Radboud University Nijmegen

Implementing a Process Scheduler Using Neural Network Technology

Master's thesis

Author	:	Peter Bex (peter.bex@student.ru.nl)
Student nr.	:	0115460
AI Supervisor	:	Louis Vuurpijl (vuurpijl@nici.ru.nl)
External supervisors	:	Erik Poll (erikpoll@cs.ru.nl)
		Theo Schouten (ths@cs.ru.nl)

Abstract

This Master's thesis describes the design, implementation and evaluation of a neural network in an operating system kernel to classify processes based on their behaviour and assign scheduling priorities accordingly. The design of the system allows easy addition of features, making it an excellent platform for rapid prototyping of new scheduler features without requiring a rewrite or substantial modification of an existing scheduler.

Contents

1 Introduction **3**
- 1.1 Motivation . . . 4
  - 1.1.1 Increasing performance . . . 4
  - 1.1.2 Increasing user-friendliness . . . 4
  - 1.1.3 Maximizing customizability . . . 5
- 1.2 Outline of thesis . . . 5

2 Process scheduling **6**
- 2.1 Introduction . . . 6
- 2.2 Job-shop scheduling . . . 6
- 2.3 Preemptive scheduling . . . 7
- 2.4 Multithreading . . . 8
- 2.5 Multiple CPUs . . . 9
- 2.6 Existing research . . . 9

3 AI techniques **11**
- 3.1 Genetic algorithms . . . 11
- 3.2 Decision tree learning . . . 12
- 3.3 Rule-based knowledge system . . . 12
- 3.4 Bayesian networks . . . 13
- 3.5 Neural Networks . . . 13
- 3.6 Existing research . . . 14

4 System overview **15**
- 4.1 Requirements of the network . . . 15
  - 4.1.1 The problem of representing time . . . 16
  - 4.1.2 The solution . . . 16
- 4.2 Infrastructural changes . . . 18
- 4.3 Pre-existing priorities . . . 19

5 Implementation details **20**
- 5.1 The system . . . 20
- 5.2 Typical usage . . . 20
- 5.3 Shared code . . . 21
- 5.4 Kernel level implementation . . . 22
  - 5.4.1 Pseudo-devices . . . 23
  - 5.4.2 Scheduler and feature collection . . . 23
- 5.5 User level implementation . . . 24

5.5.1 Tools for interacting with the kernel . . . 24
5.5.2 Training tools . . . 24
5.5.3 Miscellaneous tools . . . 24
5.6 Implementation issues . . . 25
5.6.1 Floating point . . . 25
5.6.2 Range and precision problems . . . 26
5.6.3 No audio mixing . . . 26
5.6.4 The X Window System . . . 27

6 Results 28
6.1 Features . . . 28
6.2 Network accuracy . . . 29
6.3 Influence of the memory layer . . . 32
6.4 Overhead . . . 32
6.4.1 Feature gathering overhead . . . 33
6.5 Perceived speed . . . 34
6.6 Assessment of features . . . 37
6.6.1 Sockets . . . 37
6.6.2 Audio . . . 39
6.6.3 Terminal . . . 39
6.6.4 Filesystem . . . 41
6.6.5 Application classification . . . 45

7 Conclusion and future work 47
7.1 Personal notes . . . 49

A Correction of the Turing-completeness proof for NNs 51
A.1 Background . . . 51
A.2 Proof . . . 53

Chapter 1

Introduction

Contemporary OSes (operating systems) use exceedingly complex process scheduling strategies (see chapter 2 for more information about OS scheduling) to handle scheduling on multiple CPUs, but when it comes to uniprocessor scheduling there has not been much new development. Current scheduling algorithms are not very suitable for modern multimedia applications ([33]) because these algorithms do not take into account the multimedia aspects of these applications. Traditional schedulers were designed for large multi-user systems and aimed at improving total throughput of the system, being fair to all users. Multimedia applications represent the opposite of this model; they need all the system's resources to generate smooth video or audio playback. Only when the system is fast enough to deliver all the resources the application needs can other applications be scheduled without resulting in a stutter in the multimedia application.

Application scheduling priorities are calculated based on simple heuristics that can improve throughput. For example, a priority could be boosted when a process is newly entered into the ready queue or lowered when a process is forcedly removed from the CPU because it exceeded its allotted time quantum. However, the base priority of a program must still be defined by the user if it should be something else than the default. It is tedious to do this for every program, and casual users often do not even know this functionality exists. Ideally, the computer should determine the priority automatically. This is a very hard problem to solve in a generic way, because different users have different needs. On a web server one would assign the highest priority to batch processing-like functionality (disk/database access and network usage), while on a personal multimedia computer, batch processing would have the lowest priority and interactive processes would have a higher priority (audio and video output mostly, but also CPU usage for decoding of audio, for example). This problem could be solved by creating operating systems (or schedulers) tuned for a specific usage scenario, but that would sacrifice generality. Most, if not all, of today's operating systems are designed to be general-purpose, thus a generic solution would be preferred.

This master thesis proposes to use the Artificial Intelligence technique of *neural networks* to create a solution to the problem described above. Thus, this thesis strives to find an answer to the following question: *How could neural networks be used to schedule processes in an operating system?* To answer this question, a *proof-of-concept* implementation was created by patching an existing

operating system kernel's scheduler such that it delegated its scheduling decisions to a neural network. One of the advantages of using neural networks is that they can be trained for specific use cases by experts and advanced users and then re-used by less technically inclined users without requiring any knowledge about neural networks or schedulers. The particular neural network used here is a backpropagation network with an added memory layer, but in future research different network topologies may be explored. Features that are used for input to the neural network ideally include everything that a human would use to determine whether a process is "multimedia", but the features used in this implementation are based on audio, disk, terminal and socket I/O. Other typical "multimedia features" would be harder to obtain under Unix-like operating systems, as discussed in section 5.6.

1.1 Motivation

1.1.1 Increasing performance

Operating systems still perform rather poorly when running multimedia applications. This is partly canceled by the ever increasing speed of computer systems, but applications also continue to demand more from computer systems. In fact, OS performance does *not* increase at the same speed as hardware does, as indicated by [35] and [41], for example.

The fact that there are specialized workshops for improving multimedia support in OSes, like NOSSDAV[1] and ESTIMedia[2], indicates that there is certainly a demand for better handling of multimedia in operating systems.

1.1.2 Increasing user-friendliness

A large percentage of computer users are nontechnical and simply want to get their work done. These users do not have the time or inclination to find out how to tweak their computer for optimal performance. For this reason, operating systems should be designed in such a way that at least their default settings will provide optimal performance to this kind of user.

On many operating systems, all processes have the same default base priority. This priority does not normally change unless the user explicitly indicates it should be different from the default. Because typically only so-called "power-users" know about the existence of this functionality, the average user's system will not have the perceived performance it could have if the process priorities were optimally assigned.

A neural network scheduler can take care of assigning priority to processes for the user in the background, without requiring any interaction. This will create the opportunity for average computer users to enjoy the same perceived performance from their system as power users do. The only thing that is required is that a technical user at one time creates and tunes a neural network for the usage pattern of the particular type of user, be it desktop user, server user or anything in between. After that, the user can simply select the network labeled,

[1] International Workshop on Network and Operating System Support for Digital Audio and Video, see http://www.nossdav.org.

[2] Workshop on Embedded Systems for Real-Time Multimedia, see http://www.codes-isss.org/.

for example, “multimedia desktop” and run the system in the configuration that is optimal for them.

1.1.3 Maximizing customizability

Each user has their own individual needs. Some users might only use an operating system to run a spreadsheet program, some users might use it for playing computer games and even others might run a personal web server. The process profiles of these uses are very different. In order to cater to all these users’ needs, an operating system needs to be versatile and customizable. Focusing purely on the requirements of multimedia applications would decrease the generality of the system, since this would necessarily mean that the system would be less than optimal in non-multimedia environments where throughput is more important than interactive responsiveness, for example web servers or batch computation systems. In order to maintain generality, the system needs to be customizable for all use scenarios. Neural networks are a useful tool to obtain this generality, because one network can be trained for different situations. This means an operating system could contain a generic “neural network driver” which can load any network that was specially designed and trained for the expected situations the machine will be used in. A proof-of-concept implementation of such a driver is presented in this thesis. However, this thesis will mostly focus on the specific aspect of scheduling multimedia applications.

1.2 Outline of thesis

This thesis has been broken up in such a way that the background material is discussed first, so the reader has a foundation to help interpret the implementation and testing results.

In chapter 2 the problem of application scheduling in an operating system is explored. Next, in chapter 3, several possible techniques from the field of Artificial Intelligence are discussed that could help us solve the application scheduling problem. These two chapters end in a short section where we investigate what existing research we can use to base our neural network scheduler design upon.

After the background material, the proof of concept implementation that was created to explore this problem is introduced. A description of the general network architecture can be found in chapter 4. How the network can be trained and tested offline and how features are gathered from running processes by the kernel, as well as some issues that had an influence on the implementation are described in detail in chapter 5. The preliminary features used to test the effect of using this type of scheduler are described in chapter 6, along with an assessment of their usefulness and the results of benchmarking the original scheduler and the new scheduler. Finally, we come to a conclusion in chapter 7. The appendix describes a minor correction in the proof of [18], such that a study that is useful but known to be presented slightly incorrectly would not have to be removed from the bibliography.

Chapter 2

Process scheduling

2.1 Introduction

Today's Operating Systems can handle large numbers of programs, all appearing to be running concurrently from a user's point of view. In actuality, what happens on a single CPU is that processes are run one after another for a small period of time in quick succession. This produces the illusion that they all run at the same time. This is called *multitasking*.

Of these processes, several may simply be waiting for some kind of hardware event to occur, so they will not all have to or be able to run at the same time. The problem of process scheduling arises when there are more processes ready to run (on the *ready queue*) than there are available processors in the system. That is, which process will be next to run on the CPU, given several to choose from? A short introduction to this problem will be given in this chapter. For more detailed information, the reader is referred to one of the many high-quality textbooks on operating systems that are available (e.g. [46], [15]).

2.2 Job-shop scheduling

Job-shop scheduling, also called *linear workshop process scheduling* ([13], [27]) stems from industrial environments. There, the 'processes' are actually jobs to *be processed* and the machines are the different stages the jobs will go through. The scheduling problem here is how to arrange the job processing order of machines so the jobs have to wait in line for a machine the shortest possible amount of time.

For example, if the job is a car to be produced, the different machines that would process it are welding and bolting machines specialized to fit particular components the car is made out of. Tires can only be bolted onto an axle, which has to be welded to the chassis by another machine. The machine which bolts the tires cannot perform its job unless the car has already been processed by the machine which welds the axle to the chassis. Dependencies like this impose a partial order in which the jobs must be processed by the different machines. The goal here is to maximize throughput (thus, minimizing total processing time) by reducing the time each machine has to wait for other machines to finish. This can be achieved by having as many machines busy at the same

time as possible by putting them to work on mutually independent parts of the production process and only later putting these bigger parts together.

This does have some resemblance to the process scheduling problem in operating systems research, and some ideas have been taken from it. For the problem at hand it is unsuitable for the simple reason that the algorithm requires knowledge of the total job completion time in advance. One cannot know the total job completion time for any application for which the lifetime depends on the input. One would have to have solved the famous Halting Problem ([10]) in order to be able to do so because one would have to calculate the time it takes for an application to complete given a specific input. Jobs in a job-shop processing system are always completely executed before taking on another job (batch processing). This makes these algorithms impractical for OS process scheduling purposes because this means it is impossible to dynamically add a new process at runtime. In short, the job-shop scheduling is too *static* for process scheduling in an operating system.

2.3 Preemptive scheduling

In *preemptive scheduling*, a process is given a time *quantum* during which it is allowed to run. If the process exceeds that quantum, the kernel reclaims the CPU and the next process on the ready queue is allowed to run. This is generally implemented by a clock interrupt which is raised after a set period of time, giving control back to the kernel so it can save and modify the CPU state in such a way that another process is set up to resume processing.

A process can also voluntarily *yield* (give up, free) the CPU by performing an I/O operation. The process will then be removed from the CPU until that I/O is completed. There are a number of other operations that will also put a process to sleep until some event occurs. In *nonpreemptive* or *cooperative* scheduling a process is expected to *always* voluntarily relinquish the CPU because the kernel is not equipped to forcedly reclaim it from a process. In this paper, only preemptive scheduling is considered. This is not particularly restrictive because modern operating systems rarely use cooperative scheduling.

When there is more than one process on the ready queue, the question "which process will the system select to run on the CPU?" arises. Most strategies are very simple. For example, in *round-robin* the head of the ready queue is always selected to run next. Processes which go into the ready state will be inserted at the tail of the ready queue. This is by far the most popular scheduling strategy. For an overview of strategies, see [9].

This is not necessarily the best strategy. It may be important to schedule a process which is doing a lot of I/O soon after it becomes ready. This will maximize throughput if the process is expected to go to sleep again after requesting more I/O. There are also classes of processes where a short response time is important. Examples include multimedia processes where even a short delay in audio or video output is very noticeable to the user. On a desktop computer used for multimedia these processes should preferably be scheduled more often than processes where a delay is less noticeable.

For these reasons, most operating systems introduce the concept of *scheduling priority*. From all processes in the ready queue, the one with the highest priority will be scheduled to run on the CPU. Care must be taken to avoid

starvation, which is said to occur when a process is never selected to run. More advanced scheduling techniques involve varying the quantum as well.

Usually the goal of process scheduling is maximizing throughput, minimizing resource usage and response times. The emphasis may vary from system to system, depending on the situations it is typically used in.

2.4 Multithreading

In the preceding discussion, concurrent threads of control within one process were not considered. Different systems implement threads differently. In this section the differences will be discussed.

The simplest model for multithreading from the kernel's point of view is to employ an N:1 mapping for user-level threads to kernel-level threads. In these systems, threads exist *only* on the user-level and are implemented as a library. There, the discussion of the previous section applies directly, as the kernel does not know about threads at all. There are very few, if any, modern operating systems which still use this model. Portable threading libraries that can be employed by applications to get threading support under such operating systems are "MIT threads" ([36]) and the GNU "pth" library ([21]).

In Linux ([30], [5]), OpenBSD ([48]) and newer versions of Solaris ([47]), threads are mapped 1:1 to so-called *light-weight processes* (LWP). In these systems, the scheduler does not need to differentiate between threads and processes. A thread is merely a LWP which shares memory resources with another LWP. This model is easy to understand: just substitute "thread" for "process" in the previous section.

The FreeBSD, NetBSD and Mach schedulers employ an M:N model called "scheduler activations" ([3], [7], [49], [24]). This model tries to profit from the advantages that 1:1 and N:1 threads offer and suffer from none of the disadvantages. In this model, a user-level threading library implements a scheduler which can communicate with the kernel's scheduler to enable maximum control over scheduling of threads within a process. The kernel scheduler views threads as "units of execution of a process". The threads themselves are mostly scheduled like in a 1:1 model. Simply put, the difference lies in what happens when a thread yields the CPU. When it yields, the thread is notified and the user-level thread scheduler is invoked. This can choose to have a different user-level thread take over or have the kernel-level thread yield entirely. The kernel sees only LWPs, just as in the above discussion of the 1:1 model. The user level thread library does the extra work of dividing the threads across these LWPs, but the kernel does not need to know about this.

As can be inferred from this section, threads do not necessarily introduce as much complexity to the kernel scheduler as one might think. From this point on, threads will be largely ignored for reasons of simplicity. A neural network scheduler can be designed to assign per-thread priorities or per-process priorities in much the same way. The current implementation assigns priorities on a per-process basis because of its usage of the *nice* value (see also section 4.3).

2.5 Multiple CPUs

Most current operating systems support multiple CPUs in one system, but in this thesis only single-CPU scheduling is considered for simplicity. The system under discussion only assigns priority to processes. It does *not* actually schedule processes. Because of this, the algorithms for CPU assignment in the operating system proper can remain in place without hindrance from the system under discussion, making multiple CPU support a non-issue. If one were to build a scheduler based on a neural network from scratch, processor affinity can most likely be implemented as a post-processing step to be applied after running the network.

2.6 Existing research

During my literature research, I mostly found articles on either scheduling for SMP (*Symmetric MultiProcessing*, where multiple processing units work in parallel) or simple job-shop scheduling. See section 2.2 for a description of why this is not very useful for process scheduling. There has not been much serious research on uniprocessor scheduling since the early eighties, with some exceptions that will be discussed below.

