Using genetic algorithms to autonomically tune the kernel

Jake Moilanen
IBM
moilanen@austin.ibm.com

Peter Williams
Aurema Pty Ltd.
pwil3058@bigpond.net.au

Abstract

One of the next obstacles in autonomic computing is having a system self-tune for any workload. Workloads vary greatly between applications and even during an application's life cycle. It is a daunting task for a system administrator to manually keep up with a constantly changing workload. To remedy this shortcoming, intelligence needs to be put into a system to autonomically handle this process. One method is to take an algorithm commonly used in artificial intelligence and apply it to the Linux® kernel.

This paper covers the use of genetic-algorithms to autonomically tune the kernel through the development of the genetic-library. It will discuss the overall designed of the genetic-library along with the hooked schedulers, current status, and future work. Finally, early performance numbers are covered to give an idea as towards the viability of the concept.

1 What is a Genetic Algorithm

A genetic algorithm, or GA, is a method of searching a large space for a solution to a problem by making a series of educated guesses. This search is done by using the mathematical equivalent of biology's natural selection process. The values of the parameters to the solution are analogous to biology's genes. The genes/values that perform well will survive, while the ones that under perform are pruned from the gene pool. Over time these genes/values evolve towards an optimal solution for the current environment.

1.1 Genetic Algorithm terms

The term *gene* refers to a variable in the problem that is being solved. These variables can be for anything as long as changing their value causes a measurable outcome. A gene is a piece of the solution.

All of the different genes comprise a *child*. Each child normally has different values for their genes, which makes each child unique. These different value and combinations allow some children to perform better than others in a given environment. A single child is a single possible solution to the given problem.

All of the children make up a *population*. A population is a set of solutions to the given problem.

When a set of children are put together, they create a *generation*. A generation is the time

that all children perform before the natural selection process prunes some children. The remaining children become *parents* and create children for the next generation.

The measure of how well a child is performing is a *fitness* measure. This is the numerical value assigned to each child at the end of a generation.

A *phenotype* is the end result of the genes interaction. In biology, an example would be eye color. There are a number of genes that affect eye color, but only one color as an end result. In a genetic algorithms specific genes impact specific fitness outcomes.

Much how evolution works in the wild, a genetic algorithm takes advantage of *mutations* to introduce new genes into the gene pool. This is to combat a limited set of genes that may have worked well in the old environment, but does not have the optimal result in a changing environment. Mutations also aid in premature convergence on less-than-optimal solutions.

2 Genetic-Library

As the name implies, the genetic-library is a library where components in the kernel can plug into to take advantage of a genetic algorithm. The advantage of the genetic-library is that components do not have to create their own method of self-tuning. The genetic-library creates a unified path that is flexible enough to handle almost any tuning that a component has need for.

2.1 Registering

Before the genetic-library is used, components first must register with it. When registering, state must be given to the genetic-library. For instance, the plugins need to give `genetic_ops`, which are implementation specific callback functions that the genetic-library uses. The child lifetime, the number of genes, and the number of children must also be included for each phenotype.

2.2 Genetic library life-cycle

An implementation of a genetic algorithm can vary, but the genetic-library uses the following one:

