

# A Study of Genetic Algorithms for I/O Scheduling

Natasha Gaitonde    Parag Sarfare  
{natashag, parags}@cs.arizona.edu

*Department of Computer Science  
The University of Arizona*

## Abstract

This project is an experiment in Computer Science carried out for the CS-630 course, under Dr. Richard Snodgrass. We studied the use of Genetic Algorithms (GAs) to improve disk throughput by tuning the disk scheduler to handle dynamically changing workloads. At the start of our experiment, we proposed a few hypotheses about the behavior of the GA-based input-output scheduler for the Linux kernel, when subjected to different workloads. We setup the experiment and obtained graphs depicting the throughput of the scheduler by conducting several tests with different sequences of workloads. The results indicate that GAs may not be directly applied to disk scheduling strategies where a highly responsive system is desired.

## 1 Introduction

The nature of workloads differs significantly between various applications and even within the lifetime of a single application. It is a non-trivial task for a system administrator to manually keep up with the changing workloads. One solution is to use an intelligent system that can self-tune its behavior to the nature of the workload. To develop such an intelligent system a good candidate is a widely used method in artificial intelligence, the Genetic Algorithm. Moilanen & Williams [1] have developed a generic Genetic Algorithm library that can be plugged-in the Linux kernel. It may be used for disk scheduling and other system strategies wherein intelligence can be built into the decisions the system makes. The intelligence of the Genetic Algorithm is in its self learning behavior. However, it has not been empirically proven that such a self tuning system disk scheduler would work for all types of hardware and software configurations.

We intend to profile the performance and behavior of genetic algorithms based disk scheduler. The profiling would be done using varying workloads from a set of commonly observed workloads. The experiments would provide the statistics that would justify as to what extent the use of genetic algorithms is suitable in an everyday computing environment.

### 1.1 Previous Work

We are not aware of any previous work that has been done, to verify feasibility of use of genetic algorithms in an everyday computing environment. Moilanen & Williams [1] have cited empirical results, but the computing machinery they use is a high end enterprise workstation, explained in section 5 whereas ours was a desktop PC.

## 1.2 Paper Organization

We start with explanation of the concepts that are involved to understand the premise of the experiment in Section 2. We will go on to stating the hypotheses we propose in Section 3. The infrastructure setup for conducting the experiments is covered in Section 4. The observations and results are in Section 5 & 6 respectively. Section 7 describes the possible directions for going forward with this experiment. Following that we state some general observations and experiences we had in this project.

## 2 Background

### 2.1 Disk Scheduling

Disk scheduling, also known as I/O scheduling, is the process that an operating system performs to order disk operations. Sending out request to block devices in the order that they are received results in awful performance. One of the slowest operation in a modern computer is disk seeks. Each seek positions the hard disk head at the location of the specific block and takes several milliseconds. Thus minimizing seek is absolutely crucial to the system performance.

An operating system (referred as the kernel henceforth) does not issue block I/O request to the disk in the order they are received or as soon as they are received. Instead it performs merging and sorting of disk operations to greatly improve the performance of a system as a whole. The sub-system of kernel that performs these operations is called I/O Scheduler. The disk operations that are submitted by the I/O Scheduler are prioritized and ordered depending upon various factors such as to - a) Minimize latency due to hard disk seeks. b) Prioritize a certain process' I/O requests. c) Give a share of the disk bandwidth to each running process. d) Guarantee that certain requests will be issued before a particular deadline.

In Linux, the kernel is responsible for controlling the disk access, and it does this using a sub-system called as the Elevator. The elevator is an interface that can be implemented by any disk scheduling strategy. Currently, the various strategies or schedulers that are available in the Linux 2.6 kernel include Anticipatory scheduler, Deadline scheduler, NOOP scheduler, Completely Fair Queuing scheduler. The scheduler can be chosen or changed at booting time or at runtime.

Moilanen & Williams [1] have used the Anticipatory scheduler for plugging in the genetic algorithm, as it was the default scheduler in the Linux 2.6 kernels when they developed the idea. The Anticipatory scheduler is based on the idea that synchronous disk requests come after brief periods of inactivity [7]. This "deceptive idleness" of the disk head, a situation where a process appears to be idle when viewed from the disk end, because a small delay between two fairly consecutive synchronous pieces of I/O will cause a normal work conserving disk scheduler to switch to an unrelated I/O. This situation is detrimental to the throughput of synchronous workload, where it degenerates into a seeking workload thus causing a considerable degradation in disk throughput. The Anticipatory scheduler is thus fine tuned for disk operations that are sequential. We will see its implications on the experiment later.

### 2.2 Genetic Algorithms

Genetic Algorithms (GAs) are one of most commonly used methods in artificial intelligence. They are an adaptive heuristic search algorithm based on the concepts of evolution such as natural selection. These algorithms generally simulate processes in an ecosystem using the principles laid down by

Charles Darwin of survival of the fittest. GAs use an intelligent exploitation of a random search within a defined search space to solve a specific problem. Many real world problems involve finding optimal parameters for a solution and GAs have generally proven to be outstanding optimizers.