The emphasis in SMP scheduling is mostly on the difficulty of communicating between the different processors. Some advanced research focuses also on the case where not all processors are equal, but all this is not very relevant to the case of simple uniprocessor scheduling.

Some newer research painstakingly tries to make use of the job shop scheduling algorithms. Deadline notification is introduced in (soft) real-time schedulers such as those mentioned in [34], [16] and [26]. With deadline notification an application notifies the OS about its deadline for a certain (sub)computation. The OS can then calculate when the application must be scheduled and for how long in order to reach the deadline. In hard real-time systems the deadline *must* be reached, or the computation is worthless. For example, in a medical environment a system could be used to regulate a patient's heartbeat. The exact moment and duration of electric shocks emitted to the heart are literally vital. In short, hard real-time system essentially must ensure the deadlines can be met, without exception.

In soft real-time systems the deadline is important, but exceeding it is not a problem. Generally, the deadline will be met *on average*, sometimes exceeding it, sometimes being early. Multimedia applications are a typical example of soft real-time applications. Using deadline notification is a good solution for this type of applications, because for example a video player can drop frames when it does not meet its deadlines and an audio player might switch to a lower sampling frequency.

Introducing deadline notification entails a change in the API (Application Programming Interface) of the OS which in turn requires a change in the applications. This is not an option for simple non-realtime scheduling, for two reasons: a) Every application must be altered to use the new API, which is impossible for many applications. For example sometimes common applications are no longer maintained and the source code may not be available. b) Not every application knows its deadline (either total running deadline or short-term deadline for a

job). Some applications have semi-infinite data to process in unlimited time. For example, consider a video encoding application that encodes data from an incoming video feed and writes it to a file. This can start at any time and can use all the system resources assigned to it until it is finished (someone presses a "stop recording" button). There is no time limit for the entire process and there are no intermediate deadlines either.

Chapter 3

AI techniques

There are a number of available AI techniques that could be used for scheduling. The most appropriate ones will be listed here, along with a short discussion on their applicability to this particular problem. For an in-depth discussion of available AI techniques, the reader is referred to any good textbook on artificial intelligence techniques, for example [43].

3.1 Genetic algorithms

Genetic algorithms are a technique where a population of possible solutions to a problem is (often randomly) generated. These solutions are then ordered on the basis of their fitness and the top (best) n are randomly grouped in pairs which are combined by genetic operators like *crossover* and *mutation*. The rest is discarded[1]. The crossover function selects a random point at which to split the representation of each of the "parent" individuals of a pair. It attaches the first part of one individual to the second part of the other individual and vice-versa, generating two "children" which both have part of their parent's genetic material. The children are optionally mutated at a random point in their representation. The new generation contains either only the children, or the children and the parents. This process is repeated until there is a solution in the set that exceeds a minimum fitness.

This method requires a simple string-like representation of the target scheduling algorithm which must still be valid after random mutation. Also, a fitness function must be formulated for that representation. In our case, the fitness function could very simply be the sum of the per-process deviation from the desired priority. The goal of the evolving individuals would thus be to assign a priority to each process that matches the priority as assigned by the user on a training run[2]. Algorithms represented as strings to be used as genes are by nature reasonably static in what they can represent. The genetic information would have to represent a general *algorithm* which can schedule *any kind of process.* It would be interesting to see research that solves this in a generic way

[1] Usually some of the unfit individuals survive, with lower probability. This keeps the population more versatile and can prove beneficial in a later stage.

[2] This method is very similar to the way the neural network under discussion is trained; see chapter 4.

for the problem domain at hand.

There has been at least one known research project on implementing genetic algorithm-based schedulers [20]. In this scheduler the evolving entity was not the scheduler algorithm, but the applications. The scheduler used the fitness of the applications as priority, and applications could attempt to change their resource usage profile in order to get scheduled more often. The evolution in this case is in the applications, which also means that applications have to be modified in order to work correctly with this scheduler. The advantage of this specific system is that it is backwards-compatible in a way, so old applications do not necessarily have to be adapted for this new scheduler. This would mean that one would have to calculate a static fitness for every existing application or use a default fitness and assign that to every existing application that does not have a fitness value.

3.2 Decision tree learning

Decision tree learning is a way to build a decision tree (or “classification tree”, in our case) from a full set of examples that are expressed in discrete values. It recursively looks for a predicate that would divide the set as equally as possible at the branch under consideration. These predicates do not have to be binary. This process creates a shallow, balanced tree with the predicates at every node and a final decision (or class) at every leaf. This decision tree can be used to quickly classify new problems. One can simply feed an example into the tree and at every node let the result of applying the predicate to the example decide down which branch to descend.

The algorithm is very simple and easily implemented. The biggest disadvantage for using it as a priority assignment function is that it requires discrete values to work with. This can only be done by looking at the input and, for example, introducing “bigger than x?”-like statements into the decision set. This is possible, but it is less precise than continuous values. Also, the data about processes is continuous in nature since most are either counters or ratios of some sort. Furthermore, this method is very sensitive to noise, since every value controls which branch is taken at a particular node, which means the classification process can be easily thrown off by an odd value [38]. Yet another possible disadvantage is the fact that the granularity of the now-discrete values is to be determined in advance. If, in the target use scenario, a variable fluctuates around a small range of values, the other possible values the variable can attain are less important than the small differences in that range. This has to be reflected in the way the values are turned into discrete ranges. The problem can be eliminated by asking the user who builds the tree to supply the ranges, but this also makes the system harder to use.

3.3 Rule-based knowledge system

This technique is basically a simple set of “if-then” rules. This has most of the disadvantages as decision tree learning and it also means the rules have to be hand crafted. This would make the entire system quite static and it would be tiresome to produce new rule sets.

This solution drastically reduces the number of people who can produce schedulers that perform well. Only programmers can do this, generally speaking. The goal is to design a scheduling system that is easily modified by slightly advanced users.

3.4 Bayesian networks

Bayesian belief networks are very similar to neural networks. They represent probability values by connecting two nodes a and b and assigning a weight to that connection which has the value of $P(b|a)$. The certainty of a particular proposition can be calculated by traversing a path from the input to the node that represents the proposition. Every node's probability can be calculated by doing a summation over the incoming connection weights multiplied by the certainty of the nodes they are connected to.

The disadvantage of this approach is that all the network's nodes must have a meaningful interpretation to be able to assign a probability value to its connections. There is no real structure that can easily be imposed on the data the system produces in our case. This structure is not required for a neural network. The backpropagation training algorithms for neural networks determine the underlying structure required for the network to function correctly, but this structure is usually not "meaningful" to human observers.

3.5 Neural Networks

Neural networks, or more precisely, *Artificial Neural Networks (ANNs)*, are a simplified model of biological neural networks like those found in the brains of humans and animals. ANNs consist of *nodes* (also called *units*) which are interconnected with certain weights. Usually the nodes are arranged in layers, and nodes are only connected to the nodes in the next layer.

There is an input layer, optionally a number of "hidden layers" and an output layer. The network is run by *clamping* input values on the nodes in the input layer as *activation*. A node's *net input* is calculated by multiplying every incoming connection's weight by the associated node's activation and summing these values. The node's actual activation is calculated by applying a function to its net input, which usually is a sigmoid or "step" function.

A commonly used learning rule is *backpropagation learning*, which takes the difference between the desired activation value of the output node and the actual output provided by the network. This is the *network error*. This error is propagated backwards through the network, adjusting each weight in the direction that would lessen the error for its node.

This type of learning is called *supervised learning*, because the target value of the output is known and the network is adjusted to respond as desired. There are also networks which are trained by using unsupervised learning techniques, for example *Kohonen's Self Organizing Map* (see for example [29]). These have a different topology. They have not been used in this research because time did not allow it.

Neural networks are the ideal AI technique to use here because they can be deployed in different situations by training them for these. The input and output

are continuous numbers, which the available process data usually is as well. It also allows for a kind of "brute force" search by experimentation. One can simply add features, even if it is still largely unclear how these features can be used to calculate process priorities. The network can "figure out" how to do this, if it is possible. Also, in case of noise in the input, neural networks degrade gracefully. When features are introduced by subsystems that are "equally important" for classification, the problem starts to resemble so-called *P-type problems* [38]. If this is the case, neural networks will perform better than algorithms that look at their input values one at a time in a hierarchical fashion, e.g. decision tree algorithms.

A disadvantage of neural networks is that it is very hard to get any meaning out of their structure, because their knowledge is encoded *sub-symbolically.* When a network is trained for a new feature, it will not give much insight to how the feature relates logically to other features. None of the other techniques discussed above have this disadvantage. On the other hand, there has been some research with the intent to extract rules from neural networks ([50], [44]). This could be useful to create more optimized rule-based schedulers after experimentation with the neural network scheduler.

3.6 Existing research

All neural network scheduling research I have been able to find focuses solely on the job shop scheduling problem (see section 2.2). This is not relevant for this project because job shop scheduling makes assumptions about available knowledge about processes that is not available (see the previous section). This can not be generalized to the problem at hand, because job-shop scheduling deals with static data, where change across time is not involved.

Therefore, using neural networks for process scheduling is a relatively unexplored field. The project under discussion is an ideal catalyst for starting more research in this area. Hopefully it will be picked up by operating system researchers as well as neural network researchers.

Chapter 4

System overview

4.1 Requirements of the network

To repeat our problem statement: We want to control a running process' scheduling priority automatically and *dynamically*[1] without requiring manual intervention from the user. The basic assumption we work with is that a process' desired priority can be determined by analyzing its *behaviour*. We define behaviour to be (any kind of) direct interaction of the process with the kernel, as this is most easily measured by the OS. [2]. This behaviour classifies a process as "interactive", "multimedia", "cpu-bound" or any other classes one wishes to distinguish from. The user defines the classes and the priorities to be assigned to processes that fall into these classes.

The activities of a process will fluctuate quite a bit depending on availability of resources, user instructions[3] and other factors. To mitigate these factors, we will furthermore assume that the basic nature of a process will be *"rather stable"*. In concrete terms, this means we assume its behaviour will either stay the same, or evolve gradually over time. Any *abrupt* fluctuations are purely the result of the changes described above and will not fundamentally change the process class. This is a simplifying, but realistic assumption since in most standard schedulers processes already have a constant (albeit user-changeable) priority as well, and most applications have a well-defined task. That the most common tasks do not feature such erratic behaviour is a "common sense" observation we will leave for later research to (dis)prove.

If the neural network classifies a process based only on its most recent activity profile, processes will most likely be assigned different priorities on each classification cycle because of these fluctuations, despite our assumption that on

[1] As opposed to complete *a priori* analysis of the processes, like in job-shop scheduling (see chapter 2).

[2] We do not exclude the possibility that a program can be inspected or even instrumented to provide more information, especially if emulation (eg, QEMU [8] or Xen [6]) is employed or the OS is run as a regular process (eg, User-mode Linux [17] or NetBSD/usermode [31]), but this is beyond the scope of this project

[3] Examples of user instructions that influence a program's behaviour: A desktop environment will perform different tasks ("clean up the trash", "run program", "find this file", "bring program to foreground" etc) depending on the commands entered by the user. A computer game's behaviour is almost completely dependent on the user's input and in many cases on his reaction time.

average a process has a rather stable behaviour. To lessen this fluctuation, the neural network has to be enhanced with some form of memory. This memory can only be effective if the above assumptions hold, otherwise a process which frequently changes its behaviour will be regarded as a constant process which fluctuates because of instabilities in resource availability or user response time.

The neural network used in this project is a simple back-propagation network. To represent time, an approach similar to what Elman ([19]) suggested is taken, but generalized so the memory is input pattern-specific. The following subsection will explain how time is represented.

4.1.1 The problem of representing time

Usually in networks that have a memory, this memory is used to store the feature values on the inputs at previous time steps. In our case the inputs at any given time step have nothing to do with the previous (see also section 4.3). Every network cycle a new process is selected, and its features clamped onto the input nodes. Generally speaking, processes are unrelated. Thus, at one point in time we have the features of one process clamped onto the input, but the memory will contain a value related to the inputs of the previous, unrelated processes. This will hamper learning rather than facilitate it.

The input at network cycle t is completely unrelated to the input at cycle $t-1$. If there are k processes, the input at cycle t is related to that at cycle $t-k$, but only if no processes were created or destroyed in that time span. This becomes too complicated too quickly. What we really want is a memory of input history *per process*. The global history which represents the relationship *between* processes is important in scheduling as well, but these issues are offloaded onto the existing scheduler (see also section 4.3). This means the neural network scheduler can focus on the main issues of calculating a priority for each process based on their isolated behaviour.

4.1.2 The solution

Input history per process is realized by viewing memory nodes as "secondary input". This means that when the features of a process are presented to the network by clamping them onto the input units, the memory of the process will also be clamped onto the memory units. This logic is captured in pseudo-code in Algorithm 4.1.

Algorithm 4.1 Calculate priorities

for all *proc* in *proclist* **do**
  gather_features(*proc*) {Reset counters, perform preprocessing}
  for $i = 0$ to $nfeatures$ **do** {Clamp input and memory}
    $network.input[i] \Leftarrow proc.features[i]$
    $network.memory[i] \Leftarrow proc.memory[i]$
  end for
  run_network(*network*, *proc*)
  $proc.nice \Leftarrow$ output_to_nice($network.output[0]$)
  for $i = 0$ to $nfeatures$ **do** {Remember new memory values}
    $proc.memory[i] \Leftarrow network.memory[i]$
  end for
end for

After running the network and updating activations, the new activations of the memory nodes are extracted from the network and stored in the process control block (PCB)[4] for the next cycle. This is repeated for the next process. This network is shown in Figure 4.1.

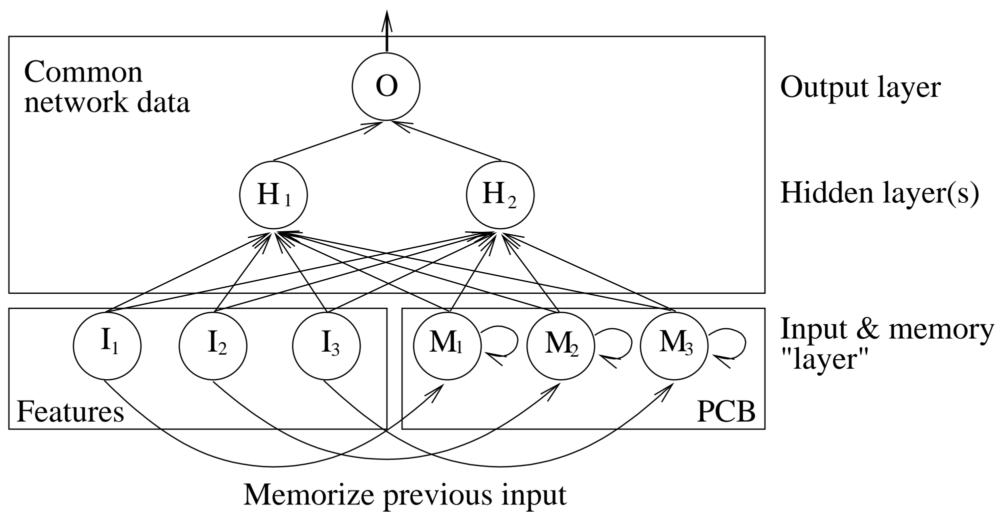


Figure 4.1: Design of the network

When we consider a particular process, the memory will be a function of the features at the previous time steps of only that process. The network's weights will still be updated on the basis of the network error value of every process, but the memory will be process-specific for every network cycle. The strength of the memory can be modified by changing the weight values of the recurrent connections.

[4] The PCB is the structure where all the data relevant to a process is stored.

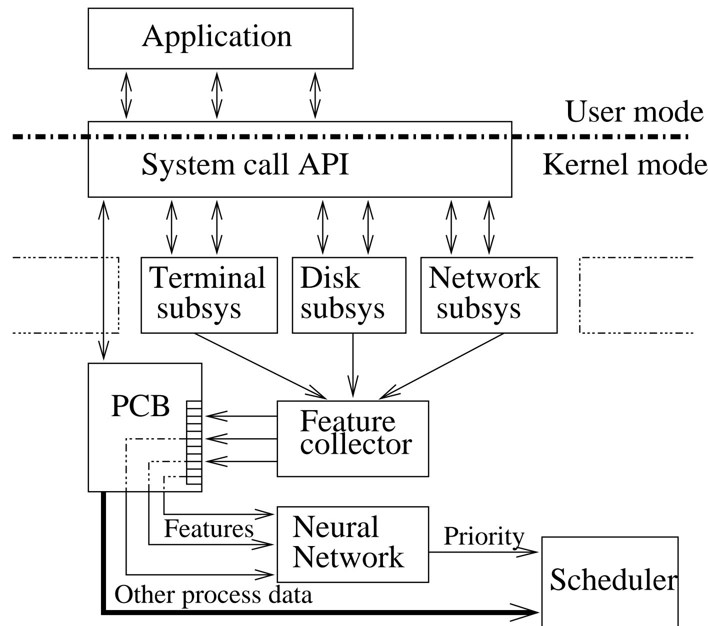


Figure 4.2: The data-flow of the system

4.2 Infrastructural changes

To augment a process scheduler with a neural network, the operating system's scheduling infrastructure will necessarily become more complex because a neural network requires *features* to operate, which means the kernel needs to be set up to collect these.

