Idletime Scheduling with Preemption Intervals

Lars Eggert[*]
NEC Europe Ltd.
Network Laboratories
Kurfürstenanlage 36
69115 Heidelberg, Germany
+49 (6221) 905-1143
lars.eggert@netlab.nec.de

Joseph D. Touch
University of Southern California
Information Sciences Institute
4676 Admiralty Way #1001
Marina del Rey, CA 90292, USA
+1 (310) 448-9151
touch@isi.edu

ABSTRACT

This paper presents the idletime scheduler; a generic, kernel-level mechanism for using idle resource capacity in the background without slowing down concurrent foreground use. Many operating systems fail to support transparent background use and concurrent foreground performance can decrease by 50% or more. The idletime scheduler minimizes this interference by partially relaxing the work conservation principle during *preemption intervals*, during which it serves no background requests even if the resource is idle. The length of preemption intervals is a controlling parameter of the scheduler: short intervals aggressively utilize idle capacity; long intervals reduce the impact of background use on foreground performance. Unlike existing approaches to establish prioritized resource use, idletime scheduling requires only localized modifications to a limited number of system schedulers. In experiments, a FreeBSD implementation for idletime network scheduling maintains over 90% of foreground TCP throughput, while allowing concurrent, high-rate UDP background flows to consume up to 80% of remaining link capacity. A FreeBSD disk scheduler implementation maintains 80% of foreground read performance, while enabling concurrent background operations to reach 70% throughput.

Categories and Subject Descriptors

D4.7 [**Operating Systems**]: Organization and Design. D.4.4 [**Operating Systems**]: Communications Management – *buffering, input/output, message sending.* D4.1 [**Operating Systems**]: Process Management – *concurrency, scheduling.*

General Terms

Algorithms, Performance, Design.

Keywords

Idletime scheduling, background processing, preemption interval, resource scheduler, queuing, network scheduler, disk scheduler.

1. INTRODUCTION

Many computer systems are often idle. Studies that focus on CPU utilization [23][24][41] report that approximately 70% of the monitored machines in a network are idle at any given time. Idletime scheduling focuses on the means to utilize such idle capacity for productive background work, without delaying or otherwise interfering with regular foreground work. For any given workload, a single *bottleneck* resource limits performance [2]. Even when the bottleneck is fully loaded, other resources often remain partially idle. For example, a system with a fully loaded disk drive may still possess significant idle CPU or network capacity. Using this idle capacity productively, without delaying execution at the bottleneck, can improve system efficiency and user-perceived performance.

Idle resource capacity is an opportunity "to get something for nothing" when utilized for background work. Examples include system maintenance tasks such as virus checking or file system optimization. Such tasks should execute regularly and can delay foreground user tasks due to their heavy use of resource capacity. However, it typically matters little exactly when or at which speed such tasks execute. Scheduling them with capacity that is not otherwise in use can eliminate delays for user tasks and improve the system performance.

Another group of services that benefits from idletime service is caches and prefetching systems, *e.g.*, prefetching of likely future FTP or web requests [27][35][36]. Conventional prefetchers must explicitly limit their speculative transmissions to avoid excessive interference with regular network traffic. Idletime use of the network enables aggressive prefetching while delaying regular network traffic only minimally. Likewise, idletime use of storage resources (such as memory or disk space) allows the prefetch cache to grow without affecting foreground storage use.

Several existing application- and kernel-level approaches establish different levels of service in an operating system (OS) or applications. Many of these use simple priority queues. Although these establish prioritization, many resources cannot preempt processing of a started request. Frequent background use hence decreases foreground performance under a simple prioritization scheme. Additionally, many other approaches are resource-, workload- and application-specific. They fail to utilize available idle capacities for background work, require widespread changes to systems and

[*] This research was performed at the USC Information Sciences Institute while Lars Eggert was a Ph.D. student at the University of Southern California.

applications or do not sufficiently protect the performance of foreground tasks in the presence of high-volume background work. The idletime scheduler addresses these limitations.

Ideally, the presence of idletime "background" use in the system should be transparent to regular "foreground" use, in terms of both performance and side effects. The first aspect of this idletime transparency mandates that the side effects of idletime execution must remain hidden from regular tasks. For example, when a file system holds idletime data, side effects include the visibility of idletime files to regular processes, the count of free disk blocks and the position of the disk head. To prevent idletime use from interfering with regular foreground tasks, the mechanism must carefully hide all such side effects.

The second aspect of idletime transparency addresses performance: the execution time of a foreground process should be the same with or without concurrent idletime use of a subset of resources. Idletime execution should fill the "gaps" in resource utilization, without delaying regular use. In such a system, resources could be busy with either foreground or background work at all times, improving overall efficiency. In other words, the system should intentionally starve idletime jobs to free sufficient capacity for foreground use.

In existing systems, protecting the execution performance of foreground tasks from the presence of idletime use is extremely difficult. Most schedulers cannot switch between two jobs instantaneously and incur a delay when doing so. Systems can avoid these preemption costs by enforcing reservations, *i.e.*, by *a priori* control of such preemption costs. Resource reservations require extensive application modifications and can lead to low overall resource utilization. Thus, they may fail to support common workloads in general-purpose systems.

Because of these preemption costs associated with stopping the idletime work and starting foreground tasks, idletime scheduling can delay regular foreground execution. Minimizing preemption costs is the key objective for effective idletime scheduling. Overall system efficiency improves only when the gain through idletime use is greater than its associated preemption costs. The goal is to create a system behavior that appears preemption-like, even if the resources do not support immediate preemption.

To support a wide variety of applications and services, a useful idletime mechanism must be tunable. Certain foreground applications may tolerate a larger decrease in performance and the idletime scheduler can consequently schedule larger background workloads. Conversely, the idletime scheduler must completely stall background resource use whenever the foreground application requires full resource capacity. A successful idletime mechanism must dynamically adapt its background scheduling decisions based on the current foreground workload.

Finally, a useful idletime mechanism must support existing applications without modifications. Consequently, it cannot require significant modifications to an existing OS or its APIs, nor can it require a completely new OS.

Idletime scheduling establishes background use as a separate, isolated, low-priority service class and allows background use of transient idle capacities with little impact on foreground performance. Based on traditional priority queuing, the idletime scheduler selectively relaxes one aspect of the work conservation property for background tasks. The idletime scheduler establishes background service as a localized modification to a small subset of system schedulers and does not require application modifications to execute them in the background. As an incremental modification to existing systems, it is thus easier to deploy and can enable idletime use of existing applications and services. The idletime scheduler can utilize short, transient idle capacity for background work even when the bottleneck resource is nearly fully loaded.

The idletime scheduler relaxes one aspect of the work conservation property for idletime jobs. Work conservation requires that a resource must not remain idle while jobs are waiting for service and that it must not destroy partially completed work. The idletime scheduler relaxes this first aspect – never remaining idle – in a controlled manner. It introduces a time delay, called the *preemption interval*, before switching to idletime tasks.

During a preemption interval, background jobs remain stalled and do not receive service even if the resource is idle. This relaxation of work conservation allows new foreground jobs appearing at the resource during the preemption interval to receive service immediately. Preemption intervals therefore increase resource availability for foreground service. In the absence of a preemption interval, *i.e.*, with simple priority queues, a higher-priority job must often wait until ongoing idletime work finishes or is preempted. This delay reduces foreground performance. Preemption intervals avoid delays when starting foreground tasks and thus increase their performance in the presence of idletime use.

