for them without warning the application programmers to change their expectation of the scheduling controls. On the other hand, for those who pursue evolutionary computation, this is good news. It allows them to carry out natural competition in an easy way consistent with their operating system and programming language. Each evolving process, one with the ability to improve its adaptation to the environment, reflects on the CPU usage rate and I/O throughput load with its fitness function, and the evolution scheduler elaborates the optimization winner for the next quantum. So there are many benefits for both scheduler designers and evolutionary programmers.

1 Introduction

We are interested in adding reactive/adaptive and reflective facilities to concurrent object-oriented programming languages. It goes without saying that this is an effective method to directly apply artificial intelligence to concurrent systems. However, there is always another approach [1, 2, 3]. We would like to propose an evolutionary approach to the same goal, which will provide a new framework with much more intelligence from stochastic optimization to indirectly but eventually achieve reactive/adaptive and reflective intelligence. Nevertheless, this kind of intelligence is somewhat more natural than artificial intelligence.

On the other hand, there are many evolutionary computation people trying every effort to make good uses of parallel and distributed computers to improve speed and efficiency [10]. Nowadays, the concurrent programming languages have been implemented on many platforms including both uniprocessor and multiprocessor architectures. So the problem of speed-up is no longer how to distribute the system load to parallel machines, but on how to give parallel machines the ability to perform evolutionary and concurrent computation.

Also, parallel and real-time research groups are seeking for a real-time algorithm for the task allocation [6, 7, 8]. Most of the scheduling algorithms gain optimizations based on some unreliable assumptions, such as shortest jobs or earlier deadlines [9]. These are prone to type II errors. We would like to propose the evolution scheduler,

which follows the laws of nature. An evolution scheduler is an innovative dynamic on-line scheduler, which gives each evolving process a different vector of objective coefficients (health indices, from the viewpoint of processes) against system variables in addition to a different process id [5]. With the fitness function (health function) an evolution scheduler knows how well this process fits the current environment. The only rule to be used is "survival of the fittest," and however, we admit that the miserable processes can still live well under community welfare, which is the guarantee value of at least one for each process in a scheduling cycle. Also, the guarantee value can be used to improve the development of scheduling results for imprecise computation -- a situation where tasks obtain a greater value the longer they execute, up to some maximum value, as well as to prevent starvation [7].

There are two major comprehensive scheduling controls on existing concurrent programming languages [4]. One is named availability control, and the other is named race control. Sometimes, we call them synchronization control and scheduling control respectively. As we can see all the race decision mechanisms are usually implemented implicitly by using FIFO or round robin semantics. We assert that a good scheduling control should be naturally fair and also prevent deadlock, starvation and bottleneck. The explanation of the deficiency in concurrent programming languages is the lack of stochastic optimization algorithms built-in inside operating systems and primitive functions. However, this situation can be improved by implanting evolutionary computation into a concurrent object-oriented system. To achieve this purpose we have created process objects and scheduler objects to make the implementation of evolving processes and evolution schedulers easier. Currently evolution scheduling can be used for concurrent programming languages as well as parallel operating systems.

2 Definitions

We define a model of evolving processes and evolution schedulers. Some trivial parameters and machine-dependent arguments are left for implementation. This model is focused on the optimization solution to the winner order for evolution schedulers. Evolving process optimizations are accomplished through the processing of the acquisition of the scheduling optimization for an evolution scheduler.

2.1 Relative Priorities and Desire Functions

Resources are limited, but desires of human beings are limitless, so to design an effective and fair scheduler is always the most important matter for concurrent scheduling controls. A traditional priority scheduler gives each process a different priority. The process with the highest priority will be granted the next quantum of resource usage right. However, when it uses up the timeslice, its priority will be reduced by one in order to let other processes have chances to gain the resource. At the beginning of the next scheduling cycle each process will revert to its original priority (also known as a guarantee value), which cannot be changed by general users. If it can be changed, every user should have promoted his process to the highest priority with no questions. We call this fixed original priority an absolute priority.

However, we would like to introduce a relative priority, which can be based on the proposed desire functions. We define a desire function (or an objective function) as follows:

$$f_{d_{c,p}}(\boldsymbol{O},\boldsymbol{S}) = \frac{2^8-1}{n}\left(\sum_{i=1}^{n} O_{i,p}S_{i,c}\right)10^{-4} \text{ with } 0 \le O_{i,p}, S_{i,c} \le 100\,,$$

where the $S_{i,c}$ is any system variable (resource), and the $O_{i,p}$ is an objective coefficient for process p, which is the strength of desire of the specific resource for a process. Each process is born with an objective coefficient vector (o_1, o_2, o_3, … , o_n). Also, at the beginning of each scheduling cycle the scheduler will use it and the system variable vector (s_1, s_2, s_3, … , s_n) to measure how badly the process needs the resource competed for. Note that the system variables are changing from time to time, so the desired value can be looked on as a relative priority. The scheduler has a sufficient reason to decide which process should be granted the resource based on the desire indices of all processes waiting in the same queue. We suggest that the scheduler should give the resource to the most needed one, which has the highest desire index.