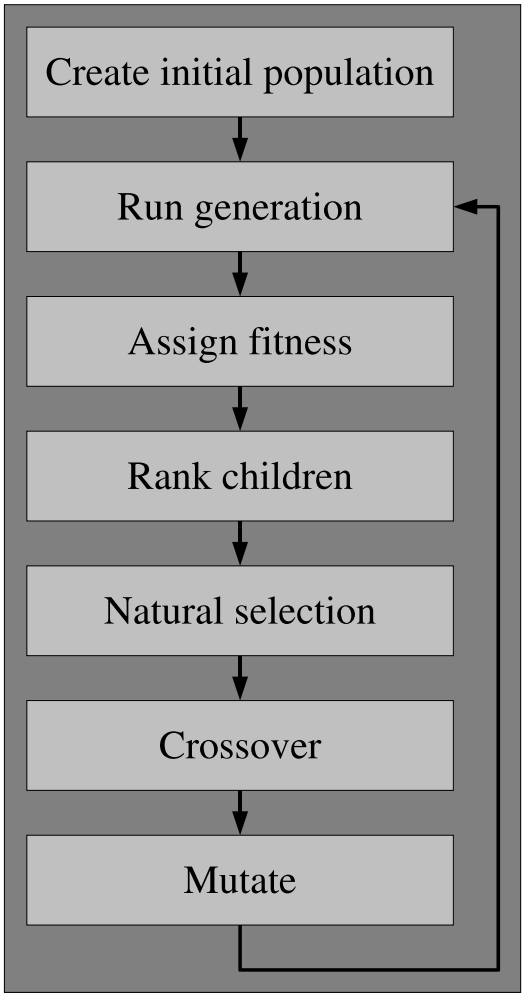

Figure 1: Life Cycle

2.2.1 Create the initial population

The first step in a genetic algorithm is to create an initial population of children with their own set of genes. Usually, the children's genes are given values that spread across the entire search space. This helps facilitate the survival of the fittest.

The genetic-library makes calls into the components through the registered `genetic_ops`. For each phenotype, all of the children are created through the `create_child()` callback. This callback can initialize genes in a number of ways. The most common is by spreading the gene values across the entire search space.

2.2.2 Run generation

In a genetic algorithm, all the children in the current generation are run in serial. The children plug their genes into the system and run for a slice of time. Once all of the children in the generation have completed their run, the generation is over.

In the genetic-library, the first child in every phenotype calls `genetic_run_child()` to kick off the generation. This function sets the genes to be used with the `set_child_genes()` callback. Next, it takes a snapshot of performance counters for the fitness measurement to determine how well this child performed. Finally, a timeout is set that will conclude the child's lifetime. That timer function is used to switch to the next child through `genetic_switch_child()`.

2.2.3 Assign fitness to children

One of the most difficult pieces of a genetic algorithm is assigning an accurate fitness number to a child. This fitness value is used to rank the children against each other. Depending on implementation, the fitness calculation is either done at the completion of a generation, or at the end of a child's lifetime.

For the genetic-library, the fitness calculation is done at the conclusion of a child's lifetime through the `calc_fitness()` callback. This function looks at the snapshot of the performance counters from the beginning of the child's lifetime, and takes the delta of the counters at the end of the lifetime. Since these number are usually normalized between all the children, the delta is usually all that is needed.

There are certain other phenotypes where the fitness calculation must be done at the end of a generation. This is usually when the phenotype contains general tunables that affect other phenotype's outcome. In this case `calc_post_fitness()` is used. This routine normalizes all the different fitness values by taking the average ranking of all the children in the affected phenotypes. The average ranking is used as a fitness measure.

2.2.4 Rank children

Using the fitness value assigned, children are then ranked in order of their performance. Children with well-performing genes get a higher ranking.

In the genetic-library's `genetic_split_performers()`, a bubble sort is used to order the children according to their fitness.

2.2.5 Natural selection operation

The same way Darwin's natural-selection process works in the wild, it works in the genetic algorithm. Those genes that perform well in

the given environment, will survive, and those that perform poorly will not. This enables the strongest genes to carry on to the next generation.

In the genetic-library, the bottom half of the population that under performs is removed. This replacing of part of the population is known as a steady-state type of algorithm. There is also a generational type of algorithm where the entire population is replaced. For instance, in implementations that make use of a roulette wheel algorithm, the whole population is replaced, but the children that have higher fitness have a proportionally higher chance of their genes being passed on.

2.2.6 Crossover Operation

This operation is the main distinguishing factor between a genetic algorithm and other optimization algorithms. The children that survived the natural selection process now become parents. The parents mate and create new children to repopulate the depleted population.

There are a number of methods for crossover, but the most common one in the genetic-library is similar to the blending method. For all of the phenotypes that have genes to combine (some phenotypes are just placeholders for fitness measures and their child's rankings are used to determine fitness for another phenotype), each gene receives X% of parent A's gene value and add in 100-X% of parent B's gene value. X is a random percentage between 0 and 100. The end result is that the child has a gene value that is somewhere randomly in the middle of parent A's, and parent B's genes.

2.2.7 Mutation Operation

To combat premature convergence on a solution, a small number of mutations are introduced into the population. These mutations also aid in changing environments where the current gene pool performs less-than-optimal.