The length of the preemption interval controls the impact of idletime use on regular tasks. With a short preemption interval, the scheduler is more aggressive in utilizing idle capacity for background use, but permits a higher impact on foreground tasks. With a longer preemption interval, the impact is lower, but idletime performance also decreases, because a longer preemption interval shortens the usable fraction of an idle period. Changing the length of the preemption interval allows tuning of the mechanism according to user policy and current workload, within limits.

Another feature of the idletime scheduler is that it requires only minor modifications to a subset of system resources. Conventional priority schemes require modifications to all resource schedulers in a hierarchy to support arbitrary workloads. When some schedulers remain unmodified, one of them can control overall system behavior under specific workloads and effectively disable prioritization. The preemption interval of the idletime scheduler introduces controlled delays for the lower-priority idletime service class. These delays cause the formation of idletime queues that absorb the scheduling (mis-)decisions of non-idletime schedulers earlier in the resource hierarchy. This establishes idletime service in a queue hierarchy where only a small number of schedulers support preemption intervals.

2. IDLETIME SCHEDULING

The key feature of the idletime scheduler is limiting the aggregate preemption cost by introducing a *preemption interval*. A preemption interval is a time period following each serviced foreground request during which no background request starts – the resource remains idle even when background requests are queued. The preemption interval limits the overhead to at most a single preemption per interval, when the preemption interval length is greater than the service time.

The basic operation of the idletime mechanism is as follows: the resource scheduler begins a preemption interval whenever an active foreground request finishes. While the preemption interval is active, the resource does not start servicing any idletime requests. It does service any queued foreground requests, however, and starts a new preemption interval after each foreground request finishes. It also immediately services newly arriving foreground requests. If no more foreground requests exist in the queue, the resource will remain idle until the preemption interval expires. As a result, the resource starts servicing idletime requests only after the expiration of a full preemption interval in which no new foreground requests arrive or are waiting.

The idletime scheduler reduces foreground delays compared to a simple priority queue. Instead of immediately starting service for queued background requests whenever the last foreground request finishes, the resource remains idle. When a new foreground request arrives at the resource, it can immediately receive service. Traditional priority queues delay a new foreground request until the resource preempts the active idletime request or finishes serving it. This decreases foreground performance.

The idletime scheduler minimizes foreground delays by limiting the number of required idletime preemptions to at most one per sequence of successive foreground requests ("bursts.") This amortizes idletime preemption cost over a burst of foreground requests. The defining characteristics of these bursts are inter-request gaps that are shorter than the preemption interval.

Figure 1 illustrates this behavior. Here, the preemption of *I* at t_2 delays the start of R_1 until t_3. After R_1 finishes, the resource enters the preemption interval $P(R_1)$. Because the $P(R_1)$ is longer than the inter-arrival time between R_1 and R_2, *I* remains blocked and the resource remains idle and ready to serve R_2 immediately at t_4. If it had started to serve *I*, another preemption delay would have delayed R_2. The following foreground requests R_3 and R_4 are also part of the burst, because their respective inter-arrival gaps are less than their respective preemption intervals. Consequently, they receive service without additional preemption delays. The resource allows idletime use and starts servicing *I* only after R_4*'s* preemption interval $P(R_4)$ expires at t_7.

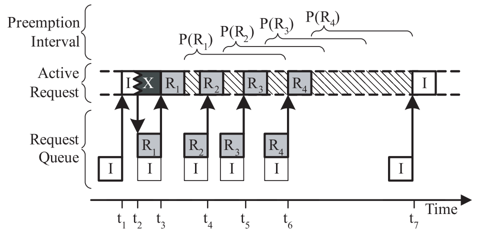

Figure 1. Formation of foreground request bursts.

Introducing artificial delays before idletime service relaxes one aspect of the property of work conservation. Traditional schedulers are work-conserving, because they do not allow the resource to remain idle while work is queued. The idletime scheduler relaxes this property and allows the resource to remain idle for a limited amount of time before starting queued background work. This relaxation applies only to background requests – work conservation for regular foreground tasks remains unchanged. A second aspect of work conservation – never destroying partially completed work – remains unchanged as well.

2.1 Preemption Interval Length

The length of the preemption interval is a parameter that controls the tradeoff between aggressive use of idle capacity and impact on foreground use. With a longer preemption interval, the performance of idletime tasks decreases, because each idletime request following foreground use incurs a long delay before it can start. Corresponding foreground performance increases with a longer preemption interval, because the likelihood that the resource is busy serving idletime requests decreases. This characteristic illustrates the fundamental tradeoff between foreground reactivity and background performance.

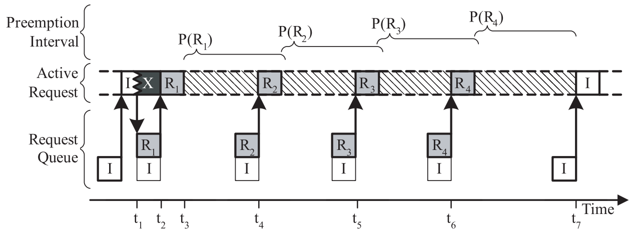

Figure 2. Effects of long preemption intervals.

Changing the length of the preemption interval thus tunes the idletime mechanism for particular resources and workloads. This section identifies simple heuristics for setting the length of preemption intervals for a resource. More refined schemes for setting and dynamically adapting preemption interval lengths are an area of future research, discussed in Section 5.

In the extreme case of an indefinite preemption interval, no idletime work ever receives service and foreground performance is identical to a system without idletime scheduling. In the other extreme, a zero-length preemption interval, foreground performance is identical to a system that uses traditional priority queues.

Figure 2 shows an example where a long preemption interval prevents idletime service. At t_1, idletime request *I* is preempted and regular request R_1 receives service at t_2. When it finishes at t_3, its preemption interval $P(R_1)$ starts. Shortly before $P(R_1)$ expires, R_2 starts at t_4. This is followed by a new preemption interval $P(R_2)$ after R_2 finishes, again preventing *I* from starting. The same occurs for R_3, R_4, and their corresponding preemption intervals $P(R_3)$ and $P(R_4)$. *I* receives service only after $P(R_4)$ expires at t_7.

A useful upper bound for the preemption interval length is the average foreground inter-arrival time. A preemption interval longer than the inter-arrival gaps leads to a situation where idletime use remains disabled. Useful preemption interval lengths are thus shorter than the foreground inter-arrival gaps.

However, preemption interval lengths cannot become arbitrarily short. The worst-case scenario for idletime scheduling is one-request foreground bursts with gaps longer than the preemption interval. In such a case, the mechanism is ineffective and each foreground request still incurs the full preemption overhead.

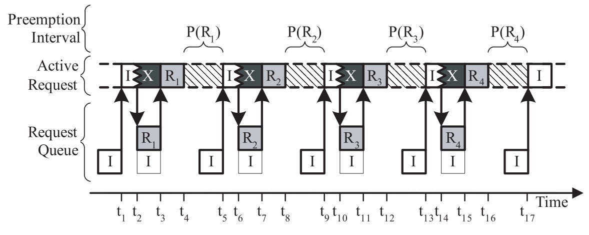

Figure 3. Effects of short preemption intervals.

Figure 3 gives an example of a worst-case scenario. Here, the preemption intervals are shorter than the inter-arrival gaps and *I* receives service after each of the preemption intervals $P(R_1)$ to $P(R_4)$ expire. Each time, the next new foreground request arriving while *I* is active incurs the preemption delay. Consequently, *I* does not finish until t_{17}.