2.2 Fitness Functions and Guarantee Functions

The range of valid values for the desire (objective) function and the fitness function is set from 0 to 255. So we define the initial fitness (needing) function at the beginning of each scheduling cycle to be:

$$f_{c,p} = f_{c,1,p} = (f_{c-1,G,p} + f_{d_{c,p}})/2$$

where the $f_{d_{c,p}}$ is the desire (objective) function for the process p.

This means that each process inherits half of its fitness value from the preceding scheduling cycle. This design is reasonable and avoids extreme fitness values (255 or 0) as well. Each process reflects on a new fitness at the beginning of a scheduling cycle. Since the fitness value is ranging from 0 to 255, and the priority value is ranging from -20 to 20 (in most Unix operating systems), we would like to define a guarantee value to be a function of a fitness value as follows:

$$g_{c,p} = g_{c,1,p} = f(f_{c,p}) = f(f_{c,1,p}) = f((f_{c-1,G,p} + f_{d_{c,p}})/2)\,.$$

We recommend that the guarantee value is an increasing function of the fitness value, its maximum is below 20, and it should be greater than one. The guarantee value is used to prevent starvation. During a cycle of competition each process will be assured at least one chance to get the resource (such as CPU). A process with guarantee value of ten will be allocated ten folders of system resources compared to the process with guarantee value of one. In practice, at the beginning of each scheduling cycle, the scheduler has known that it will grant each process the guarantee number of timeslices of a resource. The total guarantee value is the number of generations in the scheduling cycle. For each generation the scheduler will choose a winner to grant it the resource. When the process uses up the timeslice, its guarantee value will be decreased by one. The only thing still unknown to the scheduler is the order of winner selection. The optimal winner order is dependent on the operations of recombination, mutation, and population selection. There are three reasons for which we do not use the objective coefficients as the operands of recombination, mutation, and population selection. (1) The fitness value is an increasing function of all

objective coefficients. There are too many objective coefficients to save computation time. (2) The fitness value is a multiple-one function of objective coefficients. The scheduler is interested in the fitness results, but not the combinations of process objective coefficients. (3) We try to solve the optimization for the scheduler, but not for each process (even though the process optimization is a side effect).

2.3 Feeding Functions and Scheduling Optimizations

We define the feeding function as follows:

$$F_c = \sum_{g=1}^{G} f_{c,g} \ .$$

where the G is the total guarantee value, i.e. $\sum_{p=1}^{P} g_{c,p}$, or the total generation at a given scheduling cycle, and the $f_{c,g}$ is the fitness value of any potential winner in the generation g, i.e. $f_{c,g} \in \{f_{c,g,1}, f_{c,g,2}, ..., f_{c,g,P}\}$.

The optimal winner order for the scheduler should be the maximum of F_c, which will feed most needs from processes for a scheduling cycle. We yield:

$$Max(F_c) = Max(\sum_{g=1}^{G} f_{c,g}) \geq \sum_{g=1}^{G} Max(f_{c,g}) = \sum_{g=1}^{G} Max\{f_{c,g,1}, f_{c,g,2}, ..., f_{c,g,P}\} = \sum_{g=1}^{G} f_{c,g,1} \ .$$

We admit that the summation of $Max(f_{c,g})$ might not be the optimal solution, but it is a better solution that can be found quickly. This means that to pick the process with the highest fitness for each generation is an optimization solution to the winner order. The scheduler uses the scheduling optimization to react/adapt to the whole system cycle by cycle. This is an infinite loop for the scheduler to seek for a scheduling optimization and for the system to provide the fluctuating evolutionary environment with changing system variables. In the concurrent scheduling system, the scheduler plays an active role, but the processes are passive roles. However, processes also evolve their fitness values under the processing of scheduling optimizations.

2.4 Evolving Processes and Evolution Schedulers

If a process is confronting the competition of evolution scheduling, we will call the process an evolving process. Evolving processes can adapt to the environment by changing their fitness values. This kind of reaction can make a process always suitable for the fluctuation of CPU usage rate and I/O throughput load. Note that processes will automatically change their fitness values (the same effect as changing the fitness functions) according to the processing of winner orders of scheduling optimizations, but they can not change their original fitness functions, which can only be changed by the authorized users, such as a system administrator.