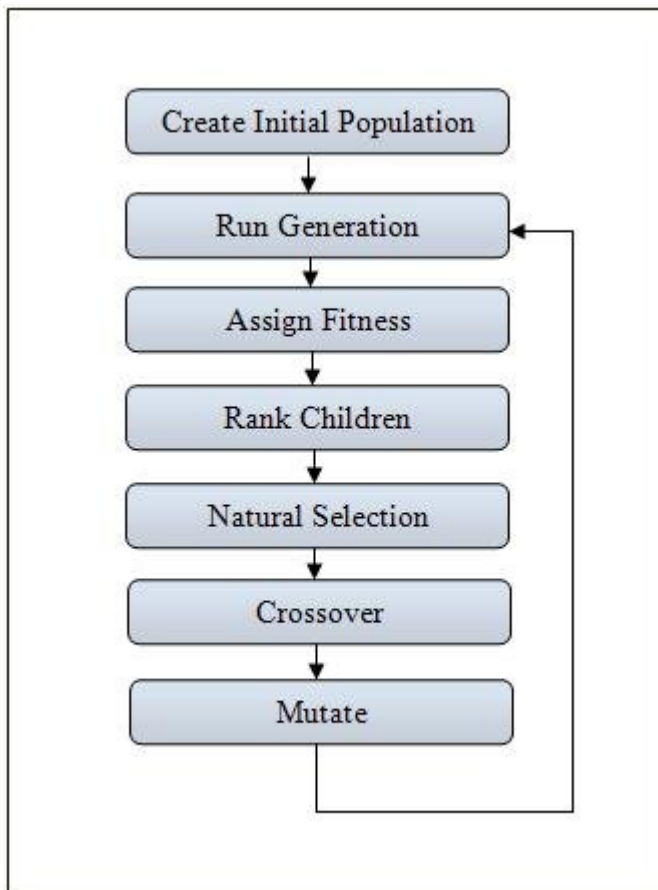


Fig 1: The Genetic Algorithm evolutionary process

GAs are implemented as a continual process in which a population of abstract representations (called *chromosomes* or the genotype or the genome) of candidate solutions (called individuals, creatures, or *phenotypes*) to an optimization problem evolves toward better solutions. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified or mutated to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. If the algorithm has terminated due to a maximum number of generations, a satisfactory solution may or may not have been reached.

### 2.3 Why a disk scheduler based on genetic algorithms?

There are currently 4 available a) Noop Scheduler b) Anticipatory IO Scheduler c) Deadline Scheduler d) Complete Fair Queuing Scheduler(CFQ).

The Noop Scheduler only implements request merging, this is used for systems where the disk seeks don't matter or occur, like flash devices. While the Anticipatory IO Scheduler implements request merging, a one-way elevator, read and write request batching, and attempts some anticipatory reads by holding off a bit after a read batch if it thinks a user is going to ask for more data. It tries to optimize for physical disks by avoiding head movements if possible - one downside to this is that it probably give highly erratic performance on database or storage systems. Deadline Scheduler implements request merging, a one-way elevator, and imposes a deadline on all operations to prevent resource starvation. Whereas the Complete Fair Queuing Scheduler implements both request merging and the elevator, and attempts to give all users of a particular device the same number of IO requests over a particular time interval.

The Anticipatory Scheduler is optimized for the common case [6] [7]. If the system configuration has a single disk (i.e., no RAID - hardware or software) and meant for desktop use, then this scheduler works best. If it's a multiuser system, probably CFQ or Deadline Scheduler provide better performance, while deadline gives the best performance for database systems [6].

All these schedulers are thus tuned to either specific type of workloads or specific types of systems, and none of them is capable of working in a completely diverse environment. This is where the GA comes into picture. Moilanen & Williams [1] have attempted to use the evolutionary nature of GA towards approaching a solution as disk scheduling strategy, so that optimal scheduling is possible in a diverse and dynamic environment.

### 3 Our Hypotheses

The intelligence of the GA has been used in the I/O Scheduler to self-tune the system. The premise behind this as discussed in the earlier section was to get optimal scheduler performance in a diverse and dynamic environment. We, in this experiment, make an endeavor to verify this premise. For this we first propose hypotheses on the expected ideal behavior from this GA enable I/O scheduler. Let's first define the terms that are expected in these hypotheses:

1. **Throughput** - A measure of the performance of the IO Scheduler measured in MB/sec. The higher the throughput the better is the response time of the system. Random read or write operations have a lower throughput as compared to their sequential counterparts as a higher seek time limits the response of the system.
2. **Convergence Time** - The time taken by the scheduler to reach a stable throughput, with acceptable variation around this stable value.
3. **Variation in Throughput** - Deviation of the throughput value after it has stabilized at a particular Throughput.

Based on our discussion on use of GAs in the I/O scheduler here are the hypotheses we propose:

**Hypothesis I** - *The time for the throughput to stabilize (Convergence Time) is not consistent.*

The GA is an evolutionary process. We make an assumption here that evolution within the algorithm takes varying amount of time and it is not a completely deterministic parameter. Based on this assumption, the hypothesis follows that time for the throughput to stabilize is not consistent.

**Hypothesis II** - *The maximum throughput attained varies within acceptable ( $\pm 5\%$ ) limits of the stabilized value.*

The reasoning applied to Hypothesis I applies here too. Even after reaching a stable value, we expect that the GA will keep the evolution process going and hence the stabilized throughput may deviate due to some unfit genes that might come into the current gene pool, though they would be weeded out at the next selection iteration.

**Hypothesis III** - *Convergence Time is within acceptable limits.*

The time required to attain the stabilized throughput must be within acceptable limits, for the use of GA in the I/O scheduler to be justified. Hence we expect that Convergence Time has an upper bound within acceptable usability limits.

The hypotheses can be very well described with Fig 2. Initially Workload 1 will be injected in the system. Assuming that the scheduler is not tuned for this workload, the throughput will be low. With

the self learning behavior of GA, the scheduler should gradually increase its throughput and attain some stable limit in acceptable time (the Convergence Time). When the workload is changed and a new Workload 2 is introduced in the system, the throughput is again expected to drop. And again the evolutionary process of GA should gradually improve the throughput till it attains stability. When stable the throughput is expected to alter with acceptable deviations which we call the Acceptable Variation.

As seen in Fig. 2 both the convergence times and variation in throughput should be under acceptable limits (Hypotheses II & III) and the convergence time for the same workload at different times is expected to vary (Hypothesis I).

