Performing Under Overload

Luke Macpherson

Doctor of Philosophy
School of Computer Science and Engineering
The University of New South Wales

2007

Dissertation Sheet

Surname:	Macpherson
Given Names:	Luke David
Abbreviation for Degree:	PhD
School:	Computer Science and Engineering
Title:	Performing Under Overload

Abstract

This dissertation argues that admission control should be applied as early as possible within a system. To that end, this dissertation examines the benefits and trade-offs involved in applying admission control to a networked computer system at the level of the network interface hardware.

Admission control has traditionally been applied in software, after significant resources have already been expended on processing a request. This design decision leads to systems whose algorithmic cost is a function of the load applied to the system, rather than the load admitted to the system.

By performing admission control at the network interface, it is possible to develop systems whose algorithmic cost is a function of load admitted to the system, rather than load applied to the system. Such systems are able to deal with excessive applied loads without exhibiting performance degradation.

This dissertation first examines existing admission control approaches, focussing on the cost of admission control within those systems. It then goes on to develop a model of system behaviour under overload, and the impact of admission control on that behaviour. A new class of admission control mechanisms which are able to perform load rejection using the network interface hardware are then described, along with a prototype implementation using commodity hardware.

A prototype implementation in the FreeBSD operating system is evaluated for a variety of network protocols and performance is compared to the standard FreeBSD implementation. Performance and scalability under overload is significantly improved.

Abstract

This dissertation argues that admission control should be applied as early as possible within a system. To that end, this dissertation examines the benefits and trade-offs involved in applying admission control to a networked computer system at the level of the network interface hardware.

Admission control has traditionally been applied in software, after significant resources have already been expended on processing a request. This design decision leads to systems whose algorithmic cost is a function of the load applied to the system, rather than the load admitted to the system.

By performing admission control at the network interface, it is possible to develop systems whose algorithmic cost is a function of load admitted to the system, rather than load applied to the system. Such systems are able to deal with excessive applied loads without exhibiting performance degradation.

This dissertation first examines existing admission control approaches, focussing on the cost of admission control within those systems. It then goes on to develop a model of system behaviour under overload, and the impact of admission control on that behaviour. A new class of admission control mechanisms which are able to perform load rejection using the network interface hardware are then described, along with a prototype implementation using commodity hardware.

A prototype implementation in the FreeBSD operating system is evaluated for a variety of network protocols and performance is compared to the standard FreeBSD implementation. Performance and scalability under overload is significantly improved.

Originality Statement

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

Publications

Portions of this work have been published in the following articles:

- **Overload protection for commodity network appliances**
  Luke Macpherson
  Asia-Pacific Computer Systems Architecture Conference,
  Lecture Notes in Computer Science, Volume 4186, Pages 203–218, 2006

- **Ipbench: a framework for distributed network benchmarking**
  Ian Wienand and Luke Macpherson
  AUUG Winter Conference, Melbourne, Australia, September, 2004

- **Maintaining end-system performance under network overload**
  Luke Macpherson and Gernot Heiser
  Technical Report UNSW-CSE-TR-0412, School of Computer Science and Engineering, March, 2004

Acknowledgements

Thanks to all those who have provided valuable feedback on this work at various times over the last few years. To my supervisors, Gernot Heiser and Kevin Elphinstone, and to my assessors, Felix Rauch Valenti, Thomas Stricker and Timothy Roscoe.

I would also like to thank the fellow members of the kernel engineering group who have freely given assistance when needed. Andrew Baumann, Daniel Potts, Harvey Tuch, Ian Wienand, Ihor Kuz and Peter Chubb. Your help has been invaluable.

Finally, thanks to my family for your unfailing support. Your faithfulness and consistency continue to have the greatest impact on me.

Contents

List of Figures

List of Tables

1 Introduction

This dissertation considers the behaviour of overloaded systems. A new approach called *edge limiting* is proposed, which prevents performance degradation under overload.

1.1 Problem statement

1.1.1 Overload

It is evident that all systems have finite performance bounds, since all systems are subject to physical constraints, such as processor speed, memory capacity, bandwidth and latency, peripheral interconnect bandwidth and latency, power consumption etc. Because all systems are subject to physical resource constraints, it is generally possible to demand more of a system than it can physically provide. It is merely a matter of applying enough load to a system that it no longer has enough resources to process all of the applied load; a situation known as overload.

The observation that it is possible to overload any system based on real physical hardware is not merely hypothetical. Overload has been the subject of operating systems research for more than a decade [19], over which time the performance of computer hardware has changed dramatically.

Despite continuing improvements in hardware performance, the occurrence of overload is not decreasing in current systems. Gigabit-per-second networking has been available for more than ten years, and is now commonly integrated with new computers; yet a single Gigabit Ethernet link is still capable of overloading many current systems. This problem is only likely to worsen given the recent introduction of 10 Gigabit-per-second Ethernet interfaces.

Since the possibility of overloading a system always exists, and is also commonly observed in real systems, it is important that the behaviour of overloaded systems be well understood, and taken into account when making system design decisions. While it is desirable that a system maintain maximum throughput when overloaded, performance degradation under overload is a common characteristic of many systems. The extreme case of such performance degradation is livelock, where given a suitably high load, the throughput of the system drops to zero.

This thesis aims to identify the modifications to existing systems which are necessary to eliminate performance degradation under overload. Such modifications should aim to endow a system with the ability to maintain maximum performance even when overloaded.

1.1.2 Problem uniqueness

Given the description of overload in the preceding section, a reasonable person would naturally expect computer systems to be overflowing with real-world examples of overload. In reality, there are very few ways in which overload may be induced in a typical computer system.

One reason for the scarcity of examples of overload in real systems is that overload requires that the load generator and load recipient operate independently, without feedback or flow control. This situation rarely occurs in systems, since communication primitives and system structures prevent it from occurring. For example, the Unix pipe communications primitive provides both limitations on the number of outstanding requests which may be generated and the ability to block the load generating process. Even systems which use asynchronous IPC typically limit the number of messages which can be queued. So while it is theoretically possible to produce overload in any asynchronous system, there are usually mechanisms in place which prevent this from occuring.

Similarly, hardware subsystems and drivers do not usually induce overload because the load is generated internally to the system. For example, in a disk subsystem, the driver issues a request to the disk, and the disk generates a response at some later point. The rate of responses is proportional to the rate of requests, and the rate of requests is naturally limited by resource availability.

It is possible to hypothesise about scenarios which could generate overload on multi-processor systems. For example, in the disk subsystem scenario, if requests were generated on one processor, and responses handled on a different processor, it may be possible to overload the processor handling responses. We are not aware of any such examples occurring in practice.

There are a few well-known exceptions to the above generalisation which are known to impact existing operating systems. For example, operating systems using virtual memory may overcommit memory, leading to excessive swapping and degraded performance, or processes may perform excessive forking, such as in a fork-bomb, leading to degraded performance. These are examples where some feedback exists, however that feedback is inadequate to prevent overload from occuring. In these examples, the load generator will see decreased performance due to the load they generate, however the cost of load generation to the process is much lower than the cost of handling the load, such that loads may be generated which cannot be processed by the system. Such situations can be handled by artificially limiting the ability of a process to generate load.

There is another class of systems in which overload is a known issue; networked systems. Networked systems are unique because load generation is not under the direct control of the system, since it is externally generated. This class of systems presents a unique challenge, since it may be impossible to limit the ability of a load generator to generate load.

This thesis is strongly focussed on preventing performance degradation under overload for networked systems. While the problem of overload in networked systems shares some similarities to the problem of overload in other situations, because of the inherent inability to limit load generation, networked systems are likely to require a significantly different approach to the problem of performance degradation due to overload.

1.1.3 Admission control

The general solution to the problem of performance degradation under overload in networked systems is to prevent a system from receiving loads high enough to induce performance degradation. Ultimately this can be achieved in two ways: reducing the number of requests being generated by clients, or discarding excess requests after they have been generated. In both cases, the aim is to reduce the incoming request rate such that it does not exceed the maximum request rate supported by the system.

Reducing the number of requests generated by clients is in the realm of network flow control and congestion control, and requires protocols which support feedback to limit the rate of request generation. Internet protocols such as TCP provide flow control to individual connections, however there is no guarantee of global flow control.

Some networks standards such as ATM attempt to provide solutions to the congestion control problem at the data-link layer, however because the wider Internet is made up of many different network standards, their effectiveness as a solution to the problem of overload in the wider internet is limited. Practically, it seems reasonable to expect that we will be stuck with Ethernet for some time to come, which means that solutions which do not require congestion control are likely to remain necessary.

In situations where network flow control is unavailable or inadequate to prevent overload, admission control is typically implemented on the server. Admission control works by rejecting units of work when resource availability is inadequate to meet all requests.

Admission control systems typically consist of solutions to two orthogonal problems; firstly, they must provide a way to prevent excess load from being processed by the system, and secondly, they must determine how much load can be admitted to a system without inducing overload. There are many existing combinations of solutions to these two problems, and these are discussed in Chapter 2.

1.2 Scope

Our search for solutions to the problem of performance degradation under overload has been constrained by a number of requirements which were designed to maximise the real-world applicability of any solutions found.

Common Internet protocols (TCP/UDP/IP) We are interested in solutions which apply to common Internet protocols, such as IP, TCP and UDP. While the IP protocol does not provide any flow control, higher level protocols may provide some degree of flow control. For example, TCP provides flow control on a per-connection basis, such that for small connection establishment rates and long lived connections, TCP-based applications are unlikely to cause overload. Unfortunately there are also many situations where IP-based protocols can cause overload in a connected host. These are widely known for services such as HTTP and DNS, where both malicious and legitimate traffic can cause spikes in server load resulting in service interruption.

Commodity network hardware (Ethernet) We are primarily interested in solutions that work with commodity network hardware. We want our solution to work with the majority of existing hardware in the world, rather than produce a proprietary system which is not compatible with existing systems. In essence, this leads us to looking at solutions which work with commodity Ethernet hardware such as switches and network interface cards (NICs). Such solutions reduce the cost and adoption difficulty of a solution, since they may be readily integrated with existing systems.

Existing Gigabit Ethernet hardware usually supports the 802.3x standard, which does allow some control over the incoming data rate of the NIC. Unfortunately, the design of the operating system driver usually prevents useful flow control information from being propagated into the network, since the driver generally runs at a high priority and dequeues all available packets even if the system cannot process them.

Finally, a significant limitation of commodity hardware is that it does not allow differentiation between packets based on their protocol headers. All packets are treated equivalently by the NIC. This makes it impossible to intelligently choose which packets to accept or reject without consuming additional resources on the server.

Applicable to existing commodity operating systems We are looking for a solution that is simple to implement in current operating systems without major restructuring or reimplementation of those systems. Solving performance degradation under overload would undoubtedly be easier if it had been a consideration in the initial design of a system, however we do not have the luxury of reimplementing systems from scratch in order to solve the problem.

Not limited to specific protocols Finally, we are not merely interested in fixing overload of the HTTP protocol. While significant research has been done into providing admission control for such systems, other essential services such as DNS, which is primarily UDP based, have not received adequate attention. We are interested in finding a solution which may be applicable to HTTP, but is general enough to provide admission control for a wide range of Internet applications and protocols.

1.3 Contributions

The thesis of this work is that the cost of admission control must be bounded in order to guarantee scalability under increasing overload. Several contributions are made to the field:

i) The cost of admission control in existing systems is examined. Focus is given to the relationship between the location of the admission controller within the system, and the overall scalability of that system under overload.

ii) A model of system behaviour under overload is provided, and this model is used to demonstrate that providing rate-limiting at the edge of the system is adequate to prevent overload from occurring.

iii) Bounded-cost admission-control mechanisms which may be implemented on commodity hardware are introduced. These mechanisms are able to scale to any level of applied load.

iv) A new mechanism for detecting overload based on measuring system throughput is described. This new mechanism has much lower implementation and run-time costs than existing approaches to overload detection.

v) An experimental evaluation of our approach is provided, which demonstrates the feasibility of implementing bounded-cost admission control on commodity hardware. The difficulties and tradeoffs encountered in this approach are also discussed.

1.4 Synopsis

Related work Chapter 2 surveys existing approaches to admission control, providing comparisons of their strengths and weaknesses. Other related issues are also discussed.

Model of overload Chapter 3 provides a model of system behaviour under overload. The behaviour of individual components of a system under overload is first addressed, then used as a basis for modelling the behaviour of systems composed of these components. The model is then used to motivate placement of admission control as early as possible in the processing of incoming network packets.

The edge limiting approach Chapter 4 introduces a number of rate-limiting mechanisms which may be implemented in the driver, and used to implement admission control at the level of the Network Interface Card (NIC) using commodity network hardware. These implementations are compared in terms of expected performance and implementation difficulty.

Rate selection Chapter 5 introduces a new method of determining the maximum rate at which load may be admitted to the system based on throughput monitoring.

Experimental evaluation Chapter 7 examines the behaviour of NIC-based rate limiting and throughput-based rate selection for workloads varying from simple UDP echo through to HTTP workloads. The performance impact of the approach is examined in detail, with both positive and negative results shown.

Conclusions Chapter 8 provides a summary of this dissertation and its contributions, followed by critical assessment and a discussion of future work.

2 Related Work

This chapter examines a number of existing systems based on criteria which are relevant to system behaviour under overload. Specifically, we are interested in the cost of admission control in the system, whether it is possible to cause performance degradation even with that admission control in place, the level at which session differentiation is performed, and the feedback mechanism used to decide how much load to admit into the system. Finally, we consider how intrusive the approach is on the structure of the system, and how readily the approach may be applied to current systems. Table 2 gives a summary of these features for a number of existing systems.

2.1 Admission control

In order for a system to provide acceptable behaviour under overload, it must implement some form of admission control. This section examines the admission control mechanisms used in a number of existing systems.

One early instance of kernel-level admission control can be seen in Mogul and Ramakrishnan's work on eliminating receive livelock (ERL) in an interrupt-driven kernel [19]. This solution uses a polling thread to service network interfaces in a round-robin fashion and limits the number of packets which may be handled each time a queue is serviced.

While the main intent of this work was to curtail excessive interrupt rates and poor prioritisation of packet processing, their solution also implemented a simplistic form of admission control by temporarily disabling input from the network interface when queues become full. This method of admission control is too coarse to work well on modern network interfaces, which tend to support very high packet rates, and transfer many packets per interrupt.

Lazy receiver processing (LRP) [8] uses early packet demultiplexing to individual queues to separate individual data-paths within the system. There were two implementations, a hardware-based solution which demultiplexes based upon ATM virtual-circuit identifiers (VCIs), and a software implementation which demultiplexes based upon fields in the IP protocol header. Admission control is performed

System	Admission control cost	Performance degrades	Session differentiation	Feedback mechanism	OS restructure	Ref.
ERL	$O(admitted)$	N	Packet	Full queue	Y	[19]
Hardware LRP	$O(admitted)$	N	Socket	Full queue	Y	[8]
Software LRP	$O(total)$	Y	Socket	Full queue	Y	[8]
SRP	$O(total)$	Y	Socket	Full queue	Y	[5]
SYN Policing	$O(total)$	Y	TCP	Resource monitor	N	[29, 26, 27, 28]
WebQoS	$O(total)$	Y	HTTP Request	Queue length	N	[2]
Yaksha	$O(total)$	Y	HTTP Request	Response time	N	[15]
Quorum	Independent	N	HTTP Request	Response time	N	[4]
SEDA	$O(total)$	Y	Multiple	Multiple	N	[30]

Table 2.1: Existing admission-control approaches.

by dropping packets which are destined for a socket whose queue is full.

TCP SYN policing [29] is an admission control system which operates by rejecting the connection setup packets of a TCP connection, in order to limit the rate at which TCP connections can be established. It has been used particularly in the context of web servers, whose network behaviour is typically characterised by large numbers of short-lived TCP connections. Admission control occurs by only allowing TCP SYN packets to enter the system at a controlled rate. This approach is only effective for loads which are responsive to such negative feedback, rather than performing load shedding at the overloaded host.

The staged event-driven architecture (SEDA) [30] is a comprehensive solution to the problem of overload. SEDA wraps every stage in a data-path with its own admission controller and feedback loop. This results in the possibility of load shedding occurring at many points in the system, depending on which stage of the data-path is overloaded. Non-admittance to a stage allows upstream stages to respond by altering their behaviour in order to reduce the load applied to the downstream stage.

While SEDA provides an excellent framework for the propagation of information on overload within the system, it does not proagate admission control all the way to the edge of the system, and hence cannot achieve $O(admitted)$ scalability. Moreover, SEDA's approach is difficult to retrofit to existing operating systems, as it necessarily impacts all levels of network processing.

2.1.1 Admission control cost

Those systems which do perform admission control may be divided into two groups based upon the cost of admission control in the system. Specifically, we are interested in whether the cost of performing admission control scales with the number of admitted requests, $O(admitted)$, or with the total number of requests, $O(total)$.

This is an important question when considering admission control. If the admission control mechanism itself scales according to the total number of requests, then it is reasonable to say that the admission control mechanism may itself be subject to overload, for large enough input loads. If an admission control mechanism scales with the number of accepted requests, on the other hand, we may be sure that it will not contribute to performance degradation under overload.

The ERL approach of disabling input from the network interface is inherently $O(admitted)$, since the system only performs processing on data once it has already passed the admission control mechanism. This mechanism results in short bursts of traffic being admitted to the system during overload, rather than applying a consistent load to the host.

The hardware-based LRP approach is able to discard traffic at the network interface when a socket's queue is full, and hence achieves $O(admitted)$ performance. The software LRP approach, on the other hand, must perform packet processing on every packet entering the system prior to performing admission control. Therefore, the LRP approach is only able to achieve $O(total)$ performance, and given suitably high loads will exhibit performance degradation.

In a similar manner to software LRP, *signaled receiver processing* (SRP) [5] also performs work on all packets entering the system prior to admission control. SRP is therefore also subject to performance degradation under overload, due to the expenditure of resources on packets which are later discarded.

The case of SYN policing is interesting, because the performance of admission control is strongly dependent on the behaviour of the clients which are generating the load. Fundamentally, the performance is $O(total)$, since all packets are examined for the SYN flag when admission control is occurring, however the extent to which performance degradation actually occurs is determined by the ratio of SYN packets to other network traffic. For traffic which consists of short-lived connections, we still expect this admission control to have significant overheads, however for traffic which consists mostly of long-lived connections, it is reasonable to expect that performance degradation will be minimal. Unfortunately traffic which consists of many short-lived connections is most likely to generate overload, since such flows are too short-lived for TCP flow control to reduce the load on the system.

2.2 Performance under overload

System behaviour under overload is the primary focus of this thesis, and is therefore the ultimate property by which we judge existing systems. The behaviour of a system under overload falls into two general categories; those whose performance becomes degraded under overload, and those who maintain peak performance even when exposed to excessive request rates.

Systems whose admission control cost is $O(total)$ exhibit degraded throughput under overload. The rate at which degradation occurs in such systems is dependent on resource consumption which occurs prior to admission control. The less work performed prior to admission control, the slower a system's degradation will be. For this reason, late admission control is still better than no admission control, even though some performance degradation will still be present in the system.

Meanwhile, those systems whose admission control cost is $O(admitted)$ or better will not experience degraded throughput under overload. Of those systems whose performance does not degrade under overload, we recognise two sub-categories; those whose peak performance is comparable with that of a standard system, and those whose peak performance is significantly lower.