After the new population is created, genes are picked randomly and randomly modified. These mutations keep the population diverse. Staying diverse makes the algorithm perform a global search.

In the genetic-library, mutation is done on some percentage of all the genes. Mutations are randomly done on both new children, and parents. Once the individual from the population is picked, a gene is randomly selected to be mutated. The gene either has a new value picked at random, or else is iteratively modified by having a random percentage increase or decrease in the gene's value.

On a system, workloads are always changing. So the population needs to always be changing to cover the current solution search space. To counteract this moving target, the genetic-library varies the rate of mutation depending on how well the current population is performing. If the average fitness for the population decreases past some threshold, then it appears as if the workload is changing and the current population is not performing as well. To counteract this new problem/workload, the mutation rate is increased to widen the search space and find the new optimal solution. There is a limit on the mutation rate, so not to have the algorithm go spiraling out of control with mutations bringing the population further and further away from the solution. Conversely, if the fitness is increasing, then it appears that the population is converging on an optimal solution, so the mutation rate decreases to not introduce excessive bad genes.

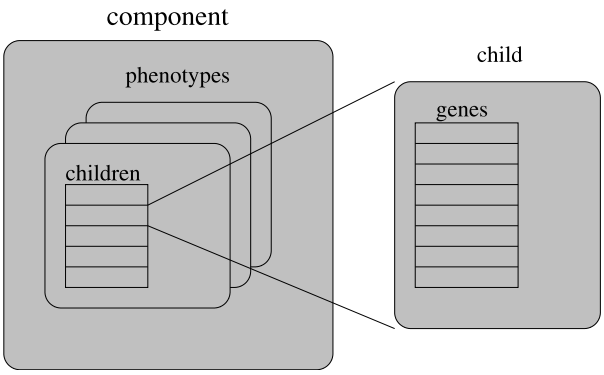

Figure 2: Structure Layout

2.3 Framework

The `struct genetic_s` is the main struct that contains the state for each component plugged into the genetic-lib.

This general structure contains all of the phenotypes in the `struct phenotype_s`. A phenotype is created for each specific measurable outcome.

Within each phenotype, is an array of `struct genetic_child_s` or children. Each child will contain an array of genes that are specific to that phenotype. Since some genes may affect multiple fitness measures, those genes are usually put into a phenotype that encapsulates other phenotypes. This will be discussed further in the next section.

Each gene has a `struct gene_param` associated with it. In this structure, the gene's properties are given. The minimum and the maximum value for a gene, along with its default value are given. If a gene has a specific function to mutate it, that can also be provided.

2.4 Phenotypes

Some other genetic algorithms refer to phenotypes as something comparable to what the genetic-library calls a child. However in the genetic library context, it refers to a population of children that affect a specific fitness measure. Phenotypes were introduced into the genetic library to increase granularity of what could be tuned in a component. Before phenotypes there was one fitness routine per component. This fitness function could look at multiple performance metrics, but all the genes for the component would be affected regardless if they had nothing to do with some of the performance metrics. For example, some of the genes that impact real-time process scheduling were being judged by fitness metrics that looked at throughput. With the introduction of phenotypes, the fitness measure of real-time performance would only affect the genes that impacted real-time.

The next problem that came about with phenotypes was what to do with the genes that affect a number of fitness metrics. For example, time-slice affects fitness measures like number of context switches, and total delay. The solution lies with adding a hierarchy of phenotypes that affect other phenotypes. This is done by assigning a unique ID's, or *uid*, to each phenotype. A *uid* is really a bitmask of phenotypes that affect it.

3 Hooked components

The genetic-library can be hooked into pretty much any component that can be tuned. For the initial implementation, the Zaphod CPU scheduler and the Anticipatory I/O scheduler were picked.

The Zaphod CPU scheduler was attractive to use because of its heavy integration with sched stats. Having extensive scheduler statistics made it much easier to create good fitness routines.

The Anticipatory I/O scheduler was also desirable because modifying the tunables could affect the schedulers performance greatly.