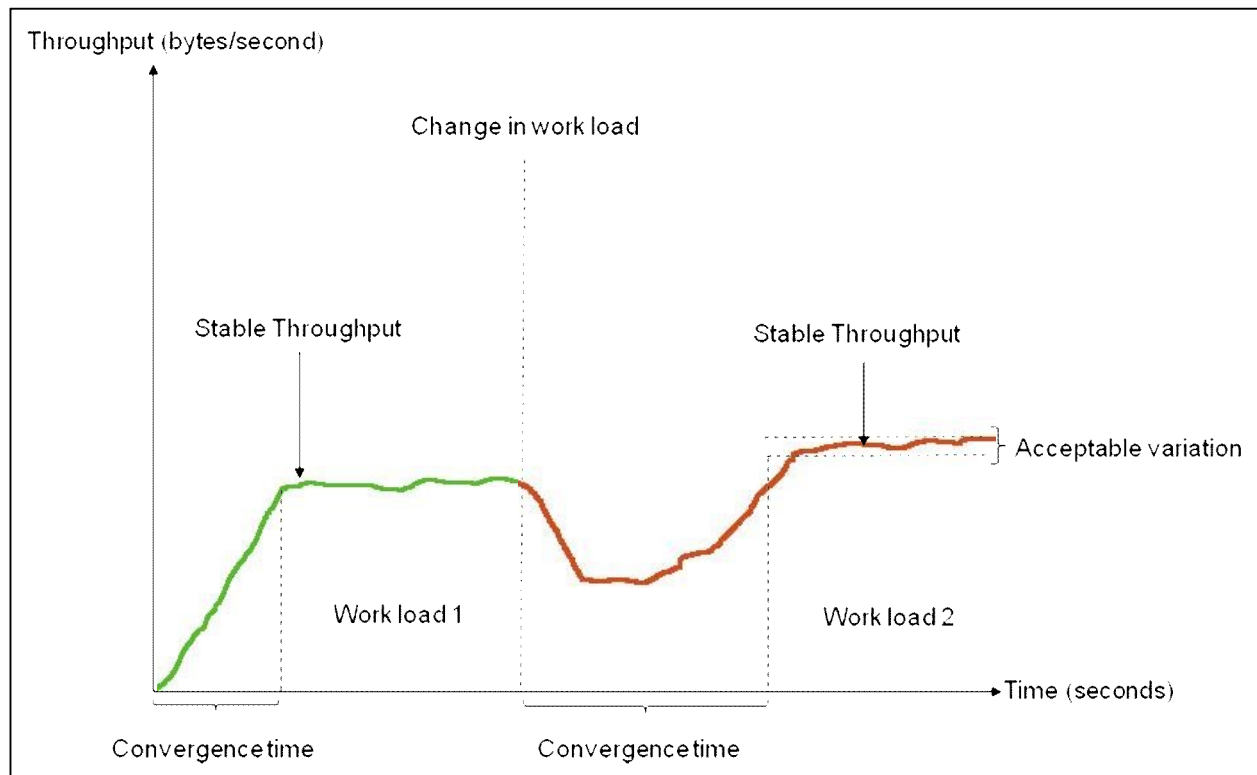


Fig. 2: Expected behavior of the I/O scheduler performance

## 4 Experiment Setup

Setting up the infrastructure for the experiments was the most challenging part of this project. For simulating various workloads we used the Flexible File System Benchmark (FFSB) tool [4], which is a widely used micro-benchmarking tool. We used a modified Linux 2.6.16 kernel with patches by Jake Moilanen [3] for the pluggable genetic library, IO scheduler framework and the Anticipatory Scheduler. Our desktop machine was a 2.4GHz Pentium 4 processor and an ATA hard drive of 80GB with a 7200rpm drive speed. The machine was running Fedora Core 6 GNU/Linux.

The different workloads we used for the experiments were Read Random, Read Sequential, Write Sequential, Meta-data and Direct I/O. All of these are workload profiles are provided with the FFSB distribution, we made few alterations to the profiles for the hardware we were using. This included

changing the sizes of file generated, the number of files that get created and the read and write block sizes. Read Random profile creates a read workload that causes random disk seeks, while Read Sequential and Write Sequential create read and write type of sequential disk seek type of workloads respectively. Meta-data profile generates disk operations on the file-system meta-data, whereas the Direct I/O profile generates read and write I/O using the O\_DIRECT flag which makes the system bypass the buffer cache.

We wrote scripts for dynamically extracting data out of the kernel, filtering the data to get required information and render graphs using this information. Fig. 3 shows the experiment infrastructure in detail.