This situation can occur in only two cases: very light foreground workloads or very short preemption intervals. In the first case, this behavior may be acceptable. When it is not, lengthening the preemption interval causes multiple spaced-out requests to form longer bursts and thus increases amortization of preemption costs. However, a longer preemption interval also prevents utilizing the idle gaps for background use. In both cases, lengthening the preemption interval avoids worst-case behavior.

Because the primary requirement for an effective idletime scheduler is the minimization of foreground interference, useful minimal preemption interval lengths will cause the formation of foreground bursts. This allows amortization of preemption costs. Longer preemption intervals often result in longer bursts and consequently reduce preemption overheads. The ideal preemption interval for a given workload is sufficiently long to cause the formation of foreground bursts that limit the preemption costs to within limits of a user-specified policy.

A strictly secondary objective is maximizing idletime work. These two requirements can conflict. This occurs under high foreground workloads, when the inverse of the inter-arrival rate approaches the service time. In such borderline cases, the first rule takes precedence and mandates idletime starvation, protecting foreground performance.

2.2 Resource Queue Hierarchies

Many approaches for service differentiation require extensive changes to both an OS and its applications, such as specification and enforcement of resource reservations. One primary benefit of the idletime scheduler is that it can establish service differentiation through localized modifications.

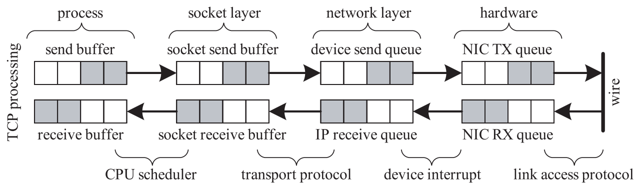

Figure 4. Hierarchical queuing in the network stack.

Computation inside an OS can be modeled as a directed graph, in which nodes correspond to resources and arcs denote producer/consumer relations. Processes and physical resources are producer/consumers of resource requests. A request serviced at a resource may generate zero or more associated requests that in turn propagate to other resources for service. To execute a specific execution step, a producer issues requests that flow as a request stream through a succession of queues managed by various schedulers before terminating at a consumer node.

As an example of such a queue hierarchy, Figure 4 shows the queues involved in network communication using TCP. When sending TCP traffic (top queue chain in Figure 4), data flows from the send buffer of an application (on the left) into the socket send buffer in the kernel. TCP's congestion control algorithm then places the corresponding IP packets into the device send queue. The network driver (on the far right) finally transfers these packets to the network interface. TCP inbound processing (bottom chain in Figure 4) is similar.

Certain systems, such as *Scout* [22], directly implement this path-based graph model. A central scheduler manages all paths in the queue hierarchy and decides which path to service at any given time. Adding idletime support to such systems is straightforward through an extension of the central scheduler. However, the major drawback of *Scout* and similar systems is that their non-standard APIs severely limit their usefulness.

Idletime support for a conventional OS, such as a UNIX-based one, is more complicated. The also consist of a large number of resources and schedulers that require modification, but additionally include implicit processing rules that interfere with idletime use. One example of such implicit prioritization is giving higher priority to internal kernel processing by preempting user-level processes for kernel-level events, such as interrupt handling. Even inside the kernel, lower-level events such as hardware interrupts take priority over higher-level functionality, such as system call processing. Furthermore, kernel processing is often work-conserving. The OS services all pending events at a given level before resuming execution of higher-level events or user processes. This can cause priority inversion, where a higher-priority request at a higher level must wait for the completion of a lower-priority one at a lower level [16].

This internal prioritization interferes with idletime resource use. Giving higher preference to kernel-level events can counteract the prioritization principle, because the system may interrupt execution of a foreground process in order to service a kernel event associated with a background task. Furthermore, work conservation drains existing queue contents before giving processes a chance to schedule more work. A series of queued background requests at a lower level may receive service while a foreground process must wait to schedule more requests at a higher level. This effectively defeats prioritization even with priority queues.

The idletime scheduler counteracts these priority inversions. Preemption intervals after foreground use at lower levels of the hierarchy stall processing of queued background work. This allows higher level processing to potentially enqueue additional foreground requests.

3. IMPLEMENTATION

The previous section presented the idletime scheduler and discussed the detailed effects of scheduling with preemption intervals. This section discusses the implementation of idletime scheduling for two different resources, a disk drive and a network interface. Both are simple, localized modifications to release 4.7 of FreeBSD.

A resource with support for idletime scheduling using a preemption interval can be described as a Moore machine with four possible states. It can be idle (state *I*, the start state), serving an active foreground request (state *F*), serving an active background request (state *B*) or halting background use during the preemption interval (state *P*). As a result, the set of states is $S = \{I, F, B, P\}$.

The transitions between these resource states are based on the following four events. First, a foreground request is at the head of the queue (event *f*). Second, a background request is at the head of the queue (event *b*). Due to priority queuing, event *b* cannot occur while a foreground request exists in the queue. Third, the preemption interval expired (event *t*). This can occur only when there is no *f*. Finally, the queue is empty (event *i*). Therefore, the set of events is $E = \{f, t, b, i\}$.

Each $X \rightarrow Y$ state transition can occur on a given event $e \in E$. Clearly, a large number of state machines exist for the given sets of states and events (4^{16}). The remainder of this section reduces the number of variants to arrive at four viable candidate algorithms that conform to the idletime scheduling principles.

Several scheduling algorithm variants conform to the idletime principles described in the previous section. Appendix A discusses the differences between the variants and identifies the most promising variant as a basis for implementation, shown in Figure 5.

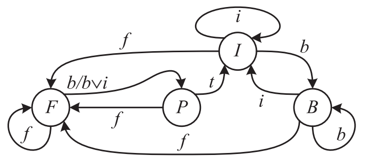

Figure 5. State transitions of the implemented scheduler.

The scheduler implementation tags each resource request as either regular or idletime. Idletime schedulers in the system use the tags to prioritize the request stream. The implementation defines a new idletime option for file descriptors (including sockets) that indicates whether resource operations occur in the foreground (the default) or background. A process can explicitly set and clear the idletime option on a file descriptor. The idletime mechanism will schedule all resource requests issued from a tagged background descriptor as idletime requests.

A new file descriptor option requires application changes. However, alternative idletime APIs allow unmodified applications to execute during idletime. One simple API overloads the meaning of CPU priority and treats all resource requests from processes running with less than a specific CPU priority as idletime. Another API variant redefines the POSIX idletime flag (originally defined for CPU scheduling) as a general idletime flag covering other resources [28]. Finally, specific reserved port ranges or IP aliases can indicate idletime use for idletime networking.

Both the modified network and disk schedulers implement the preemption interval mechanism with the standard BSD timing facilities [37]. A separate timer is associated with each network and disk device. The timer restarts whenever the scheduler for the given resource reenters state *P*. While the timer is active, the resource is in its preemption interval (resource condition *q* is satisfied). Upon timer expiration, the resource either starts serving background requests (entering state *B*) or becomes idle (state *I*).

Neither the disk nor the network scheduler implementation modifies the device drivers of the resource for which they implement idletime use. This is important, because idletime use would otherwise incur a significant deployment issue due to the multitude of drivers in a typical system that all require modifications. Instead, the implementations are simple, localized modifications at a higher system layer. The idletime disk scheduler operates at the border between the buffer cache and the block device driver, whereas the idletime network scheduler operates at the border between the network protocols and network interface drivers.