An evolution scheduler is the final arbiter, who applies evolution scheduling, including genetic algorithms and evolutionary programming, to solve the scheduling optimizations for needing functions. The evolution scheduler also reflects on changing system variables and evolving process fitness values by determining a winner order of scheduling optimization for a scheduling cycle. The optimization decision will make the evolution scheduler improve its scheduling performance in return as well.

3 Comparisons

Comparison is the only way to prove performance. We would like to compare our evolution scheduling with the dynamic adaptive scheduling. Also, we find our design principle "Keep It Simple and Smart" instead of "Keep It Simple and Stupid."

3.1 Evolution Scheduling and Adaptive Scheduling

Adaptive scheduling follows three rules. (1) The priority is reduced by one, if the process consumes its timeslice. (2) The priority is boosted by one, if the process has decayed and remains unscheduled for one second. (3) The process reverts to its original priority, if it is blocked. On the other hand, our evolution scheduling behaves as follows. (1) Process dispatching follows genetic algorithms. (2) Queue dispatching follows evolutionary programming. (3) The unblocked process reflects on its new fitness value.

Queue Dispatching. Adaptive scheduling always picks the highest priority one. A process in queue B will never be scheduled before all the processes in queue A have been scheduled first. When a process in queue B has a priority decay for one second, it will be boosted to a higher-priority queue A. However, evolution scheduling does not use the concept of priority decay. Instead, it uses evolutionary programming for the queue dispatching, so all the processes in the queue A, queue B, and queue C will follow some kind of distribution, and these distributions overlap one another over the scheduling period. If only the process's fitness value is strong enough, no matter which queue it is in, even if it is in the lowest-fitness queue C, it still has a chance to be selected for the next run. Fig. 1 is a schematic of some of these operations.

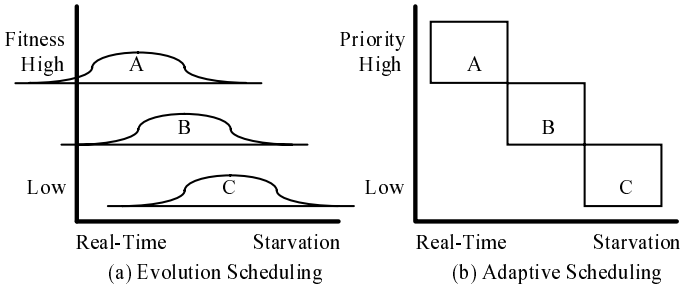

Fig.1. Evolution scheduling vs. adaptive scheduling in queue dispatching

Process Dispatching. For process dispatching in a queue, adaptive scheduling uses a round robin mechanism, so the processes a, b, and c will each be selected according to its arrival order in each scheduling cycle. However, genetic algorithms always pick the highest fitness value. This mechanism can be used in soft real-time systems. Also, the higher the fitness value is, the higher the guarantee value is. If we assume that processes a, b, c have guarantee values of eight, five, one respectively, then we can expect that process a might have eight timeslices, process b might have five timeslices, and process c might have one timeslice, and also their own timeslices

will mostly conjugate immediately. This mechanism can be used to reduce the context switches. This is interesting and also reasonable. Evolution scheduling allows the process with a strong fitness value to have more timeslices than the one with a weak fitness value. An evolution scheduler is a smart one, and it knows how to give the most needed process more system resources than other processes. Note that we use the relative priority, and the fitness value is composed of two parts, a half is from its original objective coefficients and a half is from the fitness of the previous generation, so we cannot expect that the highest fitness process for this scheduling cycle will remain for the next cycle. See Fig. 2 for the process dispatching comparison.