The kernel can be logically divided into subsystems that service requests to programs. These requests can be translated to features, for example the number of disk reads or writes a process performs. These features need to be gathered for every process. Thus, every subsystem that can report features must add its feature data to the process control block of the relevant process so the scheduler can make use of these features. This implies an extra record in the PCB for every feature that gets added in this fashion, which will quickly lead to a huge number of new records in the PCB. Usually a PCB already contains many fields to begin with, so adding feature fields to the existing ones will make the PCB too large to handle.

To control this complexity, a central feature collection API was implemented to which the various subsystems report their feature data. The layout of this system is shown in Figure 4.2. As can be seen, there is only one new API that was added, and subsystems have no knowledge of features of other subsystems. The feature collector dynamically allocates and deallocates memory in PCBs to hold this feature data whenever new features are registered or new processes are created. Only one extra field needs to be added to the PCB in this case, namely one which holds the list of feature data. An added advantage of this is that features can be introduced dynamically while the system is running, which can happen when a new loadable kernel module (LKM) is loaded, for example.

Whenever the scheduler updates process priorities (once per second), the feature collection subsystem asks every subsystem that registered features for a calculation of these features. The feature data is not extracted directly from the PCB because some features may require preprocessing or special initialization. For example, calculating the ratio of reads versus writes on a (pseudo-)terminal is done by storing reads and writes in a substructure of the PCB. Only the terminal subsystem knows how to interpret this substructure. When the features are to be calculated, the feature collection subsystem asks the terminal subsystem to extract the read/write ratio feature. The terminal subsystem then extracts the reads and writes from the substructure, returns the ratio and resets the number of reads and writes again for the next cycle.

4.3 Pre-existing priorities

All throughout the preceding discussion, it has been implicated that the neural network scheduler calculates the priority completely on its own. This is not exactly true. The intention is to leave the existing scheduler in place, so we can focus on the pure problem of assigning the priorities.

Selection of processes based on priority, in such a way that no *starvation*[5] can occur, is a separate problem that has already been solved by the designers of the operating system in which our scheduler will be integrated. There is no reason to implement it ourselves. It will suffice to simply use the existing scheduler, while only manipulating the base priority for a process. This is consistent with the notion of the neural network scheduler as a helpful tool for the user that takes away the work of deciding on a base priority by hand.

[5]Starvation occurs when a process never gets its turn on the CPU.

Chapter 5

Implementation details

This chapter describes implementation details that are useful mostly to those who wish to work with the code or use the programs. First we will describe the system used for testing, then an overview of a typical usage scenario is presented. The chapter is concluded by a list of the various subsystems in the kernel and the various utilities that were implemented.

The source code to the implementation described in this chapter will be made available for download from `http://nnsched.sourceforge.net`.

5.1 The system

The implementation of the project was done under NetBSD 2.0 and later on 4.0 Release Candidate 1[1]. Tests and measurements were performed with NetBSD 4 on an iBook G4 (PowerPC processor) with 512 Mb of RAM. The kernel was tweaked by adding an `options HZ=1000` line to the configuration file in order to have its clock interrupt at 1000 Hz instead of the normal 100 Hz for improved precision[2], as described in [22]. NetBSD was chosen for its clean, modular design, the availability of its source code and the author's familiarity with its code base.

5.2 Typical usage

A description of how to use the collection of tools created for this project will be described below. The words printed in `typewriter font` are names of programs. More information on these programs can be found in section 5.5.

Training a network is done completely offline. This means it is not required to have a modified kernel to train the networks. In fact, the networks can be trained (as well as tested) on any other OS if this is required. The feature data is extracted from a running modified kernel and written to a file which can

[1] See `http://www.netbsd.org`.

[2] With this modification, the clock interrupts more often. Processes that go to sleep soon after they are allowed to run on the CPU will cause the system to idle for a shorter time, because a new process can be selected sooner. Increased clock resolution also means more overhead since the scheduler is invoked more frequently.

be used for training and testing. The training application writes the trained network to a file which can be uploaded to a modified kernel at any time.

To create a working network, the first thing one must do is set up the machine in such a way that it will run all programs that are typically used. These must all be assigned a priority at which they are supposed to operate. In essence, a normal working situation is created, with the difference that a lot more programs are run at the same time. This can be automated by running `progstarter` with a suitable configuration file.

The activity patterns of the processes are captured by running `nnfmon`. The resulting feature file is then given as input to `nntrain`, which will train a network. This network can be tested on another dataset with `nntest` to see how well it generalizes. This can be repeated while tweaking parameters like the learning rate, number of hidden layers and units in those layers and so on until the user is satisfied with the results. `nngather` is intended to automate this process somewhat. It just requires a training and a testing dataset and trains a number of networks with different settings, compares the settings and generates statistical output on which performed best.

The trained network can be installed with the `nnconf` utility. This will "upload" the network to the kernel's neural network scheduler. From then on, it will be activated and the processes should be assigned a fitting priority automatically, in a way transparent to the user. For the user, it simply should "just work", without having to think about it. The system can be set up to load this network every time at startup, to make use of the neural network scheduler as unobtrusive as possible.

Training a network and fine-tuning it is too technical for the typical desktop user. For this type of users, a set of pre-trained networks for certain common usage situations could be shipped with the OS. The installation process could improve user-friendliness by asking what the computer will be typically used for, install the appropriate network and ensure it will always be loaded at startup.

5.3 Shared code

Since the neural network is the engine of most of the programs and subsystems described below, it is only natural to share as much code as possible.

Everything that is used in both the kernel and the utilities is located in the file `sys/kern/kern_nnnetwork.c`. This is the code used to build a network from scratch and to calculate net inputs and activations per layer. The utilities' code pulls this in and makes use of it.

There is some utility-specific code under `usr.sbin/nntrain/nnnetwork.c`, which is used in `nntrain` as well as in `nntest`. It contains the functions required to generate random weights, calculate network error and functions to select and clamp inputs. The reading code for network and feature files is also shared between the testing, training and configuration programs in `usr.sbin/nntest/nnread.c` and `usr.bin/nntrain/nnfeat.c`.

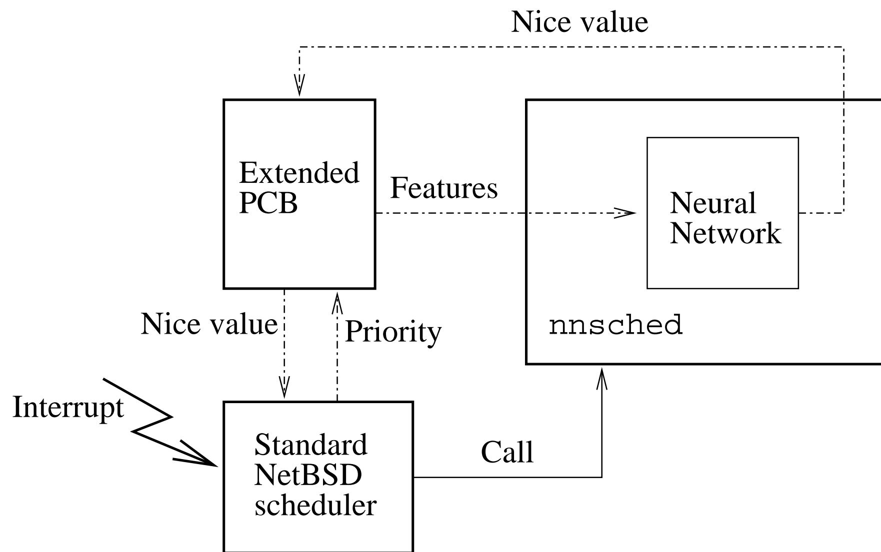


Figure 5.1: Overview of interaction between scheduler parts

5.4 Kernel level implementation

The existing kernel code was largely left untouched. The process scheduler was slightly modified to call a new function to calculate priorities and a number of subsystems were modified to report features to the feature collector. The feature collector itself is was built as a mostly independent subsystem. One extra field was added to the *proc* structure (NetBSD's PCB) to hold feature and memory data. The neural network running code, the feature calculation and the feature monitor device are all mostly independent from the rest of the kernel. See Figure 5.1 for an overview of the interaction between the parts.

To save time, it was decided that the easiest way to implement a new scheduler is to simply change the *'nice'* value[3] and leave the existing scheduler almost untouched. This means control over the priority is less precise because the kernel will add other factors into the calculation (for example, if a process did not voluntarily release the CPU, its priority will drop but its nice value will remain unchanged). The nice value is the same knob for tuning priorities that is available to the user, so this approach is still consistent with the original idea of taking work out of the user's hands. The main disadvantage of this approach is that power users can no longer influence or override the priority by changing the nice value. The advantage is that the existing algorithm needs almost no modification at all. There is also no problem regarding the interface with which the user needs to define the target priority. The user can simply run programs and tweak their `nice` values to indicate the target priority. The user can use the familiar interface that existed before and no new tools have to be implemented to define priority.

As has been said before, the end user does not necessarily have to do this as

[3]The nice value is a priority value which the user can modify. The real priority of a process is determined by the nice value and the estimated CPU time.

developers and distributors can ship the system with pre-trained networks for a variety of common usage scenarios and have the installer offer to select one of them for loading when the system boots.

5.4.1 Pseudo-devices

The following pseudo-devices are available to applications for retrieving feature data from the kernel and uploading configuration data to the kernel.

nnfmon

The neural network's feature monitor pseudo-device, `nnfmon`, exports feature data from the kernel to applications. Every second, just before the features are updated, `nnfcoll` sends the old feature values to `nnfmon`, which stores these in a freshly allocated buffer. These buffers can be accessed by a process by `open`ing and `read`ing from the pseudo-device.

nnconf

The configuration pseudo-device for the neural network. This allows applications to configure the neural network scheduler and upload a neural network's layout and connection weights to the kernel.

5.4.2 Scheduler and feature collection

nnsched

The existing scheduler in `kern_synch.c` is modified to call the *nice* calculator every `hz` clock ticks (every second), if the neural network has been initialized. The nice calculator (which is located in `kern_nnsched.c`) simply runs the network from `kern_nnnetwork.c` in feedforward mode, resulting in an activation value on the output node. It is assumed that there is only one output node, which represents the processes' priority. This priority is expressed as a value between 0 and 1, which is inverted and scaled back to an integer in the range of 0 to 40, from which 20 is subtracted to get a value between -20 and +20 (which is the range of Unix *nice* values). See also section 6.1.

nnfcoll

On startup, all subsystems which collect data for a feature will register this feature with the neural network's feature collection subsystem, `nnfcoll`. This allocates a unique feature identifier (`ufid`, for short) and associates this with the callback function pointers for this feature in the list of known features. There are three callback functions: One for allocation of process-specific feature data (which is treated as opaque data by anything but the subsystem for which it is intended), one for deallocation of this data, and one for calculating features.

The feature calculation function is called every time process priorities are updated, for every process. This happens just before the features are fed into the neural network for a new epoch. The allocation and deallocation functions are called on process creation and exit.

5.5 User level implementation

5.5.1 Tools for interacting with the kernel

nnfmon

The neural network's feature monitor `nnfmon` is both the pseudo-device discussed in subsection 5.4.1 and the name of a program which reads data from this device. This application simply empties all `nnfmon` feature buffers, storing the information in them in a file. This file is later read in by other neural network applications.

nnconf

This simple program reads a saved neural network and 'uploads' its layout and weight data to the kernel by means of `ioctl`s on the `nnconf` pseudo-device.

5.5.2 Training tools

nntrain

The training application for the neural network. This application reads out a file as stored by `nnfmon` and allows the user to create a network and train it with these data as input. It accepts network configuration on the command line in the form of size and number of hidden layers, maximum net error, maximum number of epochs to train, learning rate, momentum and initial weights for memory connections. It initializes the other weights and biases randomly.

nntest

An application intended for comparing existing pre-trained networks. It displays a net's error with regard to an `nnfmon` feature file.

nngather

This is a simple program used to gather data on the effectiveness of combinations of features and settings in the `nntrain` program, by training a number of nets and testing them on a training and testing set.

5.5.3 Miscellaneous tools

typesim

This is a very small program that will attach to a terminal and send simulated keystrokes to it. These keystrokes will trigger writes to the terminal as though they were real ones, thus creating a "fake interaction" with an interactive process.

progstarter

This is a program that sets up a system to run a number of programs with specified priority settings and calls `nnfmon` to record their feature values. It can

optionally also start an instance of `typesim` to create a fake interaction with some of these processes.

featmerge

This little tool can merge two output files from `nnfmon`. It is necessary when one wishes to combine application runs which claim the audio device, if there is no in-kernel mixing available. It can also be very useful to try out different combinations of programs. One can simply run each program on its own, capture its features and combine these output files to create usage scenarios.

gentest

A simple program used to generate test sets. One can define a number of "features" and a number of "processes". It will divide the range of nice values equally among the simulated processes. The feature values that are written are random, but they all add up to their converted nice value (meaning -20 is mapped to 0 and 20 is mapped to 1). This program is useful if people would want to test different kinds of network implementations or modifications to the existing one.

5.6 Implementation issues

This section will deal with some practical problems that arose during implementation of the network.

5.6.1 Floating point

In NetBSD, it is not allowed to use floating point instructions inside the kernel. At first, when all subsystems are initialized, the floating point unit (FPU) is not initialized yet. About halfway through the main initialization process, when the hardware is probed, the FPU is initialized.

The features are registered during hardware probe, as well, since their data is generated by device drivers. We cannot control the order of device driver initialization and probing, so there is no guarantee that the FPU has been initialized at feature initialization time. When a feature is registered, any process which is forked afterwards will have its NN data structures initialized. This will cause the system to crash if the FPU is not yet initialized at that time, because these data are in floating point format.

A solution would be to wait until the FPU is initialized and only then initializing the NN data structures for every existing process. This was implemented and worked for a while, until a process used the FPU. This would cause a crash as well because the kernel changes the FPU status while calculating features. Saving and restoring of FPU register status is very slow on many machines, so the kernel does not do this on every switch to kernel level, only when doing a process switch. As a consequence, the kernel can not use any floating point instructions at all.

Therefore, the current implementation uses fixed point arithmetic. This considerably slowed down the implementation of the project, because the author had no previous experience in writing fixed point arithmetic and there are many

subtle things that can go wrong while using fixed point. A decent tutorial to fixed point can be found in [28]. On the upside, because this required close attention to the representation of the numbers, it provided some insights that would otherwise probably have gone unnoticed.

5.6.2 Range and precision problems

Theoretically, neural networks can be used to calculate any mathematical function one can think of. In other words, they are Turing-Complete (See [18] for a proof[4]). In practice, however, the computational power of neural networks is directly proportional to the numerical range and precision of the implementation.

Consider a simple network with one input node and one output node. For simplicity, let us assume we try to learn only two patterns, namely (0.001) where the output shall be (1) and (0) where the output shall be (0). In order to represent a network that can compute this, we must allow numbers as large as 1000 in the weights. Also, in certain situations only very small changes in weights can make a big difference. If the chosen representation does not allow such precision, it may be impossible to learn such a pattern.

Fortunately this was not a real problem in practice, but it might matter in the future for certain new patterns or features. Extra preprocessing could probably be used to change ranges or clustering of activity in such cases. It must be noted that the same problems can occur with other algorithms as well. Even floating point has a maximum precision. The major difference is that fixed point is much less flexible, because one has to decide a priori what the position of the point will be. On the upside, fixed point is many times faster than floating point on most systems, making it more suitable for use in kernels.

5.6.3 No audio mixing

NetBSD does not provide a possibility to multiplex the audio device. Once one process has claimed the audio device, others can not access this device. The kernel does not mix sound channels. This is not a real problem because there are various ways to multiplex audio using userland applications, but this means feature information about the application that outputs the audio is lost. The kernel only sees audio output from the mixing application. To remedy this, a program was created to combine feature outputs from several monitoring runs. One can start a number of applications which includes up to one process with direct kernel audio output. These can then be monitored for features, after which a different combination of applications can be started for feature monitoring. Finally these two feature files can be combined. This leads to a feature file which interleaves the feature values from the first two feature files. This circumvents the audio mixing problem by mixing the feature values afterwards.