3.1 Disk Scheduler

To issue background disk requests, a process uses the *fcntl* system call to set the idletime option on an open file descriptor. The kernel then tags all underlying block transfers for idletime scheduling at the buffer queue. The current system implements idletime scheduling by replacing the standard *bufqdisksort* algorithm, which normally implements the *C-LOOK* variant of the *elevator seek* algorithm [40]. The idletime implementation establishes prioritization through two *C-LOOK* queues for foreground and background requests together with the preemption interval mechanism described in Section 2.

Many disk drives enqueue multiple operations concurrently and manage the service order internally. Often, the on-disk hardware queue does not support request priorities. Therefore, many drives may internally reorder pending foreground and idletime operations to lower the combined service time. Preemption intervals can counteract these effects and still support idletime use. The device driver stalls idletime requests during the preemption interval and prevents them from entering the hardware queue. This way, they cannot interfere with regular operations on-disk, decreasing the impact of idletime use on foreground performance.

Besides its buffer cache, a second performance-enhancing feature of UFS is read-ahead: during *breadn* or *cluster_read*, UFS uses a heuristic to predict whether the most recent request history corresponds to a sequential read. When it does, UFS speculatively generates additional reads for the next disk blocks. When predicted correctly, these blocks will already reside in the buffer cache when the application requests them, improving performance.

For idletime use, both caching and speculative read-ahead are problematic. First, the buffer cache is of limited size. Using it for idletime data can decrease regular foreground performance, if caching background items flushes foreground data from the cache. One approach to deal with this issue would be to treat the cache itself as a resource that should support background use. The scheduler implementation does not support such idletime caching. Instead, the current idletime scheduler simply disables caching of idletime data. (Only the buffer cache inside the OS was disabled. On-disk hardware caches remained active and could lead to reduced performance when the disk drive flushed buffered foreground data to cache new idletime information.)

Background-initiated read-ahead is also problematic, because it can further decrease foreground performance by causing additional preemptions. These speculative I/O operations receive service along with the single application-requested I/O operation and lengthen the duration of idletime use of the device. This delays new foreground requests arriving during this time and lowers foreground performance. The implementation accordingly disables read-ahead for idletime use.

3.2 Network Scheduler

Similar to the *fcntl* option used by the idletime disk scheduler, a process uses the *setsockopt* system call on an open socket to set the idletime option. The idletime scheduler operates at the network layer. The kernel tags IP packets sent from a socket with the idletime option enabled using a particular type-of-service value. The current implementation uses *0x20*, which the Internet2's Q-Bone Scavenger Service uses for a similar purpose [31]. The current implementation supports IPv4, but IPv6 support is straightforward using the "traffic class" field in the IPv6 header.

When receiving a packet with the type-of-service field set to "idletime," the kernel enables the idletime option on the corresponding receive socket. This causes future application responses on the same socket to use idletime traffic automatically also. A future release will make this behavior optional to give applications explicit control over their service level.

The preemption interval scheduler for network idletime service extends the *ALTQ* queuing framework [4]. *ALTQ* replaces the standard FIFO outbound queue with configurable queuing disciplines, including a priority queue, which served as the basis for the idletime scheduler. The idletime scheduler replaces the indirect *ifq_dequeue* call that all network drivers use to obtain the next packet to send. In the original implementation using a FIFO queue, this call simply returns the packet at the head of the queue.

ALTQ modifies only outbound queues. However, systems also queue inbound packets during receive operations, using a FIFO by default. An earlier research effort produced an *ALTQ* extension to support different queuing disciplines for the inbound queue [9]. However, an earlier research has demonstrated that inbound scheduling is ineffective for a conventional OS due to its inherent prioritization [9]. Therefore, the current idletime scheduler does not manage the inbound queue.

3.3 Implementation Considerations

One key difference between the implementation and the initial description of idletime service in Section 2 is the starting point of the preemption interval. The implementation starts the preemption interval timer when the resource starts a foreground request, instead of immediately after it finishes, as described above.

This difference is a design decision and allows investigation of scenarios where the preemption interval is shorter than the service time of the resource. This difference also simplifies the implementation for resources that internally batch together or reorder requests. They would otherwise require individual driver modifications instead of a single modification at a higher layer.

The preemption interval timers use the standard FreeBSD timing facilities [37], which offer efficient, constant-time operations. However, the current scheduler incurs a timer restart for each foreground request. The timer management overhead could become noticeable for high-rate resources. A first improvement is batching the foreground requests together and restarting the timer only at the end of a batch. Additionally, improvements to the timer facility itself can further decrease this overhead [1]. Very high-rate resources may require direct use of hardware timers.

One limitation of the current prototype implementation is that the length of preemption interval is a per-scheduler parameter. Thus, all idletime schedulers for a specific type of resource use the same preemption interval length. Although this does not affect the validity of the benchmarks presented in Section 4 (because they measure only idletime use on a single resource) a future revision must implement preemption delays as per-resource properties.

4. EXPERIMENTAL EVALUATION

Section 3 described implementations of the idletime scheduler defined in Section 2. This section discusses the measured performance of both disk and network scheduler implementations under example workloads. Due to space limitations, only some of the performed experiments are shown. A much more detailed evaluation is available in [10].

Figure 6 shows the experimental setup that this section analyzes. Two processes run concurrently on a single machine, one issuing foreground requests, the other issuing background requests for the same resource managed by an idletime scheduler. The objective is to quantify the concurrent foreground and background performances achieved by the two processes.

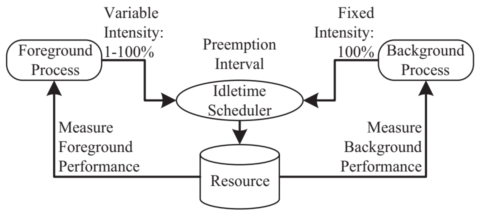

Figure 6. Experimental setup.

Each process generates resource requests at a certain rate, called the "intensity" of the process. It specifies the fraction of CPU time a process uses to generate resource requests. For example, at 50% intensity a process spends half its cycles generating load and half its cycles performing other tasks. (Note that the CPU is never the bottleneck resource in these experiments and CPU scheduling of the benchmark processes hence does not influence the results.)

The intensity of the foreground process is one variable of the experiment and varies between 1-100%. Varying the foreground intensity allows studying the impact of idletime use as the foreground workload increases. For this purpose, the exact request patterns are not relevant and the relatively coarse intensity-based mechanism is sufficient.

The background process is greedy and tries to consume as much resource capacity as possible, *i.e.*, its intensity is always 100%. This models the desired scenario for background use, where any idle resource capacity becomes utilized. This models the worst-case scenario for idletime use, because each time a new foreground request arrives, it incurs a preemption delay due to ongoing idletime use. The resulting performance measurements therefore also determine the worst-case performance of the scheduler.

A second variable of the experiment is the length of the preemption interval. Depending on the resource, it varies from zero up to a few hundreds of milliseconds. When the preemption interval is less than the service time of a request, the behavior of the scheduler is identical to a simple priority queue. A preemption interval larger than the CPU quantum (100 ms) completely stalls idletime use, because the benchmark processes generate at least one resource request per CPU quantum.