Some papers in this area have produced results which obscure the degradation inherent in their approaches. For example in [27,28,29], by presenting results in terms of connections per second, reduced throughput under overload can be obscured by preferentially dropping longer connections or connections which would consume more resources. Such presentations can make systems with $O(total)$ admission control cost appear to eliminate performance degradation, or even provide significant improvements in peak throughput. It is clear that in these situations a reduction in useful work is occurring, and that the chosen performance metric of connections per second is inadequate to characterise systems which exhibit this behaviour.

2.3 Differentiation

Admission control may be performed at different granularities depending on the requirements of the system. Most systems perform admission control on either a per-packet, per-TCP-connection, or per-user basis.

Performing admission control on a per-packet basis is primarily used because it does not require specific knowledge about the contents of a packet in order for admission control to take place. This approach has the advantage that admission control cost can be made independent of the load applied to the admission controller, allowing the approach to scale to very high applied loads. The disadvantage is that it is not possible to preferentially treat packets based on their contents.

Admission control on a per-TCP-connection basis has been proposed as a solution for protocols such as HTTP, which typically generate large numbers of short-lived TCP connections. The advantage of such approaches is that existing connections can be prioritised over new connections, such that once a connection is established, that connection can be given enough bandwidth for the connection to complete within a well defined time. The disadvantage of this approach is that it requires that the admission controller be able to process the IP and TCP packet protocol fields, in order to differentiate between incoming packets. Such processing implicitly consumes resources for each additional packet received, even if it is not admitted. If those resources are also used for further processing of requests, performance degradation under overload can be expected.

Admission control can also be performed at higher levels, according to information available to the application-level protocol. Such information could allow for per-user service differentiation, for example [29].

While the granularity at which admission control is performed is conceptually orthogonal to the cost of admission control, the cost of admission control is closely related to the implementation details of session differentiation.

There are two basic classes of admission control implementations; those which perform admission control at the network interface, and those which perform admission control during protocol processing on the host CPU. Typically, performing admission control at the network interface is necessary to achieve $O(admitted)$ scalability, however performing session differentiation at greater granularities requires increased processing capacity on the network interface itself.

For commodity IP over Ethernet, performing admission control at the network interface on a per-packet basis is simply a matter of accepting packets at the appropriate rate. Performing admission control at higher levels requires the examination of higher level protocol fields in order to differentiate between packets. Such functionality is not typically available in commodity Ethernet interfaces, however in other research contexts, such as user-level network protocol implementations, programmable Ethernet interfaces have been modified to provide hardware packet filtering into multiple input queues [23].

For TCP/IP over ATM, performing admission control at the network interface on a per-connection basis is feasible, since TCP connections are mapped to individual ATM virtual circuits, of which the network interface may be aware [17, 3]. This is the approach taken by the hardware-based implementation of LRP [8].

2.4 Feedback

We now examine feedback mechanisms which are used to control the acceptance rate of the admission controller. There are three fundamental approaches which are taken in the literature, explicit flow control, resource-based feedback, and performance monitoring.

Explicit flow control is used in systems which are structured with queues between connected components. Such systems allow upstream components to signal downstream components that they are overloaded, by allowing their incoming queues to become full. Upon encountering a full receive queue, the downstream component can presume that the upstream component is unable to process the incoming datastream at the required rate, and is overloaded.

Resource-based feedback is implemented by monitoring key system resources, such as CPU time and memory consumption, in an attempt to determine when a system has become overloaded. The main problem with resource-based feedback is that resource usage is not always an accurate indicator of overload.

The third feedback mechanism which is commonly used is performance monitoring. Most systems use some variation of response time measurement, where the feedback mechanism uses a control loop to maintain response time guarantees. For example, SEDA, Yaksha and Quorum aim to maintain a well-defined percentage of traffic below a maximum response time.

2.5 Effect on system structure

The choice of control mechanism and feedback source may dictate the structure of the system. In many cases (ERL, LRP, SRP), the system was significantly restructured in order to support queueing models which allowed for explicit flow control.

Systems such as SYN policing, which have more loosely coupled feedback and control, are somewhat simpler to insert into an existing system without requiring major system restructuring. In this case, the admission controller may simply be inserted on the incoming data-path, and the feedback mechanism simply performs measurements on the system.

2.6 Data-path separation

One of the significant problems encountered on commodity Ethernet network interfaces is that it is not possible to differentiate between different data streams at the network interface. Unlike asynchronous transfer mode (ATM) networks, which can utilise a virtual circuit identifier (VCI) to differentiate between individual data streams, Ethernet interfaces have no concept of virtual circuits. Therefore, a solution supporting Ethernet would require that the network interface have the ability to demultiplex packets based on fields located in the headers of higher level protocols.

The initial implementation of Nemesis [17] used ATM VCIs for packet demultiplexing, however later implementation on Ethernet hardware used a software-based packet demultiplexor [3]. More recent work on the Arsenic project [23] has focussed on providing hardware packet demultiplexing to user-level protocol stacks in Linux. Such hardware would enable data-paths to be rate-controlled individually.

Both *Scout* [20] and *Nemesis* operating systems share the approach of explicitly separating data-paths within the system. They also have the desirable property that it is simple to monitor the input and output of individual data-paths, which may assist in detecting overload. Our solution naturally lends itself to implementation in such systems, as they avoid data-path entanglement and its inherent problems, as discussed in Section 3.7.

2.7 Loosely related issues

This section provides a quick overview of related work which is loosely related to the problem of overload. While not essential in understanding this dissertation, it is useful to clarify their impact on the behaviour of systems under overload.

2.7.1 TCP flow control and overload

In some instances, high-level protocols such as TCP are able to prevent the occurence of overload when the majority of network traffic belongs to a relatively small number of long-lived TCP connections. This is achieved by relying on the sender to decrease its transmit rate in response to network and receiver feedback.

TCP also assumes packet loss indicates network congestion. When packet loss is detected, the sender decreases its transmit rate. This has implications for instances of overload whose behaviour is to drop packets in the receive path. When such packet loss is observed, the sender will reduce its transmit rate, and therefore the load applied to the overloaded host is reduced[1].

There are a number of practical limitations which prevent this being a general approach to solving the problem of overload. Firstly, it requires that protocols respond to congestion feedback by reducing their transmit rates. Many protocols do not provide such behaviour, and reimplementing every protocol is not a viable option.

Moreover, overload need not result in packet loss or congestion at or below the network protocol level. In such cases, network protocols will be unable to limit the sender's transmit rate without explicit feedback from the application. Such feedback may be available as a side effect of the IP stack's application programming interface, however such feedback is dependent on application behaviour.

Finally, a problem which is frequently encountered in real systems such as web servers is that protocol-based flow control is only really effective for long-lived protocol sessions. This means that for protocols such as HTTP, which typically have high connection rates and short lived connections, protocol-based flow control is ineffective in preventing overload from occuring.

2.7.2 Impact of optimisations on performance under overload

Many optimisations which improve performance when a system is not overloaded result in reduced throughput when a system is overloaded. Optimisations which result in performing work on packets which were later discarded tend to produce reduced throughput under overload.

[1]In many circumstances, reducing throughput when packets are lost is not a desirable behaviour. For example, on a wireless network which experiences packet loss due to interference rather than congestion or overload, the network protocol should not respond to packet loss by reducing the transmit rate.

A common approach to network driver and protocol stack implementation is to process batches of packets at a time, rather than processing an individual packet to completion before beginning processing on the next packet. For example, a network interface will typically queue a number of packets prior to raising an interrupt, the network driver will dequeue all of those packets and add them to the IP stack's input queue, the IP stack will process all of those packets and add them to the application's input queue.

The advantage of this approach is that it provides good instruction-cache locality, and tends to minimise the number of context switches which are performed while processing packets. When a system is not overloaded, there is only the slight disadvantage that jitter may be increased slightly, while average latency is reduced.

When a system becomes overloaded, such optimisations can have a pronounced negative effect on the overall performance of the system. Under overload, it is inevitable that some requests must be discarded, since there are not enough resources in the system to process all requests. In this case, processing batches of packets tends to cause partial processing of many packets which are later discarded. Such partial processing further reduces the resources which are available for processing requests to completion. Such behaviour inevitably leads to system performance which will progressively degrade as applied loads increase beyond the maximum capacity of the system.

It is desirable that a solution to the problem of performance degradation under overload not significantly impact the performance of the system when it is not overloaded. This means that optimisations which may cause degradation under overload may need to be retained, and any additional overheads must be minimal.

Ultimately, the solution to this dilemma is to perform admission control as early as possible, such that performance optimisations are always performed on packets which will be processed to completion. In this way, the performance optimisations can remain in place without introducing undesirable behaviour when overloaded.

2.7.3 Event notification mechanisms

There are two general classes of event notification used in modern systems; interrupts and polling. In interrupt-driven systems, the maximum rate of event notifications is determined by the entity which generates events (for example, the network interface). In systems which use polling, on the other hand, the maximum rate of event notifications is determined by the entity which is receiving event notifications (for example, the device driver).

Because interrupt-driven systems allow the event generator to determine the notification rate, it is possible for the event generator to produce event notifications at an excessively high rate, causing performance degradation under overload. On early hardware there were few mechanisms for controlling the event notification rate, other than disabling interrupts. This has lead to the widespread belief that overload is usually the result of an excessively high interrupt rate. While this was indeed an issue on early network hardware [19], modern hardware does not suffer the same problems.

Modern network interfaces provide a variety of mechanisms for controlling the interrupt generation rate, and typically provide a reasonable upper bound on the event notification rate. This means that on modern hardware event notification is largely orthogonal to the problem of overload — both polling and interrupt driven systems provide control of the event notification rate, such that it is reasonable to build systems which are not susceptible to performance degradation under overload due to excessive event notification rates.

2.7.4 Load balancers and multi-server systems

Many large-scale internet services are distributed over multiple servers, in order to increase peak serving capacity. Such systems typically utilise a front-end load balancer, that is responsible for distributing incoming requests over a pool of back-end servers. This approach minimises the work done by the load balancer, such that it is able to handle a greater volume of traffic than a single general-purpose server. On the other hand, it is simply a matter of adding enough back-end servers for the front-end load balancer itself to become the limiting factor in increasing service capacity. In this case, the load balancer itself is likely to be subjected to overload.

Front-end load balancers form a natural point for the protection of back-end services from excessive load. In order to perform admission control, front-end load balancers need to solve similar problems to those addressed in this dissertation. Firstly, they need to implement some mechanism for rejecting excess load, and secondly, they need to detect when the back-end servers are overloaded.

The front-end load balancer is in essence an extension of the edge-limiting approach described in this dissertation. The difference being that the edge of the system has been pushed one step beyond the boundary of the back-end server. In this case the front-end load balancer may be able to afford to dedicate CPU time to differentiating between incoming packets prior to performing load rejection, since processing power is not needed for performing other functions.

Despite the increased resources which can be devoted to admission control in a front-end load balancer, our rate limiting approach may still be useful in such systems for protection of the load balancer itself, since it is still conceivable that the load balancer may be subjected to more load than it has resources to handle. In this case, our approach may be useful to protect the load balancer itself from overload.

3 Modelling Overload

In order to understand the overall behaviour of an overloaded system, it is useful to develop a model of the way individual components of that system behave and interact. This chapter presents a model of the behaviour and interaction of network communications components.

The purpose of this model is to develop an understanding of the impact and tradeoffs of the application of admission control at different points within a system. Ultimately, it will show that limiting at the edge of the system is adequate to prevent performance degradation under overload, with the limitation that the throughput of other applications whose load also passes through the admission controller will be impacted.

3.1 Background

Overload occurs when the load applied to a system exceeds the system's maximum capacity. A carefully designed system should continue to provide maximum capacity even as the load applied increases beyond the system's maximum capacity.

Most systems exhibit gradual performance degradation as the applied load increases beyond the maximum capacity of the system. This performance degradation typically continues until the applied load is capped by the exhaustion of a required resource, such as PCI bus bandwidth, or until the system becomes live-locked, at which point it ceases to perform any useful work.

3.2 Resource types

There are many resources in a computer system which may become overloaded, thereby limiting maximum system capacity. We divide these resources into two categories; physical resources and logical resources.

Physical resources are those which correspond to various aspects of the system hardware, such as processor time, memory bandwidth, data bus bandwidth and network interconnect bandwidth. They are present and inherent in all systems.

Logical system resources may also become overloaded. Examples of logical resources which can become overloaded in a real system include queues, buffer allocators and DMA rings that utilise fixed memory areas or a fixed number of entries. Any logical resource which is allocated from a finite set may become overloaded when all elements in the set are in use. Any further allocations will fail until existing allocations are rescinded, causing a rate-limiting effect to occur.

3.3 Sources of system load

Any operation performed by a system which consumes resources contributes to system load. While not an exhaustive list of load sources, the following examples describe common sources of load on a typical system.

User-level applications directly generate load on the CPU and memory bus. The operating system can control this load by adjusting the time allocated to the application by the operating system's scheduler.

Secondary storage devices generate load on the device bus, the system bus, and the memory bus. This load can be controlled by the operating system by adjusting the rate at which read and write requests are issued to the device controller.

Network interface devices generate load on the system bus and the memory bus by performing DMA. Network interfaces are also capable of placing significant load on the CPU, by generating a high rate of interrupts.

Unlike other types of devices, the operating system cannot control the packet arrival rate of the network interface. There is no implicit feedback mechanism to prevent the network load applied to a system exceeding the maximum capacity of that system. This means that network interfaces are considerably more likely to be sources of overload than other devices, since the rate of incoming packets is not under the control of the operating system, and may exceed the total capacity of the system.

3.4 Modelling overload

We model real systems as a graph of logical resources traversed by data passing through the system. Logical resources may map to one or more physical resources, and multiple logical resources may share an underlying physical resource, such that placing load on one logical resource may reduce the capacity of another logical resource.

Figure 3.1 is an example resource graph for a simple system with a single datapath that is performing packet echo. Circular nodes represent logical resources, rectangular nodes represent physical resources, super-nodes (indicated by dashed

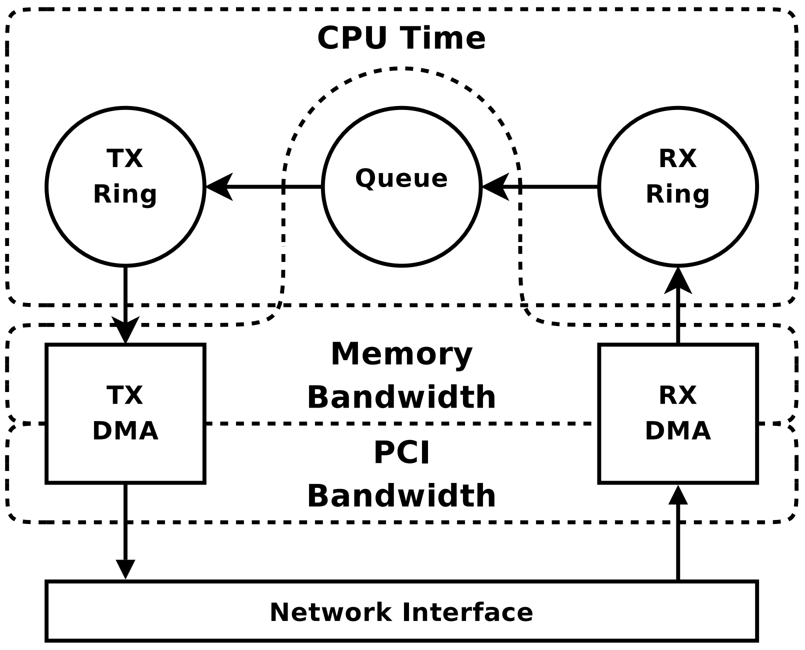

Figure 3.1: Example graph of resources for a simple system.

lines) show shared physical resources. In Figure 3.1, a packet begins at the network interface. The network interface then consumes an entry in the DMA ring (which is a logical resource), and uses DMA to copy the packet into memory, consuming both PCI and memory bandwidth, but not CPU time. The driver's receive routine is triggered by an interrupt, consuming CPU time and freeing an entry in the receive ring. The driver then causes the packet to become enqueued using a copy operation, consuming both CPU time and memory bandwidth. The driver's transmit routine then adds the packet to the transmit ring, consuming a transmit ring entry and some CPU time.[1] The network interface then uses DMA to copy the packet into its local memory for transmission, consuming both memory and PCI bandwidth.

Resources in real systems fall into two categories; those that exhibit degraded performance under overload, and those that do not. We refer to these as degradable and non-degradable resources, respectively. Degradable resources occur when two or more resources on the same data path share the same underlying physical resource, and those resources do not receive a fixed share of the underlying resource.

Consider the following scenario: a network interface is connected via a peripheral bus to main memory. The maximum throughput of the peripheral bus is lower than the maximum throughput of the network interface. In order to avoid

[1]In reality most software operations use some memory bandwidth.

receive-buffer overruns, the network interface prioritises the DMA of receive packets over the DMA of packets ready for transmission. The host application is a simple UDP packet echo.

When the peripheral bus becomes saturated, half of the bus bandwidth will be used for the DMA of incoming packets, and half for the DMA of outgoing packets. As the rate of incoming packets is increased beyond the point at which the bus becomes saturated, the bus bandwidth used by incoming packets reduces the bus bandwidth available for outgoing packets by a corresponding amount. In the extreme case, incoming packets will consume all available bus bandwidth, and no useful work will be done by the echo server.

Mogul and Ramakrishnan [19] observe a similar phenomenon. In the case they examine, the shared system resource is time on the CPU, and processing of incoming packets by the interrupt handler occurs at a higher priority than other packet processing. This causes a reduction in CPU time allotted to processing packets for transmission, leading to the operating system transmit path (and user application) behaving as a degradable resource.

3.5 Modelling independent resources

We model independent resources using output throughput as a function of input throughput. Non-degradable resources are characterised by functions that have non-negative derivatives for all input throughputs.

We also define a special type of non-degradable resource, which we call a rate-limiting resource, $f_{\text{rate}}(x)$ whose behaviour is characterised by output throughput which equals input throughput for input throughputs which are less than a determined rate-limit, and equal to the rate-limit for input throughputs above the rate-limit.

$$f_{\text{rate}}(x) = \begin{cases} x, & \text{for } 0 \leq x \leq \text{rate}, \\ \text{rate}, & \text{for } x > \text{rate}. \end{cases} \tag{3.1}$$

Figure 3.2 shows characterisations of three resources. The rate-limiting resource $f_{\text{rate}}(x)$, is an example of a rate-limiting resource with rate-limit of 600.[2] The non-degradable resource $f_n(x)$ is an example of a simple non-degradable resource, with throughput which increases until the resource becomes saturated, then continues to perform at that level. The degradable resource $f_d(x)$ is an example

[2]Figures 3.2 and 3.3 do not specify units. Packets per second or bytes per second are typical metrics which could be used.

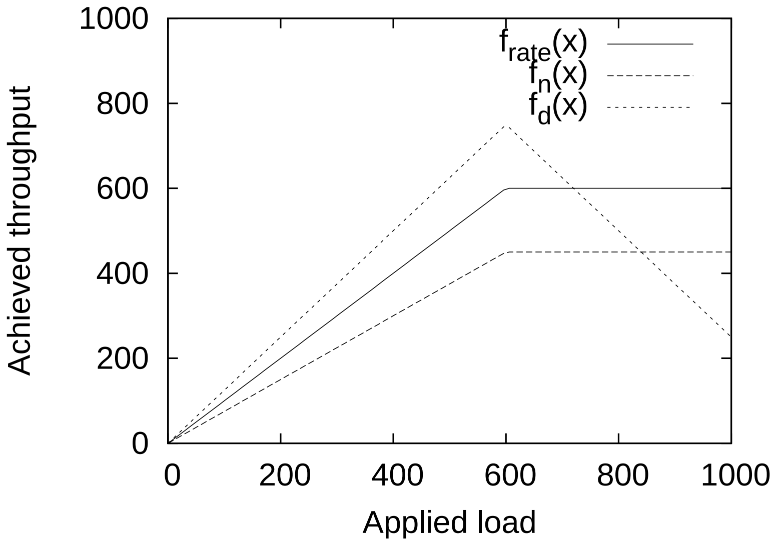

Figure 3.2: Characterisation of rate-limiting resource $f_{\text{rate}}(x)$, non-degradable resource $f_n(x)$ and degradable resource $f_d(x)$.

of a simple degradable resource, with throughput which increases linearly until the resource becomes saturated, and throughput which decreases linearly once the saturation throughput is exceeded.