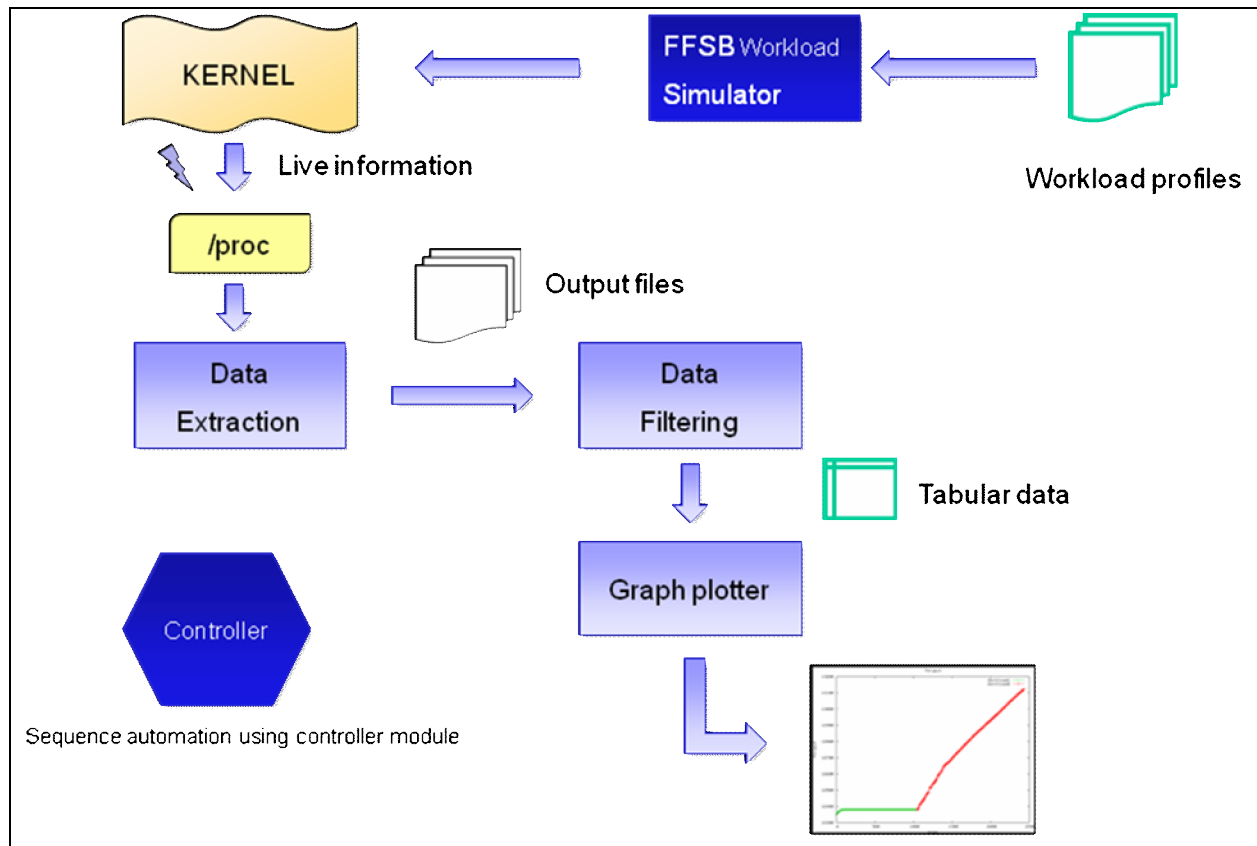


Fig. 3: Experiment infrastructure setup

#### 4.1 Experiment workflow

Different workload profiles mentioned earlier, two at a time, were fed to the FFSSB workload simulator. FFSSB would then generate the I/O corresponding to the workload. Once the workload is injected in the system, it would cause the disk seeks and the modified Anticipatory scheduler then creates schedules for the disk operations. The /proc file-system interface was used to read the live information out of the kernel. This was done periodically at the interval of every 2 seconds, which is equal to one genetic child lifetime, by the data extraction module. The output generated by the extraction process was then filtered by the data filtering module that converts it to tabular data suitable for further processing. The tables would then be used to dynamically plot graphs by the graph plotting module using GnuPlot [5]. We created GnuPlot [5] scripts on-the-fly to plot data from a sequence of two workloads on the same graph. The entire process of experiment was automated using a Controller module that governed the sequence and synchronization of operations.

We performed numerous experiments combining different workloads in different order of execution. To get reliable results, we repeated our experiments varying the workload execution time, for 1sec, 10secs, 500secs and 1000secs.

## 5 Observations

Using the FFSB Workload simulator, we subjected the GA-patched Linux kernel to all combinations of the five workloads mentioned in section 4. We paid close attention to the response of the scheduler when a newly injected workload causes a change in system throughput. When a workload is running for certain predefined time (1000secs, Section 4.1), we expected that the system throughput to stabilize. When a new workload comes in, the GA will try to gradually improve the scheduler performance to finally converge to a stable throughput whose value depends on two criteria. First, the nature of the new workload, and second, the past history it has captured about the workload.

To determine the convergence time, five different workloads were simulated and interleaved. These included a random read, a random write, a sequential read, a sequential write and a direct I/O. When we start with one workload and then to go another one of an opposite nature, e.g. a sequential write followed by a random read, the GA evolution process begins from scratch, with a initial set of probably unfit genes. The only exception is when we have run enough workloads of a certain type; the GA can learn from the history and start tuning the scheduler at a higher throughput level.

### 5.1 Expectations

At the start of our experiment we assumed the behavior of the system to follow the behavior indicated in Fig. 2.

1. Every newly injected workload will attain a stable throughput after a brief transition period. This might result in variations in throughput (mostly gradual improvements) until a stable throughput is attained.
2. The time taken to reach a stable throughput is within acceptable limits.
3. Random operations will have a lower throughput than sequential operations as their throughput is limited by high disk seek times.

### 5.2 Key Points to Observe

The reader should correlate the observations with the following intentions of the experiment:

1. Does the throughput for a workload converge?
2. How long does the convergence take?
3. Is there an acceptable variation once a stable throughput is attained?

### 5.3 Workload Sequences

Our experiments consisted of different sequences of workloads. We will discuss a few from which we could observe most of the important points mentioned above. More results are included in the Appendix.

### 5.3.1 Meta-Data followed by Read Random Workload

Both operations are dominated random read operations and as we expected, they converged within an acceptable amount of time, 100 seconds for Meta-data and 10 seconds for Read Random. It can be noted that the throughput of these operations is not very high and once a stable value is reached, the throughput varies within our predicted limits of  $\pm 5\%$ . Hence for both of these workloads, our hypotheses were verified. Fig. 4b is the same as Fig. 4a with the throughput scale expanded to enhance our understanding of how the throughput for the two workloads is obtained as flat lines in Fig. 4a.