Each experiment consists of five separate 30-second iterations for each data point (unique pair of intensity and preemption interval) to compute the average foreground and background performances and their standard deviations. Standard deviations are usually less than 5% and are therefore omitted to avoid cluttering the density plots. Normalizing the average measured performances against the performance of the baseline case (without background work present) allows comparison of the relative impact of idletime use.

The experiments used PC workstations running release 4.7 of FreeBSD together with a modified *ALTQ* framework [4] to support idletime use. Each PC was equipped with 512 MB of RDRAM and dual 733 MHz Intel Pentium-III processors. FreeBSD 4.7 can run user processes simultaneously on multiple processors, but allows only a single CPU to execute kernel code at any time. This eliminates contention between the foreground and background load generators – increasing the offered load without affecting in-kernel processing.

4.1 Performance Expectations

A formal model and quantitative analysis of the expected behavior of idletime scheduling is presented in [10] and has been shown to describe the scheduler behavior to within 15% accuracy for different resources. Due to space limitations, this section summarizes the expected behavior. Figure 7 illustrates the expected density plots for an ideal idletime scheduler and identifies several regions of interest. For example, the scheduler should support close to 100% foreground performance with a preemption interval larger than the resource service time, independent of the foreground intensity. This appears as the larger, lightly shaded area in the left graph in Figure 7.

The bottom dashed line cutting across both graphs signifies the service time of the resource. With a preemption interval of less than the service time, the scheduler is expected to be ineffective, as described above. For reference, the baseline case without idletime use would appear as a single, evenly shaded area of the lightest shade of gray, independent of foreground intensity or preemption interval lengths.

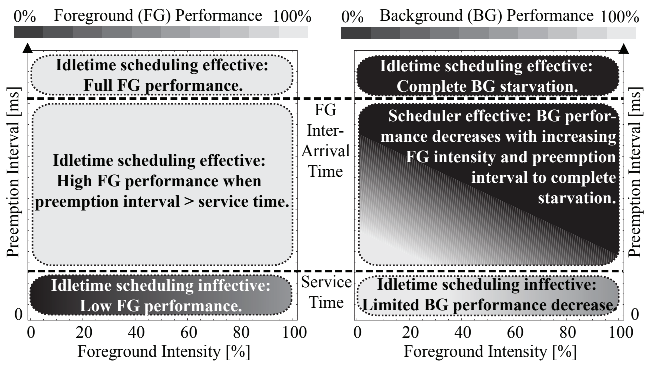
0% Foreground (FG) Performance 100%
0% Background (BG) Performance 100%
Idletime scheduling effective: Full FG performance.
Idletime scheduling effective: High FG performance when preemption interval > service time.
Idletime scheduling inffective: Low FG performance.
FG Inter-Arrival Time
Service Time
Idletime scheduling effective: Complete BG starvation.
Scheduler effective: BG performance decreases with increasing FG intensity and preemption interval to complete starvation.
Idletime scheduling inffective: Limited BG performance decrease.
Preemption Interval [ms]
Foreground Intensity [%]
0
20
40
60
80
100

Figure 7. Illustration of expected foreground (left) and background (right) performance under idletime scheduling.

The graph on the right in Figure 7 shows the corresponding background performance in the same scenario. An ideal idletime scheduler utilizes available capacities for background work (especially at low foreground intensities), but begins to starve background work as foreground intensities rise. Consequently, background performance decreases with increasing foreground intensity and preemption interval lengths, as illustrated by the gradient in the larger area in the right-hand graph.

With a preemption length shorter than the service time, the mechanism is again ineffective. Background performance still decreases with higher foreground intensities, but does not completely stall (smaller, bottom area in the right-hand graph in Figure 7), and foreground performance therefore decreases.

The two primary criteria for evaluating the effectiveness of an idletime scheduler are first, the impact on foreground tasks and second, the amount of background work scheduled. An ideal idletime scheme exhibits performance identical to the baseline case across the full range of foreground intensities for preemption intervals longer than a specific resource-dependent bound. In the density plots, this shows as very light shades of gray. It also succeeds in scheduling some background work during idletime, especially at light foreground intensities. This shows in the graphs as lighter areas.

4.2 Disk Scheduler Evaluation

During the disk benchmarks, the foreground and background benchmarks generate fixed-size disk read requests on the same test file containing random data. The 8.2 GB test file spans a UFS file system that completely utilizes a Western Digital Caviar AC28200 disk connected on a separate ATA channel. The manufacturer-reported maximum mean seek time of this drive is 15 ms and the mean latency is 5 ms, including controller overhead. Thus, the resulting total average service time for a random access to a single disk block is 20 ms, which has been empirically verified.

Two separate experiments investigate the performance of the idletime scheduler under random and sequential disk accesses. This paper, however, only discusses the first experiment due to space restrictions. The sequential-access experiments are described in [10]. In the random-access scenario, the benchmark processes read single bytes from random locations across the test file, causing accesses to single random disk blocks. The benchmark processes also re-mount the test file system before each run, which flushes the buffer cache to eliminate cache effects across successive runs.

In this experiment, both foreground and background process read 512-byte disk blocks from random locations in the 8.2 GB test file. During each run, they read up to 1,600 blocks at a rate of up to 60 blocks/second with a standard deviation of less than 0.5 blocks/second in all cases.

Figure 8 shows the throughput of the foreground (left graph) and background (right graph) benchmarks. Lighter shades of gray indicate areas of higher throughput. The graphs break down into three areas of interest: less than 20 ms (preemption interval less than service time), greater than 100 ms (preemption interval greater than CPU quantum), and the middle range of preemption intervals between 20-100 ms.

With a preemption interval of less than 20 ms, the idletime scheduler is not expected to be effective. The background benchmark monopolizes the disk, receiving 50-100% throughput, while the foreground throughput degrades to 50% with increasing intensity. The average access delay (seek time plus rotational delay) of the drive in this benchmark is approximately 20 ms. A preemption interval shorter than its service time expires while its corresponding foreground request is still active. Thus, any enqueued background request immediately receives service after the last active foreground request finishes. The behavior is therefore identical to a traditional priority queue. However, although preemption intervals shorter than the service time fail to address foreground performance degradation, they do affect corresponding idletime performance.

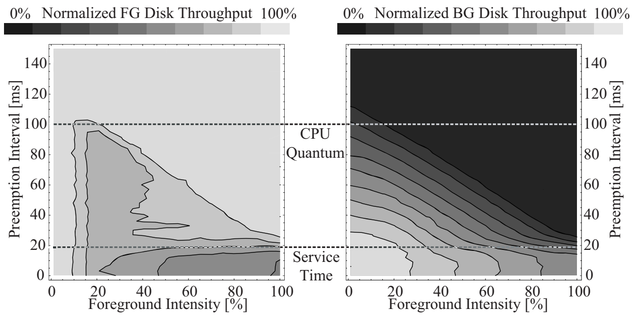

Figure 8. Measured random-access disk throughput.

The right graph in Figure 8 illustrates that idletime throughput already starts to decrease with preemption intervals longer than 10 ms. Foreground throughput, however, does not benefit from this reduction of idletime load until the preemption intervals exceed the service time. Preemption intervals in the second region of interest exceed the CPU quantum of 100 ms. Here, the idletime scheduler almost completely suppresses service of background requests. The preemption interval is longer than almost all inter-request gaps of the foreground request stream, independent of the intensity. This allows immediate back-to-back execution of foreground requests. While foreground throughput and latencies are very close to 100%, background throughput is almost.