3.6 Modelling independent data-paths

Given a model of the behaviour of individual resources under load, it is possible to model the behaviour of an individual data-path within the resource graph. When resources are connected in series, the input throughput seen by a resource is the output of the previous resource in the data-path. Therefore, the characterisation of resources $f_a(x)$ and $f_b(x)$ connected in series is $f_b(f_a(x))$.

Figure 3.3 is an example of a data-path where a rate-limiting resource precedes a degradable resource. We see that when a rate-limiting resource $f_{\text{rate}}(x)$ precedes a degradable resource $f_d(x)$, throughput does not degrade below $f_d(\text{rate})$. Therefore, if a rate of $f_d^{-1}(max(f_d))$, overall throughput is maintained once peak throughput is reached.

Figure 3.4 shows the effect of selecting a rate limit which is lower than the maximum of f_d. Rate limiting occurs before peak throughput is reached, however the overall behaviour of the system is still non-degradable.

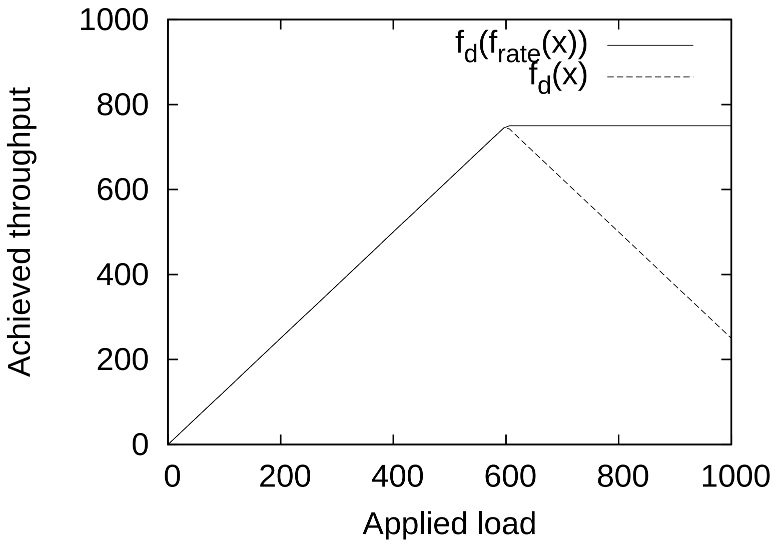

Figure 3.3: A rate-limiting resource preceding a degradable resource

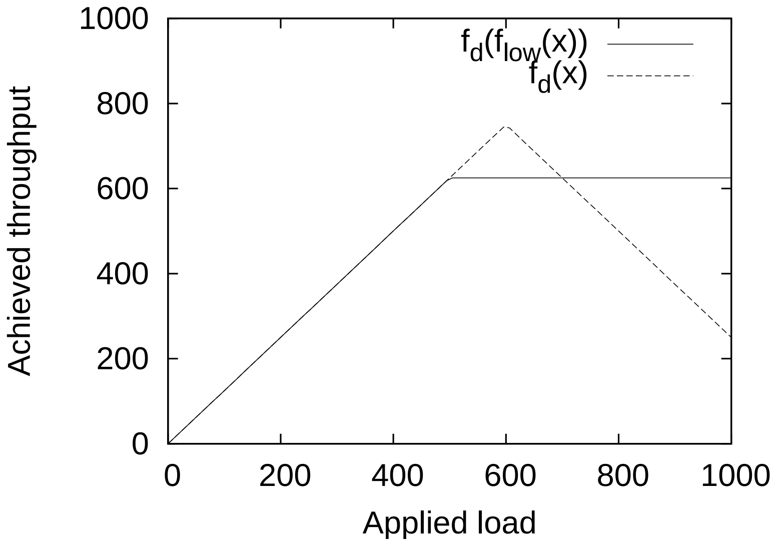

Figure 3.4: A rate-limiting resource preceding a degradable resource, with rate set too low.

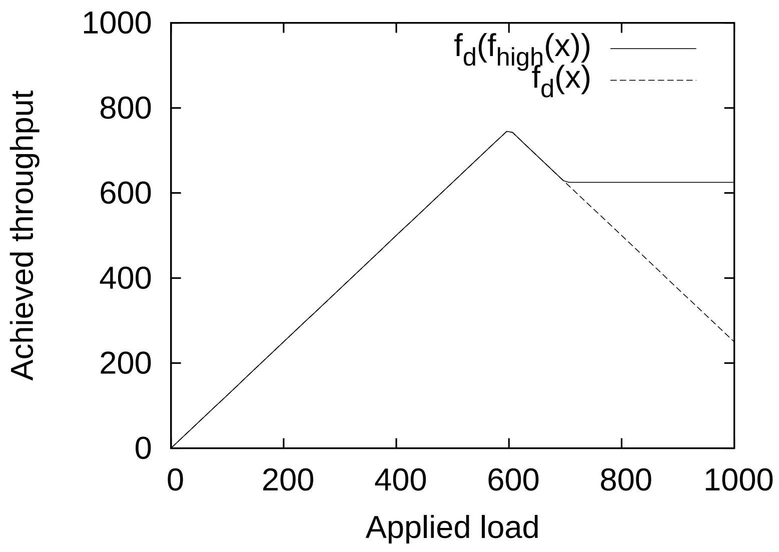

Figure 3.5: A rate-limiting resource preceding a degradable resource with rate set too high.

Figure 3.5 shows the effect of selecting a rate limit which is higher than the maximum of f_d. Rate limiting occurs after the system has begun to exhibit degradation, however once the rate limit is reached, further degradation is prevented.

3.7 Modelling shared resources

The previous section models the behaviour of data-paths in isolation, however in many systems individual resources may be shared between many data-paths. This section will propose a model for the effect of shared resources on the throughput of individual data-paths.

3.7.1 Shared prefix

Consider the scenario shown in Figure 3.6, where two data-paths, y and z are prefixed by path x. Assuming an ideal demultiplexor, the sum of the resource's inputs should equal the sum of the resource's outputs. Non-ideal demultiplexors can be modelled using an ideal demultiplexor and independent resources in series. An example of this situation in a real system is a network protocol stack which demultiplexes data

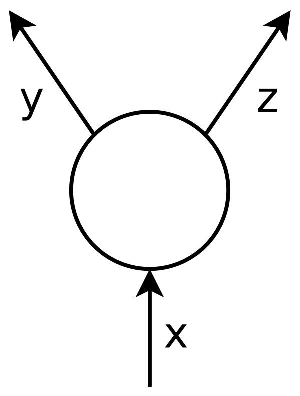

Figure 3.6: Two data-paths sharing a common prefix

between multiple applications.

$$x = y + z \tag{3.2}$$

For some proportion, c in $[0, 1]$, of x destined for y, the data rates y and z are proportional to x:

$$y = cx \tag{3.3}$$
$$z = (1 - c)x \tag{3.4}$$

If the proportion c is beyond the control of the system (as is the case for packets generated externally to the system), there are two ways in which rate limiting can be applied to y. The first option is to simply drop packets between x and y. Doing this will mean that the path x consumes resources handling packets which are later discarded. The second option is to propagate the rate limiting to x, however this necessarily also causes a reduction in the throughput of z.

This means that when two or more data-paths share a common resource, reducing the throughput to one of those data-paths by reducing the throughput applied to the common resource will necessarily reduce the throughput to the remaining data-paths.

The solution to this dilemma is to structure systems such that data-paths avoid sharing a common prefix. Such an approach is reminiscent of systems such as LRP [8] and Nemesis [3], which perform early packet demultiplexing.

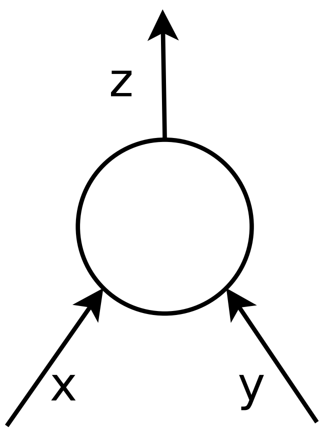

Figure 3.7: Two data-paths sharing a common suffix

3.7.2 Shared suffix

Consider the scenario shown in Figure 3.7, where two data-paths, x and y are suffixed by path z. Assuming the resource itself is not overloaded, the sum of the resource's inputs should equal the sum of the resource's outputs. An example of such a resource in a real system is on the transmit path of a network protocol stack, where multiple senders are multiplexed onto an underlying protocol. For example, when merging TCP and UDP packets into the network interface's output queue.

$$x + y = z \tag{3.5}$$

Unlike the shared prefix case, there is nothing to be gained by performing rate limiting at the point where data paths merge. Instead, the output z should be rate-limited by controlling the rates of x and y. In this case, the proportion of z's throughput assigned to x and y is at the discretion of the resource.

3.8 Summary

This chapter has presented a model of the behaviour of a system comprised of connected resources, and the effects of overload have been explained in terms of that model.

The model makes two important points very clear: that a rate limiting resource may be used to provide peak throughput for a particular data path, and that systems in which data-paths have a shared prefix (or late demultiplexing) make it difficult to prevent the overload of one data path without reducing the performance of other data paths.

4 The Edge-Limiting Approach

This chapter introduces mechanisms which my be used to provide a rate-limiting resource at the edge of the system. In particular this chapter will focus on mechanisms which can be implemented using unmodified commodity Gigabit Ethernet network interface cards. The principles used in this chapter may be applicable to other devices, however they do depend heavily on the nuances of network interface hardware design.

As discussed in the previous chapter, in order to prevent degradation due to overload, it is necessary to implement admission control wherein the rate at which data enters a system is limited to the maximum capacity of that system. This allows the system to provide peak throughput even when the applied load is beyond that peak.

In Chapter 2, we introduced the concept of admission control cost. In order to guarantee that degradation does not occur under overload, an admission control system must have an admission control cost of $O(admitted)$ or better. In order to achieve this, the network interface itself may be used as a non-degradable resource, since it has separate processing resources to the main CPU, and has enough capacity to process a saturated link. Chapter 3 also argued that by performing admission control at the start of the data path all resources along that path could be prevented from exposure to excessive loads.

For these reasons, we propose the use of the network interface card itself as a rate-limiting resource which may be used to perform admission control for the entire system. This chapter will focus on admission control mechanisms which can be implemented in off-the-shelf network interface cards.

Determining the maximum capacity of the system is an orthogonal but equally important problem, and will be dealt with in the next chapter.

4.1 Rate control mechanisms

Since we require the use of commodity hardware, we will examine rate-limiting mechanisms which can be implemented in the driver, and which cause packets to be dropped by the network interface before they can be copied to main memory using DMA.

4.1.1 Introduction to network interfaces

Early Ethernet network interfaces and some modern embedded network interfaces operate by receiving an incoming packet into a local buffer before interrupting the host CPU. The operating system's interrupt handler then invokes the driver, which reads the packet from the NIC into a buffer in main memory before enqueueing it for higher level processing.

This simple design is cheaper and easier to build due to reduced hardware complexity, but is unable to scale to the high packet rates seen in modern Ethernet networks. Polling schemes can be used to increase the number of packets received per interrupt, however the design is fundamentally limited by buffering constraints.

Some optimisations are possible with such interfaces, such as using the host CPU to look at the packet headers in the network interface's packet buffer prior to copying the packet. This could allow cheaper demultiplexing of packets between recipients, or intelligent admission control with reduced loading on the host CPU and memory bus.

Modern Ethernet interfaces use an entirely different scheme for packet reception. Received packets are automatically copied into pre-defined locations in main memory by the network interface. These memory locations are defined by descriptors in the DMA ring, a data structure usually maintained in main-memory and used to communicate between the driver and network interface. Conceptually, the DMA ring is a circular list, implemented as a circularly-linked list or simple array, with the specific structure dictated by the network interface itself. Each entry in the DMA ring is a descriptor containing a pointer and meta data describing a location in memory which may be used by the network interface to store an incoming packet. Minimally, the meta data includes the size of the buffer and whether it is owned by the network interface or the host. In general, the buffers pointed to by the DMA ring are large enough to store one incoming packet, such that one DMA ring entry will contain one received packet.

Interrupt generation on modern network interfaces is generally programmable, with different interrupt mitigation schemes used by different manufacturers. Interrupt mitigation typically allows the maximum interrupt rate generated by the network interface to be specified, but may also include various thresholds to prevent the delay of packet processing under low load [14, 25].

4.1.2 Generalised rate limiting

The following rate-limiting mechanisms rely on the behaviour of the network interface when all receive buffers in the DMA descriptor ring have been used, which is to drop any subsequent incoming packets. This approach gives the behaviour of a

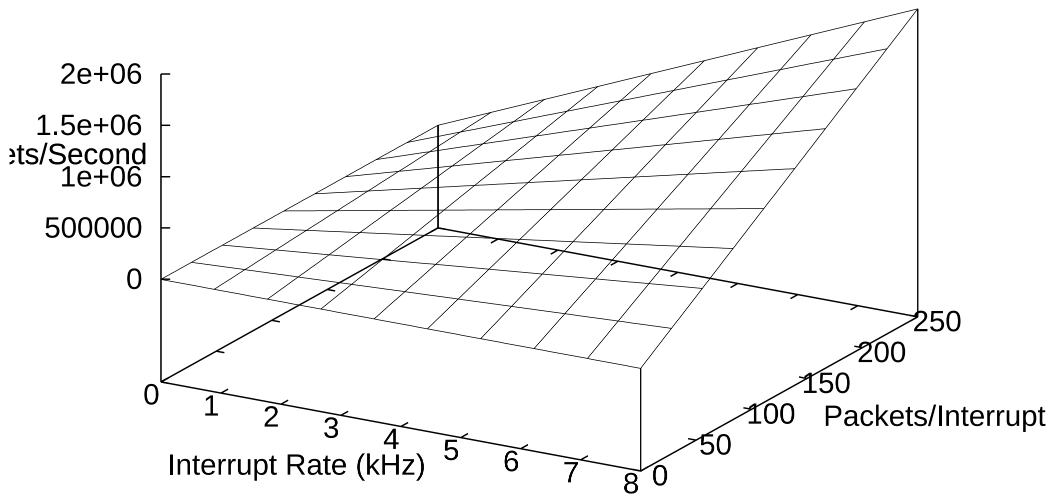

Figure 4.1: Model of rate-limiting mechanism

non-degradable resource, because packets which are discarded do not consume any additional system resources. Resource consumption of the discarded packet is limited to processing on the network interface, which is capable of handling the full load of the network link.

If n packets are dequeued per interrupt, and interrupts occur at frequency f, the data rate d can be calculated:

$$d = n \times f \tag{4.1}$$

Figure 4.1 shows the relationship between the achieved packet rate d, the number of packets dequeued per interrupt n, and the interrupt frequency f for Equation 4.1. Given this model, we can choose to control the incoming packet rate by adjusting the number of packets dequeued per interrupt, or by adjusting the interrupt frequency.

All of the following rate-limiting mechanisms operate on the same principle. This means that under most circumstances there is little performance difference between the approaches. On the other hand, there may be significant non-performance reasons to choose one mechanism over the other. The choice of rate control mechanism should be made primarily on the basis of how well the approach suits the existing hardware design and driver framework.

4.1.3 Controlled dequeue rate

This rate-limiting mechanism is implemented by controlling the rate at which packets are dequeued from the DMA ring by the driver.

Traditional driver implementations aim to dequeue as many packets from the DMA ring as possible before returning from the interrupt handler. We modify this behaviour by dequeuing only the number of packets required to maintain the desired data rate. Excess packets are left in the DMA ring, while consumed packets are freed in the DMA ring. If the incoming data rate exceeds the packet dequeue rate, the DMA ring becomes full, and the network interface begins dropping packets. If the network supports flow control, the rate limit will be propagated through the network, causing the sender to be blocked.

The main advantage of this approach is that it is extremely simple to implement, requiring only a few lines of code in the driver's interrupt service routine.

The disadvantage of this approach is that it incurs additional per-packet latency if the applied load is greater than the admitted load, as packets must sit in the DMA ring until the driver is ready to process them. According to Little's law [18], the incurred latency T is dependent on the size of the DMA ring N, and the admission control rate λ, as shown in Equation 4.2.

$$N = \lambda T \tag{4.2}$$

Figure 4.2 shows the latency incurred while a packet waits to be dequeued by the driver, as modeled by Equation 4.2. If the admission control rate is low relative to the number of DMA buffers in use, the latency incurred can be significant.

The impact of added latency depends on the network protocol and application being used. In the case of protocols which do not retransmit in the event of packet loss (for example, a time synchronisation protocol), minimising the time packets spend waiting in the DMA ring can be useful. On the other hand, for protocols such as TCP, the fact that some packets must be dropped under overload means that end-to-end latency is determined by retransmission timeouts, and latency added to the delivery of an individual packet has little impact on overall performance.

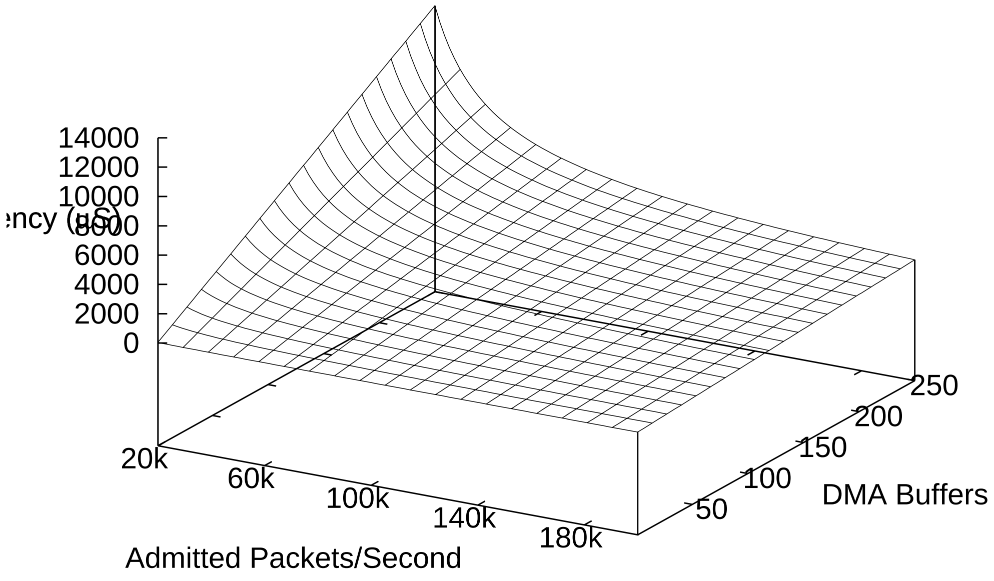

Figure 4.2: Latency incurred by the controlled-dequeue-rate approach

4.1.4 Controlled free rate

An alternative to the controlled dequeue rate approach is to dequeue packets as soon as they are available, but delay notification of packet consumption to the network interface. In this solution packets are freed in the DMA ring at the desired packet rate.