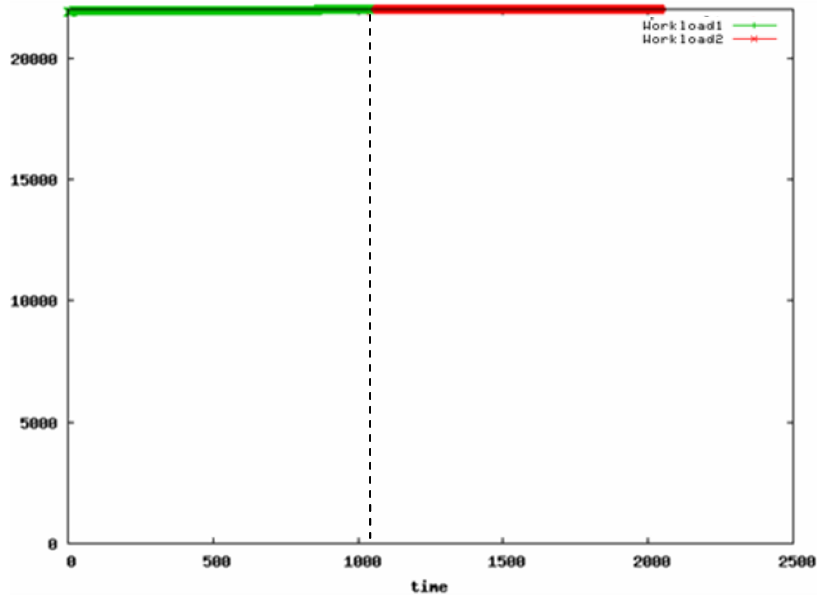


Fig. 4a: Meta-data – Read Random, with compressed throughput axis\*

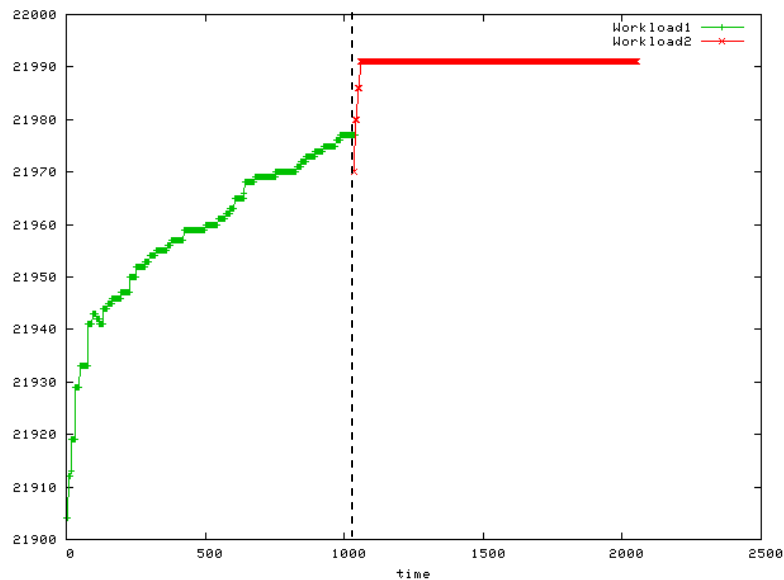


Fig. 4b: Meta-Data – Read Random, with enlarged throughput axis\*

\* Please note that all the graphs in this document are Throughput(Y) Vs Time(X) plots.

### 5.3.2 Write Sequential followed by Read Random Workload

As can be observed from Fig. 5, Sequential Write workloads have a steadily increasing throughput. In fact only on one occasion we were able to achieve a stable throughput for it, wherein we ran several Sequential Write workloads for prolonged periods of time and then we plotted the last one. This demonstrates the learning nature of the GA and how it utilizes a history of previous results.

The behavior for sequential writes that we observed every other time is shown below as Workload 1. There is no convergence similar to what we observed for the Meta-data or Read Random workloads.

Also note that the behavior of the Read Random workload is consistent with 5.3.1, i.e. convergence within 10 seconds followed by stable throughput.

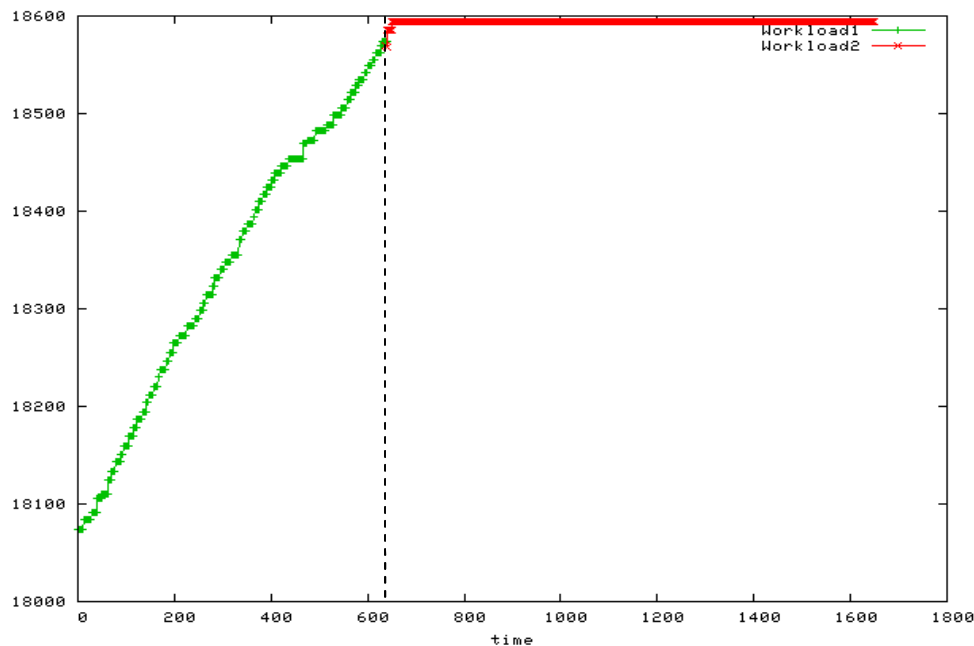


Fig. 5: Write Sequential – Read Random\*

### 5.3.3 Meta-Data followed by Write Sequential Workload

Fig. 6 shows the throughput for Meta-data in accordance with our observations in 5.3.1. Meta-data workloads converge within 100 seconds of running time and remain at the stabilized throughput until there is a change in the workload. Since meta-data operations are mostly short bursts of random read operations, the system response for the two workloads is similar. However Meta-data can achieve a higher throughput as compared to Read Random, and hence take longer to converge.