Note that at low foreground intensities of less than 20%, a few background requests continue to receive service at preemption intervals of 100-120 ms. This is due to an implementation limitation of the FreeBSD *usleep* system call used by the benchmark processes. It can cause sleep intervals to lengthen under high system load, leading to slightly longer inter-arrival times that cause servicing of extra idletime requests.

The third region of interest consists of preemption intervals between 20-100 ms. Here, idletime scheduling improves foreground performance compared to shorter preemption intervals, but does not completely suppress background use, the way that longer preemption intervals do.

For foreground requests, throughput lies between 70-100% of baseline. As preemption interval length increases, idletime use stalls at lower foreground intensities. At a preemption interval of 40 ms, background requests do not receive service past 90% foreground intensity, whereas with a 80 ms preemption interval, this happens at 40% intensity. Section 2.1 discussed how the higher arrival rate of foreground requests lowers the possibility of preemption interval expiration and thus more easily preempts idletime use.

Another observation is that the idletime scheduler is effective at low intensities (less than 10%), no matter what the length of the preemption interval is. This effect is due to priority queuing. With a queue full of background requests, an incoming foreground request always moves to the head of the queue and receives service next. At low intensities, this is sufficient to achieve throughput comparable to the baseline case. (However, it must be noted that latency – not shown due to space limitations – is higher, because queued foreground requests block until the active background request finishes. As foreground intensity increases, priority queuing alone is not capable of maintaining throughput comparable to the baseline due to the impact of aggregate blockage delays.)

The two primary criteria for an idletime scheduler are minimal foreground performance impact and effective utilization of idle capacity. The previous measurements show that the idletime disk scheduler sustained foreground throughputs of 70-100% of the baseline case under idletime load. With sufficiently high foreground intensities, the scheduler completely preempts idletime use. Consequently, throughputs are practically identical to the baseline case.

One major reason for the high foreground performance impact lies in timing granularities. The average service time of the disk drive used for the experiments is approximately 20 ms for random requests. This means that the disk can serve only approximately five random-access requests in 100 ms. Thus, whenever the idletime scheduler starts servicing a single background request, it reduces foreground performance by up to 20%. This occurs, because the benchmark generates at least one foreground request per CPU quantum. This effect also causes the difference between the expected and measured foreground performance for preemption intervals less than the service time – the unlimited background load has a greater impact.

These results are not surprising. The setup of the disk benchmark scenario violates the heuristic for determining preemption interval lengths described in Section 2.1. The preemption interval should be at least an order of magnitude longer than the service time of the resource, in order to allow amortization of preemption costs across a burst of foreground requests. It should also be significantly less than the inter-arrival time of foreground requests to allow utilization of some idle capacities for background tasks.

The disk benchmark scenario does not satisfy both rules. The service time of the resource is 20 ms, but preemption interval lengths are less than 150 ms. Furthermore, foreground arrival rates even at lightest intensities reach 8-10 requests per second, corresponding to an inter-request gap of only approximately 80 ms. According to the heuristic in Section 2.1, given the arrival pattern in relation to the service time, the scheduler in this scenario should use a long preemption interval (~ 200 ms) and preempt background tasks to prevent interference with foreground use.

4.3 Network Scheduler Evaluation

The previous section presented experimental measurements of the idletime disk scheduler and analyzed them. This section presents a similar discussion for the idletime network scheduler.

Crossover patch cords established an isolated, directly connected link between two machines, using Intel PRO/1000F Fiber 1 Gb/s Ethernet interfaces. One machine acted as the traffic source, sending a mix of foreground and background traffic towards the sink machine. Each set of experiments evaluates a combination of two different network protocols (UDP and TCP) for foreground and background traffic, resulting in four different experiments to evaluate all protocol combinations. The TCP benchmark process at the source opens three separate, parallel connections to the *discard* service [29] on the sink.

The bandwidth-delay-product of a 1 Gb/s link with 1 ms delay (which exceeds the propagation delay of a local link) is approximately 128 KB. In order to eliminate the socket buffers or system calls as potential bottlenecks, the benchmark process increases socket buffers to 128 KB and then proceeds to send 128 KB chunks of random data to the sink [13]. Likewise, the *inetd* process implementing the *discard* service on the sink machine increases its socket buffers to 128 KB. Similarly, the UDP benchmark process uses three separate sockets to send 1,400 bytes of random data to the *discard* service on the sink. This avoids fragmentation, because the Ethernet MTU is 1,500 bytes.

Unlike TCP send operations, UDP send operations do not block for completion, but will instead return an error value if a message was not sent. This usually occurs due to outbound queue exhaustion. In such a case, the process sleeps for a random time between 10-15 ms to allow the queue to drain before sending more data. The preemption interval length for a given run was in effect on both source and sink hosts. Receiver-side preemption intervals enable correct idletime scheduling of the TCP acknowledgement stream flowing from the sink to the source.

Four separate experiments are required to cover all possible combinations of TCP and UDP foreground and background benchmarks. Due to space limitations, this paper presents only the two more interesting combinations: first, a simple scenario with UDP foreground and background senders and second, a more realistic scenario with a congestion-controlled TCP foreground sender competing against an unregulated, high-rate UDP background sender.

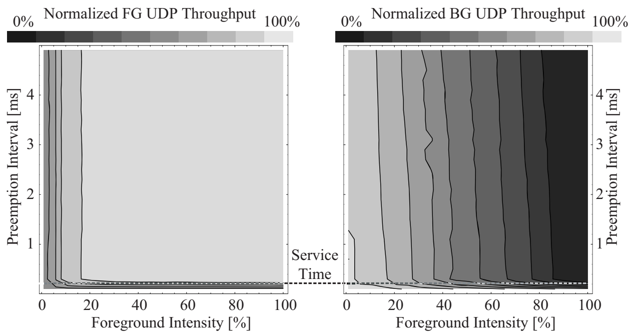

Figure 9. Measured 1 Gb/s Ethernet UDP/UDP throughput.

Figure 9 shows the throughputs of UDP foreground (left graph) and background (right graph) benchmarks. The graphs break down in two major areas based on the length of the preemption interval: less than 0.05 ms and greater than 0.5 ms. With a preemption interval less than 0.05 ms, the idletime scheduler is not effective. Foreground throughput is 50% of the baseline. The background traffic can monopolize the link at lower foreground intensities and performance of both traffic classes evens out as foreground intensity reaches 100%. This is expected, as the empirically measured service time of the resource is approximately 0.05 ms.

With a preemption interval longer than 0.05 ms, idletime scheduling becomes effective. At foreground intensities over 10%, foreground throughput achieves over 90% of the baseline case. With lower foreground intensities, foreground performance still reaches 60-80% of the baseline.

Unlike during the disk measurements, the idletime scheduler does not suppress background tasks completely to maintain unchanged foreground performance. Instead, it gradually reduces the amount of background traffic as foreground intensity increases. Background traffic stalls only at very high foreground intensities (> 90%).

The background performance prediction of the network interface on the right side in Figure 7 looks very different from the illustration overview in Figure 9. These two illustrations are juxtaposed in Figure 10. The overview shows two distinct triangular regions that split the middle area along its diagonal. In the top right triangle, background performance is extremely low, because idletime use stalls. In Figure 9, however this does not seem to occur.