Because network interfaces typically consider a slot in the DMA ring to be either full or empty, the driver must introduce a third state to keep track of slots which are no longer in use, but which have not yet been returned to the network interface for reception of additional packets. How cleanly this can be implemented is somewhat dependent on the structure of the DMA ring, whose design is dictated by the network interface.

The primary advantage of controlling the free rate is that it does not incur any additional latency, since packets may be dequeued as soon as the interrupt is raised, rather than waiting in the DMA ring, as is the case for the controlled dequeue rate approach.

4.1.5 DMA-ring length modulation

A variant of these approaches is to control the number of DMA buffers which are available in the receive ring. At each interrupt, the entire DMA ring is dequeued and freed. This approach is suitable when receive processing interrupts occur at well known intervals, as is the case for many Gigabit Ethernet interfaces which support interrupt moderation.

As in the controlled free rate approach, the difficulty of implementing DMA-ring length modulation is largely dependent on the interface provided by the network interface.

Figure 4.3 shows the time packets spend waiting in the DMA ring before being dequeued for the DMA-ring length modulation approach. The worst case latency added by rate limiting is equal to the interrupt period. This occurs when a packet must wait the entire interrupt period before being dequeued, and is no worse than that incurred by normal interrupt moderation.

4.1.6 Interrupt frequency modulation

Equation 4.1 shows how the data rate can also be controlled by modulating the frequency of interrupts, while maintaining a fixed number of entries in the DMA ring.

The advantage of this approach is that on most Gigabit Ethernet network interfaces, controlling the interrupt rate is simply a matter of setting a value in a

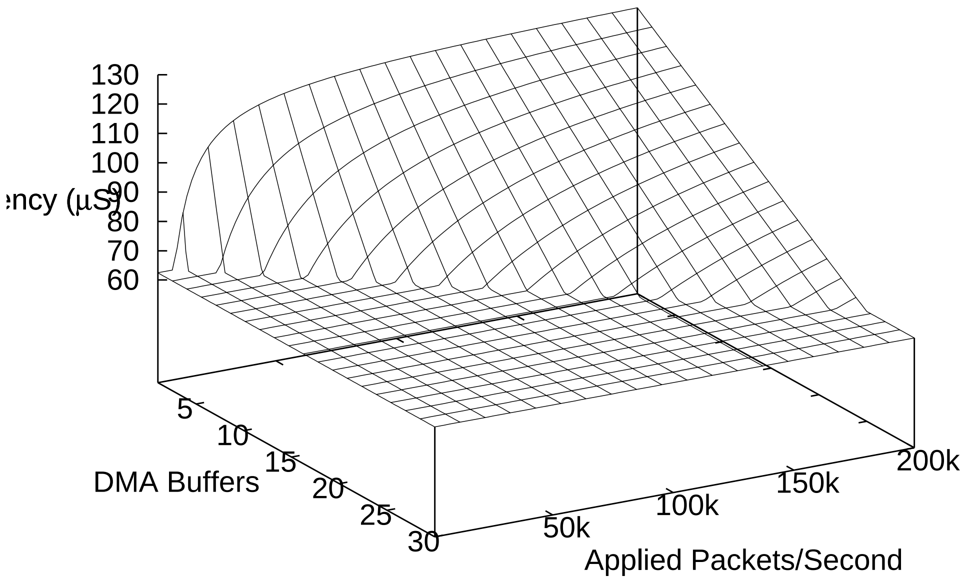

Figure 4.3: Latency incurred in the DMA ring (8kHz interrupts)

device register. This means that the only change necessary in the device driver's interrupt service routine (ISR) is to prevent more than one iteration over the DMA ring from occurring. This is necessary to limit the number of packets dequeued per interrupt, since most driver implementations attempt to reduce the interrupt rate by not returning from the interrupt handler until the input queue is empty.

There are several disadvantages to this approach. It results in an interrupt frequency which increases as the admitted data rate increases. It therefore tends to increase load on the system as overload is approached (although it still maintains O(admitted) performance).

This approach also has the disadvantage of increasing latency at low admission control rates, as the interrupt frequency may need to be very low for low packet rates.

4.1.7 A hybrid approach

It is also possible to implement rate control by modulating both the interrupt frequency and DMA ring length simultaneously. Such an approach can have the advantage of allowing increased interrupt frequency under low load in order to minimise latency, while reducing interrupt overheads at high loads.

This approach goes beyond simply eliminating overload and has become an exercise in performance optimisation. The performance of this approach is evaluated in Section 7.6 using a controlled-free-rate admission controller and interrupt frequency modulation.

4.2 Queueing behaviour

It is worth noting that the rate limiting mechanisms presented in this system all provide "tail drop" queueing semantics, where the most recently arrived packet is dropped. These semantics are typical of those used by default in many internet routers.

While tail drop is extremely popular in existing internet routers, there are a few well known disadvantages to this approach [1]. The first disadvantage is that the phenomenon of lock-out, where a small number of traffic flows may be able to monopolise space in the queue. The second problem is that tail drop results in full queues, which means that bursty traffic tends to be dropped.

Proposed solutions to these problems include active queue management approaches such as Random Early Detection [11] (RED), and congestion notification mechanisms such as Explicit Congestion Notification [10,24] (ECN). The implementation of such approaches is not possible using the edge limiting approach without explicit queue management on behalf of the network interface.

4.3 Limitations

Commodity Ethernet interfaces typically only support a single DMA ring, and as such that resource must be shared by all data paths used by that interface. We are not aware of any commodity Ethernet cards which support hardware packet demultiplexing based on high level network protocol fields, however some interfaces do support demultiplexing packets into individual DMA rings based on the Ethernet packet's virtual local area network (VLAN) tag.

There has been research into providing hardware packet demultiplexing [23] by replacing the firmware of certain programmable Ethernet cards. This work has been done in order to support user-level network protocol implementations in experimental operating systems, where such packet demultiplexing allows packets to be delivered to different user-level protection domains for processing.

In Section 3.7 we discussed the effects of having shared resources along a data path. Because the hardware shares a single DMA ring between multiple data paths, it is not possible to apply separate rate limits to individual data paths. Therefore, when rate limiting is implemented using the network interface's DMA ring, the throughput of all data paths which are prefixed by that DMA ring is limited. This situation could be resolved by having hardware that supports packet demultiplexing into multiple queues, or by limiting each network interface to a single data path.

In practice many applications, such as those typically used in network servers, put the majority of traffic through a single data path. Therefore despite this limitation, this approach achieves useful results.

5 Rate Selection

While a rate-limiting mechanism makes it possible to prevent excess traffic from entering the system, it is also necessary to determine exactly how much traffic should be allowed into the system in order to prevent overload and maximise throughput, as discussed in Section 3.6. This chapter will briefly discuss existing methods of detecting overload. It will then look at a new approach which uses throughput based traffic analysis to determine when a system has become overloaded.

5.1 Overload detection

There are many possible ways to detect overload within a system. One is to have each stage along the data path detect when it is overloaded.[1] When a stage becomes overloaded, that information could be propagated backwards along the data path to the data source. This is the approach taken by the staged event driven architecture (SEDA) [30]. Although detection of overload within an individual stage may be simpler than detecting overload in the system as a whole, retrofitting such detection to an existing system is difficult, as it necessarily requires modifying the interfaces and behaviour of all components in a data path.

One proposed approach to solving this problem is to use resource monitoring to detect when a system has become overloaded [26,27,28,29]. These systems typically monitor resources such as CPU usage as an indicator of system load. Unfortunately, resource monitoring is often a poor indicator of overload, since many optimisations take advantage of any spare resources which are available. Additionally, resource consumption may not be a function of network load if resources are being consumed by some independent process.

An alternative approach is to use traffic monitoring to detect overload. Such a monitor observes the response of the system to incoming traffic in order to determine when the system is performing sub-optimally. If overload is detected, the monitor can reduce the data rate entering the system by directly controlling data source(s) in the system.

The advantage of this approach is that it is relatively simple to retrofit to an existing system, and it is more accurately able to detect the occurrence of overload

[1]This ignores the issue of how overload is detected within an individual stage.

than simple resource monitoring schemes.

5.2 Traffic monitoring

The goal of the rate selection algorithm is to determine the maximum throughput of the system at runtime. A rate-limiting mechanism may then be used to limit the input rate to the maximum throughput of the system.

There are a number of properties which may be useful to monitor in order to detect overload. Such properties include throughput, latency, request rate vs. response rate, and other properties depending on the nature of the network traffic being monitored. Traffic analysis may be performed passively, using traffic already entering the system, or actively, by injecting additional probe traffic into the system.

Once a given property can be monitored, the rate selection algorithm needs to know what behaviour is indicative of overload. In the case of throughput, the characteristic behaviour of an overloaded resource is throughput which decreases with increased load. In the case of latency, a significant jump in latency typically occurs as the system becomes overloaded. We expect that other properties will provide additional behaviours which can be used to detect overload.

Our system uses throughput monitoring to detect overload, rather than the latency monitoring approach which has been used in some existing systems. The primary advantage of throughput monitoring is that it has extremely low overheads when compared to latency monitoring. In order to monitor latency, it is necessary to track request-response pairs. Such tracking incurs high overheads due to the detailed analysis of traffic passing through the system which must occur.

A worst case scenario of such latency monitoring would be tracking the response time of a HTTP/1.1 compliant web server. In this case, a complete implementation would require full TCP stream reconstruction in order to search for individual requests and responses within the stream. For a simpler protocol such as HTTP/1.0, which performs only a single request per TCP connection still requires decoding the IP and TCP headers, maintaining the state of all active TCP connections, and measuring the time between TCP connection establishment (SYN) and connection close (FIN) packets.

In contrast to latency measurement, throughput monitoring can be accomplished cheaply, requiring only a few cycles per packet to increment packet counters, greatly reducing the amount of state information which must be maintained.

5.3 Control approaches

Throughput monitoring is a control problem which requires finding the input throughput which produces the maximum output throughput for the system. This is actually a difficult problem from a control theory perspective, since the behaviour of the system is not well defined. There are a number of reasons for this:

- The system's response to load is affected by the particular hardware configuration being used. Adding physical resources will usually increase the maximum throughput which may be sustained.
- The system's response is affected by the particular software configuration of the machine. Changing the application software and services will change the response of the system. This means the response function will change as a result of changes to the system configuration by the system administrator.
- The system's response is affected by the current workload. Different network requests will produce very different behaviours within the system.

These factors combine to make it infeasible to provide a static definition of the response of the system to input load, such that offline analysis approaches cannot be applied in the development of a control approach.

This leaves us with several potential approaches to the problem which do not require offline analysis:

1. Learn (and relearn) a model of system behaviour at runtime, which may then be used to calculate the optimal input rate [16].
2. Use a fuzzy-logic approach to emulate a human decision making process [7,22].
3. Develop a domain-specific control approach, taking advantage of properties specific to this domain.

Since we have useful domain-specific knowledge about the behaviour of our system which is largely independent of the configuration of the system, we have focussed on the implementation of a domain-specific control approach. The implementation and testing of other control approaches is left as future work.

5.4 Rate selection algorithm

Our rate selection algorithm seeks to solve the problem of maximising the output throughput of a system by controlling the input throughput seen by the system.

The previous section described the difficulties associated with solving this problem for traditional control approaches, and has outlined the need for a domain-specific control approach to solving this problem.

Fortunately our domain has certain properties which may be used to assist in the development of a domain-specific control approach. Significantly, Chapter 3 has already given characterisations of degradable and non-degradable resources which may be used by a domain-specific control approach. Specifically, we know that the throughput based characterisations of degradable resources have positive derivatives when not overloaded, but negative derivatives when overloaded.

5.4.1 Maximum power-point tracking problem

As it turns out, a very similar control problem exists in the power electronics domain. Maximum power-point trackers (MPPTs) are a special class of DC-DC converters which are used to interface a photovoltaic array with batteries or other electronics. The control problem which MPPTs aim to solve is the maximisation of output power of a photovoltaic array based on the control of array terminal voltage.

Figure 5.1 shows array power as a function of array terminal voltage with constant insolation and temperature for a simulated photovoltaic array. The most important features from a control algorithm perspective are that $\frac{dP}{dV}$ is zero at the maximum power point, and that power decreases at an increasing rate as the terminal voltage moves away from the maximum power point.

5.4.2 Maximum power-point tracking solution

The perturb and observe (PO) control algorithm [13,9] is used by photovoltaic maximum power-point trackers (MPPTs) to match a photovoltaic array to an electrical load under varying insolation and temperature conditions, in order to maximise the power output of the photovoltaic array.

In MPPTs, the perturb and observe algorithm operates by perturbing the array terminal voltage and observing the corresponding change in array power. If the perturbation produces an increase in output power, then the direction of perturbation is maintained. If the perturbation results in a decrease in array power, it is concluded that the perturbation has moved away from the photovoltaic array's maximum power point, and the direction of perturbation is reversed. Because the PO algorithm works by selecting values above and below the maximum, the average power output is necessarily slightly lower than the theoretical maximum.

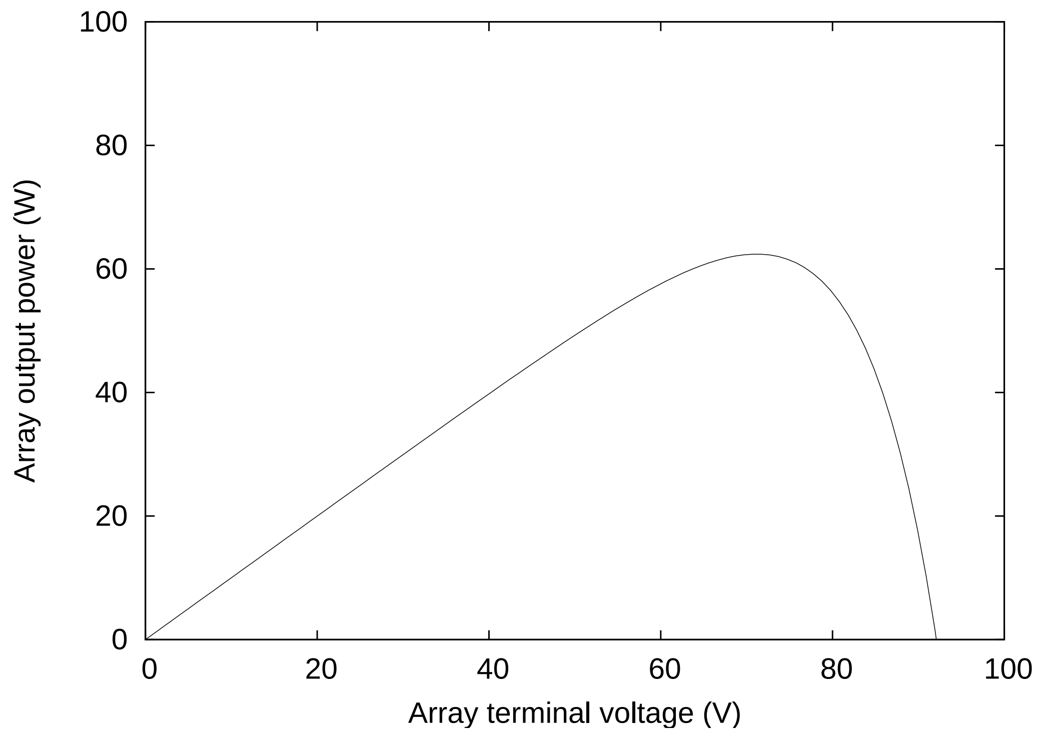

Figure 5.1: Example photovoltaic Power vs. Voltage function

5.4.3 Application to rate selection

Our control problem shares some similarities to the MPPT problem. Where the MPPT aims to maximise output power, we aim to maximise output throughput. Where the MPPT is able to control array terminal voltage, we are able to control input throughput. Where the maximum power point varies with insolation and atmospheric conditions, the maximum throughput point varies with the behaviour of the application and the requests contained in the incoming network traffic. In both applications, the control algorithm attempts to find the peak output over a family of related functions whose derivative is positive to the left of the peak, and whose derivative is negative to the right of the peak.

Our scenario, however, is slightly different to that of the photovoltaic application; the response time of our system is longer and may be variable, since there is considerable buffering of requests and results within a system. This means that changing the input request rate will not have an immediate effect on the system's output.

While the MPPT has complete control of the array terminal voltage, we can only control the maximum input throughput. Therefore, if the applied throughput is greater than the maximum throughput being allowed by the rate limiter, the throughput seen by the operating system is exactly the maximum throughput allowed by the rate limiter. On the other hand, if the applied throughput is lower than the maximum throughput allowed by the rate limiter we can measure the throughput seen by the operating system, but we are not controlling it directly.

Fortunately, this situation can be used to our advantage. A drop in input throughput, whether it is caused intentionally or by as a result of deliberate perturbation, can be used to detect a corresponding change in output throughput. In some situations this allows us to determine that the maximum throughput point has changed without needing to discard any additional packets.

5.4.4 Algorithm tradeoffs

Changes of workload have the effect of changing the maximum throughput slightly. The time taken by the perturb and observe approach to adjust to changes in workload is proportional to the size and frequency of the perturbations. This gives a tradeoff between speed and accuracy, with larger perturbations being less accurate but more responsive to changes in input load.

Clearly, the PO approach will respond best to gradual changes in workload which occur over several seconds, while adjustment to large instantaneous changes of workload will be slower. Such gradual changes are more likely to occur in Internet services where load is determined by changing user demands, however there are probably other scenarios where workloads do change more rapidly, such as in compute clusters.

6 Implementation

A prototype implementation was completed by modifying the FreeBSD operating system. The modifications consisted of implementation of a rate limiting mechanism in the device-driver's packet receive path, and a control thread implementing our rate selection algorithm.

6.1 Rate limiting

The rate limiting mechanism is implemented using the controlled-dequeue-rate approach, by fixing the interrupt rate using the interrupt rate control registers on the network interface, and by modulating the number of packets which may be dequeued per interrupt.

In our driver this is done in the receive interrupt processing function, which normally iterates over the receive DMA ring, dequeueing all available packets from the DMA ring, and enqueueing them for further processing by the network protocol stack. This function was modified to stop iteration once a pre-determined number of packets had been dequeued, leaving any remaining packets in place in the DMA ring.

The number of packets to dequeue is stored in a per-adapter data structure, which is shared with the rate selection thread. This data structure is also used to store related statistics such as the number of packets received and transmitted, and the number of interrupts that have occurred since the last iteration of the rate selection algorithm.

6.2 Rate selection

The rate selection algorithm runs in a single thread that monitors the behaviour of the entire system. The control algorithm is invoked at 100Hz using the kernel's thread sleep mechanism. At each invocation, the control thread first iterates over the adapters, locking, accumulating and unlocking the statistics data structure. The number of packets transmitted since the last iteration of the rate selection algorithm is then calculated, and used to determine whether the derivative with respect to time is positive or negative.

The control thread maintains a current direction of perturbation, either increasing or decreasing. If the derivative is positive, the direction of perturbation is unchanged, while if the derivative is negative the direction is reversed. In the case that the derivative is zero, the direction of perturbation is set to decreasing.

The number of buffers to dequeue per interrupt is then incremented or decremented based on the newly calculated direction of perturbation. In the event that this results in exceeding a defined minimum or maximum number of buffers to dequeue per interrupt the value is capped, and the direction of perturbation is reversed.

Finally, the adapter list is again iterated over, and the number of packets to dequeue per interrupt is stored in the per-adapter data structure, where it is later used by the rate-limiting mechanism.

7 Experimental Evaluation

In order to evaluate the ability of our admission-control approach to scale to very high levels of applied load, we have implemented it in the FreeBSD kernel by modifying the Intel Gigabit Ethernet adapter device-driver.