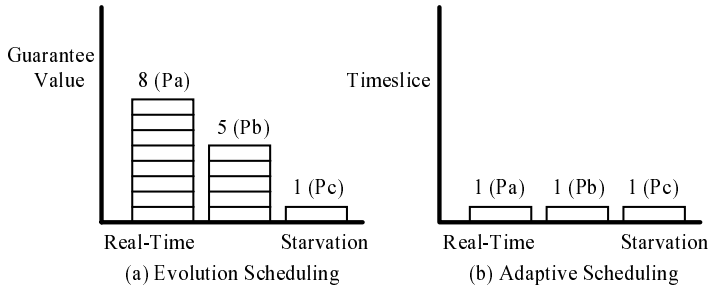

Fig.2. Evolution scheduling vs. adaptive scheduling in process dispatching

Design Principles. In Fig. 3, the vertical axis is the CPU utilization, and the horizontal axis is the process response time. The SS line is a CPU supply line, which is the maximum capacity for one unit of CPU, and the DD curve is the CPU demand curve at a given time. Suppose that we currently use the adaptive scheduling, and the CPU utilization rate is still under a full load. We have a balance point at A. Now we change from adaptive scheduling to our evolution scheduling. Because evolution scheduling is smarter than adaptive scheduling and can meet real-time criteria better than adaptive scheduling, we may find another point on the DD curve on the left-hand side of the A point, say the E point. The E point, for the evolution scheduler, uses a little more CPU time to achieve better real-time performance. So this assumption is reasonable. Suppose that we can also find another scheduling algorithm, we call it "complicated scheduling," balanced at C. It can get better real-time performance than evolution scheduling and adaptive scheduling. All these three scheduling approaches are still under full load in the current situation. Whenever there are more processes going into this system, the CPU demand curve is moved up and to the right from the solid DD curve to the broken dd curve. In this case, if all these three scheduling approaches still want to keep the same real-time performance, adaptive scheduling has to use more CPU, and let its new balance point reach the F point. New evolution scheduling balance point is at G. However, the complicated scheduling will reach the B point, which is a bad point, because its utilization rate is over the current capacity of one unit of CPU. This diagram reminds us that if we have to make our evolution scheduler smarter than the adaptive scheduler, we also have to keep our evolution scheduler simpler than the complicated scheduler. We can get our design principle for evolution schedulers that is "Keep It Simple and Smart."

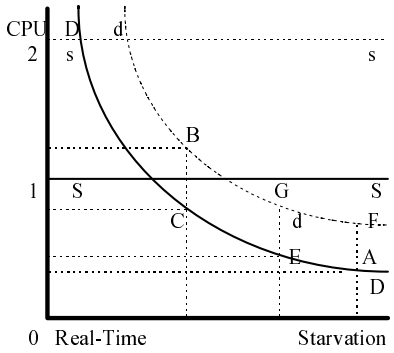

Fig.3. Evolution scheduling vs. adaptive scheduling in performance

3.2 Advantages of Evolution Scheduling

We would like to conclude a list for the advantages of evolution scheduling compared to the traditional scheduling approaches used in Unix-like systems.

A Dynamic Scheduler. Evolution scheduling is a dynamic scheduler, because it uses the fitness functions to measure the needs of each processes. The evolutionary operations of "recombine" and "mutate" give each process a hope to be chosen as the next winner.

Partially Deterministic. Evolution scheduling is partially deterministic, because we use both evolutionary scheduling and genetic algorithms. From the viewpoint of schedulers, genetic algorithms always pick the highest fitness value for the next run, so they are deterministic mechanisms. The partially deterministic are more flexible in designations and applications.

Parallel Scheduling. Each evolving process will have a chance to be selected for the next run, so it is a parallel scheduling algorithm. To be noted that even the weakest process in the weakest queue can be chosen as the next winner through the "mutate" operation on the fitness value of the process.

Exempt from Type II Errors. We assume nothing about the shortest job or the earliest deadline, so our evolution scheduler is exempt from type II errors. Evolution scheduling is a selection mechanism that obeys the laws of nature.

Soft Real-time Scheduling. To give each evolving process a different fitness value makes the evolution scheduler a potential soft real-time scheduler, because the evolution scheduler schedules the most important process first as well as giving it more timeslices.

Less Context Switches. Evolution scheduling allows each evolving process to have a different guarantee value. The higher the guarantee value, the more timeslices are granted. This can reduce the number of context switches to a large extent, and promote the performance of the scheduling algorithm.

A Total Solution to the Queue and Process Dispatching of OS and PL Scheduling Controls. Evolution scheduling provides us with a full set of scheduling options for queue dispatching and process dispatching. Also, it can be applied in operating system scheduling controls and programming language scheduling controls.

4 Implementations

In order to get more beneficial evidence in practice, we have implemented our evolving processes and evolution schedulers in the latest version of the Linux operating system. You may get the source codes on the public Sunsite, at http:// sunsite.unc.edu/pub/Linux/kernel and http://sunsite.unc.edu/pub/Linux/system/status. Also, you are cordially invited to visit our research web pages, at http://www.iit.edu/ ~linjinl/esep.html & xesep.html for the latest information. We have used this implementation to prove the two major significant benefits for evolution schedulers, which are (1) to reduce context switches in a large amount and (2) to provide concurrent scheduling systems with a soft real-time scheduling mechanism.