This has no negative effects, because this output would look exactly the same if the processes were actually started together, if in-kernel audio mixing was available.

[4] This proof is slightly incomplete, see the appendix for a discussion.

5.6.4 The X Window System

In Unix, a graphical shell is optional. The most common *Graphical User Interface* (GUI) for Unix is X11, which is generally not integrated in the kernel. The system consists of a server and a number of clients. The server program controls the display resources of the workstation while the clients send commands to it over a socket. This means the clients look to the kernel like simple network programs. Generally, the server does not know (indeed it does not *need* to know) which process is requesting graphical output. This makes it very difficult to gather process information from X.

Probably the most interesting features for multimedia are the frequency and nature of graphical requests by a process. To get this information available to our scheduler, it must be extended with a way for X to communicate this information to it. X would need to be modified as well. Because of a lack of time this project did not include the task of making these modifications to X and the kernel, but theoretically these should certainly improve the recognition of multimedia processes. Etsion et al. [23] describe a system that does something very similar, and it would be interesting to see this system re-implemented in terms of features for our neural network scheduler.

Chapter 6

Results

Here we will present the results of running the neural network on certain example program sets and a benchmark of the overhead of running a neural network in the kernel.

The kernel that was used in the tests below is the `GENERIC` kernel for the Macintosh PowerPC ("macppc") port. There is a `GENERIC` kernel for every port in the NetBSD source tree. These kernels include almost all drivers and subsystems supported by the OS on that particular platform. This is the kernel that one will find in a default installation of the operating system because it is most likely to work on a wide range of hardware configurations.

Using such a "default" configuration means these results will be easier to reproduce in other experiments if needed, since the configuration is not special or tuned to specific hardware in any way. As stated before in section 5.1, there was one change in this configuration: an `options HZ=1000` line was added to increase feature gathering granularity. The neural network kernel used is simply a `GENERIC` configuration with the added neural network scheduler subsystem.

6.1 Features

The features that were used were the same for all subsystems: The number of reads and writes were recorded, as well as the number of reads divided by the number of writes. The number of reads divided by the number of writes was deemed a good feature because it can not be calculated by the wiring of a simple network with few layers. It does provide useful information, since a process may do a lot of reads and writes or just a small number of reads and writes, but the ratio might be the same. This will hopefully remove the differences in hardware and/or bandwidth, making pre-trained networks more generally deployable. The following subsystems were monitored for these three features:

- Sockets (which can be both network and interprocess communication). The expectation is that applications like HTTP daemons will need to be given an elevated priority on server machines and filesharing programs will need to be given a lowered priority on desktop machines. Another reason to use this feature is that X window clients use sockets to communicate

with the X server, which handles graphics, so it can help us in further distinguishing background processes from interactive processes.

- Terminal interface (including pseudo-terminals like `xterm`). In most situations, terminal emulators should have a reasonably fast response because they provide the main interaction with the system.

- Audio. Multimedia applications generally produce audio output, which would make this a very suitable feature for detecting this kind of application on a desktop system.

- Filesystem. Indexing of the filesystem, archiving, making backups and many other typical background tasks interact solely with the filesystem, which makes this a useful feature to detect these kinds of applications.

Target values were nice values, specified by a `progstarter` configuration file. Nice values are in the range $[-20, +20]$. The network works with a linear activation function which has the range $[0, +1]$, so target nice values were originally normalized by the following function:

$$\texttt{convert_nice}(n) = \frac{n + 20}{40}$$

After some experimentation, it was observed that the network would not learn very well. After some consideration it was determined that the reason for this was that the calculation of the network would be biased in favor of *high priorities*, because low nice values close to zero actually give high priority to the processes. When a process does not have particularly interesting features, it will result in zero values being clamped upon the input nodes of the network.

Uninteresting processes should run at a default nice value and only be elevated if they are interesting. There is no way by which a zero value can be multiplied so that it will result in a more useful value. Biases introduced in the network will help a little bit, but the training process tended to drop the bias connection weights down, so this would often result in the network favoring elevated priority. By *inverting* the output node value, this problem is solved. This results in the following function to convert a nice value to a target value:

$$\texttt{convert_nice}'(n) = \frac{-1 \cdot n + 20}{40} = \frac{20 - n}{40}$$

6.2 Network accuracy

To measure how well the network was able to learn feature sets for running processes, data was gathered by running one program at a time with a target nice value and monitoring its features using `nnfmon`. The target nice values selected for each of the observed programs can be observed in Table 6.1. These nice values are just an example of what a typical desktop user would most likely prefer.

It is assumed most desktop users want their multimedia applications to run at the highest priority (lowest nice) and processes that only perform a lot of disk access (like find) to run at the lowest priority. Usually an editor does not need the highest priority to achieve acceptable response times, but it should

nice	Program name	Type of program
-20	Mplayer	Movie player
-8	Firefox	Graphical web browser
-4	Vi (600 kpm)	Text editor
-2	Vi (120 kpm)	Text editor
6	Ftp	Network application
18	Bzip2	Compression application
18	Find	File searching application
20	Mencoder	Movie recoding program

Table 6.1: "Nice" values for a typical desktop situation

be assigned an elevated priority so it will not become too unresponsive on high system load. Network applications should be fast, but as soon as system load drops, they should yield to more important processes since downloads are not as important as interactive applications. The current distribution of nice values should be seen as an example, or proof of concept. The feature values of all these processes were merged together to create a file for training and a file for testing.

The rationale behind having two editors running with different typing speeds (in keystrokes per minute) is that fast-typing users will benefit more of a fast response time. As they type, they want to see immediate feedback. Slower typists will not require such speedy feedback because they will not have hit so many keys before noticing they made a typo.

The training set and the test set consisted of mainly the same programs, but the files on which these programs operated were different between the sets. For example, the movie player was instructed to play a different movie in a different format, the web browser was used to visit different sites and the vi sessions had different keystroke files for simulated typing.

To obtain a good network, `nngather` was run to try a number of combinations of settings, running the network with the same settings 50 times to eliminate random influences. This run took two full days, which the author considers to be too long. It would be interesting to see if major speed improvements could be made in the training and testing algorithms, which would be needed for measurements concerning larger datasets. The generic backpropagation algorithm has been improved upon by a number of studies, for example *Quickprop* [25], *Rprop* [39] and *GRprop* [2]. Any of these could probably be used as a drop-in replacement for the standard backprop algorithm. The author did not expect backpropagation to be as slow as it proved to be. Nevertheless, the results discussed below still provide some interesting insights.

The best network that was trained produced a network squared error of 0.169540 on the training set and one of 0.188568 on the test set. The error values for the respective programs are presented in Table 6.2. Interesting results were that find and the simulated vi sessions were very constant in error values. One might think that this is a result of the fact that the keystrokes were simulated. Looking

at the vi session with manual keystrokes, which are much less constant in speed, the error was just as constant as with the simulated test runs. This is

Average squared error	Program name
0.480523	Mplayer
0.305481	Firefox
0.059364	Vi (600 kpm)
0.058685	Vi (360 kpm)
0.058640	Vi (manual typing)
0.036316	Vi (120 kpm)
0.121804	Ftp
0.127144	Bzip2
0.169342	Find
0.136162	Mencoder

Table 6.2: Error values of the tested programs

an indication that the memory layer does exactly what it was designed to do; make the input smoother.

We can also observe that the features are apparently not quite suitable enough for classifying multimedia processes. The errors of the movie player and the web browser are notably higher than the other processes. The fact that the web browser was classified badly was expected since it does not have any very distinguishing features. It mostly does some disk reads and some network access, like a combination of the compression and download programs. On the other hand, the movie player can be accurately identified by its sound output, so it was expected that it should be recognized by the network. The other features apparently drown out the importance of the audio features. Introducing more multimedia-specific features would hopefully balance this out.

In [23], a number of "Human-Centered" metrics are explored which can be used for priority heuristics. The results shown in this studies are promising. It would be trivial to implement these metrics as features for our neural network. The biggest problem with these metrics is that the data required to measure them are not available to the kernel directly. To make the kernel aware of, for example, GUI features like horizontal and vertical size of a window, one needs to implement a feature reporting system. Userlevel applications need a way to notify the kernel about the service requests they receive from other applications. In Unix much information is not available to the kernel, because for example the graphics device is multiplexed to other userland applications by X. This is also true for audio and pseudo terminals on certain systems. Where such systems are in use, these would have to be modified as well as the kernel to realize the gathering of these additional features.

On microkernel systems (examples are QNX [37], Windows NT [32], EROS [45] and Mach [1] and its descendants Hurd [11] and Darwin/Mac OS X [4]) this would probably be less of a problem. Microkernel systems are systems which have a tiny kernel that performs only the minimal number of tasks required for the OS to function. Functionality that is provided by the kernel in traditional systems is provided by several processes or threads. This also allows easy replacement of kernel subsystems by simply starting a different implementation of that particular subsystem. This architecture is more open to userland processes, so on these systems it may be easier for these resource-multiplexing applications

to access process information and report this to the neural network scheduler. On systems with a monolithic design, where each and every resource is handled by the kernel, these problems are nonexistent because all needed information is directly available to the kernel.

6.3 Influence of the memory layer

One interesting, but possibly problematic result of introducing the memory layer is that training a network toward a low error does not mean a testset will produce the same error. If one trains a network, the memory layer is trained as well, producing a memory "fingerprint" unique to this particular training run. The weights are updated to reflect more or less importance of the relevant feature's history with regards to the current value of the feature (ie, the degree of "decay" of the memory). The history that has been built up to this point is still subject to the previous memory weight values. This is mathematically expressed in Equation 6.1. Here a_m and a_i are the activations of one memory unit and its input unit, respectively. w_{im} and w_{mm} are the weights from the input unit to its memory unit and the recurrent weight from the memory unit to itself, respectively.

$$netinput_m(t) = a_i(t-1) * w_{im}(t) + a_m(t-1) * w_{mm}(t) \tag{6.1}$$

This is not incorrect, but when one runs a test, the memory history will always be different even on the same dataset, since the equation does not rely on changing weights but assumes they are constants as can be seen in Equation 6.2.

$$netinput_m(t) = a_i(t-1) * w_{im} + a_m(t-1) * w_{mm} \tag{6.2}$$

Introducing the `LOCK_MEMORY_WEIGHTS` option (which is a C preprocessor `define`) into `nntrain` allowed experimentation with this phenomenon. When one trains a network using `nntrain`, the network error is printed during training. When testing this network with `nntest`, we *only* get exactly the same error `nntrain` printed after the last epoch when memory weights are locked and the number of runs of the training set is equal to the number of training epochs. When the number of "epochs" of the test program is different, the memory layer does not store the exact same history for all processes as the training program recorded during *its* run.

6.4 Overhead

To measure the overhead of the neural network scheduler, a number of test runs were performed with the benchmarking program `hackbench` [42], which was specifically developed for benchmarking scheduling algorithms. This benchmarking program creates a number of process groups which communicate through a socket. It creates an equal number of senders and receivers in each group. All of the senders in a group send one hundred bytes to all of the receivers in its group, repeated a hundred times. The size of a group is 40 processes (meaning, 20 receivers and 20 senders).

The overhead calculation is performed by measuring the total execution time or "wall clock time" of a `hackbench` session. By subtracting the end time of a

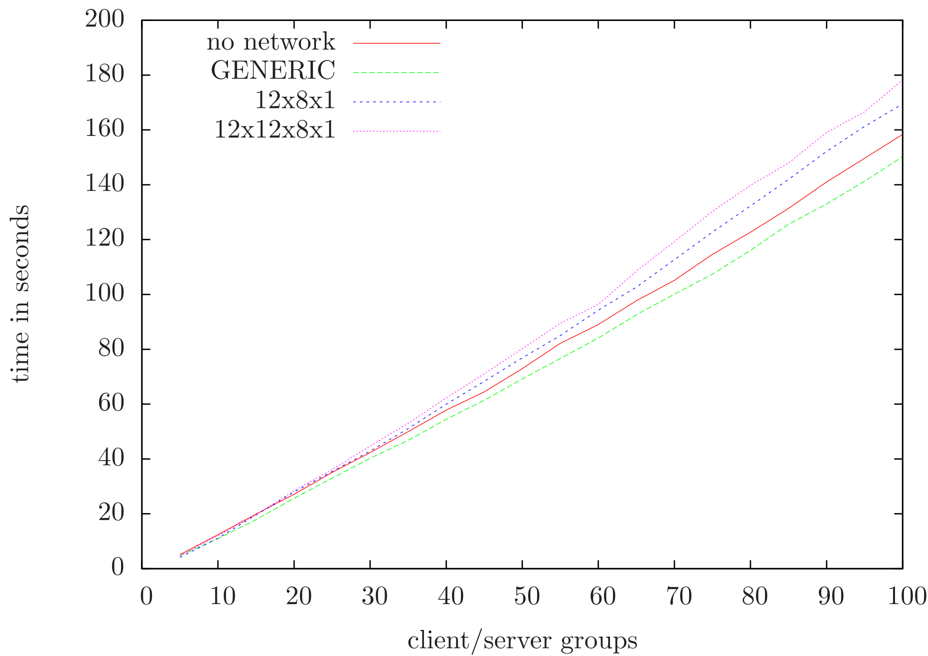


Figure 6.1: Runs of `hackbench` with different kernels and networks

session from the start time, a result is obtained. These results can be compared between different kernels to see how much scheduling overhead is involved by subtracting one kernel's total time of the same number of `hackbench` groups from the other kernel's total running time of these groups. The difference is the extra time spent inside the scheduler, considering everything else is equal.

These tests were conducted in the so-called single-user mode, which is an operating mode in which no services are started except for the ones that are vital to keep the system running. This was done to minimize interference from other processes that otherwise might be running (for instance, `cron` which might run a daily script that performs quite heavy disk seeks). Each test was run 5 times and the results averaged to reduce further anomalies. After every batch of 5, the system was rebooted to ensure earlier tests would not influence later tests. The results are displayed in Figure 6.1.

6.4.1 Feature gathering overhead

The original NetBSD scheduler (marked `GENERIC` in the Figure 6.1) has a very slight curve, indicating quadratic complexity of the scheduler. This curve becomes more pronounced as more time is being taken by the neural network scheduler. The complexity of running the network and feature gathering for one process is constant with respect to the number of processes. The features for one process are gathered and the network for that process is run completely independent from all other processes. Because the feature gathering is run for every

process, the total overhead of the feature gathering is linear. The network is run inside the scheduler so, having constant complexity of itself, the scheduler as a whole will be of quadratic complexity since $O(1) \cdot O(n^2) = O(n^2)$. The feature gathering is not run inside the scheduler. Considering the total overhead of the new system, this will still be of quadratic complexity as $O(n^2) + O(n) = O(n^2)$.

In the graph we can see the overhead of the feature gathering (marked *no network* in the figure) compared to a normal `GENERIC` kernel without feature gathering. At around 15 groups of processes of hackbench (making 600 processes in total) one starts to see a clear difference between a regular `GENERIC` kernel versus the feature gathering kernel. The difference in feature gathering overhead as compared to the kernel without feature gathering is about 8 milliseconds per process group or 0.2 milliseconds per process.

This overhead consists of three things: First, when a process is created, the kernel allocates of a buffer to hold bookkeeping information which is needed to track the data required for calculating the feature's input value for the network. It initializes this buffer for each feature by copying from the parent process that invoked the new process. Second, at every call in the subsystem which registered the feature, there will be a call to a procedure that will increment a tally for this feature or do something else, depending on the feature. Third, when a process is destroyed, the kernel needs to deallocate the feature buffers.

To determine the overhead of the neural network itself, the above process was repeated after loading a network into the kernel, causing it to go through the steps of calculating the features as well as running the network in feedforward fashion to calculate `nice` values for all the processes. The nice values would all be exactly equal, so as not to cause skews in the measurements. This was ensured by simply setting all the network's weights to zero, thereby making all processes get an equal nice value of 20 (lowest priority).

We can see that the overhead of a network is quite noticeable. The overhead of the 12x8x1 network over the feature gathering is about as much as the overhead of the feature gathering over a `GENERIC` kernel; it is roughly 12.5 milliseconds per process group or 0.31 millisecond per process. The total overhead of this network as compared to a `GENERIC` kernel is thus about 20.5 milliseconds per process group or 0.51 milliseconds per process. The overhead of adding an extra hidden layer of 12 nodes to the network is roughly 8.5 milliseconds per process group or 0.21 milliseconds per process.