Ideally, we would like our admission control approach to cause the overall system to scale with admitted traffic rather than the load applied to the system. Results which show that performance degrades in the absence of the admission controller, but does not degrade in the presence of the admission controller will demonstrate the effectiveness of our approach.

Our evaluation will focus on establishing the effectiveness of our approach in preventing overload, the genericity of the approach to a number of existing protocols, and on measuring the performance impact of admission control under normal operating conditions. Comparisons will be made between the behaviour of a standard FreeBSD kernel and a FreeBSD kernel which has been modified by the addition of edge-limiting admission control in the network interface device driver.

7.1 Kernel configuration

For the purpose of evaluating our system, we will be comparing two different configurations of the FreeBSD kernel, which we refer to as the *standard* and *dynamic* configurations. All benchmarks are run using the same kernel configurations, and no hand-tuning of the control algorithm has been done for specific applications. This section will summarise the similarities and differences between these configurations.

The *standard* configuration has minimal changes when compared to the generic kernel configuration[1]. The primary change is that the timer interrupt rate was increased from 100Hz to 1000Hz. This change is recommended by FreeBSD developers to improve performance of network servers, especially if polling is being used. More recent FreeBSD kernel revisions have used 1000Hz as a default setting.

The interrupt moderation setting of the Ethernet driver was left at the default 8000 interrupts per second. Reducing this value could be expected to improve peak performance by decreasing interrupt overheads.

[1]The generic kernel is the *GENERIC* kernel and corresponding kernel configuration file distributed with *FreeBSD 5.3-RELEASE*.

The *dynamic* configuration implements our proposed solution, and consists of the *standard* configuration, with the addition of a controlled-dequeue-rate based rate-limiting mechanism in the driver, a control thread to determine the appropriate receive rate based on throughput monitoring, and some statistics collection required by the control loop. The interrupt moderation settings remain the same as for the *standard* configuration.

These configurations have been selected to be a fair and reasonable representation of typical server configurations. The standard configuration is a typical FreeBSD install that might be found on an Internet server. The dynamic configuration is modified only by the addition of device driver modifications to implement admission control. By keeping our modifications to a bare minimum, a fair evaluation of the performance impact of our admission control approach can be made.

7.2 Hardware description

The FreeBSD machine being tested was based on an Intel Xeon 2.66GHz processor with hyper-threading disabled, 1GB RAM, and an Intel PRO/1000 Gigabit Ethernet adapter[2] connected via PCI-X. A D-Link DGS-1216T managed Gigabit Ethernet switch was used, and was configured with pause frames disabled on all ports. This was done to prevent flow control information from propagating within the network, which was necessary to ensure that the full range of loads could be applied to the machine being tested.

7.3 Firewall benchmark

The firewall benchmark is designed to simulate an overloaded network firewall. The benchmark utilises *ipbench*[3] [31] to generate a prescribed load on the firewall, then measures the throughput and CPU utilisation at that load.

7.3.1 Firewall configuration

Figure 7.1 shows how our hardware was configured for benchmarking. The configuration consists of two VLANs, each containing four hosts connected via a Gigabit Ethernet switch. The two VLANs are connected via a single FreeBSD machine acting as router and simple firewall. The hosts on the first VLAN run the ipbench

[2]Both ports of a dual-port card were used for the firewall benchmark, two ports of a quad-port card were used for the NFS benchmarks.

[3]More information on ipbench can be found in Appendix A.

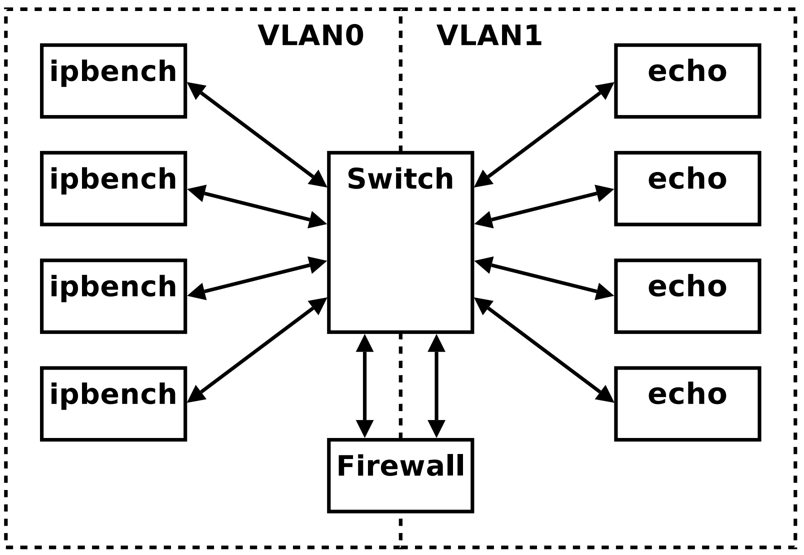

Figure 7.1: Firewall configuration.

distributed network benchmark's UDP load generator, while the hosts on the second VLAN run UDP echo servers.

The FreeBSD firewall is configured to use the *pf* packet filter, with a simple rule set which checks that all packets have reasonable and valid protocol headers, and that the source and destination addresses are valid before forwarding. The specific rule-set used is shown in Figure 7.2.

The ipbench distributed benchmark was run on the hosts in VLAN0. These hosts generate 512-byte UDP packets at a specified data rate, which we refer to as the *applied load*. The UDP packets traverse the switch and firewall, to the hosts on VLAN1, which are running UDP echo servers. The UDP echo servers simply send an identical copy of the UDP payload back to the ipbench host, which again must travel via the switch and firewall. The ipbench host then records the rate of echo replies, which we refer to as the *achieved throughput*.

We note that packets must traverse the firewall twice before they are measured as achieved throughput. It is also worth noting that the firewall's input packet rate on VLAN0 will be higher than the firewall's input packet rate on VLAN1 when the firewall is overloaded. This is because some packets from VLAN0 will be dropped before they are seen by VLAN1. Therefore, those packets which are dropped before reaching VLAN1 will never be echoed, and hence cannot generate additional load on the firewall. This advantages the standard configuartion, since as it becomes overloaded and drops more packets, it receives a reduced load from VLAN1.

```
#normalize incoming packets
scrub in all

#block everything by default
block all

#pass all on loopback and management port
pass on {lo0 fxp0} all

#only allow 192.168.0.0/24 on em0's network
pass  in  on em0 from 192.168.0.0/24 to any
pass  out on em0 from any to 192.168.0.0/24

#only allow 192.168.1.0/24 on em1's network
pass  in  on em1 from 192.168.1.0/24 to any
pass  out on em1 from any to 192.168.1.0/24
```

Figure 7.2: Firewall rule-set

7.3.2 Firewall measurements

This section will discuss measurements taken to evaluate our implementation. We begin by establishing a performance baseline for the *standard* configuration, by comparing applied load with achieved throughput. We then compare our *dynamic* configuration to the *standard* configuration on the basis of achieved throughput, CPU utilisation and cycles per delivered packet, under a variety of applied loads.

Figure 7.3 shows achieved throughput as a function of applied load in the firewall benchmark, for both the *standard* and *dynamic* configurations. The *standard* line establishes the baseline throughput of the firewall benchmark. The results for the *standard* configuration show that the system does indeed exhibit significant performance degradation under overload.

The *dynamic* line in Figure 7.3 shows the effect of introducing our rate-limiting to the system. The *dynamic* line shows that limiting the receive rate by controlling the number of packets dequeued per interrupt causes the receive DMA ring to behave as a non-degradable resource, preventing performance degradation under overload.

Our approach achieves 97.3% of the ideal result of matching the peak throughput of the *standard* configuration, while avoiding the effects of performance degradation under overload. The slight decrease in peak throughput when compared with the *standard* configuration can be attributed to our rate selection algorithm needing to occasionally select throughputs above and below the maximum rate achieved by the *standard* configuration in order to detect overload.

We measure CPU utilisation by counting the percentage of cycles spent in a

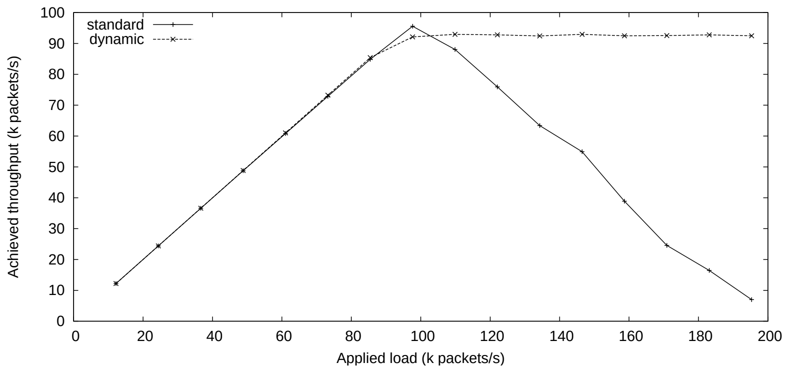

Figure 7.3: Firewall: achieved throughput vs. applied load

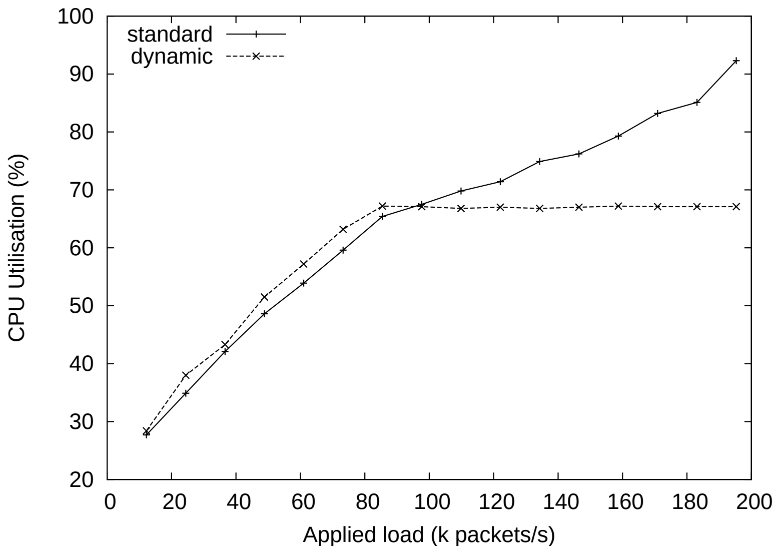

Figure 7.4: Firewall: CPU utilisation vs. applied load

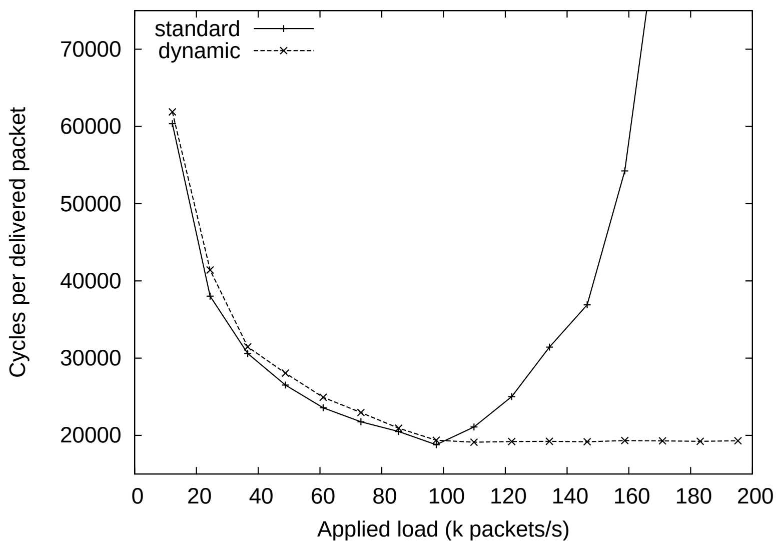

Figure 7.5: Firewall: Cycles per delivered packet vs. applied load

low-priority user process. This lets us accurately determine the number of cycles left for other user processes for a given load. In Figure 7.4, we see that the *dynamic* configuration incurs slight (on average 2.1% additional CPU time) overheads when compared to the *standard* configuration, however once overload is reached, the *dynamic* configuration does not increase its CPU utilisation with applied load, showing that our rate-limiting mechanism is effective in preventing the system from exposure to excessive network loads.

Once the system becomes overloaded, the *standard* configuration begins wasting cycles on packets which are later discarded. This can be seen in Figure 7.5, where cycles per delivered packet increases considerably with applied load. Meanwhile the *dynamic* configuration maintains consistent overheads even when overloaded, since it does not waste resources handling packets which are not processed to completion.

Figure 7.6 shows the effect of overload on latency. The admission control implementation used in this case was controlled-dequeue-rate, as described in Section 4.1.3. As predicted, this rate limiting approach causes additional latency due to packets waiting in the DMA ring to be processed.

The measurements shown in Figure 7.6 are for packets which are traversing the firewall twice. For those two traversals the additional latency is around 3.25ms. This is in line with the 2.85ms additional latency predicted by Equation 4.2, since the return path's rate data-rate is close to the rate limit, such that additional latency

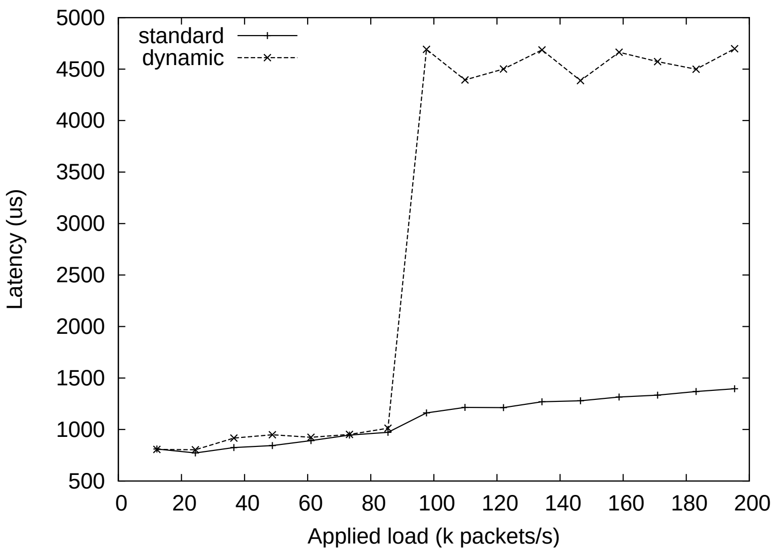

Figure 7.6: Firewall: Round-trip latency vs. applied load

is not incurred by rate-limiting on the return path.

It is worth noting that latency only increases under overload, where the dynamic configuration is returning more traffic than the standard configuration. The infinite latency of unreturned packets is omitted.

7.4 NFS benchmark

The NFS benchmark is designed to test the behaviour of an NFS server under overload. We utilise ipbench to generate very high NFS request rates, and monitor the corresponding response rate.

7.4.1 NFS configuration

The NFS benchmark configuration consists of nine load generators running ipbench, a Gigabit Ethernet switch, and a FreeBSD NFS server, as shown in Figure 7.7. The load generators are connected to the Gigabit Ethernet switch, which is in turn connected to the NFS server via two Gigabit Ethernet links, which are trunked using FreeBSD's Fast EtherChannel (FEC) netgraph module.

It is important to note that rate-limiting is performed in the network driver, which is not aware of the existence of FEC. We could have instead configured the

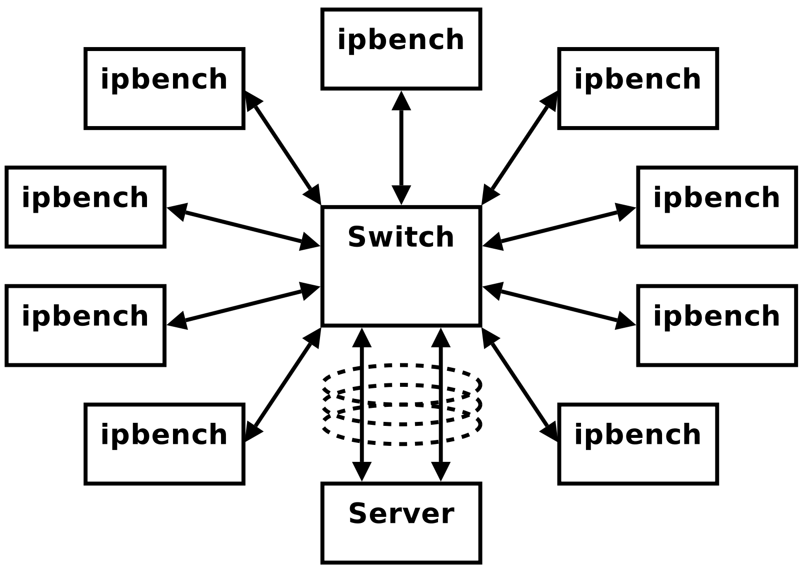

Figure 7.7: NFS configuration

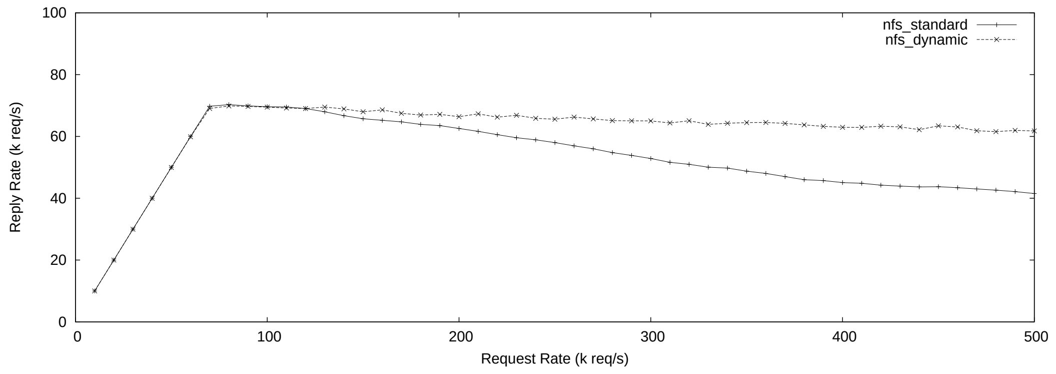

Figure 7.8: NFS: replies vs. requests

NFS server with multiple independent Internet addresses, however we chose to use port trunking because it allowed us to generate loads exceeding one Gigabit-per-second while maintaining a simple benchmark configuration.

The NFS server is configured to serve data from an MFS RAM-backed filesystem. This configuration was chosen primarily because we did not have access to a high-performance RAID array for benchmarking purposes.

Load was generated using ipbench, which was configured to generate read and write requests with equal probability. Individual request sizes were randomly selected between 16 and 1024 bytes. All NFS operations were performed on a single 400 megabyte file.

7.4.2 NFS measurements

The results of the NFS benchmark are shown in Figure 7.8. We see that the results of the NFS benchmark have similar characteristics to the results of the firewall benchmark. For both the *standard* and *dynamic* configurations, response rate matches the request rate for request rates which are less than the maximum capacity of the system (in this case, seventy thousand requests per second). Once the system becomes overloaded, the *standard* configuration experiences gradual and continuous performance degradation as the applied load increases.

The rate of performance degradation of the *standard* configuration is considerably less than that seen in the firewall benchmark. This tells us that the kernel is doing less work on incoming requests before discarding them than is the case for the firewall benchmark. While the rate of performance degradation has improved, it is clear that performance degradation under overload is significant. Changing the amount of work done by enabling additional packet processing features (such as a packet filter) would increase the rate of degradation observed.

We observe that the *dynamic* configuration still displays a small amount of performance degradation under overload. Because all measurements are performed by ipbench, results indicate the behaviour of the entire benchmark configuration. Therefore the observed decrease may be the result of packet loss which occurs externally to the NFS server.