The second workload is a sequential read and being a sequential operation, can achieve a high throughput. We were unable to see a convergence to a stable throughput for all the time slices we ran this workload. This goes to show that the GA tries to achieve a very high throughput for sequential operations, but at the cost of an increased convergence time.

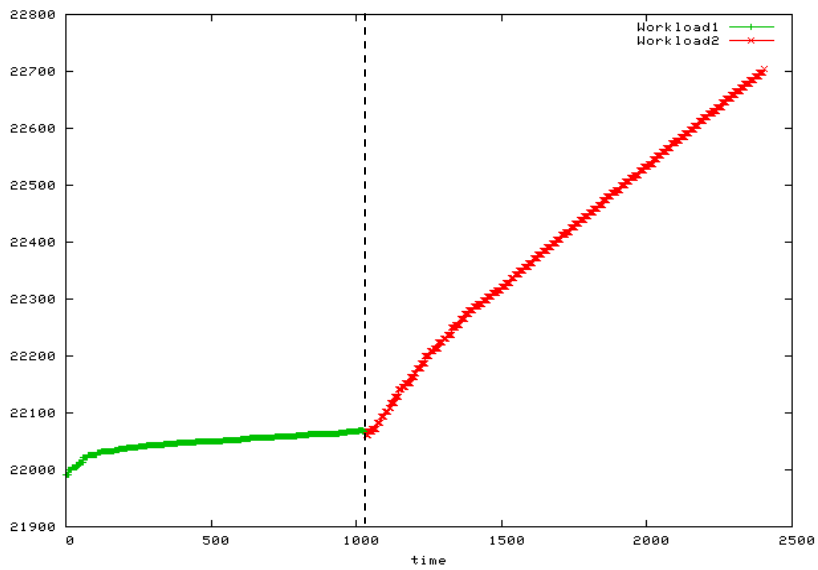


Fig. 6: Meta-data – Write Sequential\*

### 5.3.4 Non Convergence of Write Sequential Workloads

When we ran a Write Sequential workload first followed by any other workload, we failed to observe any convergence. We can see in Fig. 7 that the behavior of Meta-data and Read Random workloads is consistent with what we have seen throughout this experiment. Thus we can conclude that sequential write operation does not converge in acceptable time.

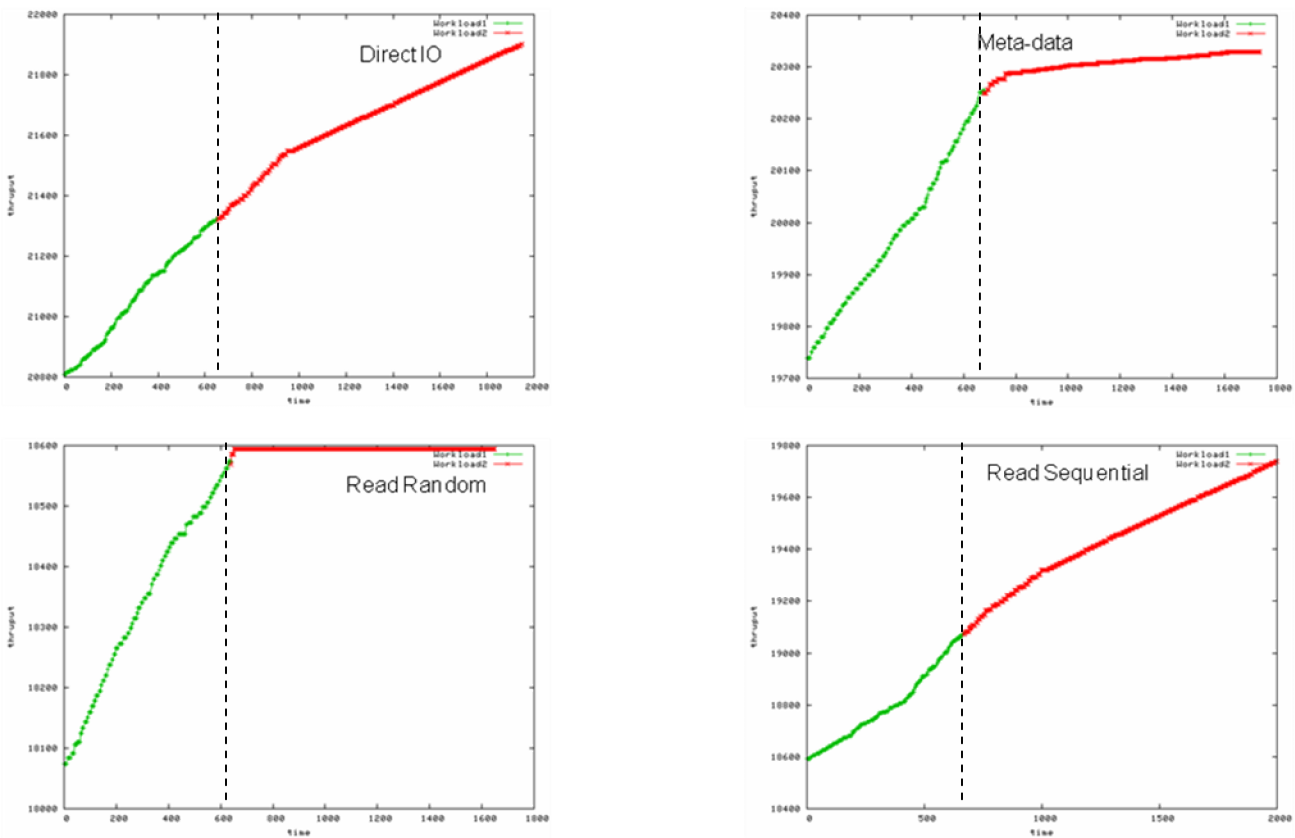


Fig. 7: Write Sequential does not converge\*

## 6 Verification of Hypotheses

After completing our observation of the behavior of the system to different combinations of workloads, we concluded that all workloads have a consistent time to convergence, verifying our Hypothesis I as true. All our experiments consisted of running the first work load for approximately 1000 seconds, followed by the second one for another 1000 seconds. However, since these times were set in the script files of FFBSB, we cannot be certain that the transition occurs at exactly the 1000<sup>th</sup> second. The reason for this is the latency for syncing the pages with the disk, which is taken care of by FFBSB.