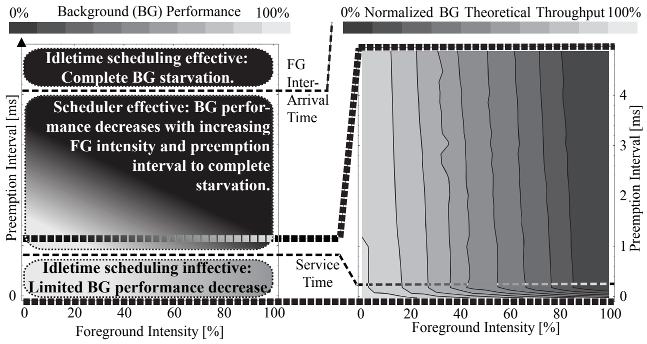

Figure 10. Comparison of expected (right graph, from Figure 7) and measured performances (left graph, from Figure 9).

The reason for this apparent discrepancy is that the service time of the network interface (0.05 ms) is orders of magnitude less than the assumed inter-arrival rate of the foreground requests (100 ms). The maximum preemption interval shown in Figure 9 is 5 ms and it shows thus only the small strip of the overview graph located above the service time, where the effects of the preemption interval length on background throughput are not yet significant.

Figure 11 shows an experiment where the foreground benchmark uses TCP while the background benchmark uses UDP. In a sense, this represents the worst-case scenario: foreground congestion-controlled TCP flows share a bottleneck path with greedy, high-rate UDP senders. With FIFO schedulers, the UDP traffic can significantly affect, or even starve, foreground traffic.

In such a scenario, an effective idletime mechanism should still sustain foreground performance at levels that are comparable to the baseline case without background load. Foreground throughput is 90-100% of the baseline case with a preemption interval longer than 1.25 ms. With preemption intervals shorter than 1.25 ms, the idletime mechanism is not effective.

Unlike with foreground UDP traffic, the service time of the resource (0.05 ms) is not a useful lower bound for effective service times for TCP foreground traffic. Significantly longer preemption intervals of 0.9-1.25 ms are required to raise foreground performance to levels comparable with the baseline case.

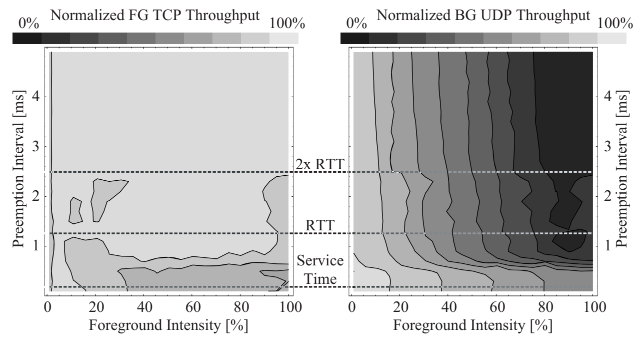

Figure 11. Measured 1 Gb/s Ethernet TCP/UDP throughput.

This lower bound of 1.25 ms is not arbitrary. The round-trip time (RTT) estimator in FreeBSD's TCP implementation uses 10 ms timers and averages the measurements using fixed-point arithmetic with a scaling factor of eight. This means that 1.25 ms is the smallest possible RTT estimate for TCP connections. This performance shift could therefore indicate a correlation between effective preemption interval lengths and the estimated RTT of foreground TCP connections. Furthermore, a second, minor performance improvement occurs with preemption intervals of over 2.5 ms (twice the RTT). This may indicate a correlation with delayed acknowledgements, which FreeBSD enables by default.

An additional experiment across a network with a longer transmission delay investigates this hypothesis. *Dummynet* [30] is a FreeBSD kernel mechanism to apply artificial delays, queue limits and loss rates to selected flows. *Dummynet* can simulate a wide-area link by buffering packets in a transmission queue, sized to accommodate the bandwidth-delay-product of the chosen link, for a given delay.

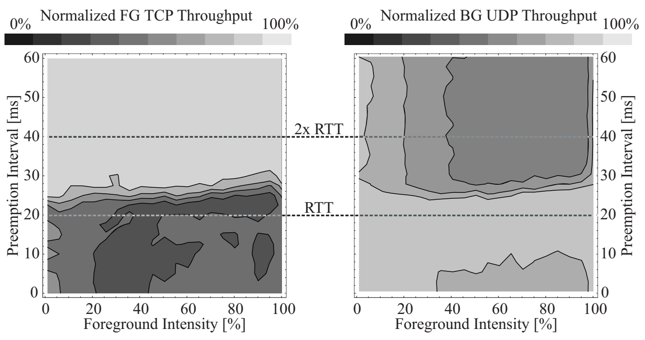

Figure 12. Measured 100 Mb/s TCP/UDP throughput with 10ms propagation delay.

However, simulating wide-area 1 Gb/s links with *Dummynet* is problematic due to its per-packet processing overhead [42]. The next experiment thus replaces the 1 Gb/s link of the previous experiment with a slower 100 Mb/s Ethernet connection. (A prior empirical analysis finds that the testbed hardware is sufficiently powerful to support *Dummynet* at speeds beyond 100 Mb/s.) Again, a crossover cable connects the test machines, using two Intel PRO/100 Fast Ethernet adaptors. *Dummynet* now simulates a 100 Mb/s wide-area link with 10 ms transmission delay.

Figure 12 repeats the previous experiment, where TCP foreground traffic competes with UDP background traffic, across the simulated wide-area link. The estimated RTT of the foreground connections clearly affects the minimum effective preemption length. In the corresponding LAN case, the idletime scheduler became effective with preemption intervals longer than the 1.25 ms RTT. Here, in the WAN case with 10 ms delay, the required preemption length for effective idletime scheduling is over 20-30 ms.

With a preemption interval less than 20-30 ms, foreground throughput reaches only approximately 50% of the baseline. Likewise, with preemption intervals less than 20-30 ms, the background throughput is high (70-90% of the baseline).

It is interesting to note that effective preemption lengths of 20-30 ms are slightly longer than the simulated RTT of 20 ms. Two factors may contribute to this effect. First, TCP's RTT estimator is conservative by design and over-estimates the RTT to avoid overloading the network. Second, the link is fully loaded during these measurements because background transmissions are greedy. Additional queuing delays can therefore increase the apparent RTT.

5. FUTURE WORK

This section briefly discusses areas for future improvement of the idletime mechanism described in this paper, including idletime scheduling in cases where a fixed, limited reduction of foreground performance may be acceptable, automatic adaptation of preemption interval lengths, idletime use of storage capacity, and specific improvements to idletime networking.

5.1 Fixed-Overhead Idletime Scheduling

The implementation of the idletime scheduler described in Section 4 bases its scheduling decisions strictly on current state, such as queue contents and preemption interval timers. It does not accumulate usage history or track usage statistics. This reliance on current state alone significantly simplifies operation and analysis of the idletime mechanism. However, it also eliminates possible optimizations of the idletime mechanism. For example, in certain scenarios, the user's foreground delay policy may allow for a specific, fixed decrease of aggregate foreground performance. Under such a policy, the idletime scheduler may skip the preemption interval in a controlled fashion when switching from foreground to idletime use. This can increase idletime performance.

For example, consider a policy that permits a 10% reduction of foreground performance. Whenever the resource serves ten foreground requests without incurring a preemption delay, it can immediately switch to idletime use. Even if it must immediately preempt the idletime request for a newly arriving foreground request, the aggregate foreground performance will still exceed 90%. In general, the ratio between the number of serviced foreground requests that did not incur preemptions and the total number of preemptions bounds foreground performance.