Table 7.1 shows a breakdown of the fields in an NFS request generated by the benchmark. The average size of the randomly generated requests is 370 bytes. Assuming Gigabit Ethernet can carry 1×10^9 bits per second, or 1.25×10^8 bytes per second, this gives an upper bound of $1.25 \times 10^8/370 = 337837$ NFS requests per second per Ethernet link for our workload.

Moreover, the FEC trunking being used to bond the two Ethernet channels uses source and destination MAC addresses to determine which channel a given

Component	Read Size	Write Size
Ethernet header	14B	14B
IP header	20B	20B
UDP header	8B	8B
RPC header (approx)	32B	32B
NFS header (approx)	32B	32B
Expected payload size	0B	520B
Ethernet CRC	4B	4B
Total	**110B**	**630B**
Average	**370B**	

Table 7.1: Total size and breakdown of NFS request fields.

packet should traverse. Given that we have an odd number of load generators, one of our two Ethernet links must become saturated before the other.

The observation that we are operating near the saturation point of Ethernet is important in understanding these results, since under these conditions, we expect very high contention and hence packet loss within the switch. This is certainly the case for request traffic, since it is generated by multiple unsynchronised senders.

7.4.3 Pause frames

Pause frames are a special type of Ethernet frame which tells the receiver to stop sending packets for a small period of time, so that the sender of the pause frame may have adequate time to process incoming packets.

Pause frames can have a significant impact on the behaviour of benchmarks which are aimed at measuring system behaviour under overload. Unfortunately, it is difficult to make generalisations about behaviour of Ethernet networks when pause frames are enabled.

Pause frame support is not required by IEEE 802.3x for standards compliance, so many implementations lack the ability to generate or receive pause frames. Vendor support for IEEE 802.3x pause frames varies between different manufacturers and models of switches [6]. Systems may ignore pause frames entirely, generate pause frames only when incapable of processing incoming packets at link speed, not generate pause frames at all but comply with pause frames by suspending transmission temporarily if requested, or provide complete support for pause frames.

Pause frames were originally intended to prevent packet loss on high speed Ethernet links where the receiver is unable to keep up with the sender. It is intended as a single-link solution, rather than an end-to-end flow control solution. That is, it is only intended to operate between devices connected directly via Ethernet cable, such as a host and a switch, rather than between hosts on a switched network.

If pause frames are propagated within the switch, "External Head of Line Blocking" can occur. This is caused by pause frames preventing the receiver from transmitting any traffic, not just from transmitting traffic to the blocking port. Doing so can degrade performance of an entire LAN when only a single port is actually overloaded. These issues have been addressed by researchers [21], however alternatives to pause frames have not yet appeared in the ethernet standards, and are therefore not supported by commodity hardware.

Our relatively low-end switch does generate pause frames when it experiences internal congestion on a given port. This has the effect of propagating the flow control afforded by pause frames through the switch once the switch runs out of transmit buffers for the port connected to the overloaded host.

Because the *standard* configuration dequeues packets as quickly as they arrive, the network interface rarely sends pause frames to the switch, so that there is no limit to how much load can be placed on the interface. By enabling pause frames, this switch allows us to limit the amount of traffic which can be sent by the load generation hosts in our benchmark.

Because our rate-limiting approach allows the incoming DMA ring to fill, the network interface generates pause frames which effectively limit the data rate which may be applied to the system. This flow control information is propagated through the switch to the load generation hosts, preventing them from generating additional load.

It should be noted that while pause frames reduce packet loss on individual Ethernet links, they do not provide any guarantees of lossless transmission within an Ethernet network, especially in the case of multiple hosts transmitting to a single switch port.

Figure 7.9 shows the effects of enabling Ethernet pause frames on the NFS benchmark. Once the dynamic configuration begins rate-limiting its input, pause frames are generated by the network interface, causing queues within the switch to fill, and the switch to generate pause frames on the load-generating switch ports. Eventually, ipbench is no longer able to generate increasing loads, as the network interfaces themselves cannot transmit the packets generated by ipbench fast enough.

While in this particular example pause frames were able to prevent excessive load from being generated, this result would not occur in LANs using switches which did not propagate pause frames, or on the wider Internet. On the other hand, there may be some cases where if carefully used, pause frames can be used to move packet admission further upstream of the overloaded host within a network; for example by moving packet admission to a front-end load balancer in a server cluster.

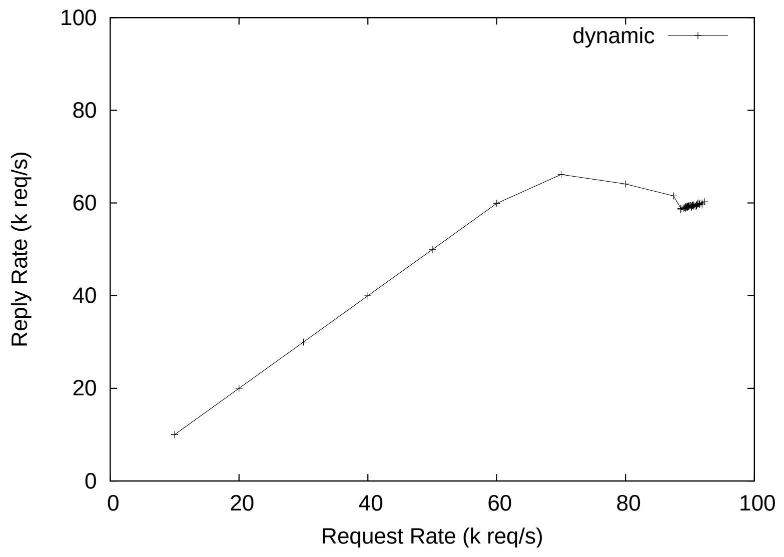

Figure 7.9: Effect of pause frames on load generation

7.5 HTTP performance

Although the admission control mechanism is intended to control non-responsive network loads, it is useful to know how the system behaves in the case of a web server. In particular it is interesting to see whether useful admission control can be achieved for TCP-stream based protocols which have significant inter-packet dependencies.

It is important to note that this is a completely different problem to an overloaded router or firewall which is carrying HTTP traffic, and hence the results presented in this section are not useful in understanding the router case.

7.5.1 HTTP benchmark configuration

In this section we consider HTTP performance for four system configurations, based on two independent variables. The first variable is the web server, for which we have tested both Apache and THTTP. The second variable is the kernel, for which we have used a FreeBSD kernel with and without our admission control mechanism.

The primary difference between these web server applications is in their connection handling architecture. Apache uses a large number of threads to handle the applied workload, while THTTP uses a single-threaded event-driven model. This difference in web server architecture gives THTTP a large performance advantage when serving static content.

We perform our tests for two different workloads. The first workload consists of transferring small files which fit within a single Ethernet packet. The second workload transfers a 1MB file, increasing the number of packets per request and the impact of TCP flow control.

In both kernels, the maximum interrupt rate generated by the network interface has been reduced from 8000 to 1000 interrupts per second, by changing the network interface's interrupt-moderation registers. This was done to allow the admission control mechanism to work more effectively at low throughputs, since the minimum granularity of admission control in the implementation tested is one packet per interrupt. In some of the following benchmarks we will see that the incoming data rate is still too low for NIC based rate limiting to be useful.

7.5.2 Realistic expectations

In the vast majority of HTTP benchmarks, the standard system configuration does not exhibit significant degradation under overload. In these cases, there is little hope of admission control itself providing any benefit. Rather than seeking a performance improvement, in these cases we are primarily interested in minimising the performance impact of admission control.

7.5.3 Large-file workload

Figures 7.10 and 7.11 show the behaviour of a system with a single Gigabit Ethernet NIC running apache and serving a 1MB file. In Figure 7.10, the horizontal axis shows HTTPerf's attempted connection rate, while the vertical axis shows the rate at which successful HTTP transactions occur. In Figure 7.11, the horizontal axis shows the HTTP request rate and the vertical axis shows the HTTP response rate.

Figures 7.10 and 7.11 show little performance difference between the standard and dynamic configurations. In this benchmark the network interface's transmit bandwidth becomes saturated, limiting the maximum transmit rate that can be achieved.

Figures 7.12 and 7.13 show the behaviour of the same benchmarks when using the THTTP web server instead of apache. In this case, THTTP's event-driven model results in the system continuing to accept additional connections when the network interface has reached peak transmit throughput. This behaviour causes the standard configuration to exhibit degraded throughput once the peak transmit throughput has been reached. The dynamic configuration successfully limits performance degradation in this scenario.

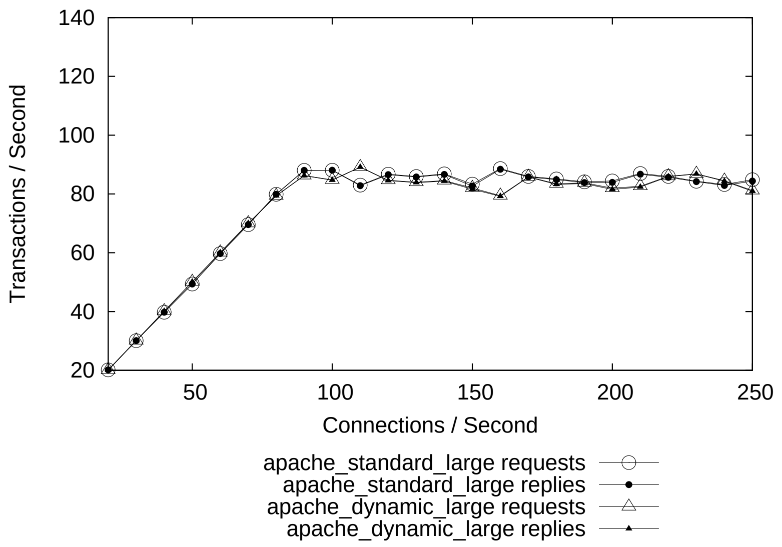

Figure 7.10: Apache Large-file HTTP load

Figure 7.14 shows the response times of all configurations. The response times of the standard and dynamic systems are very similar for both the Apache and THTTP servers. It is interesting to note that Apache's behaviour of rejecting additional connections once the maximum transmit throughput has been reached causes a significant increase in response time, since connections must wait for a free server thread before being served.

7.5.4 Small file workload

Figures 7.15–7.19 utilise the same benchmark configurations as the large-file workload shown in Figures 7.10–7.14, with the exception that a 93-byte HTML document is used.

Changing from the large-file to small-file workload has a significant impact on the behaviour of the network protocol. Firstly, it means TCP connections are very short-lived, since the HTTP request and response may each be contained in a single packet. This means that the protocol overheads are higher, and that TCP flow control is less likely to impact the behaviour of the system. It also shifts the performance bottle-neck to connection establishment rather than outright data-transfer rate.