Seeing these numbers compared to each other is a strong motivation to choose features very carefully in a production system, so that the total number of nodes can be kept to a minimum. In actual usage of the system, it is more interesting to see what the *perceived speed* is, as experienced by the user. This will be dealt with in the next section.

6.5 Perceived speed

Most computer users are not interested in minor speed differences between processes and schedulers. The reason for using a neural network scheduler on a desktop system in a multimedia context is to see if it is possible to improve the *subjective speed* or the speed the user *perceives* the system to have. When an interactive or multimedia process slows down on a desktop system, this is much more noticeable and annoying than when any long-running background process

Kernel	config	fdrops$\pm\sigma$	%fdrop	fps$\pm\sigma$
GENERIC	default	1389$\pm$8.5	96.46%	0.65$\pm$0.08
nnsched	default	1389$\pm$13.9	96.46%	0.71$\pm$0.27
GENERIC	manual	2$\pm$3.4	0.14%	21.95$\pm$0.08
nnsched	manual	2$\pm$3.0	0.14%	21.96$\pm$0.05
nnsched	network	41$\pm$11.3	2.85%	17.57$\pm$0.30

Table 6.3: Dropped frames with `mplayer` for one minute of play time

is taking a little longer to complete. The idea is then, to *deallocate* computing time from "uninteresting" processes and reassign this time to more "interesting" processes. A neural network can be trained on different data sets in order to specify what behaviour profiles are considered "interesting" on a particular system.

To measure if this works well, we have set up a system with a number of processes that simply exist to slow down the system: two `bzip2` processes are enough to make this happen. To measure "responsiveness", the `mplayer` process is set up to record the number of so-called *frame drops* that have occurred. A frame drop happens when the system is overloaded so much that the process gets scheduled on the CPU so infrequently that the actual "wall clock time" that has expired is so much that it is no longer possible to fit the number of frames per second that the video uses on the screen. In that case, frames have to be omitted, or "dropped". The more frames are dropped, the lower the frame rate and the higher the perceived "stuttering" effect of the video is because it lacks smooth transitions between two frames.

The video that was used in this example is an AVI file with audio component that is encoded with the AC3 codec at 48 KHz at a bitrate of 192kbit and an XVid video component with a pixel size of 720x416, recorded with 23.976 *fps* (frames per second), which should run at 3289.4 *kpbs* (kilobit per second) for optimal playback quality. This is at a high enough resolution to be so demanding of the system that a slowdown is immediately noticeable when a few CPU-intensive processes are started along with the video player.

The experiment that was set up consisted of five test runs for each situation in order to get reliable results. After every run, the system was rebooted in order to negate any effects of file caching. The averages of these test runs are displayed in Table 6.3. The network that was loaded was reasonably simple and trained with the test configuration given in section 6.2. The network consists of 12 input nodes, 12 memory nodes, one hidden layer of 8 nodes and one output node. To produce a full comparison, four situations with no network loaded are given, compared to one situation with the network specially trained to boost priorities for multimedia applications.

The column marked *config* in Table 6.3 presents the following configurations:

- *default* is the default situation, where all processes get a default `nice` value of 0, which means there is no priority boost or priority penalty for either process type.

- *manual* is the situation where the user has manually indicated that he wants the `mplayer` process to run at maximum priority (a `nice` value of

-19 — note that the user needs to have `root` privileges in order to be able to do this) and the `bzip2` process to run at the minimum priority (a `nice` value of 20). This is the best situation imaginable when simply running only `bzip2` and `mplayer` and we are only interested in making `mplayer` run as fast as possible. Any background process (like the X server, the window manager, the user's shell etc) still run at `nice` 0, or default priority.

- *network* is the situation where the neural network scheduler is loaded and actively monitoring and scheduling processes. Any `nice` value set by the user is ignored and reassigned by the neural network scheduler. All processes are automatically assigned priorities by the scheduler, not only the `mplayer` and `bzip2` processes under discussion. This includes the user's shell, X and the window manager. The network was trained with example sets such that `mplayer` should receive a `nice` value of -19 and `bzip2` should receive a `nice` value of 20.

The column marked *fdrops* presents the average number of frames dropped by `mplayer` in the given configuration and its standard deviation on the one minute of play time in four trial runs.

The column marked *%fdrop* presents the average percentage of dropped frames. The total number of frames in one minute of the video used for the tests was 1440.

Finally, the column marked *fps* presents the average frames per second and standard deviation that was achieved on the four runs in the given configuration. This is what concretely determines how smooth the video plays for a human eye. This number can differ from the *fdrops* average because even though a minute of video was requested, the actual play time can be slightly higher depending on the amount of stutter. When audio and video get out of sync because of stuttering and video frame drops, the player needs to compensate by waiting for the audio track to catch up. This means that with higher frame drop counts, the total time taken for the video varies more. It is always higher than the total time of the video clip. The better fps rate of the nnsched kernel in the default config is the result of such fluctuation, as can be determined from standard deviation.

These results indicate that the neural network scheduler performs quite well in this task and does exactly what it was designed to do: take the nice value assignment out of the hands of the user and automate it. The GENERIC scheduler in the default situation does not take into account the fact that the `mplayer` process is more important to the desktop user than the `bzip2` processes and simply divides the CPU time equally to the processes. This is not good enough because it results in an extreme amount of dropped frames. The neural network scheduler without a network loaded behaves identically.

When a network is loaded, the neural network scheduler adjusts the priorities automatically by determining that the `mplayer` process is using audio output and uses that to boost its priority, resulting in a negligible amount of dropped frames. This certainly makes the video more pleasant to watch.

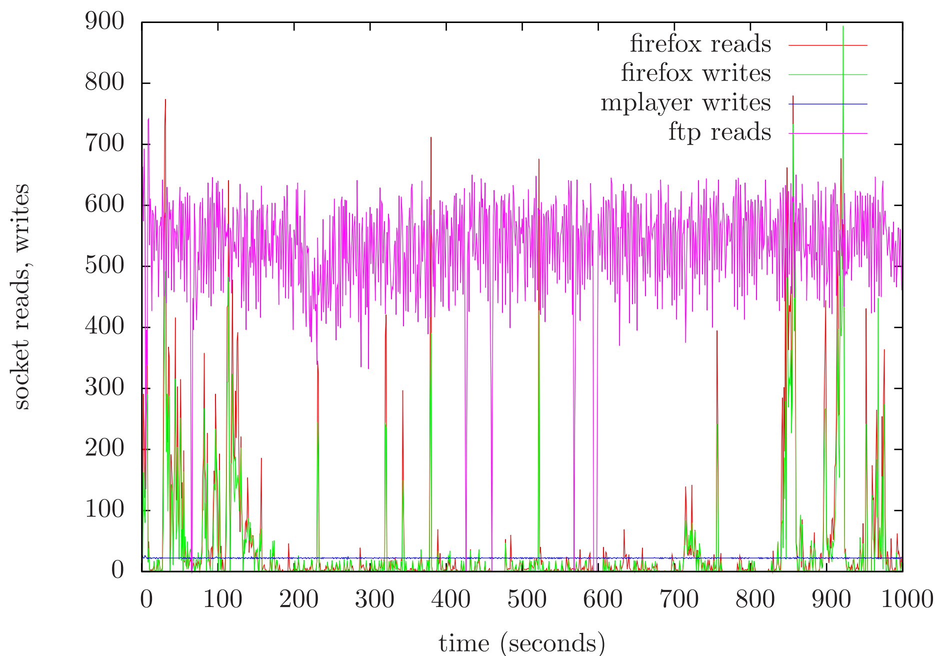


Figure 6.2: Socket reads and writes for `firefox`, `mplayer` and `ftp` processes

6.6 Assessment of features

Even though this is only a proof of concept implementation, the usefulness of the features used in the initial study will be described here.

6.6.1 Sockets

The socket features were expected to be indicative of X programs and network programs. It turned out that multimedia programs are not necessarily detectable this way because programs like `mplayer` and `firefox` use the *X video extension* [12] in conjunction with the *MIT shared memory extension* [14]. This combination of extensions to the X protocol allows client programs *residing on the same machine as the X server* to store graphics in memory and grant access to the X server to read this memory directly, instead of having to pass every pixel over the socket connection. The only information going over the socket connection is the client's request to have the server read a new frame from memory.

This is shown in Figure 6.2, where it is clearly visible that the `mplayer` process has a constant amount of socket writes per second: around 22 per second, corresponding to the frame rate. It performed no reads at all so this is not shown. The lower spikes in the `firefox` plot are when the user has scrolled or performed some other action that requires a redraw of the page. The bigger spikes correspond to HTTP requests for new pages or images on

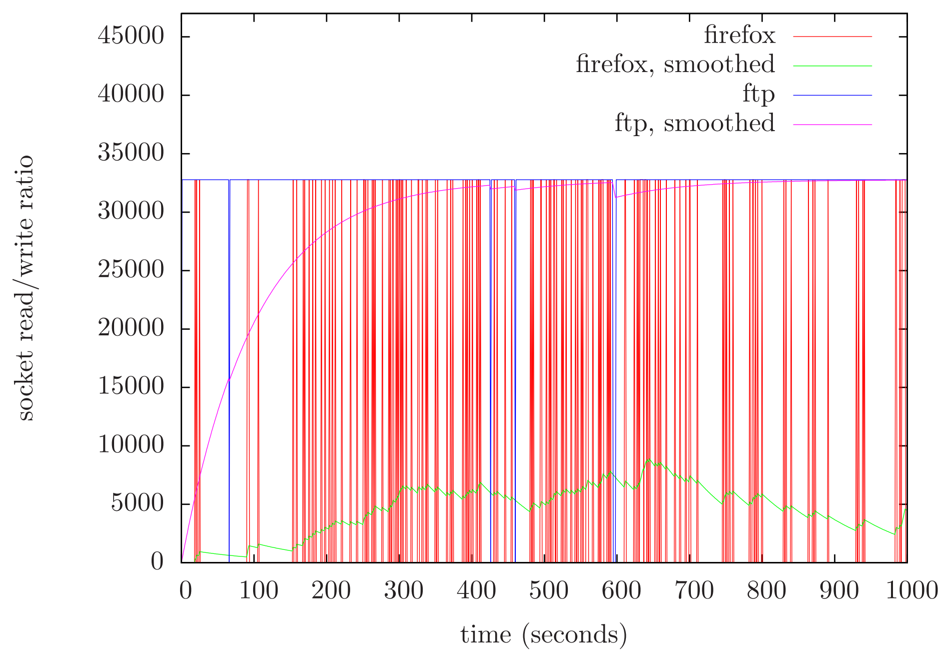


Figure 6.3: Socket read/write ratio for `firefox` and `ftp` processes

pages. The FTP download process has no writes at all during download, but it has a constantly high number of reads, which is visible in the plot as the wide bar in the center of the plot. This bar will be higher when the connection is faster and lower when the connection is slower, because the kernel will wake the process up more often if there is more data available, resulting in a new read, after which the process will go to sleep again. However, the reads divided by writes is almost constantly the maximum representable value of the features, resulting from a division by zero, the writes, (the system stores "infinity" as the maximum representable value) as can be seen in Figure 6.3. The `firefox` process fluctuates wildly between zero and the maximum value as a result of the fluctuation in reads and writes.

With the right weight settings in the memory layer this can be smoothed out and result in a fuzzy line through the center of the graph. The graph shows the smoothed-out curve for the memory node values for both `ftp` and `firefox` with a recurrent weight of 0.09 from the memory node to itself and a weight of 0.01 from the input to the memory node, thus resulting in the following set of equations to calculate the activation value of the memory node:

$$f(x_t) = \begin{cases} 0 & \text{if } t = 0 \\ f(x_{t-1}) * 0.09 + x_t * 0.01 & \text{otherwise} \end{cases} \tag{6.3}$$

Any particular network does not necessarily have to result in precisely these settings for the weights. The read/write ratio feature, when smoothed out, adds information to the raw read and write features: The firefox process is the only interactive one, and using this feature one can distinguish more easily between it and other socket-using processes than one would be able to do given only the reads or the writes.

6.6.2 Audio

About audio there is not much to say, besides that `mplayer` is clearly the only application that has any activity on audio writes, which confirms our expectation that this is a very distinctive feature for multimedia applications. The audio writes feature is displayed in Figure 6.4. The spikes and fluctuation in the figure are most likely the result of chance; if the process was scheduled on the CPU just before the feature was collected it would be higher and if the process was scheduled just after the feature sampling it would be lower.

The `mplayer` process performs no reads at all, so the read/write ratio is always 0. Here, too, the ratio seems irrelevant and could probably be removed. In case audio recording tools can be used, the read ratio may be useful but in other cases it could be removed too.

6.6.3 Terminal

The terminal features are more interesting. As can be seen in Figure 6.5, the reads and writes are almost identical for the two `vi` processes. They overlap so much that it is a little difficult to distinguish between the read and write graphs. As is to be expected, the process with a higher number of simulated keystrokes per minute has a higher number of reads and writes to the terminal device. However, the `mplayer`, `mencoder` and `ftp` processes also show terminal