3.1 Zaphod CPU scheduler

The Zaphod CPU scheduler emerged from the CPU scheduler evaluation work. It is a single priority array O(1) with interactive response bonuses, throughput bonuses, soft and hard CPU rate caps and a choice of priority based or entitlement based interpretation of "nice."

3.1.1 Configurable Parameters

The behavior of this scheduler is controlled by a number of parameters and since there was no *a priori* best value for these parameters they were designed to be runtime configurable (within limits) so that experiments could be conducted to determine their best values.

`time_slice` One of the principal advantages of using a single priority array is that a task's time slice is no longer tied up controlling its movement between the active and expired arrays. Therefore all tasks are given a new time slice every time they wake and when they finish their current time slice. This parameter determines the size of the time slice given to `SCHED_NORMAL` tasks.

`sched_rr_time_slice` This parameter determines the size of the time slice given to `SCHED_RR` tasks.

`base_prom_interval` The single priority array introduces the possibility of starvation and to handle this Zaphod includes an O(1) promotion mechanism. When the number of runnable tasks on a run queue is greater than 1, Zaphod periodically moves all runnable `SCHED_NORMAL` tasks with a `prio` value greater than `MAX_RT_PRIO` towards the head of the queue. This variable controls the interval between promotions and its ratio to the value of `time_slice` can be thought of as controlling the severity of "nice."

`bgnd_time_slice_multiplier` Tasks with a soft CPU rate cap are essentially background tasks and generally only run when there are no other runnable tasks on their run queue. These tasks are usually batch tasks that benefit from longer time slices and Zaphod has a mechanism to give them time slices that are an integer multiple of `time_slice` and this variable determines that multiple.

`max_ia_bonus` In order to enhance interactive responsiveness, Zaphod attempts to identify interactive tasks and give them priority bonuses. This attribute determines the largest bonus that Zaphod awards. Setting this attribute to zero is recommended for servers.

`initial_ia_bonus` When interactive tasks are forked on very busy systems it can take some time for Zaphod to recognize them as interactive. Giving all tasks a small bonus when they fork can help speed up this process and this attribute determines the initial interactive bonus that all tasks receive.

`ia_threshold` When Zaphod needs to determine a dynamic priority (i.e., a `prio` value) it calculates the recent average *sleepiness* (i.e., the ratio of the time spends sleeping to the sum of the time spent on the CPU or sleeping) and if this is greater than the value of `ia_threshold` it increases the proportion (`interactive_`

bonus) of max_ia_bonus that it will award this task asymptotically towards 1.

cpu_hog_threshold At the same time it calculates the tasks *CPU usage rate* (i.e. the ratio of the time spent on the CPU to the sum of the time spent on a run queue waiting for CPU access or sleeping) and if this is greater than the value of cpu_hog_threshold it decreases the task's interactive_bonus asymptotically towards zero. From this it can be seen that the size of the interactive bonus is relatively permanent.

max_tpt_bonus Zaphod also has a mechanism for awarding throughput bonuses whose purpose is (as the name implies) to increase system throughput by reducing the total amount of time that tasks spend on run queues waiting for CPU access. These bonuses are ephemeral and once granted are only in force for one task scheduling cycle. The size of the throughput bonus awarded to a task each scheduling cycle is decided by comparing the recent average *delay time* that the task has been suffering to the expected delay time based on how busy the system is, the task's usage patterns and static priority. It will be a proportion of the value of max_tpt_bonus. This bonus is generally only effective when the system is less than fully loaded as once the system is fully loaded it is not possible to reduce the total delay time of the tasks on the system.

current_zaphod_mode As previously mentioned, Zaphod offers the choice of a priority based or an entitlement based interpretation of "nice." This attribute determines which of those interpretations is in use.