Comparing Figures 7.15 and 7.17, we see that Apache's performance is much

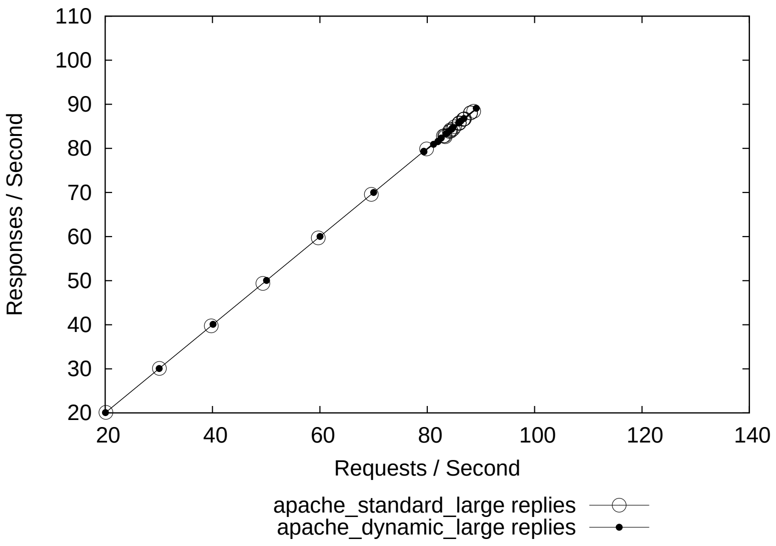

Figure 7.11: Apache Large-file HTTP throughput

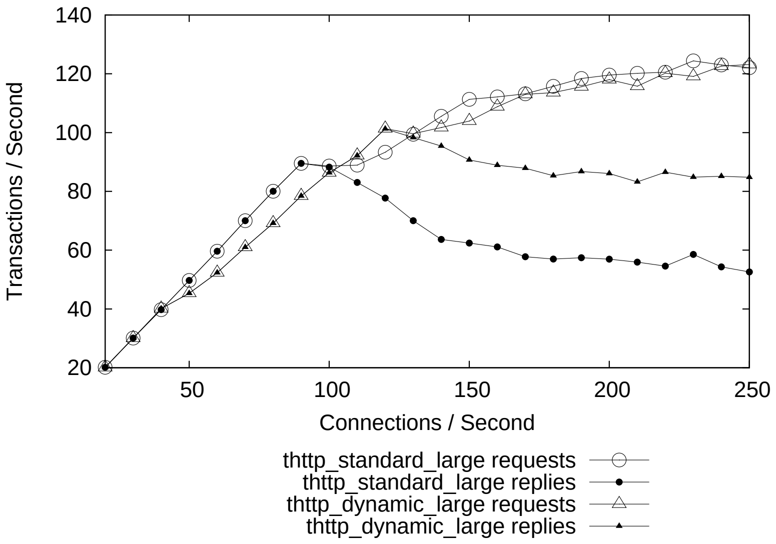

Figure 7.12: THTTP Large-file HTTP load

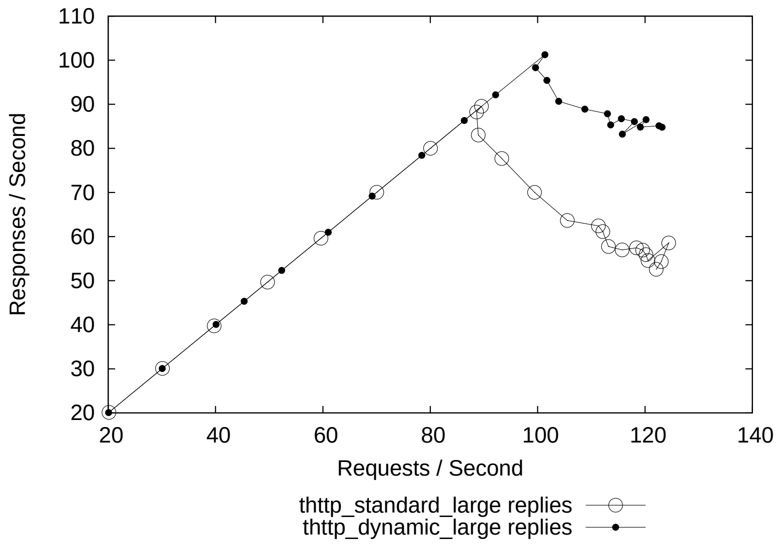

Figure 7.13: THTTP Large-file HTTP throughput

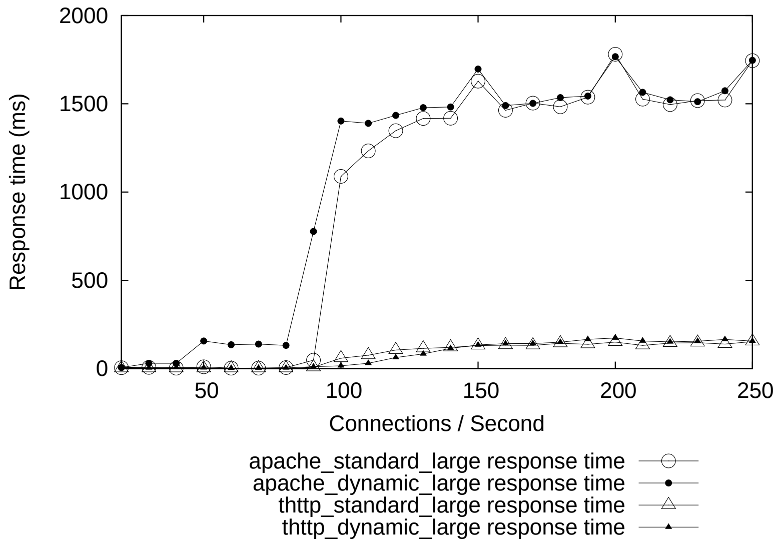

Figure 7.14: Large-file HTTP response time

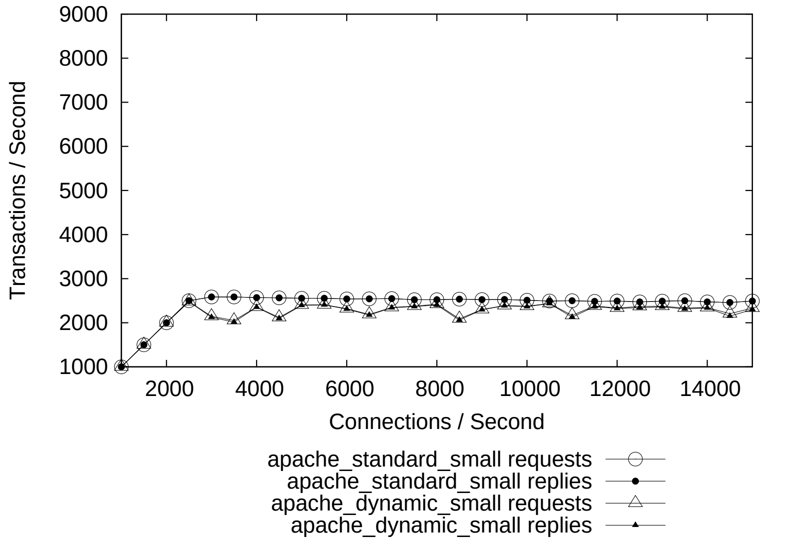

Figure 7.15: Apache Small-file HTTP load

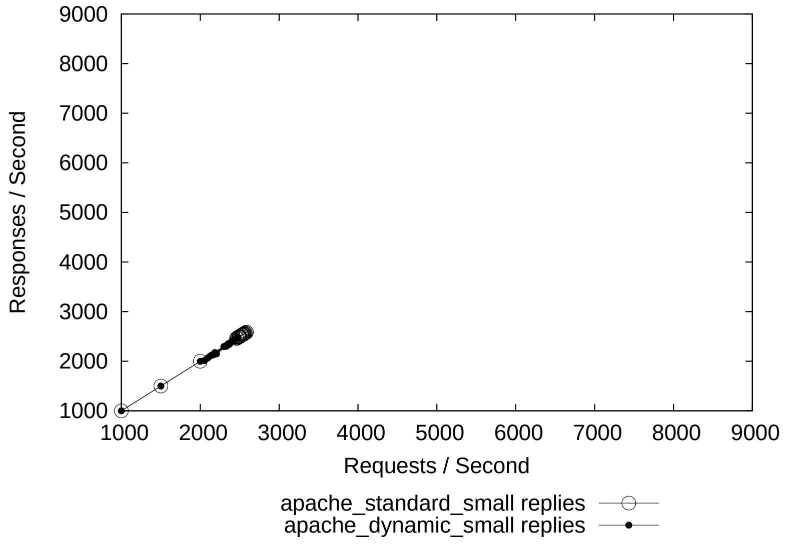

Figure 7.16: Apache Small-file HTTP throughput

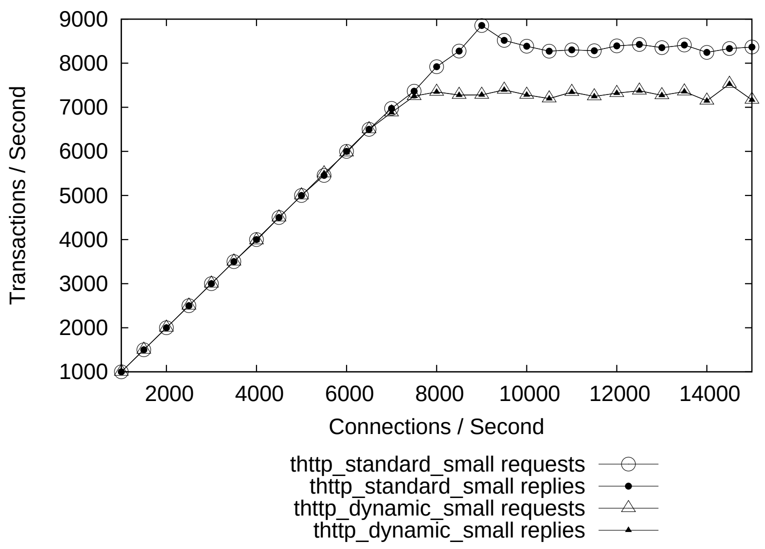

Figure 7.17: THTTP Small-file HTTP load

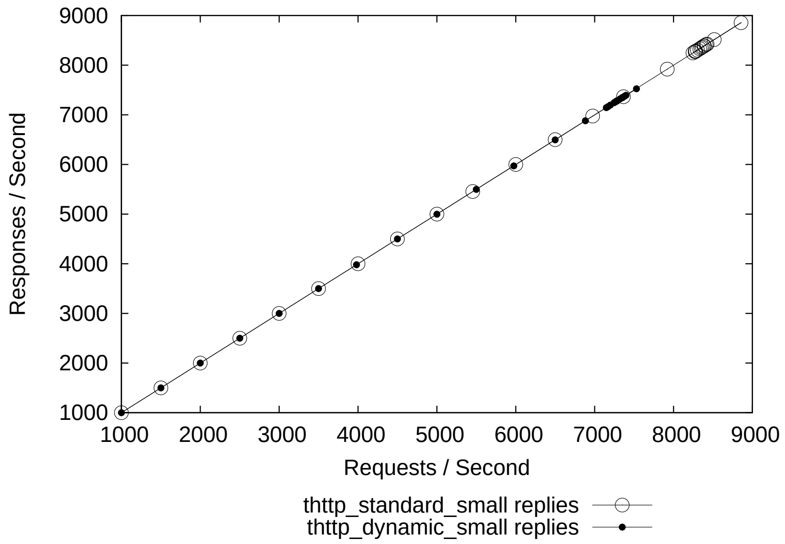

Figure 7.18: Small-file HTTP throughput

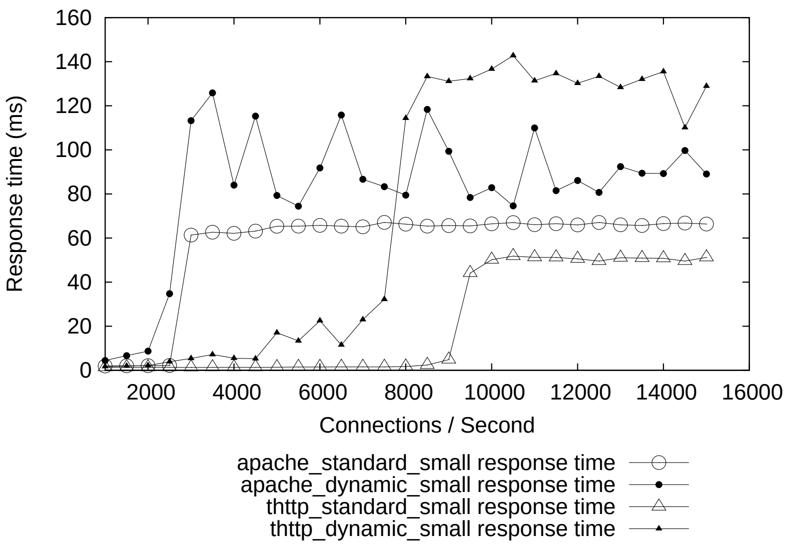

Figure 7.19: Small-file HTTP response time

worse than THTTP's. This is caused by a significant difference in the per-connection overheads of Apache's multi-threaded design when compared to THTTP's event driven design.

In Figure 7.15, the dynamic algorithm produces undesirable results with significant variation in achieved throughput once overload is reached, while in Figure 7.17, the dynamic algorithm achieves consistent throughput under overload. This difference occurs due to the extremely low peak throughput which is achieved by Apache for the small-file workload. Since in this benchmark we have fixed the maximum interrupt rate at 1000 interrupts per second, varying the number of entries in the DMA ring by one gives a change of 1000 packets per second. This granularity means the control algorithm must choose between allowing either too many packets-per-second or too few. A partial solution to this problem is discussed in Section 7.5.5.

The response times for the small-file workload are shown in Figure 7.19. In this case the dynamic approach does have a significant increase in response-time. This was predicted in Figure 4.2, where latency increases significantly for very low packet rates.

The amount of latency added would be different for other rate-limiting mechanisms, however the small added response-time delay (up to 100ms) is insignificant when compared to other factors influencing a user's perception of web-site performance [12].

7.5.5 Effect of control algorithm frequency

As mentioned in the previous section, at low rate-limits small changes to the number of packets received per interrupt have a significant impact on the number of packets admitted to the system. This problem can be alleviated by decreasing the interrupt frequency, however a side effect of decreasing interrupt frequency is that the average time a packet spends waiting in the DMA ring is increased.

An alternative solution to decreasing interrupt frequency is to rapidly switch between rate-limits that are close to the optimal rate-limit. Figure 7.20 shows the effect of increasing the frequency of the perturb and observe algorithm on the ability to track the maximum throughput point in the face of inadequate rate-limit granularity. As the control frequency increases from 10Hz to 100Hz, the average admitted throughput approaches that of the correct rate-limit for this workload. In effect the perturb and observe algorithm is being used to approximate the correct input throughput by rapidly switching between rate-limits which are above and below the ideal limit.

7.6 Results for alternative hardware

In order to demonstrate that our admission control approach can be generalised to other hardware, we have implemented our approach on a much slower machine, based on a 550MHz Intel Pentium-III processor and a National Semiconductor DP83820 based Ethernet interface connected via a 32-bit, 33MHz PCI bus.

The implementation in this case consists of modifications to the *nge* network interface device driver. In this case we have implemented a hybrid control approach, using a controlled-free-rate admission controller (Section 4.1.4) while simultaneously controlling the interrupt rate in order to minimise interrupt overheads under overload. This control approach was selected primarily because it was easy to implement using the DP83820 hardware-level interface — its DMA rings separate driver and hardware state, and it supports an easily programmable interrupt delay timer. In comparison, the standard FreeBSD device driver uses $100\mu S$ interrupt holdoff delay with a fixed number of DMA ring entries.

This particular benchmark uses UDP packet echo from user-level, and is somewhat simpler than the NFS and routing benchmarks presented earlier. Load is generated against a user-level UDP packet echo server. The echo server simply switches the sender and receiver addresses of the packet, then retransmits.

Figure 7.21 shows achieved throughput versus applied load for 1024 byte UDP payloads. The '$100\mu S$' line shows that without admission control, throughput rapidly declines under overload, while the 'variable' line shows that with admis-

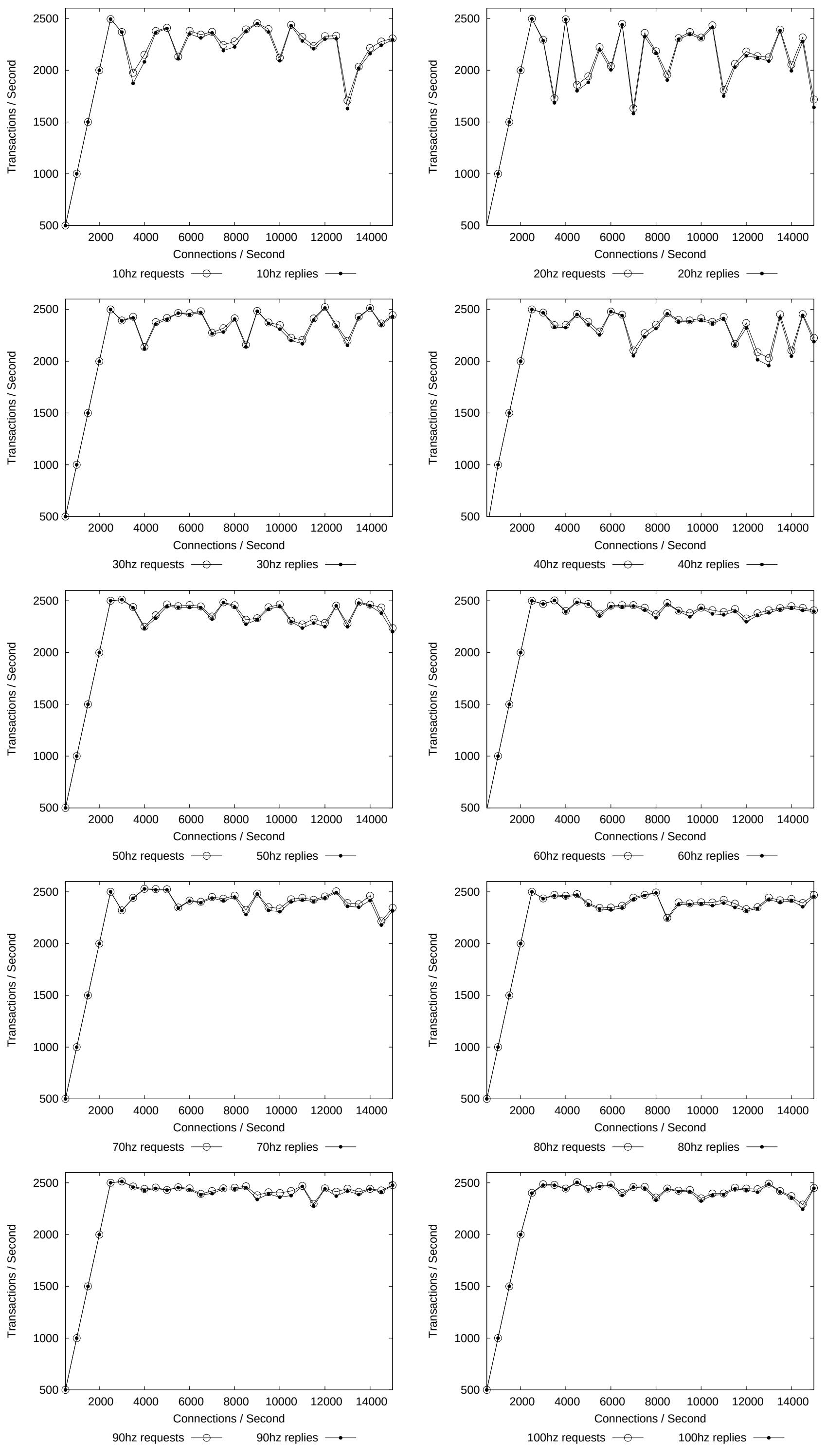

Figure 7.20: Small-file HTTP benchmark using apache with varying control algorithm frequencies.

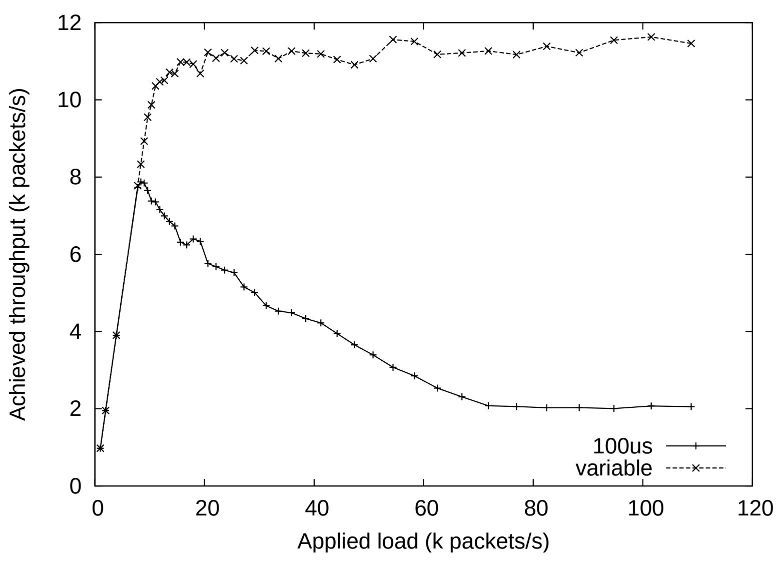

Figure 7.21: UDP echo: achieved throughput vs. applied load

sion control in place, throughput under overload remains constant. Moreover, due to the reduction of interrupts by our hybrid control scheme, we also see increased peak throughput compared to the standard case.

Figures 7.22 and 7.23 show the CPU utilisation and cycles-per-delivered-packet for a range of applied loads. Below eight thousand packets per second, both the modified and unmodified systems are operating below their maximum loss free receive rates. In this region the systems are not overloaded, and the cycles per delivered packet is reduced with applied load. The modified system continues this trend through to around eleven thousand packets per second, as the hybrid control scheme is able to reduce interrupt overheads by controlling the interrupt frequency.

The unmodified system then experiences a gradual increase in CPU utilisation as applied load is increased from ten thousand to seventy thousand packets per second, at which point the CPU becomes completely saturated, and no further increase is possible. This behaviour is reflected in the Figure 7.23 as a rapid increase in cycles per delivered packet, as during this period the rate of successful packet return also decreases.

The modified system, on the other hand, does not see any increase in CPU utilisation or cycles-per-delivered-packet once the admission controller has come into effect at around eleven thousand packets per second.

Interestingly, the behaviour of the unmodified system becomes constant (albeit

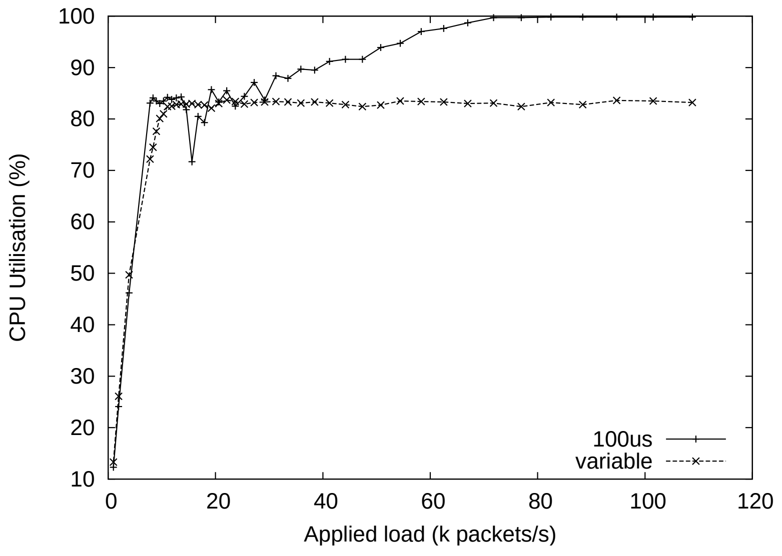

Figure 7.22: UDP echo: CPU utilisation vs. applied load

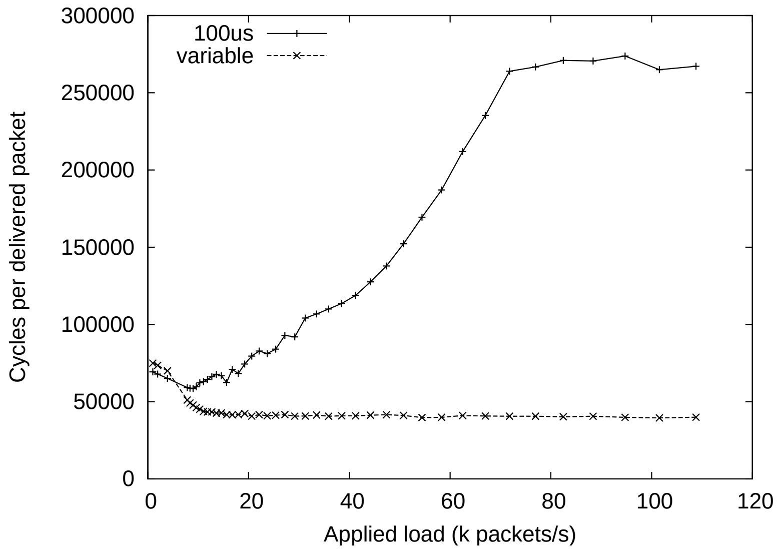

Figure 7.23: UDP echo: cycles per delivered packet

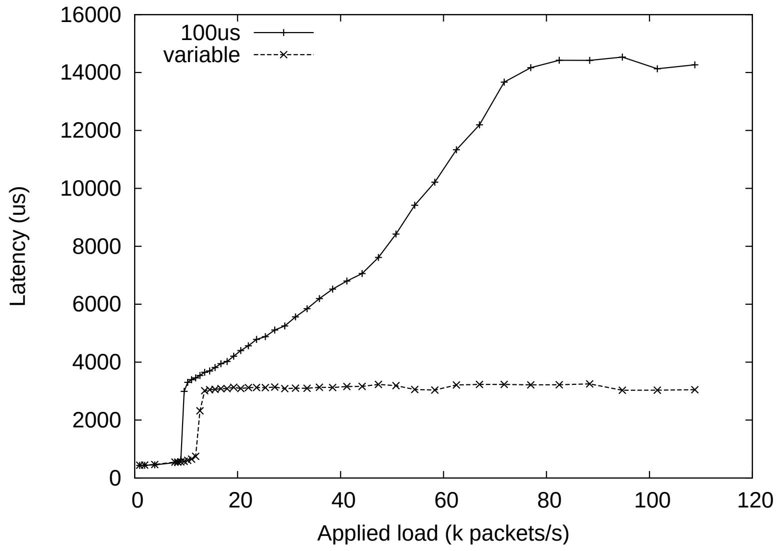

Figure 7.24: UDP echo: average round-trip latency

poor) once the CPU has become fully saturated, at a much higher applied load than is required to overload the system. By contrast, the modified system provides consistent performance under overload regardless of the additional load applied to the system.

Figure 7.24 shows the round-trip latency versus applied load for the UDP echo benchmark. We see that the unmodified system sees a large jump in latency when overloaded, followed by a fairly linear increase in latency until the point at which the CPU utilisation reaches 100%.

Like the unmodified system, the modified system also sees a significant jump in latency at applied loads which are high enough to cause overload. The reason for this is that the control algorithm used was aiming to maximise throughput rather than bound latency, and peak throughput is achieved by keeping the system slightly overloaded, without allowing it to see enough load to degrade throughput. This slight overload is enough to keep queues within the system saturated, but has the negative effect of causing increased latency.

It bears mentioning that the increased latency is not due to the admission controller itself, but by the behaviour of the rest of the system when operating at peak throughput. This is reflected in the equivalent jump in latency shown by the unmodified system.

7.7 Summary of results

7.7.1 Behaviour under normal operating conditions

A desirable feature of our admission control approach is that it has minimal performance impact under normal operating conditions. If this were not the case it would have been necessary to enable admission control only under high loads, however because our results indicate that performance is minimally effected under normal operating conditions, this is unnecessary.

7.7.2 Behaviour under overload

Our results show that in the majority of cases the dynamic configuration performs as expected; it prevents performance degradation under overload, and is able to sustain peak performance even under heavy overload. It does this without needing to know the specific cause of performance degradation within the system, and without requiring any application-specific tuning.