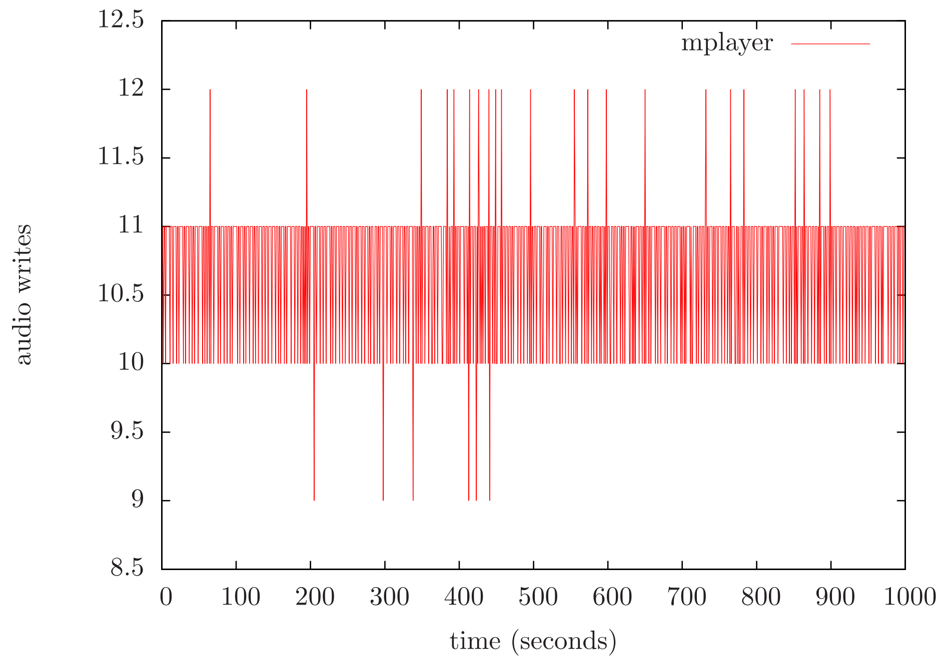


Figure 6.4: Audio writes for an `mplayer` process

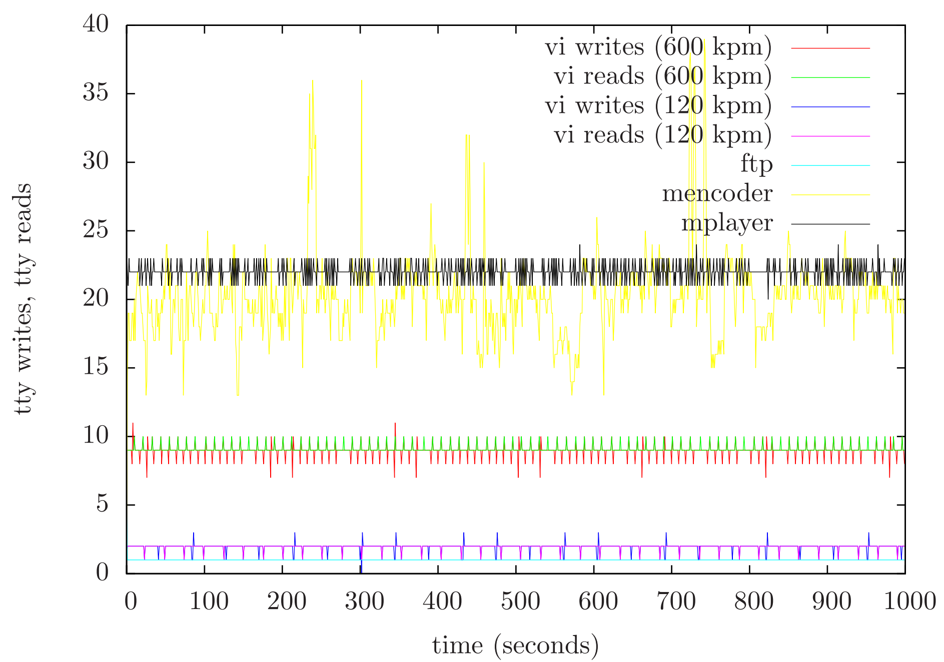


Figure 6.5: Terminal reads and writes

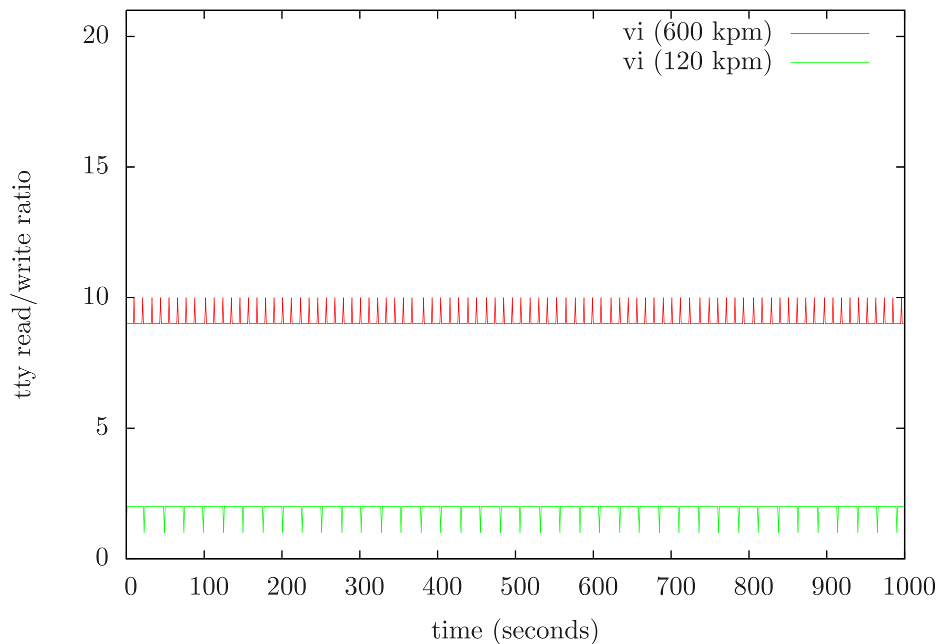


Figure 6.6: Terminal read/write ratio

writes. This is because both show status reports on the terminal. `mplayer` outputs the current position in the audio/video stream which is constantly updated, `mencoder` also shows the position which the encoding process is currently working on and `ftp` shows an ASCII progress bar which is updated whenever data has been read. It is interesting to see that programs with "typical" terminal interfaces like `vi` show less terminal output than programs which need to update a status report very often.

The read/write ratio, presented in Figure 6.6 again does not add a lot of new information. For all applications this ratio is zero, except the `vi` processes. The only thing we could possibly say about this is that the read/write ratio can reliably be used to distinguish "traditional" (i.e., console-based) interactive applications from other applications.

6.6.4 Filesystem

Almost every program in our test set uses the filesystem, even if the program does not write to regular files. This is the case in Unix because there, devices are accessed as files called "special files" or "device files" [40]. For example, mplayer does not write anything to a regular file, but its audio output is written to the device file `/dev/audio`. As discussed above, it also issues terminal writes, but the terminal driver is also accessed through a device file, which is in this case called a "pseudo device", since (on modern systems which can multiplex many terminal windows) it does not refer to a physical piece of hardware. Thus, by the time the terminal subsystem is notified of a write to a terminal device, this

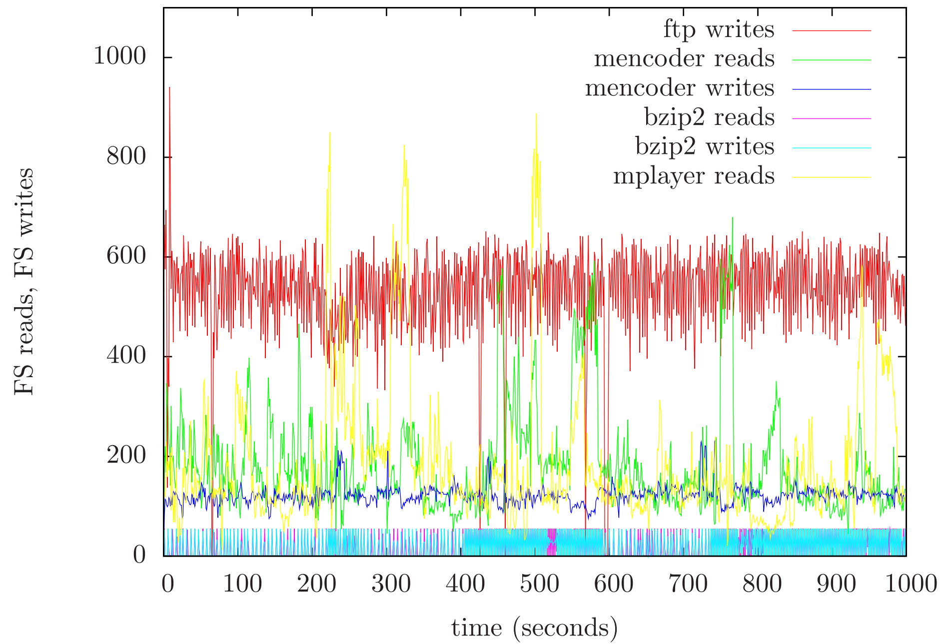


Figure 6.7: Filesystem activity on the high end

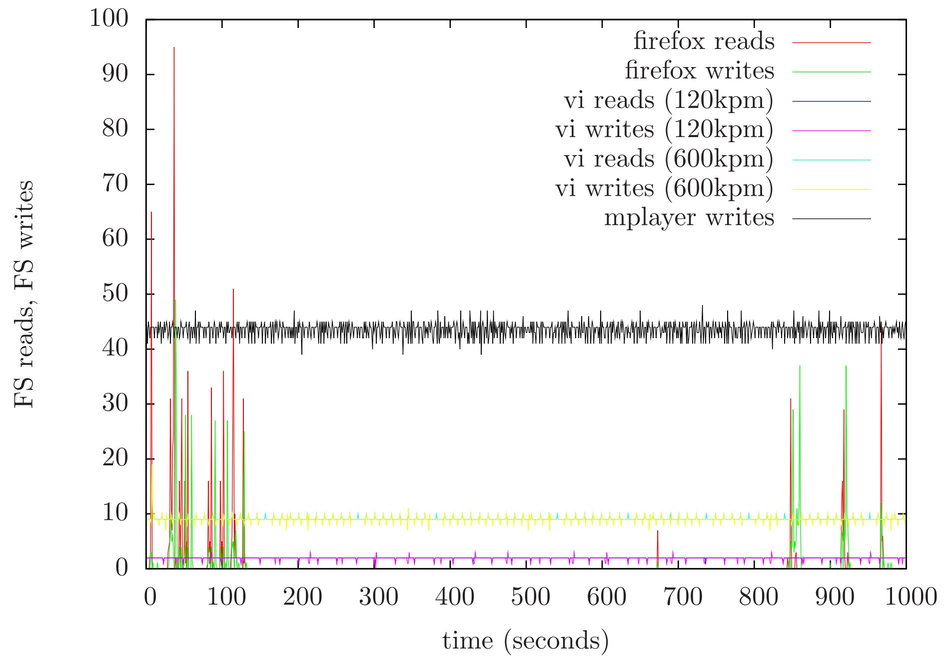


Figure 6.8: Filesystem activity on the low end

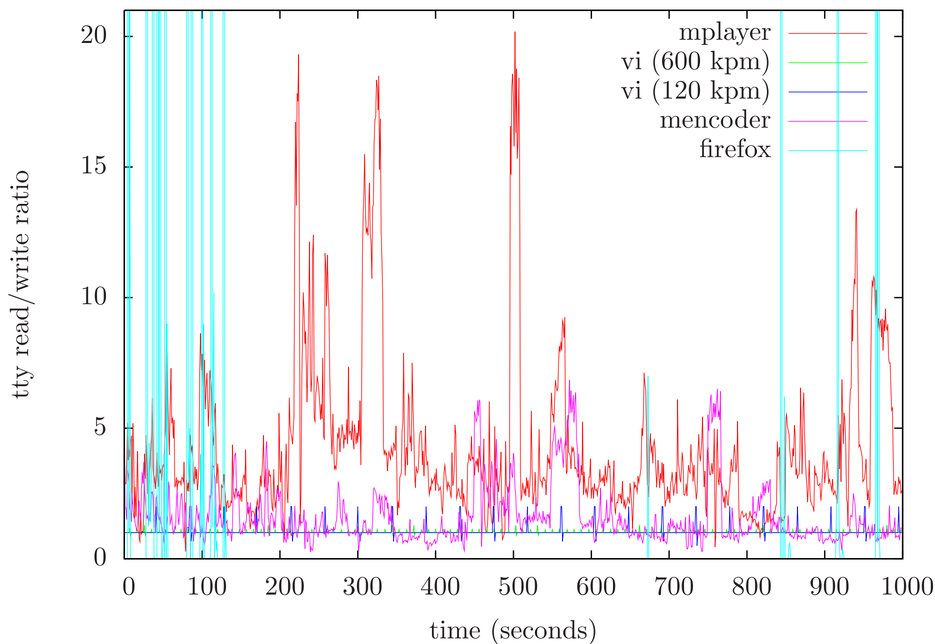


Figure 6.9: Filesystem read/write ratio

has already passed through the subsystem that deals with the filesystem.

The only application that does *not* interact with the file system is, oddly enough, the `find` application. Even though `find` searches through the filesystem, it does not actually read from or write to any files. Because accessing the contents of directories is not done through the same system calls as reading or writing files, the filesystem feature handler does not register any activity for `find` processes. This could be fixed by adding hooks for the features in the corresponding kernel system call implementations.

In Figure 6.7, we see filesystem activity of applications which issue a large number of filesystem requests per second. `ftp` shows prominently in this graph, as it simply writes back to disk whatever it reads from the network. Compare this band of the graph with Figure 6.2 and we see this relation clearly. The `bzip2` application has about an equal number of reads and writes, which overlap almost entirely. The `mencoder` process writes out a quite consistent stream of information, however the reads include peaks of activity. It may be that these peaks correspond to small transitions in frames which means the decoding process can fetch the next frames quickly without having to do a lot of decoding work, but this is only speculation. `mplayer` itself also shows peaks which could be of the same kind.

For the *write* behaviour of `mplayer`, we turn our attention to Figure 6.8, which shows the low end of the filesystem activity. From this graph, the `bzip2` activity was excluded because it would obscure the other graphs, since it has such a wide graph. The `mplayer` graph ends up in this figure because there is a lot less data that `mplayer` needs to write (only audio, not video), resulting in

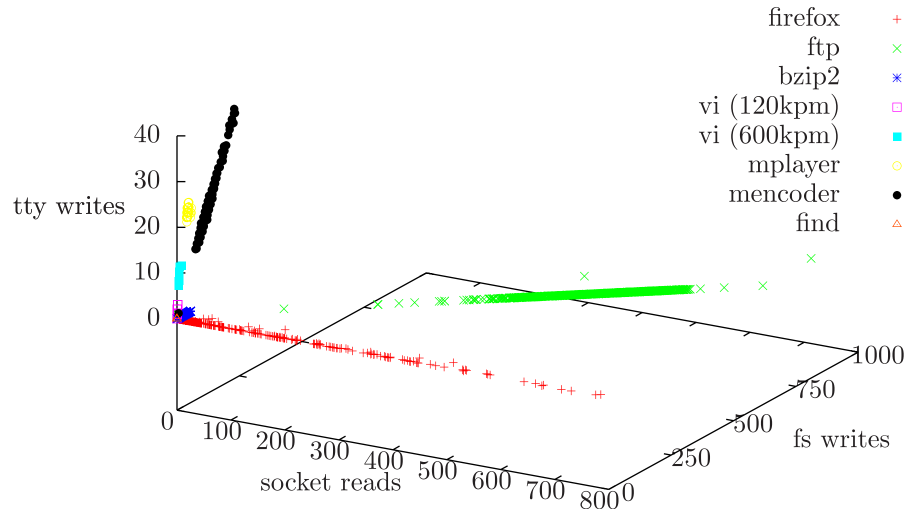


Figure 6.10: Filesystem, socket and tty features

smaller numbers than for the read activity. It is also much less erratic because the audio output must be very constant; the movie's audio layer is encoded at a constant frequency which needs to be maintained to produce the correct sounds.

We also see that `firefox` has very few filesystem requests. The few peaks are when web requests were made, and it read or wrote the received information from or to its page and image cache. During reading and scrolling of the page, it does not exhibit any disk activity. Lastly, we have a look at the `vi` processes. If we compare these graphs to Figure 6.5, we see that these do indeed, as noted at the beginning of this section, correspond to the writes to the terminal (pseudo-) device file. Here, too, the reads and writes overlap so much they become almost indistinguishable.

The filesystem read/write ratio graph, presented in Figure 6.9 contains a bit more information than the other read/write ratio graphs. On the left and right sides of the graph, we can see the "spikes" up to the maximum possible value (which were cut off to keep the rest of the graph readable) for the `firefox` process. These occur during time frames when the process reads but does not write. In the lower part of the figure, we also see the `vi` graphs from Figure 6.6 repeated here, as expected. The `mplayer` and `mencoder` follow the graph of the reads for these processes. Again, we can conclude that the read/write ratio feature does not add any information and could probably be discarded from the feature set.

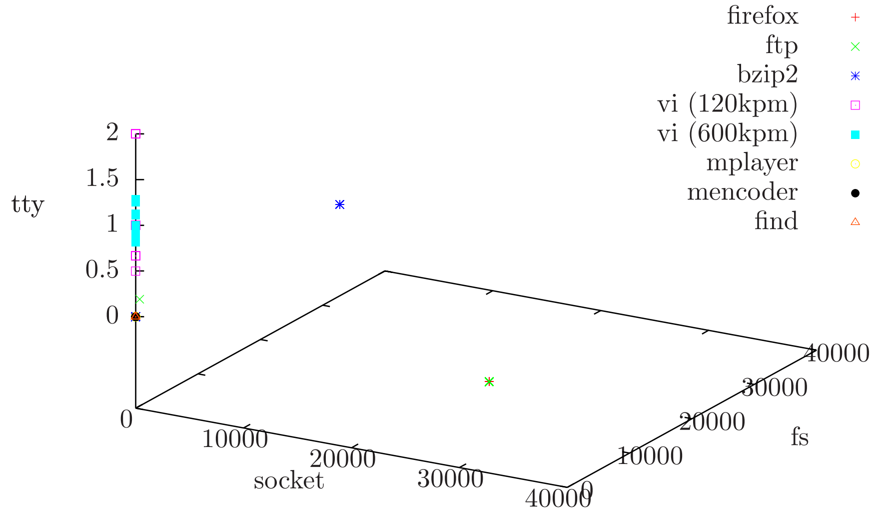


Figure 6.11: read/write ratios

6.6.5 Application classification

In this final subsection we will try to visualize the classification process for applications. In Figure 6.10 we see the data points for our processes plotted across three dimensions. The axes that can be seen are those of the number of socket reads, filesystem writes and writes to the terminal device. From this visualization it again becomes clear that `mencoder`, `bzip2` and `ftp` can best be identified by their filesystem writing activity. The correspondence between socket reads and filesystem writes for `ftp` which we noted earlier becomes very prominent in this graph. There is another correspondence between two features that becomes clear here, which we had not noticed before; namely that when `mencoder` writes more to the disk, it also outputs more information to the terminal. This is logical, as it has to update the percentage that it has completed more often if it has been able to write more to disk. What is interesting is that `firefox` can be better identified by its socket reads than became clear from Figure 6.2, the 2d graph that displayed socket read and write activity across time.

Further, the vi process belonging to the fast "typist" can be identified quite well. However, the other `vi` process ends jumbled up with the `find` process and some parts of the `firefox` and `bzip2` processes. Interestingly, the `mplayer` process is reasonably easy to pick out, even without the audio feature.