3.1.2 Scheduling Statistics

As can be seen from the above description of Zaphod's control attributes, Zaphod needs data on the amount of time tasks spend on run queues waiting for CPU access in order to compute task bonuses. The kernel does not currently provide this data so Zaphod maintains its own scheduling statistics (in nanoseconds) for both tasks and run queues. The scheduling statistics of interest to this paper are the run queue statistics as they are an indication of the overall system performance. The following statistics are kept for each run queue in addition to those already provided in the vanilla kernel:

total_idle The total amount of time (since boot) that the CPU associated with the run queue was idle. This is actually derived from the total CPU time for the run queue's idle thread.

total_busy The total amount of time (since boot) that the CPU associated with the run queue was busy. This is actually derived from the total time that the run queue's idle thread spent off the CPU.

total_delay The total amount of time (since boot) that tasks have spent on this run queue waiting for access to its CPU.

total_rt_delay The total amount of time (since boot) that real time tasks have spent on this run queue waiting for access to its CPU.

total_intr_delay The total amount of time (since boot) that tasks awoken to service an interrupt have spent on this run queue waiting for access to its CPU.

total_rt_intr_delay The total amount of time (since boot) that real time tasks awoken to service an interrupt have spent

on this run queue waiting for access to its CPU.

`total_fork_delay` The total amount of time (since boot) that tasks have spent on this run queue waiting for access to its CPU immediately after forking.

`total_sinbin` The total amount of time (since boot) that tasks associated with this run queue have spent cooling their heels in the *sin bin* as a consequence of exceeding their CPU usage rate hard cap.

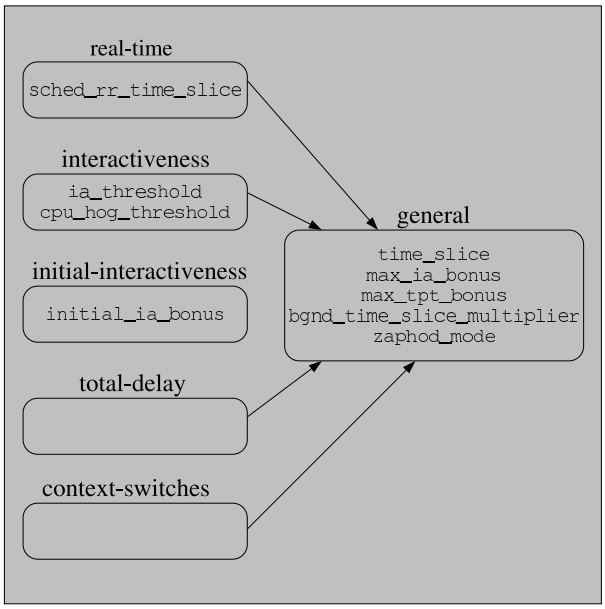

Figure 3: Zaphod Phenotypes

3.1.3 Phenotypes

In Figure 3, there are six phenotypes listed along with the genes that exist within them. All of the phenotypes have their own fitness measures. For example, `real-time`'s fitness measures takes the delta of `total_rt_delay` for each child. The fitness measure not only affects `sched_rr_time_slice`, but also affects all of the genes in the `general` phenotype.

Phenotypes might not always have genes in their children. This is done in when the phenotypes are just being used for their fitness measures. The children that perform well are ranked accordingly. The `general` phenotype looks at the average ranking of the children of all the phenotypes under it.

The `general` phenotype also has a weights associated with each of its subsidiary phenotypes. The phenotypes that have a greater impact on the general phenotype gets higher weights associated with them. For instance, the `total-delay` phenotype is three times more important than the `real-time` phenotype.

To actually calculate the fitness for the `general` phenotype's children, the first child looks at what place it ranked in each phenotype, between `1-NUM_CHILDREN`, and then multiply its place by the weight associated with that phenotype to get a final fitness number for that child. A quick example would be if a child was ranked as the worst performing in `real-time`, it would receive 1 point (top rank gets the most points, lowest gets the least) times the `real-time` weight, which is 1. However, in the `total-delay` phenotype, it was the second best performer, and receives `NUM_CHILDREN-1` points. Assume that there are 8 children. The child would receive 7 points (ranked second best), times `total-delay`'s weight, which is 3.

The final fitness number would be:

```
real-time:              1 * 1
total-delay:          + 7 * 3
                        ------
final fitness:             22
```

3.2 Anticipatory IO scheduler

The anticipatory I/O scheduler, or AS attempts to reduce the disk seek time by using a heuris-

tic to anticipate getting another read request in close proximity. This is done by delaying pending I/O with the expectation that the delay of servicing a request will be made up for by reducing the number of times the disk has to seek.