We define our desire (objective) function as follows:

$$f_{d_{c,p}}(\boldsymbol{O},\boldsymbol{S}) = \frac{2^8-1}{5}\left(\sum_{i=1}^{5} O_{i,p}S_{i,c}\right)10^{-4} \text{ with } 0 \le O_{i,p}, S_{i,c} \le 100 .$$

There is no loss of generality in taking s_1 = es_constant, s_2 = es_sys_free, s_3 = es_cpu_free, s_4 = es_ mem_free, and s_5 = es_dsk_free. S_1 is a constant of 100. S_2 is the system load capacity available. S_3 is the CPU capacity available. S_4 is the major memory capacity available. S_5 is the major disk capacity available. On the other hand, we define our guarantee function as follows:

$$g_{c,p} = f(f_{c,p}) = (f_{c,p} >> 5) + 1 ,$$

where $1 \le g_{c,p} \le 8$, for $0 \le f_{c,p} \le 255$.

You may try to add some system variables to the desire function, or change the guarantee function to build your evolution scheduler based on the source codes of esep.

5 Conclusions

Evolving processes and evolution scheduling have many advantages; however, they have a disadvantage. The disadvantage is that the evolution scheduler is a time-consuming one. Fortunately this is not a fatal issue. The time-consuming problem can be solved by (1) further simplified genetic algorithms and evolutionary programming, which will save time in the calculation of algorithms, (2) a faster CPU, which becomes possible because of the developing hardware improvement in chip technology, or (3) a slave CPU, which is dedicated to do scheduling and supports part of the communication load for a master CPU.

5.1 Significance of This Research

We can conclude many significant improvements. From the viewpoint inside evolving processes and evolution scheduling we obtain four advantages: (1) evolving processes and evolution scheduling provide object-oriented schedulers and processes for programmers, (2) evolving processes and evolution scheduling support flexible mechanisms and rich policies for end-users, (3) evolving processes and evolution scheduling apply parallel scheduling, which interests artificial intelligence experts, and (4) evolving processes and evolution scheduling distribute resources to waiting

processes according to the ratio of each process guarantee value to total guarantee values. From the viewpoint outside evolving processes and evolution scheduling we obtain two major advantages, (1) evolving processes and evolution scheduling construct a reactive/adaptive and reflective artificial intelligence framework for concurrent programming, and (2) evolving processes and evolution scheduling enable parallel/concurrent and distributed evolutionary computing on concurrent object-oriented systems for evolutionary computation programmers.

5.2 Future Work

Nevertheless, there are still five ongoing research areas: (1) simplifying further genetic algorithms and evolutionary programming, (2) defining system variables better, such as system_load, CPU_used, memory_used, I/O_load, etc., in order to be properly used by fitness functions, (3) reporting empirical results for evolving processes and evolution scheduling, (4) employing evolving processes and evolution scheduling to solve classic concurrent programming problems, and (5) migrating parallel evolutionary computation to evolving processes and evolution scheduling platforms. Each of these is currently being investigated by our research group.

References

[1] Elrad, T., Lin, J., Cork, D.J.: Scheduling controls in concurrent real-time systems through evolutionary computation. Proceedings of the 4th Annual Australasian Conference on Parallel and Real-time Systems, PART'97 (1997) 238-249.

[2] Elrad, T., Lin, J., Cork, D.J.: Evolutionary computation for scheduling controls in concurrent object-oriented systems. Proceedings of ISCA 10th International Conference on Computer Applications in Industry and Engineering, CAINE-97 (1997) 72-75.

[3] Elrad, T., Lin, J., Cork, D.J.: Evolutionary computation for scheduling controls in concurrent object-oriented systems. To appear in IJCA, International Journal of Computers and Their Applications, September (or June) (1998).

[4] Elrad, T., Verun, U.: A hierarchical and reflective framework for synchronization and scheduling controls. Future Generation Computer Systems **457** (1996) 1-14.

[5] Fogel, D.B.: *Evolutionary computation - toward a new philosophy of machine intelligence*. IEEE, New York, NY, **3** (1996) 67-120.

[6] Mok, A.: Firm real-time systems. ACM Computing Surveys **28-4es** (1996) 185.

[7] Stankovic, J.: The pervasiveness of real-time computing. ACM Computing Surveys **28-4es** (1996) 188.

[8] Stankovic, J., Spuri, M., Natale, M., Buttazzo, G.: Implications of classical scheduling results for real-time systems. IEEE Transactions on Computers **28-6** (1995) 16-25.

[9] Tanenbaum, A.S.: *Modern operating systems*. Prentice-Hall, Englewood Cliffs, NJ, **2** (1994) 27-73.

[10] Zomaya, A.: *Parallel and distributed computing handbook*, McGraw-Hill Companies, New York, NY, **40** (1996) 1118-1143.