We consider 1000 seconds to be an acceptable time for the system for convergence. In spite of this, we found that only two out of five workloads converged within this window of time (Meta-data in 100 seconds and Read Random in 10 seconds). For both we did see a variation in the stabilized throughput value in the tolerance threshold of  $\pm 5\%$ , thus proving our Hypothesis II.

Finally, Hypothesis III was proved false. Within our time frame of 1000 seconds, Direct-I/O, Write Sequential and Read Sequential workloads failed to converge. The probable reason behind this is that the GA tries to achieve maximum throughput through a series of steps in the evolution process. Hence it will eventually attain best possible throughput but the convergence time may be more than the acceptable limit.

## 7 Conclusions

With the expectations we had from the experiment and the observations we obtained it can be concluded that though the approach of self-tuning the I/O scheduler with a Genetic Algorithm can scale dynamically to a wide variety of workloads, it may not be capable of achieving this scalability in acceptable time, particular for workloads that can achieve very high throughput. This shortcoming severely restricts the use of this technique in everyday computing, where good response times are more crucial. Though, it can be argued that its use may be justified in enterprise systems where the total workload throughput has more priority over the time to scale.

In the future, more investigations could be done, to isolate the factors within the Genetic Algorithm that lead to a gradual evolution. If by some means the rate of evolution could be varied depending upon the current general fitness of the gene pool better convergence times could be attained, making its use more reliable and predictable. However, considering the widely accepted implementations of the Genetic Algorithm which posses identical behavior, this may not be trivial or feasible.

## 8 A few obstacles along the way

We faced numerous obstacles in this project; few of them even show stoppers. The most challenging part was getting the kernel with all the alterations to run. The kernel was patched to use the genetic library, the modified I/O scheduler framework and the modified Anticipatory Scheduler. After trying different kernel versions and recompiling for different architectures (Pentium Pro and i486), finally we got kernel version 2.6.16 for i486 up and running. We also had problems in automating the experiment; we kept running out of disk space and incorrect simulation terminations. It took time for us to realize that the data we were looking at was incomplete. Other grave issue was figuring out the appropriate time duration for which a workload was to be run. We did quite some trial and error to arrive on the value of 1000secs which we felt was sufficient enough to draw some conclusions.

## 9 Acknowledgements

We would like to thank Dr. Snodgrass for giving us much flexibility to conduct this experiment and helping us with acquiring a dedicated machine for our experiments. He has provided us with valuable inputs at just the right times. We would also like to thank John Luiten, from the lab staff for helping us out with the hardware setup.

## References

- [1] Jake Moilanen & Peter Williams, *Using genetic algorithms to autonomically tune the kernel*, Proceedings of the Linux Symposium, Volume One, July 2005.
- [2] Jake Moilanen, *I/O Workload Fingerprinting in the genetic library*, Proceedings of the Linux Symposium, Volume Two, July 2006.
- [3] Kernel patches were acquired from Jake Moilanen's homepage <http://kernel.jakem.net/>
- [4] FFSB tool website <http://sourceforge.net/projects/ffsb/>
- [5] GnuPlot website <http://sourceforge.net/projects/gnuplot/>
- [6] Axboe, J., Deadline I/O Scheduler Tunables, SuSE, EDF R&D, 2003.
- [7] Corbet, J., The Continuing Development of I/O Scheduling, <http://lwn.net/Articles/21274>

## Appendix

### A. Non-convergence of Direct-I/O Workload

Fig. 8 shows the system throughput when Direct-I/O is the first workload. The behavior is the same as Sequential Write, i.e. there is no convergence in acceptable time. Read Random and Meta-data on the other hand, converge in accordance with Hypothesis III.