The anticipatory I/O scheduler was developed on one large assumption, that there was only one outstanding I/O on the bus, and only one head to seek. In other words, it assumed that the disk was an IDE drive. This works very well on most desktops, however, in most server environments, they have SCSI disks, which can handle many outstanding I/Os, and many times these disks are setup in a RAID environment and have many disk heads.

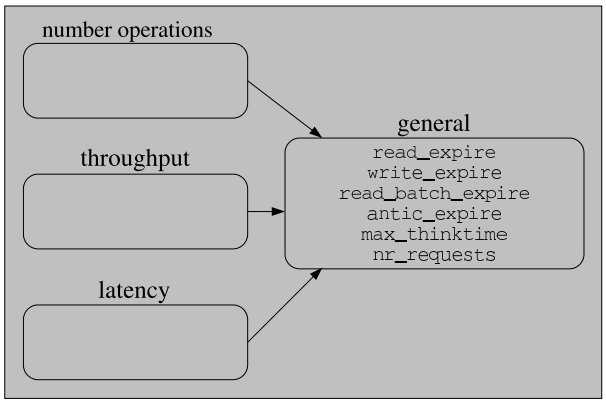

Figure 4: Anticipatory I/O Scheduler Phenotypes

3.2.1 Phenotypes

Figure 4 shows how three of the four phenotypes are just placeholders for fitness measures. Only the `general` phenotype contains genes.

The three phenotypes that are just fitness measures are in place to make sure all workloads are considered, and not favor one type of workload over another. They are agnostic towards the type of I/O, whether it is a read or write.

The `num_ops` phenotype only exists for fitness measurements. The fitness routine looks at the delta of number of I/O operations completed during a child's lifetime. This fitness routine helps balance out the idea of pure throughput. This gives a small fitness bonus to a large number small I/O's.

In the `throughput` phenotype, the fitness simply looks at the number of sectors read or written in during a child's lifetime. This phenotype makes sure data is actually moving, and not just servicing a lot a small requests.

The `latency` phenotype measures the time all requests sit in the queue. This should help combat I/O starvation.

4 Performance numbers

The main goal of the genetic-library is to increase performance through autonomically tuning components of the kernel. The performance gain offered by the genetic-library must outweigh the cost of adding more code into the kernel. While there is no hard-and-fast rule towards what percentage increase is worth adding X number lines of code, gains should be measurable.

In the performance evaluation, an OpenPower 710 system, with 2 CPUs, and 1.848 gigabytes of RAM was used. The benchmarks were conducted on a SLES 9 SP1 base install with a 2.6.11 kernel. More system details can be found in Appendix A.

The base benchmarks were conducted on a stock 2.6.11 kernel, with the PPC64 default config. On the benchmarking of each component utilizing the genetic-library, only the genetic-library patches for the component being exercised at that time were in the kernel.

4.1 Zaphod CPU scheduler

To benchmark the Zaphod CPU Scheduler, SPECjbb2000® was used. This benchmark is a good indicator of scheduler performance. Due to these runs being unofficial, their formal numbers cannot be published. However, a percentage difference should be sufficient for what this paper intends to look at.

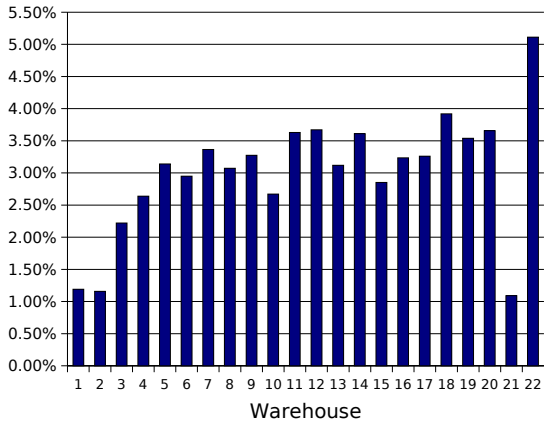

Figure 5: SPECjbb results—GA plugin to Zaphod

The performance improvement ranged from 1.09% to 5.11% in all of the warehouses tested. There is a trend towards a larger improvement as the number of warehouses increase. This indicates that the genetic-library helped Zaphod scale as the load increased on the system. The warehouses averaged an improvement of 3.04%, however the SPECjbb peak performance throughput only showed a 1.52% improvement. The peak performance throughput difference may not be valid due to the performance peaking in different warehouses. That difference makes the two throughput numbers unable to be measured directly against one another.