To test the conclusion that the read/write ratios are mostly useless for classifications, we tried to visualize the feature space with the read/write ratios for the filesystem, sockets and terminal features. This is displayed in Figure 6.11.

As we can see, there is hardly any information in this plot. All the data points collapse on just a few values, and clusters from one process split far apart, while values from different processes are clustered together. The only reasonably informational feature is the terminal read/write ratio. This confirms the conclusion made in the preceding sections, that the ratio is mostly a useless feature and should be left out, except for the terminal and possibly the smoothed-out socket feature (see Figure 6.3 and the accompanying description).

Chapter 7

Conclusion and future work

This thesis described how a neural network was integrated in an operating system. The implementation was described in detail; how process features can be obtained and used as input for a neural network, and how the output of this network can be used as a classification of the application. This output could even be used directly as a scheduler priority value. The goal of this was to recognize multimedia applications automatically and schedule on a desktop system with a higher priority than less important "background" applications. Testing results showed that this was reasonably successful, but that the features could be better.

Investigating the features resulted in the insight that the read and write features are quite useful for the processes that were studied, except for audio reads, but this feature could prove useful in situations with different types of processes. The read/write ratio was useless in all cases except the socket case, as no information was added by it that was not also in the reads or the writes themselves. It turned out that the socket read/write ratio feature, when smoothed out a little, was the distinguishing feature for the web browser process. The multimedia process could very easily be picked out by writes to the audio device and pure downloading could be picked out by the large number of reads from the network and writes to disk. The application that walked the filesystem hierarchy (`find` in this study) was the only application that showed no activity on any of the features. This could be remedied by modifying the filesystem feature such that it would also note when an application simply traversed the file hierarchy, or by adding a new feature that does this.

The implementation is a proof-of-concept, but it was designed in a very modular fashion. This makes it an ideal platform for testing new ways of classifying processes without spending a disproportionate amount of time modifying or rewriting the existing scheduler. It allows researchers to freely experiment and to evaluate the influence of different metrics on process scheduling performance. Adding a new feature to the network is trivial and the effects can be observed instantly with existing tools. Most new tools or modifications to measure or improve performance are automatically independent of particular features, benefiting everyone. Hopefully, this helps giving research on uniprocessor scheduling the impulse it needed.

Using a neural network to assign priorities instead of requiring the user to manually assign them is a design that works pretty well on the desktop. The

results show that it visibly improves multimedia performance. In retrospect, on a server it should be less useful, because the variety of applications running on any particular server is much smaller, and it could be more effective to let an administrator assign these priorities manually. Having the neural network scheduler from this paper in a scheduler would not preclude this possibility, because the server's administrator could simply not load a network at all, thereby keeping manual control of all priority assignment.

Unfortunately, one of the most important aspects of scheduling multimedia applications under Unix was not touched upon by this research simply because it would be too big an undertaking. This would be gathering process feature data from the X windowing system. This is important because multimedia applications are often graphics-intensive, and under Unix X is the system that draws windows and all graphics output goes through X. For example, the "Human-Centered" metrics explored in the study of Etsion et al. (described in [23]) could be implemented as features for a neural network. Hopefully, this will provide some features that can strongly improve the scheduling on multimedia desktop computers. This would require substantial modification of the X Window system as well as the kernel in order to get the data from X to the kernel.

Another approach for making multimedia information available to the scheduler would be to move the scheduler to an external process. This process could decide the "nice values" of other processes. This would be advantageous for desktop environments, for example. Projects like the KDE or the GNOME desktop environments could build a userspace-only scheduler. The advantage here is that if every application's operations are performed through the abstraction layers provided by these desktop environments, more multimedia information is available to the scheduler. Because the neural network part of the scheduler is in userspace, there is less overhead in the kernel. For example, desktop environments often have an audio multiplexer application.[1] These multiplexing processes should be able to register their feature with a scheduler in such a way that the feature can be attributed to the process that requests its service. As speculated in section 6.1, OSes with a microkernel architecture should not need to implement such a complex system because they are *designed* around small processes that perform system tasks and communicate with each other. This is a very promising architecture for implementing a neural network scheduler.

Another interesting subject for future research would be to use unsupervised learning techniques (for example Kohonen's Self Organizing Map, as described in [29]). This could be done by having this network identify classes of applications and then presenting the user with these classes along with the names of the processes in that class. Then the user could assign the desired priority for each class. The memory system presented in this paper could be maintained by creating a hybrid backpropagation network/Kohonen map. This can consist of a memory and an input layer which are connected to an output layer of the same size as the input layer. The resulting output can then be used as input to the Kohonen map.

It is unfortunate that AI developments do not seem to have influenced researchers in the scheduling field and the Operating System field in general. It would be interesting to see what results would come from a higher degree of

[1] For example, KDE uses aRts (`http://www.arts-project.org/`) and Gnome uses Esound (`http://developer.gnome.org/doc/whitepapers/esd/`) to multiplex their audio.

collaboration between the research areas in the future.

7.1 Personal notes

In this section I will describe what I have personally learned from this project.

Implementing a neural network appears to be quite easy, but it becomes very difficult when things go wrong. Debugging a neural network that does not crash but does contain a bug is extremely difficult, in my opinion. If there is no particular point where anything goes wrong, one has to step through the network code with a debugger. Even this is not necessarily very helpful if the network appears to learn correctly in the first number of epochs. The little steps do not give a "big picture" of what is happening, and one can easily get lost in details.

All of this means that the implementation of the neural network took a very long time. It took so much longer than expected that there was not a lot of time left for large-scale experimentation and measurements of effectiveness of features. Also because of this, it was very disappointing to see that the evaluation of the work focused *only* on the thesis and *not* on what constituted the bulk of the work and in which most time was invested, namely the code.

Before this project, I was under the impression that neural networks are formalized in a mathematical model. I learned that this may be true in theory, but that in practice there are many common issues that can only be solved empirically. For example, determining the best number of hidden units, and the best values for the learning rate and momentum are a matter of trying out what works best for the particular test set under consideration.

Another issue was that a network's activation function can make a world of difference. When the test set has continuous (instead of binary) target outputs, the best activation function is a simple linear function. In retrospect, this is very obvious, but at the time of implementation it was something I had not thought about. I also did not expect that backprop would be such a slow algorithm. It is the standard algorithm taught in neural network courses and tutorials, but it is unacceptably slow for real-world applications. If this was known beforehand, perhaps a different training algorithm could have been chosen.

The fact that floating point instructions can not be used in an OS kernel was quite surprising to me. After a short discussion on the NetBSD kernel mailing list, it became clear that there is no decent solution for this problem in a portable OS. Certain systems might have no *FPU* (Floating Point Unit) at all and on most architectures, saving and restoring the FPU registers are such slow operations that the overhead is unacceptable. Implementing fixed point is not very difficult, but it is very prone to errors. This was the source of quite a number of bugs.

The helper `progstarter` program was written in a bit of a hurry, as well as the `nnread` and `nnwrite` modules (which read and write a network from a file). I did not spend much time thinking about them and wrote the code quickly, expecting that I would not have to work with the code much after creating them. Unfortunately, this assumption was not true. After some attempts to extend `nnread` and `nnwrite`, I decided to completely rewrite these functions and also change the file format. The code is a lot cleaner now, but the file format is still very much a hack. The total time I spent writing and rewriting these modules

could probably have been spent better by designing a good, extensible format and writing a real parser for it using, for example, `lex` and `yacc` instead of `fscanf`. `Progstarter` was extended without having to rewrite it, but these changes cost more time than they should.

Appendix A

Correction of the Turing-completeness proof for NNs

The proof given in the diploma thesis [18] is not entirely correct. There is a small error that will be discussed here. The author apparently forgot that an aspect of his model required more bookkeeping than he uses in his proof.

The paper consists of a long path of proofs by simulation, starting from Turing machines, simulating these in stack machines, going from there to counter machines, adder machines, alarm clock machines, restless counters and finally sigmoidal neural networks, thus proving that these neural networks can simulate Turing machines. In other words, sigmoidal neural networks are *Turing-complete*. I will assume the reader has read the paper in question.

The flaw is in the proof of **Proposition 3.3.1**. To repeat the proposition:

Proposition 3.3.1 An acyclic k-adder machine can be simulated by a $(k^2 + k + 4)$-restless counter machine.

This proposition should be modified to read:

Proposition 3.3.1' An acyclic k-adder machine can be simulated by a $(k^2 + k + 8)$-restless counter machine.

We already know from **Proposition 3.2.1** that an acyclic k-adder machine D that computes in time T can be simulated by a $(\frac{k^2+k}{2}+2)$-alarm clock machine in time $O(T^3)$.

A.1 Background

Like in the original paper, we are simulating an *alarm clock machine* in a *restless counter machine*. An alarm clock machine (which is in turn used to simulate an *adder machine*) consists of a number of timers which all hold a number. This number represents the number of time steps it takes for them to wind down

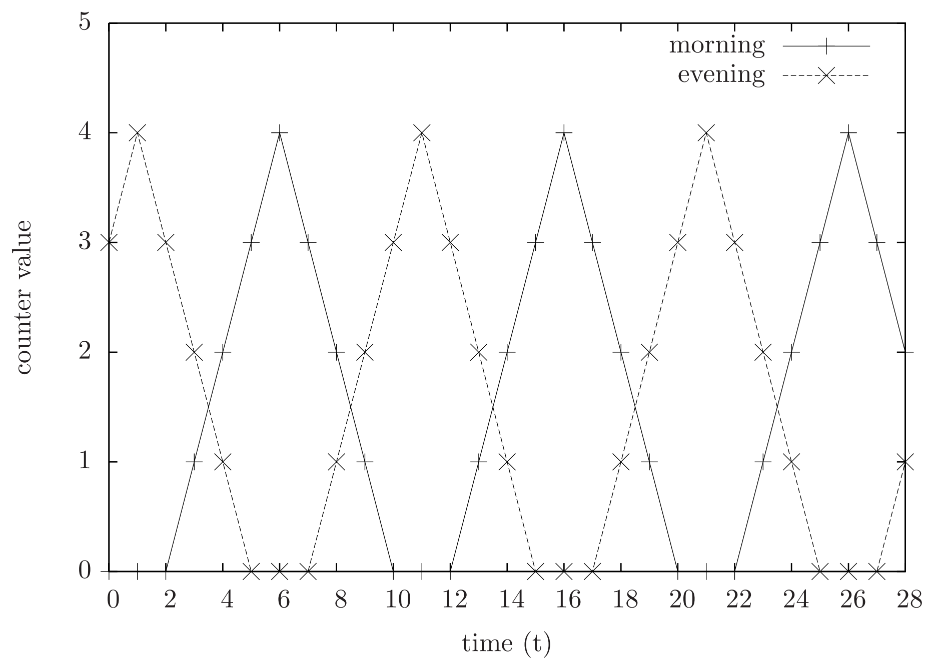


Figure A.1: One simulated alarm clock

and "sound an alarm". In a normal situation, every time step every clock's number decreases by one. The machine is described by a transition function $\delta : \{0,1\}^{5k} \rightarrow 2^{\{\texttt{delay}(i),\texttt{lengthen}(i):1 \leq i \leq k\} \cup \{\texttt{halt}\}}$. The domain of δ is $\{0,1\}^{5k}$ because we need to remember the last 5 time steps for our simulation of an adder machine. A 1 at position $5k+i$ in the vector means alarm clock A_i alarmed at timestep k. The information about an alarm clock's period and time to alarm is the input for the machine. The alarm clocks can be `delay`-ed or `lengthen`-ed by finite control, which means either the time to alarm is increased by one (`delay`) or both the time to alarm and the period are increased by one (`lengthen`).

To simulate an acyclic adder machine D with adders $D_1, ..., D_k$, we use an alarm clock machine with clocks $A_0, A_1, ..., A_k, A_{00}, A_{ij} : 1 \leq i < j \leq k$. The clocks $A_1, ..., A_k$ are used to simulate $D_1, ..., D_k$, the clocks A_{00} and A_0 are used to synchronize the machine. If any clock alarms simultaneously with A_{00}, it gets delayed. If A_{00} and A_0 alarm simultaneously, the delay and lengthen operations are completed and a new cycle can begin. For more details, please refer to [18].

A.2 Proof

In the restless counter machine, every alarm clock A_i is simulated by two restless counters, a *morning* (M_i) and an *evening* (E_i)[1] counter. The period of the clock is represented by the duration of the cycle of the two counters. The time to alarm is represented by the time it takes the *morning* counter to hit zero. See Figure A.1 for a depiction of this process.

Two counters for every alarm clock in the alarm clock machine gives us the following number of counters if we use the alarm clock to simulate an acyclic k-adder machine:

$$(\frac{k^2+k}{2}+2)*2 = (k^2+k+4) \tag{A.1}$$

An alarm clock that goes off corresponds to its *morning* counter hitting 0. An alarm clock's counters are said to be *in the morning* from the point the morning counter starts counting upwards from 0, and we are *in the evening* from the point the evening counter is counting upwards from 0 (intuitively speaking, alarm clocks generally go off in the morning).

M_i and E_i will be in a cycle with a phase shift of $2p$ time steps as described in the original paper. It will count up to $2p-1$ and then it will count back to 0 and stay there for two additional time steps, before it will start to count up again, as we can observe in Figure A.1. To be able to distinguish the three states at which the counter is 0 (the *zero states*) when we refer to them, two new zero values are introduced, $0'$ and $0''$. When a counter reaches 0, finite control will wake up and set the associated counter's direction to *Down*. So if E_i hits 0 at time t, M_i starts to drop at $t+1$, and vice-versa. This causes the pair of counters to cycle with period $4p$, simulating one alarm clock of period p. The mistake in the original paper is in the implementation of the `delay` and `lengthen` operations. These work as follows:

- **Delay.** To `delay` we switch the direction of the corresponding morning and evening counters for a duration of two time steps. Since finite control

[1] This includes the A_{ij} counters.

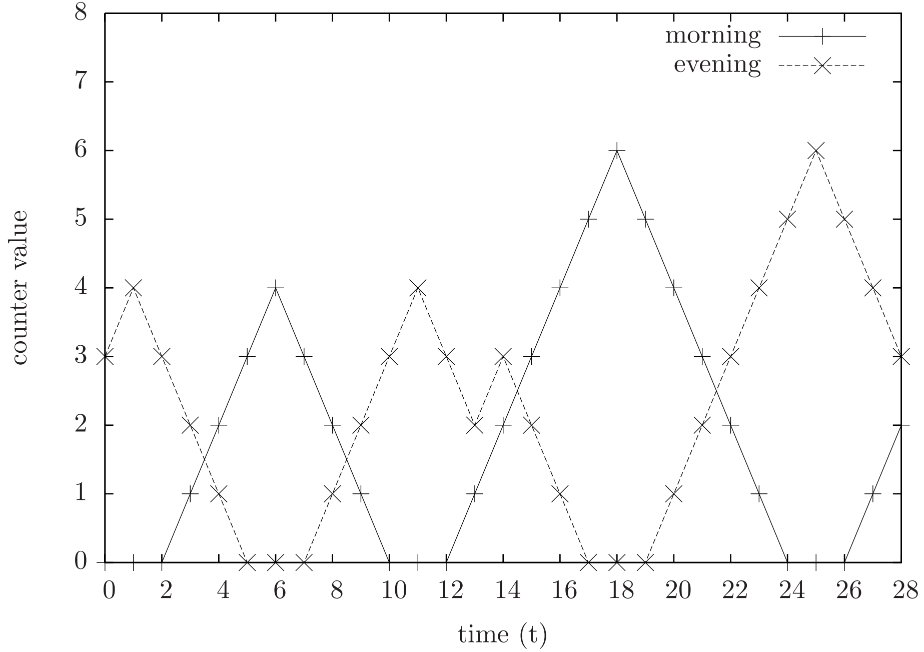
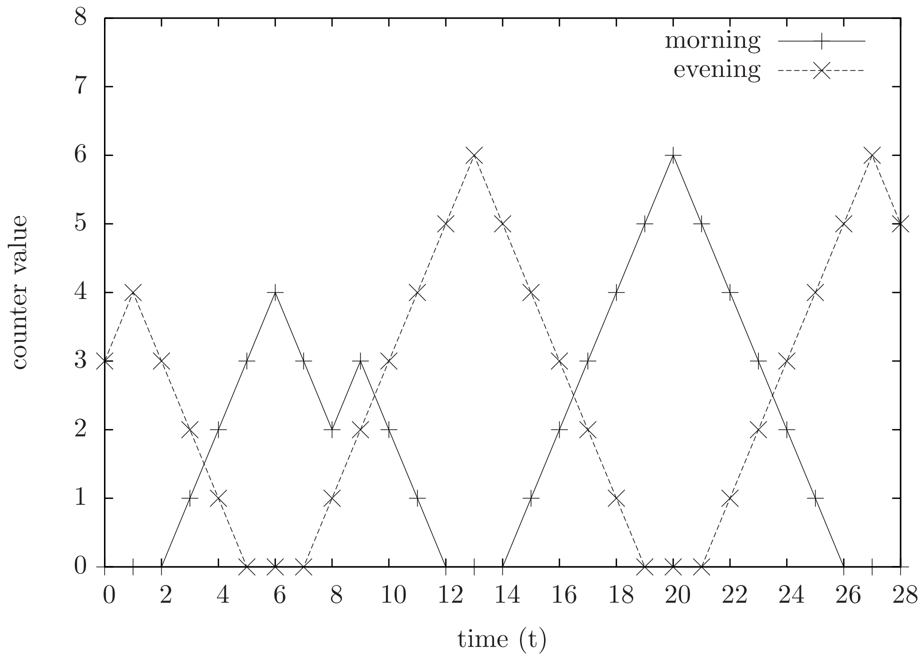