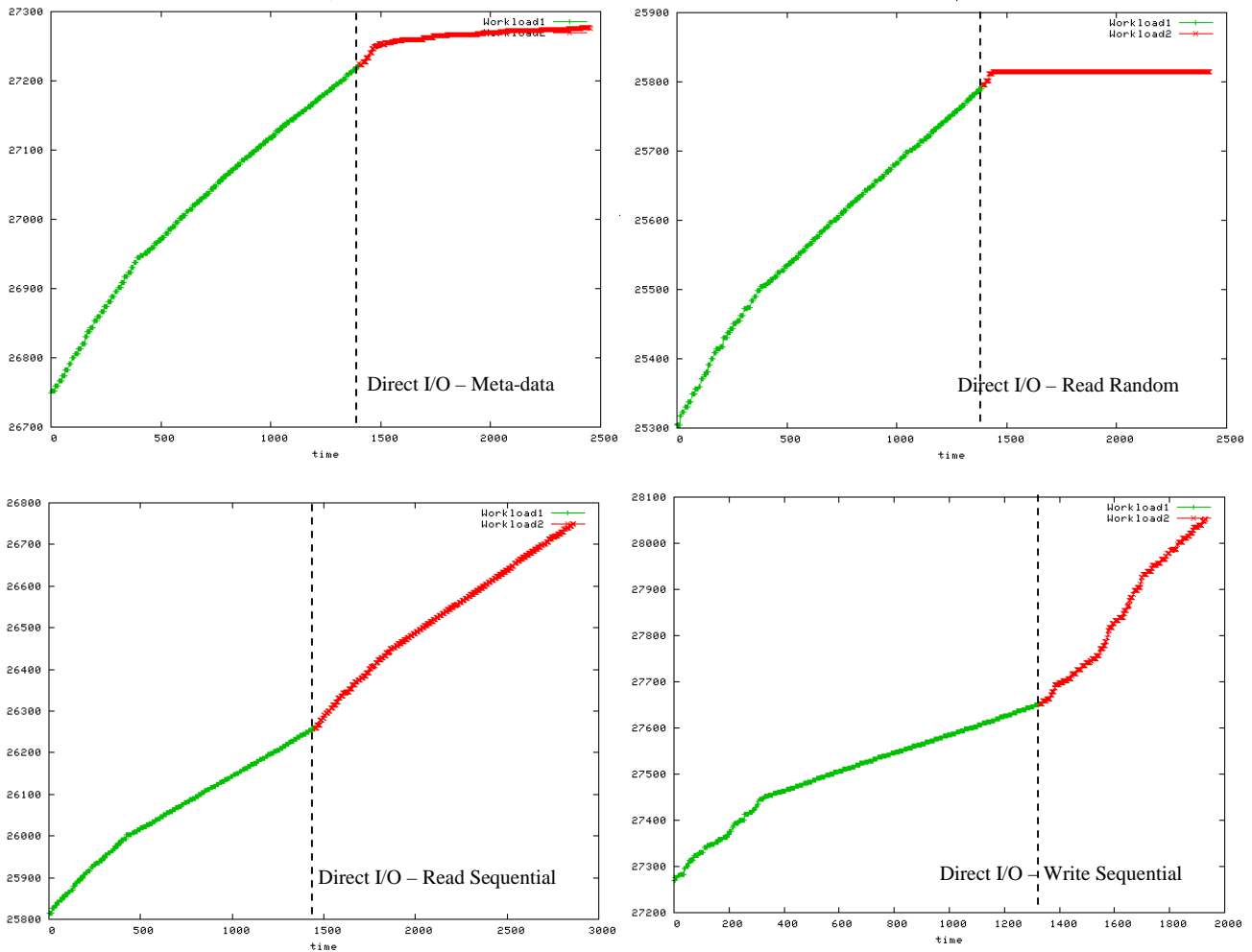


Fig. 8: Non convergence of Direct-I/O\*

### B. Non convergence of Read Sequential Workload

As seen in Fig. 9, Read Sequential is taken as the first workload and we fail to observe convergence to a stable throughput in acceptable time. This violates Hypothesis III.

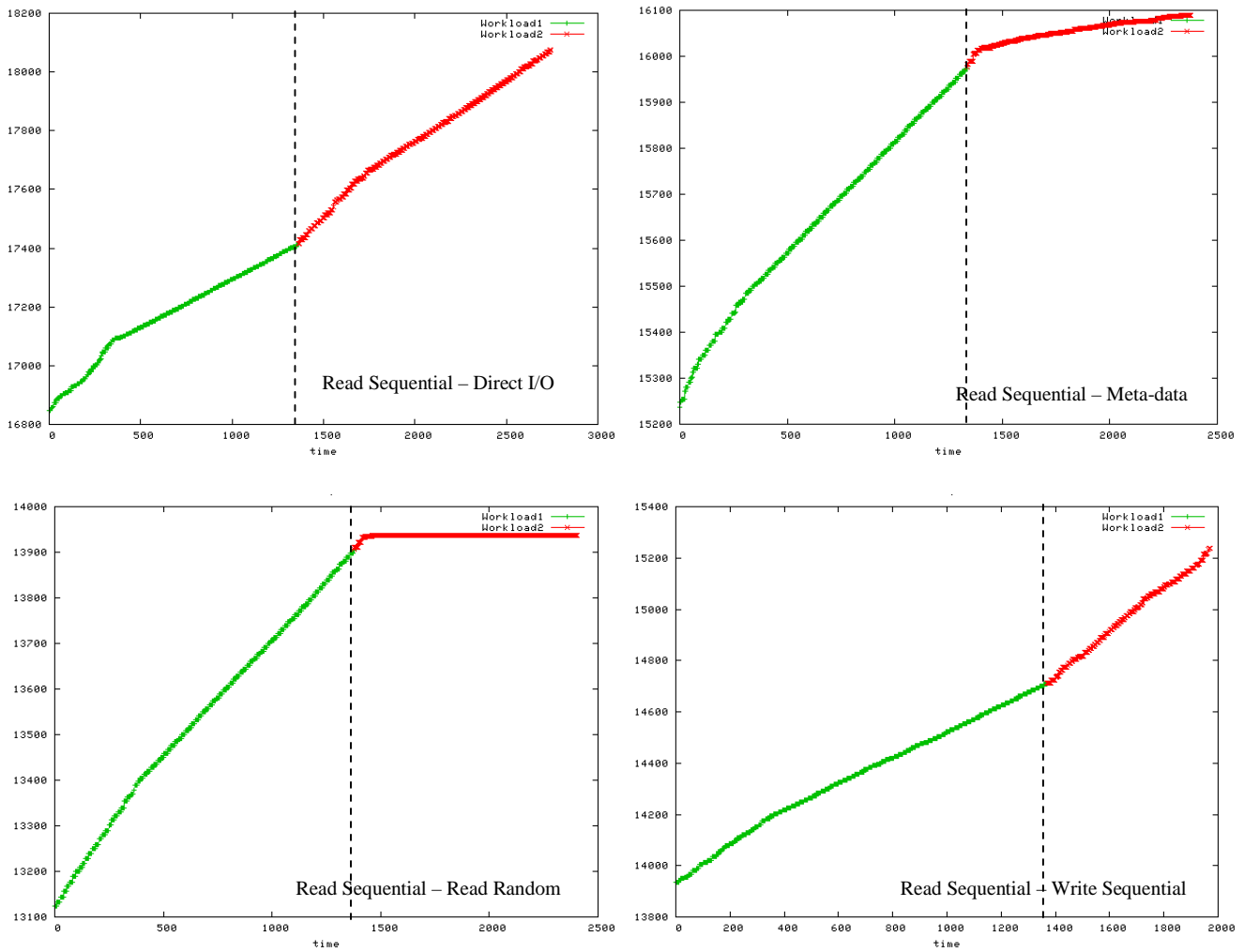


Fig. 9: Non convergence of Read Sequential\*