4.2 Anticipatory I/O scheduler

The Anticipatory I/O scheduler is tuned to do sequential reads very well[1]; however, it has problems with the other types of I/O operations such as sequential writes and random reads. These other I/O types of operations can do better when the AS is tuned for them. If the genetic-library did its tuning correctly, there should be a performance increase across all types of workloads.

To generate these workloads, the flexible file system benchmark, or FFSB was used. The FFSB is a versatile benchmark that gives the ability to simulate any type of workload[2].

For the benchmarking the AS genetic-library plugin, FFSB sequentially went through a series of workload simulations and returned the number of transactions-per-second and the throughput. This experiment was conducted on a single disk ext3 file system.

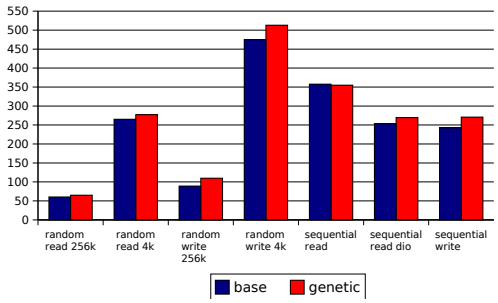

Figure 6: FFSB transactions per second—Anticipatory Plugin

With the exception of one workload, there were performance improvements across the board. The largest increase was in the 256K random write workload. The genetic-library version had a 23.22% improvement over a stock AS. The average improvement of all tested workloads was 8.72%.

The one workload where the genetic-library degraded performance was the sequential read workload at –0.74%. This is not surprising because the AS is optimized specifically for this workload, and the genetic-library's tunings might not get any better than the default settings. The performance loss can be attributed

to the genetic library's attempts at finding better tunings. When the new tuning solution is attempted it will probably be less-than-optimal.

5 Conclusion & Future work

5.1 Kernel Inclusion viability

At the present time, the Anticipatory I/O Scheduler sees large enough improvements, that a strong argument can be made to add the extra complexity into the kernel. The GA plugin to Zaphod also sees substantial gains in performance, especially when the system is under a high load. If only throughput was a concern on the CPU scheduler, then inclusion into the kernel should be considered. However, there are number of other factors that must be looked at. The biggest one is, how well the system also maintains interactiveness, which is very subjective.

In the near-term, the GA plugin to Zaphod should only be used in a server environment. This is because a desktop environment is particularly malicious for the genetic-library. There are numerous CPU usage spikes that can skew performance results. On top of the CPU usage, there are the interactiveness concerns. A user expects to see instant reaction in their interactive applications. If the genetic library goes off on a tangent to try finding a new optimal tuning, a time-slice may go much longer than is acceptable by a desktop user. New features are planned for the genetic-library to help converge quicker on changing workloads.

5.2 Future work

There are other areas of the kernel that are being investigated for their viability of being tuned with the genetic-library. Some of them include scheduler domain reconfiguration, full I/O scheduler swapping, plugsched CPU scheduler swapping, packet scheduling, and SMT scheduling.

The next major feature of the genetic-library will be workload fingerprinting. The idea is to bring back the top-performing genes for certain workloads. By being able to identify a particular workload, the history of optimal tunings for that workload can be saved. These optimal genes will be reintroduced into the population when the current workload matches a fingerprinted workload. This will enable faster convergence when workloads change.

5.3 Conclusion

The genetic-library has the ability to put intelligence into the kernel, and gracefully handle even the most malevolent of workloads. Hopefully, it will pave the way towards a fully autonomic system and the elimination of the system admin dependency.

Legal Statement

References

[1] Pratt, S., Heger, D., *Workload Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers*, 2004 Linux Symposium

[2] `http://sourceforge.net/projects/ffsb/`

Appendix A. Performance System

IBM OpenPower 710 System
2-way 1.66 Ghz Power5 Processors
1.848 GB of memory
2 15,000 RPM SCSI drives
SLES 9 SP1
2.6.11 Kernel

Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*