Figure A.2: Lengthening a clock in the morning

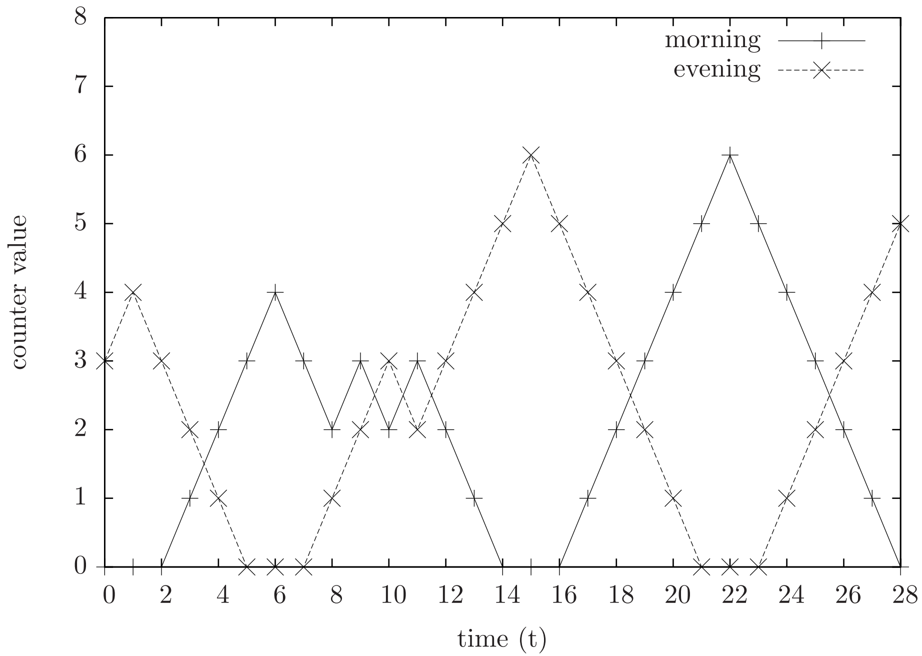
only wakes up when a counter hits 0, there must be a timer that hits 0 at the end of the two time steps in order to be able to revert the direction back to normal again. The original paper does *not* provide a counter like this. We will first look at the other operation, `lengthen`, before we try to fix this.

- **`Lengthen` while in the morning.** To `lengthen` A_i when in the morning, its *evening* counter, E_i, will be incremented for one timestep and then changed back to normal. This causes an (intended) delay which results in the evening counter dropping to 0 four timesteps later than would normally be the case. See Figure A.2 for clarification. Before the delay operation, we see that the period is 10 since after hitting 0 it takes an additional 10 time steps for the morning counter to hit 0 again. We will need another extra counter that hits zero to be able to turn it back around.

- **`Lengthen` while in the evening.** `Lengthen`-ing A_i when it is in the evening is more problematic. Simply changing the evening counter like we do when in the morning does not work, because the morning counter hits zero before the evening counter. The delay that is implicit in the `lengthen` operation does not take effect then. The morning counter will simply continue decreasing until it hits 0, as if nothing happened. This means the period will be longer from then on. Changing the morning counter directly like we changed the evening counter is effective, but the delay caused by this is only half a delay. Consider Figure A.3(a). We



(a) Wrong attempt



(b) Correct attempt

Figure A.3: Two attempts at lengthening a clock in the evening

t	c_0	c_1	c_2	c_3	Partial output (c_0, c_1, c_2, c_3)
0	$(0, Down)$	$(1, Down)$	$(0'', Up)$	$(0', Up)$	(U, N, D, N)
1	$(0', Up)$	$(0, Down)$	$(1, Down)$	$(0'', Up)$	(N, I, N, D)
2	$(0'', Up)$	$(0', Up)$	$(0, Down)$	$(1, Down)$	(D, N, I, N)
3	$(1, Down)$	$(0'', Up)$	$(0', Up)$	$(0, Down)$	(N, D, N, I)
4	$(0, Down)$	$(1, Down)$	$(0'', Up)$	$(0', Up)$	(U, N, D, N)

Table A.1: State changes of the extra control counters

can see that the original period is 10. When we do a `lengthen` operation on the morning counter (similar to the evening situation), it will hit 0 two timesteps later, making the effective period of the previous cycle 12. On the other hand, the period has become 14, as we can see from the morning counter's graph to the right of $t = 12$. This means the initial delay is *half* from what it is supposed to be. Recall that one timestep of the alarm clock machine is simulated by four timesteps of the restless counter machine. Thus, the delay is *half a timestep* of the alarm clock machine.

The correct way to get the implicit `delay` when in the evening is to delay the morning counter *twice*. Doubling the delay of Figure A.3(a) causes it to be a delay of one timestep in the alarm clock machine. This causes a new problem. The lengthening was correct in our previous attempt, only the delay needed to be fixed. By doing the operation twice, the lengthening is doubled too. This means that a `lengthen` operation would cause a lengthening of *two* timesteps. To remedy this, we also have to cause a delay in the evening counter, as shown in Figure A.3(b).

Only when A_{00} hits 0 in the alarm clock machine there will be an action of the finite control, and this translates to M_{00} hitting 0 in our restless counter machine. Only then will there be a `delay` or `lengthen` operation taking place. The `lengthen` operation in the evening requires a counter to hit zero *every* timestep, because it requires four changes in direction. So, we will need at least three extra counters, one for each timestep after M_{00} hits 0 (one or two will not be enough, because we can only take action on a counter hitting 0. $0'$ and $0''$ do not wake up finite control). Naturally, the other operations can "borrow" these counters.

Because every timestep of the alarm clock machine is simulated by four timesteps of the restless counter machine, we want our newly added "control counters" to oscillate with period 4. The state changes can be observed in detail from Table A.1.[2] Because M_{00} does not necessarily hit zero every simulated

[2] Here the *Up* direction is used in all zero states except the first. The original paper does not clearly specify if 0' is greater than 0 (in the ordering laid down by *Up* and *Down*) or if the *Up*/*Down* directions even have any effect on counters in the zero states. Since they *must* remain in the zero position for two timesteps, the actual direction encoded is irrelevant. It is unambiguous which state the counter will be in after any of the zero positions. The only important aspect is that when it *leaves* the last zero state, $0'$, it should go to the state $(1, Up)$. Using *Up* throughout the zero states except the first is perhaps the best choice conceptually, because directions may only be changed at zero events. This way, counters can switch their own direction unaided from *Down* to *Up* when hitting 0, thus going through these phases:

time step (it can be `lengthen`ed too), we require an extra counter to keep our counters under control. This is the fourth counter in Table A.1. Essentially it keeps c_1 from spinning off when M_{00} does not hit 0. The last column displays a partial output of the mapping that is the restless counter machine, namely the values for the four counters under consideration. N means *No change*, I means *Increase from next timestep onwards* and D means *Decrease from next timestep onwards*. All that is left to do when these changes are applied is to add mappings to the function that defines the machine to ensure the `delay`ed or `lengthen`ed operations are completed by reversing the affected counters again.

In conclusion, we add these four counters to Equation A.1 and we end up with:

$$(k^2 + k + 4) + 4 = (k^2 + k + 8) \tag{A.2}$$

which is what our modified version of the original proposition (**Proposition 3.3.1'**) reads.

$(0, Down) \rightarrow (0', Up) \rightarrow (0'', Up)$. from 0, then going Up because it changed. From this follows that $0 > 0'$ and $0' < 0''$.

Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevenian Jr., and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer conference*, pages 93–112. USENIX Association, June 1986. 6.2

[2] Aristoklis D. Anastasiadis, George D. Magoulas, and Michael N. Vrahatis. A new learning rates adaptation strategy for the resilient propagation algorithm. In *Proceedings of the 12th European Symposium on Neural Networks (ESANN-04)*, April 2004. 6.2

[3] Thomas E. Anderson, Brian N. Berschad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *ACM Transactions on Computer Systems*, volume 10, pages 53–79. University of Washington, ACM, February 1992. 2.4

[4] Apple Computer, Inc., 1 Infinite Loop, Cupertino, CA. *Kernel Programming*, 2005. Available from `http://developer.apple.com`. 6.2

[5] Moshe Bar. The linux process model. *Linux Journal*, 71, March 2000. 2.4

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 164–177. ACM, December 2003. 2

[7] Paul Barton-Davis, Dylan McNamee, Raj Vaswani, and Edward D. Lazowska. Adding scheduler activations to mach. Technical Report 92-08-03, Department of Computer Science and Engineering University of Washington, Seattle, Washington 98195, March 1993. 2.4

[8] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*, pages 41–46. USENIX Association, April 2005. 2

[9] William G. Bulgren and Lee-Ho Hwang. A simulation study of time-slicing in non-exponential service environments. In *ANSS '80: Proceedings of the 13th annual symposium on Simulation*, pages 161–180, Piscataway, NJ, USA, 1980. IEEE Press. 2.3

[10] Leslie Burkholder. The halting problem. *SIGACT News*, 18(3):48–60, 1987. 2.2

[11] Michael I. Bushnell. The HURD: Towards a new strategy of os design. In *GNU's Bulletin*. Free Software Foundation, 1994. http://www.gnu.org/software/hurd/hurd-paper.html. 6.2

[12] David Carver. *X Video Extension Protocol Description version 2*. Digital Equipment Corporation, July 1991. 6.6.1

[13] Thomas Murray Cook. Schedule-constrained job scheduling in a multiprogrammed computer system. In *Proceedings of the 7th conference on Winter simulation*, volume 2, pages 675–685, 1974. 2.2

[14] Jonathan Corbet. *MIT-SHM — The MIT Shared Memory Extension How the shared memory extension works*. National Center for Atmospheric Research. 6.6.1

[15] H. M. Deitel, P. J. Deitel, and D. R. Choffnes. *Operating Systems*. Pearson Education, 2004. 2.1

[16] Jayanta K. Dey, James F. Kurose, Don Towsley, C. M. Krishna, and Mahesh Girkar. Efficient on-line processor scheduling for a class of iris (increasing reward with increasing service) real-time tasks. In *SIGMETRICS '93: Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 217–228. ACM Press, 1993. 2.6

[17] Jeff Dike. User-mode linux. In *Proceedings of the 5th Annual Linux Showcase & Conference*, pages 3–14. USENIX Association, November 2001. 2

[18] Ricardo Joel Marques dos Santos Silva. On the computational power of sigmoidal neural networks. Master's thesis, Universidade Técnica de Lisboa, July 2002. 1.2, 5.6.2, A, A.1

[19] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990. 4.1

[20] Tzilla Elrad, Jinlong Lin, and Douglas J. Cork. Evolutionary computation for scheduling controls in concurrent object-oriented systems. *International Journal of Computers and Their Applications*, 5(3):11–20, Sept 1998. http://www.esep.com. 3.1

[21] Ralf S. Engelschall. GNU portable threads. http://www.gnu.org/software/pth/pth.html. 2.4

[22] Yoav Etsion, Dan Tsafrir, and Dror G. Feitelson. Effects of clock resolution on the scheduling of interactive and soft real-time processes. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 172–183. ACM Press, 2003. 5.1

[23] Yoav Etsion, Dan Tsafrir, and Dror G. Feitelson. Desktop scheduling: how can we know what the user wants? In *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 110–115. ACM Press, 2004. 5.6.4, 6.2, 7

[24] Jason Evans and Julian Elischer. Kernel-scheduled entities for FreeBSD. Technical report, The FreeBSD Project, 2003. 2.4

[25] Scott E. Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, Carnegie-Mellon University, September 1988. 6.2

[26] Wu-chun Feng and Jane W.-S. Liu. Algorithms for scheduling real-time tasks with input error and end-to-end deadlines. *IEEE Trans. Softw. Eng.*, 23(2):93–106, 1997. 2.6

[27] Mario J. Gonzalez, Jr. Deterministic processor scheduling. In *ACM Computing Surveys*, volume 9, pages 173–204, September 1977. 2.2

[28] Brian Hook. An introduction to fixed point math. *Book of Hook*, September 2004. Available from `http://www.bookofhook.com`. 5.6.1

[29] Teuvo Kohonen. The 'neural' phonetic typewriter. *Computer*, 21(3):11–22, 1988. 3.5, 7

[30] Robert Love. *Linux Kernel Development*. Novell Press, 2nd edition, 2005. 2.4

[31] Jared D McNeill. NetBSD/usermode. E-mail to the NetBSD tech-kern mailinglist, December 2007. `http://mail-index.netbsd.org/tech-kern/2007/12/28/0001.html`. 2

[32] Microsoft Corporation. MS Windows NT kernel-mode user and GDI white paper. Available from `http://technet.microsoft.com`. 6.2

[33] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX scheduler unacceptable for multimedia applications. In D. Shepherd, G. Blair, G. Coulson, N. Davies, and F. Garcia, editors, *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, volume 846 of *Lecture Notes in Computer Science*, pages 41–53, Lancaster, U.K., November 1993. Springer-Verlag. 1

[34] Jason Nieh and Monica S. Lam. A smart scheduler for multimedia applications. *ACM Trans. Comput. Syst.*, 21(2):117–163, 2003. 2.6

[35] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the USENIX Summer conference*, pages 247–256. USENIX Association, 1990. 1.1.1

[36] Christopher Provenzano. Portable threads library. `ftp://sipb.mit.edu/pub/pthreads/`. 2.4

[37] QNX Software Systems Ltd., 175 Terence Matthews Crescent, Kanata, Ontario, Canada. *System Architecture (QNX Neutrino 6.2.1)*, 2003. Available from `http://www.qnx.com`. 6.2

[38] J. R. Quinlan. Comparing connectionist and symbolic learning methods. In *Computational Learning Theory and Natural Learning Systems*, volume I: Constraints and Prospects, pages 445–456. MIT Press, 1994. 3.2, 3.5

[39] Martin Riedmiller. Rprop - description and implementation details. Technical report, University of Karlsruhe, January 1994. 6.2

[40] Dennis M. Ritchie and Ken Thompson. The UNIX time sharing system. In *Communications of the ACM*, volume 7. Bell Laboratories, Association for Computing Machinery, July 1974. 6.6.4

[41] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 285–298, New York, NY, USA, 1995. ACM Press. 1.1.1

[42] R. Russel. Hackbench: A new multiqueue scheduler benchmark, December 2001. `http://lkml.org/lkml/2001/12/11/19`. 6.4

[43] Stuart J. Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc, 1995. 3

[44] Rudy Setiono and Huan Liu. Symbolic representation of neural networks. *IEEE Computer*, 29(3):71–77, March 1996. 3.5

[45] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, December 1999. `http://www.eros-os.org`. 6.2

[46] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice-Hall, Inc, 3rd edition, 2001. 2.1

[47] Sun Microsystems. Multithreading in the Solaris operating environment: A technical white paper, 2002. Available from `http://sun.com/software/whitepapers`. 2.4

[48] Ted Unangst. `rthreads`: A new thread implementation for OpenBSD. In *Proceedings of the 5th European BSD Conference*, November 2005. 2.4

[49] Nathan Williams. An implementation of scheduler activations on the NetBSD operating system. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*. Wasabi Systems, Inc., USENIX Association, June 2002. 2.4

[50] Zhi-Hua Zhou, Yuan Jiang, and Shi-Fu Chen. Extracting symbolic rules from trained neural network ensembles. *AI Communications*, 16(1):3–15, 2003. 3.5