The small-file HTTP benchmarks did highlight some shortcomings of the proposed mechanisms when dealing with situations where the system becomes overloaded at very low packet rates. In these situations it seems clear that higher-level admission control approaches are more appropriate, since the increased cost of admission control for such low input data rates would be acceptable.

8 Conclusions

8.1 Summary

Chapter 2 compared the behaviour of a number of existing systems on the basis of the algorithmic cost of admission control within the system, the potential for performance degradation, the granularity of admission control, the feedback mechanism used, and the impact of the admission control scheme on overall system structure.

Of particular importance in Chapter 2 was the observation that many existing admission control approaches have an algorithmic cost which increases with the load applied to the system. In such systems it is reasonable to expect some performance degradation under overload, since the admission controller itself consumes additional resources under overload, reducing resource availability for processing incoming requests.

Chapter 3 provided a model of system behaviour under overload. The behaviour of individual components of a system under overload was addressed, then used to model the behaviour of a system comprised of connected resources.

The model was then used to argue that a rate-limiting resource placed at the beginning of a data path may be used to provide peak throughput for that data path. This conclusion is important because it demonstrates that admission controllers placed at the beginning of a data path (and which scale with traffic admitted to the system) are adequate to prevent performance degradation under overload.

Chapter 4 proposed a number of rate-limiting mechanisms that used the network interface itself to perform admission control. These rate-limiting mechanisms were designed to be implemented in the driver, and to be applicable to modern commodity network interface hardware.

Chapter 5 introduced the idea of monitoring the response of the system to changes in input throughput as a means of detecting that a system had become overloaded. The problem of using throughput monitoring to control input throughput was compared with a similar problem encountered in the design of photovoltaic maximum power point trackers (MPPTs), and solutions to that problem were evaluated as possible solutions in this case. An algorithm similar to the perturb-and-observe MPPT control algorithm was shown to satisfy the requirements of throughput control.

Chapter 7 evaluated the behaviour of the proposed NIC-based rate limiting and throughput-based rate selection for workloads varying from simple UDP echo through to HTTP workloads. The performance impact of the approach was examined in detail, with both positive and negative results shown.

8.2 Further work

8.2.1 Hardware packet demux

In order to provide overload elimination for a system consisting of a large number of data paths, it is necessary to provide separate rate control for individual data paths. In order to do this, it is necessary to utilise hardware which has the ability to demultiplex incoming packets into multiple queues, such that rate limiting may be applied to a queue without impacting the throughput of other queues.

Hardware packet demultiplexing has already been developed for some programmable network interfaces, and a vast amount of work on packet classification and packet filtering has been published. The issue is simply that such functionality has not been available in commodity network interface cards. Such functionality is beginning to become available in high-end network interfaces, since it is becoming necessary to support demultiplexing packets to per-processor-core DMA rings as the number of processor cores per system increases.

8.2.2 Control approaches

While the benchmark results show that a perturb-and-observe approach is feasible in detecting overload for a range of network protocols, it is also clear that there is room for improvement. Much of the disadvantage of the perturb-and-observe approach stems from the fact that in order to detect the maximum throughput of the system, it is necessary to discard some packets which might otherwise be processed.

This situation could be improved by using traffic injection to measure the effect of an increased load, rather than rate limiting to measure the effect of a reduced load. This approach is more complicated, as it requires generation and injection of valid packets, however it has the advantage of never causing packet loss when the system is not overloaded.

Another approach would be to discard the perturb-and-observe approach completely, and instead attempt to learn and periodically update a model of system behaviour at run-time. Once such a model had been learned, a rate-limit which produced optimal performance for the model could be selected. This system would have the advantage that if an accurate model had been learned, the rate limit selected would not cause unnecessary packet loss.

The model-learning approach is not without significant challenges; the model of behaviour would need to take into account the present workload of the machine, or the model would need to be adjusted whenever the workload changed. This results in either a very complicated model, or a very frequently changing model. This means that in practice, such a system may not provide better performance than the simpler perturb-and-observe approach.

8.2.3 Integration with other admission control approaches

A common approach to admission control seen in existing systems is to take a layered approach, wherein multiple admission control schemes are used at different points within the system [29, 30]. By adding edge limiting as the lowest layer of admission control within these approaches, it may be possible to achieve the best of both worlds; admission control which scales with admitted load, and admission control which chooses what to admit intelligently. Such an approach could bound the resource consumption of the software-level admission controller using the edge-limiting approach.

8.3 Contributions

i) The cost of admission control in existing systems was examined. Focus was given to the relationship between the location of the admission controller within the system, and the overall scalability of that system under overload.

ii) A model of system behaviour under overload was provided, and that model was used to demonstrate that providing rate limiting at the edge of the system is adequate to prevent overload from occurring.

iii) Bounded-cost admission-control mechanisms which may be implemented on commodity hardware were introduced. The mechanisms are able to scale to any level of applied load.

iv) A new approach to detecting overload based on measuring system throughput was described. The new approach has much lower implementation and run-time costs than existing approaches to overload detection.

v) An experimental evaluation of our approach was provided, which demonstrated the feasibility of implementing bounded-cost admission control on commodity hardware. The difficulties and tradeoffs encountered in this approach were also discussed.

8.4 Real world applications

The approach presented in this thesis, and especially the mechanisms for performing rate-limiting at the edge of the system, are highly applicable to current systems. Edge limiting is particularly suited to applications such as DNS servers and host-based routers, firewalls and load balancers for which traditional tail-drop semantics provide good results.

Because of the ease of implementation of the edge-limiting approach, there are few barriers to its implementation in existing operating systems. A practical approach would be to allow the system administrator to enable or disable edge limiting depending on the purpose and expected workload of the machine.

8.5 Closing remarks

In a perfect world, admission control would be unnecessary, since networks would support end-to-end flow control, and resource constraints would never come into play. To that end, all admission control can be viewed as an exercise in making the best out of a bad situation.

At times this dissertation has suggested approaches which appear counter-intuitive. To an idealist, the concept of detecting overload by selecting the wrong rate limit seems undesirable, and the idea that that network packets should be discarded indiscriminately appears strange. Yet by revelling in such pragmatism, we have arrived at an approach which is elegant in its simplicity, and demonstrably successful in eliminating performance degradation under overload.

A Measuring Overload

Measuring the behaviour of systems under overload is a particularly difficult task, because of the high loads that are often required. In order to generate such loads accurately and consistently, it is necessary to use a large number of load generators. Measuring overload therefore requires solutions to distributed system issues such as measurement synchronisation.

This appendix describes the experimental tools developed to perform the analysis of system behaviour under overload found in this thesis.

A.1 Distribution

The first problem encountered when trying to measure behaviour under overload is generating enough load to saturate the device or network being tested. Typically, the capacity of the device or network being tested is greater than the load that can be generated by a single host, necessitating the use of multiple hosts for load generation.

As a load generating host approaches saturation, it is normal for that host to experience increased latency and jitter effects, which can adversely impact performance measurements. It is therefore desirable to use more than the minimum number of load generating hosts required to generate a given load, in order to minimise the impact of the performance of the load generating hosts on the measured results.

The use of multiple hosts for load generation introduces a number of problems which are not seen in non-distributed benchmarks, and must be solved in order to obtain reliable and repeatable results. These problems include synchronisation of load generators, aggregation of results and increased sensitivity to network configuration.

A.1.1 Measurement synchronisation

Synchronisation of measurements is an issue which must be tackled in any distributed benchmark. If results from multiple hosts are to be aggregated, all hosts must run the test concurrently. There are two approaches to this problem; one is to ensure that

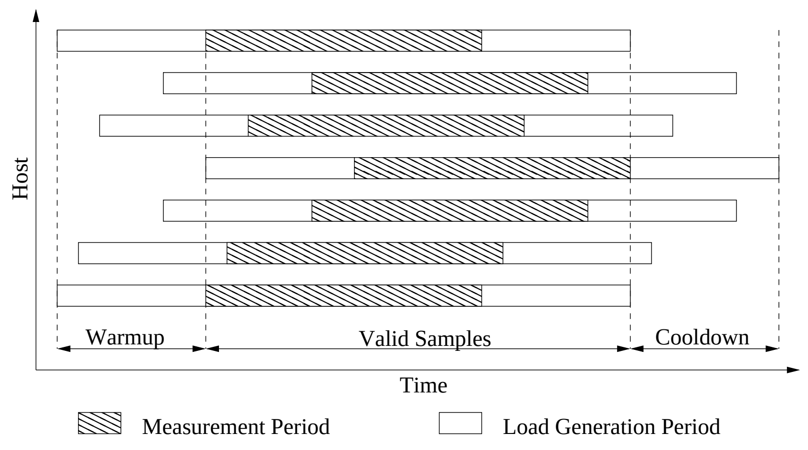

Figure A.1: Synchronisation error mitigation for multiple hosts.

all tests start simultaneously, the other is to mitigate the effect of non-simultaneous startup.

The startup delay between hosts is minimised by using a protocol where all hosts notify the benchmark controller that they are ready to start before the benchmark controller issues a start message to the load generating hosts.

The effects of non-simultaneous execution can also be minimised by eliminating measurements taken when not all load generators are executing. In practice we achieve this by discarding measurements taken during the warmup and cooldown periods. Figure A.1 shows how this approach can reduce the impact of non-simultaneous startup. All measurements occur while all hosts are generating load, however the results from each host are not guaranteed to have been measured at the same time. Therefore, this approach is only viable if load generation by all hosts occurs at a consistent rate.

The effects of non-simultaneous startup can also be minimised by taking measurements over a longer time period. This increases the degree of measurement concurrency between the hosts. When the measurement time is much greater than the synchronisation error, the number of non-concurrent samples becomes insignificant.

A.1.2 Result aggregation

There are limitations to what can be achieved when combining measurements taken by multiple hosts. Time-stamps from the same host are correct relative to each other, however because of the synchronisation problem, time-stamps taken by different hosts can only be compared with limited accuracy. The limit of that accuracy is the maximum synchronisation error.

Many useful measurements do not depend on the accurate synchronisation of results. Such measurements include statistical latency information, such as minimum, maximum, median, average and standard deviation of latency, average applied load and average achieved throughput.

Because of the synchronisation limits, the way we handle benchmark data is to simply have each host return a set of untimestamped latency measurements, which we combine into a single large dataset. It is not possible to make any conclusions about the correlation of individual samples from that dataset.

A.1.3 Network effects

When link throughputs are approaching saturation, switch configuration can have a large effect on the reliability of results. In particular, we have found that when pause frames are enabled, some switches will assign unequal priorities to individual load generators, such that it is difficult to get consistent load generation from a large number of machines.

One of the challenges of network benchmarking is that measurements reflect not only the performance of the host being tested, but also the network which it is connected to. This can make it very difficult to distinguish between poor switch performance and poor host performance. Without additional network analysis hardware it is impossible to separate the two components.

A.2 Measurement

The goal of ipbench [31] is to generate an accurate and complete characterisation of the performance of a system for a given workload. We control the packet size and packet rate applied to the system, and measure the packet return rate, the packet return latency, and the CPU utilisation of the target host. In order to measure latency, we measure the latency of responses to requests.

A.2.1 Packet generation[1]

Packets are generated by a normal user-level program using either TCP, UDP or raw sockets. This approach can generate a very high rate of system calls on the load generation host, such that reducing the system call overheads greatly improves the maximum rate at which packets are generated. Our initial implementation used

[1]The optimisations described in section A.2.1 were implemented by Peter Chubb and Ian Wienand.

the `gettimeofday` system call to generate time-stamps, however doing so effectively doubles the number of system calls required. We therefore removed calls to `gettimeofday`, and replaced them with reads of the CPU's cycle counter, in order to reduce the number of system calls and improve the peak packet generation rate of an individual load generation machine.

At low to moderate loads we keep the packet generation routine in a busy loop between sending packets, since sleeping the process was found to be unreliable. In an overload situation (i.e. the next packet is due to be sent at a time before the current time) the loop will be sending as fast as the system allows. This overload situation will appear in results as discrepancy between requested and sent throughput; in this case the test is considered invalid.

A.2.2 CPU

Accurate measurement of CPU time is critical in understanding the behaviour of a system under network overload. The usual method of measuring CPU time is to count the iterations of a loop running at the lowest possible priority. While this approach can in theory give accurate results, there are many pitfalls which can be seen in common implementations.

The first problem with counting iterations in a low-priority thread is that calibration is required. Accurate loop calibration requires the system to guarantee that no other code executes while the loop is being calibrated, and that an accurate timing mechanism available.

Even a quiescent system has some level of background interrupt processing occurring. This makes it difficult to guarantee that no other code executes during the calibration period. Therefore calibrated loops typically over-estimate the amount of CPU time available.

When running on development systems, an accurate timer may not be readily available. On such systems it is difficult to perform accurate calibration, and hence gain accurate results from a calibrated idle loop.

After calibration, some implementations also require timer signals during the measurement phase. We have found such implementations particularly problematic when measuring overload, because they produce invalid results if the system fails to deliver a signal. Because overloaded systems may fail to deliver signals on-time, or at all, implementations using signals have been known to report falling CPU utilisation under overload.

Because of these problems, a new method for measuring idle time was developed. The new method relies on the cycle-count register found on most modern CPUs. The cycle-count register is typically implemented as a monotonically increas-

```
void idle_thread(cycle_t *idle){
    cycle_t x0, x1, delta;

    x0 = get_cycles();
    while(1){
        x1 = x0;
        x0 = get_cycles();
        delta = x0 - x1;
        if(delta < THRESHOLD)
            *idle += delta;
        else
            *idle += CORRECTION;
    }
}
```

Figure A.2: Source code for idle-time measurement.

ing counter, modulo the maximum value of the cycle-count register.

The code for this new idle-time measurement technique can be seen in Figure A.2. Like the calibrated idle loop, we use an infinite loop running in a low priority thread. The body of the loop samples the cycle count register and compares the result to the previous iteration's sample. If the difference between the two samples is greater than the maximum number of cycles required to complete the loop in the absence of a context switch, it can be deduced that a context switch occurred during that iteration. Otherwise, the difference is added to a counter of cycles spent in the idle loop. The proportion of idle time can then be calculated by dividing the idle cycle counter by the total number of cycles executed in a given time period.

While this approach may miss a few cycles for each context switch, the number of cycles missed per switch is very small compared to the total cycles per context switch, such that the lost cycles are insignificant when compared to the total cycles. This error can be further minimised by calculating the number of cycles required per iteration assuming a cold cache, and adding this number to the idle counter when a context switch occurs. In practice we have found the approach accurate enough without performing this adjustment.

This method of idle time calculation does not require calibration, is very accurate, and is simpler to implement than existing approaches.

Bibliography

[1] Bob Baden, David Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, Larry Peterson, K. K. Ramakrishnan, Scott Shenker, John Wroclawski, and Lixia Zhang. Recommendations on queue management and congestion avoidance in the internet. RFC 2309, Internet Engineering Task Force., 1998.

[2] Nina Bhatti and Rich Friedrich. Web server support for tiered services. *IEEE Network*, 13(5):64–71, 1999.

[3] Richard Black, Paul T. Barham, Austin Donnelly, and Neil Stratford. Protocol implementation in a vertically structured operating system. In *Proceedings of the 22nd Annual IEEE Conference on Local Computer Networks*, pages 179–188. IEEE Computer Society, 1997.

[4] Josep M. Blanquer, Antoni Batchelli, Klaus Schauser, and Rich Wolski. Quorum: Flexible quality of service for internet services. In *2nd Symposium on Networked Systems Design and Implementation*, Boston, MA, USA, May 2005.

[5] Jose Brustoloni, Eran Gabber, Abraham Silberschatz, and Amit Singh. Signaled receiver processing. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 211–223, San Diego, CA, USA, 2000.

[6] Cabletron, Cisco, Foundry, Hewlett-Packard, and Nortel. Vendors on flow control. *Network World Fusion*, 1999.

[7] D. Driankov, H. Hellendoorn, and M. Reinfrank. *An introduction to fuzzy control*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[8] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 261–275, Seattle, WA, USA, October 1996.

[9] Nicola Femia, Giovanni Petrone, Giovanni Spagnuolo, and Massimo Vitelli. Optimization of perturb and observe maximum power point tracking method. *IEEE Transactions on Power Electronics*, 20(4):963–973, 2005.

[10] Sally Floyd. Tcp and explicit congestion notification. *ACM Computer Communications Review*, 24, 1994.

[11] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[12] Md Ahsan Habib and Marc Abrams. Analysis of sources of latency in downloading web pages. In *World conference on the WWW and Internet (WebNet)*, San Antonio, TX, USA, 2000.

[13] K. H. Hussein, I. Muta, T. Hshino, and M. Osakada. Maximum photo-voltaic power tracking: an algorithm for rapidly changing atmospheric conditions. In *Generation, Transmission and Distribution, IEE Proceedings*, volume 142, pages 59–64, 1995.

[14] Intel. PCI/PCI-X family of gigabit ethernet controllers software developer's manual. `http://download.intel.com/design/network/manuals/8254x_GBe_SDM.pdf`.

[15] Abhinav Kamra, Vishal Misra, and Erich M. Nahum. Yaksha: a self-tuning controller for managing the performance of 3-tiered web sites. In *Quality of Service — IWQoS 2004, 12th International Workshop*, pages 47–56, Montreal, Canada, 2004.

[16] P. R. Kumar. Convergence of adaptive control schemes using least-squares parameter estimates. *IEEE Transactions on Automatic Control*, AC-35:416–424, 1990.

[17] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 17(7), 1996.

[18] J.D.C. Little. A proof for the queueing formula $L = \lambda W$. *Operations Research*, 9:383–387, 1961.

[19] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proceedings of the 1996 Annual USENIX Technical Conference*, pages 99–111, San Diego, CA, USA, January 1996.

[20] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 153–167, Seattle, WA, USA, October 1996.

[21] W. Noureddine and F. Tobagi. Selective back-pressure in switched Ethernet LANs. In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*, pages 1256–1263, Rio de Janeiro, Brazil, 1999.

[22] Rainer Palm. Fuzzy signals in control loops. In *SAC '95: Proceedings of the 1995 ACM Symposium on Applied Computing*, pages 455–460, New York, NY, USA, 1995. ACM Press.

[23] Ian Pratt and Keir Fraser. Arsenic: A user-accessible Gigabit Ethernet interface. In *Proceedings of the 20th IEEE INFOCOM*, April 2001.

[24] K. K. Ramakrishnan and Sally Floyd. A proposal to add explicit congestion notification (ecn) to ip. RFC 2481, Internet Engineering Task Force., 1999.

[25] National Semiconductor. DP83820 datasheet. `http://www.national.com/ds.cgi/DP/DP83820.pdf`, 2001.

[26] Thiemo Voigt. Overload behaviour and protection of event-driven web servers. In *Proceedings of the International Workshop on Web Engineering*, Pisa, Italy, May 2002.

[27] Thiemo Voigt and Per Gunningberg. Adaptive resource-based web server admission control. In *7th IEEE Symposium on Computers and Communication*, Taormina/Giardini Naxos, Italy, 2002.

[28] Thiemo Voigt and Per Gunningberg. Handling multiple bottlenecks in web servers using adaptive inbound controls. In *Seventh International Workshop on Protocols for High-Speed Networks*, Berlin, Germany, 2002.

[29] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and Ashish Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 189–202, Boston, MA, USA, 2001.

[30] Matt Welsh and David Culler. Adaptive overload control for busy internet servers. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, USA, 2003.

[31] Ian Wienand and Luke Macpherson. ipbench: A framework for distributed network benchmarking. In *AUUG Winter Conference*, Melbourne, Australia, September 2004.