The length of the event history acts as a moving averaging period. Long event histories potentially allow the scheduler to accumulate a large number of credits to skip preemption intervals. When the scheduler uses these credits in a short period, it may skip many preemption intervals and cause transient foreground performance to decrease past the permitted reduction. It may be useful to investigate *leaky bucket* or other rate-limiting schemes that bound the number of saved skip credits and the rate at which they may be spent. The idea of accumulating and spending credits is also similar to proportional-share schedulers [39]. Proportional-share schedulers allocate different fractions of resource capacity based on a weight distribution. In the context of the idletime scheduler, such a mechanism would not manage resource use directly, but instead control the overheads of bypassing preemption intervals.

5.2 Automatic Preemption Interval Adaptation

The current idletime scheduler requires manual specification of an appropriate preemption interval for a given resource and workload. One key improvement is a mechanism that automatically adapts the preemption interval based on observed resource and workload characteristics. Effective idletime use requires amortization of preemption cost over a burst of foreground requests. Foreground bursts by definition incur at most a single preemption cost and, as a result, bound idletime overhead. The idletime scheduler could measure burst and delay statistics, and thus automatically adjust the preemption interval length.

The implementation described in Section 4 includes the beginnings of a framework to support such auto-tuning mechanisms. For each of the four possible states of the resource (*I*, *F*, *B* and *P*), the scheduler maintains event counters for the *f*, *b*, *t* and *i* events. For example, whenever a new foreground request appears (event *f*) during a preemption interval (state *P*), the scheduler increases the *P[f]* counter by one. A mechanism to adjust the preemption interval automatically can monitor these counters. For example, a rapidly increasing *B[f]* counter indicates that many idletime preemptions delay foreground requests and the preemption interval should increase. Likewise, a steady increase in the *P[f]* and *I[f]* counters could allow a reduction of the preemption interval to increase background throughput.

One interesting direction of future research is whether a TCP-like windowing mechanism can effectively manage preemption interval lengths based on these counters. For TCP, a segment loss serves as an indicator to decrease the congestion window. Similarly, an increase in *B[f]* can serve as an indicator for increasing the preemption interval length. In the absence of congestion losses, TCP slowly increases the window. In the same way, the preemption interval could shrink slowly over time.

5.3 Idletime Use of Storage Capacity

Idletime use of spatially shared storage resources, such as disk and memory space, requires additional mechanisms, due to their inherent persistence. Traditionally, when a process obtains storage capacity, it is free to use it at any time thereafter. The kernel does not withdraw that storage capacity until the process explicitly returns it. This behavior remains unchanged for foreground use of capacity resources under idletime scheduling, but background use of idle capacity follows a different service model.

The OS must reclaim unused storage allocated to idletime use when it is required to satisfy a newly arriving foreground request. This results in a service model where idletime storage can disappear at any time. Applications and services that wish to use idle capacity for storage must therefore gracefully adapt to these events. Many existing applications may not execute under this service model for idletime storage. Furthermore, new OS extensions must maintain the consistency of the overall system in the presence of preempted storage use. Finally, even with modified applications adapted to the service model of idletime storage, the OS must provide further mechanisms to merge isolated idletime data into the regular, foreground state. Without this merge operation, idletime data could never become visible to regular tasks, greatly reducing the usefulness of idletime storage.

Stateful resources, such as disk drives, are one challenge for idletime use of spatially shared capacity. File systems exploit locality by laying out data and metadata to reducing disk arm movement, increasing performance. A naïve implementation of an idletime mechanism can interfere with the layout of foreground data and reduce performance. Idletime schedulers must prevent such side effects of background use. The presence of idletime use must not affect the layout policy of the foreground file system.

Another benefit of providing these additional mechanisms to support transparent use of idle storage capacity is enabling *speculative* use of idletime capacity. The ability to store large amounts of data speculatively, without the possibility of interfering with regular, higher-priority storage requirements, allows straightforward support for aggressive optimizations such as caching and buffering. Combined with idletime use of temporally shared resources, such as the CPU or the network interface, these mechanisms provide an integrated framework for idletime use. Successful speculations become visible to other processes in the system through the integrated, transparent merge operation supported by spatially shared resources.

5.4 Idletime Networking Improvements

Another area of improvements is the idletime network scheduler. An earlier, experimental version of the idletime scheduler extended *ALTQ* to support different queuing strategies for the IP inbound queue [9]. At the time, experimental evidence showed that inbound scheduling offered only minimal performance improvements. Thus, the idletime inbound queuing code was not ported to the newer *ALTQ* release extended for idletime scheduling used during the experiments in Section 4.

However, the earlier idletime variant did not yet use preemption intervals. Preemption intervals during inbound network processing delay delivery of background IP packets to higher layers, such as transport protocols and applications. Preemption intervals during inbound processing could therefore further reduce the impact on foreground traffic. Additional experiments are required to investigate the effects of idletime inbound network scheduling.

Another possible idletime networking improvement applies to idletime use of the TCP protocol. The current idletime mechanism starts a preemption interval whenever an outbound TCP segment enters the network driver. However, TCP is a bidirectional protocol based on a stream of receiver acknowledgements for pacing transmissions. Starting preemption intervals upon the *receipt* of such an acknowledgment, in addition to scheduling them when sending data segments, may further enhance foreground TCP performance under idletime scheduling.

6. RELATED WORK

Related work falls into several broad categories. The first category includes systems that prioritize resource use, such as hard and soft realtime systems. The second category comprises of idletime execution systems, including systems for process and data migration. Finally, a third area is priority schemes for specific resources or applications. This section contrasts and compares these systems with the idletime scheduler.

Realtime systems, such as *Spring* [32], *Nemesis* [18], *Eclipse* [3], *Realtime Mach* [34] or *Omega* [25], among many others, differ in one or more of the following characteristics from a traditional, general-purpose OS: predictability, resource requirement specifications, and admission control.

Predictability requires time limits on all resource operations and scheduling overheads. Without such limits, guarantees for computation deadlines become impossible. Predictability is not required for idletime scheduling, although it might lower preemption costs. With a known service time for a request, a scheduler may let an idletime request finish instead of preempting it when a regular request arrives. If the time-to-finish of the idletime request is less than the preemption cost, this might decrease interference with regular use.

A second difference between regular and realtime systems is *a priori* specification of resource requirements. A realtime system uses this information for admission control to prevent overcommitting resource capacity. The resource requirements of dynamic workloads are difficult to predict and their worst-case resource use may be unbounded. This is one reason why realtime systems cannot easily implement idletime scheduling for general-purpose workloads. The idletime scheduler does not require resource requirement specifications. If idletime tasks choose to specify their resource requirements, the scheduler could optimize performance by not allocating available capacity to tasks that depend on fully loaded resources.

Many of the prioritized schedulers for realtime systems can implement the service prioritization required idletime service. However, prioritization is not sufficient to establish full idletime use; preemptability and isolation are also required. Realtime systems provide neither. For example, it is acceptable for a realtime system to continue servicing a lower-priority request when a higher-priority one arrives, as long as it fulfills all deadlines. In fact, it may be advantageous to avoid preemption to increase resource utilization. Isolation is a concept without an equivalent in realtime systems; side effects of execution at different priorities are always globally visible.

Several existing systems use idle remote resources for non-speculative purposes. One category of such systems is *process migration systems* (cycle harvesters), which push local processes to idle remote machines for faster execution, such as the *V System* [33] and *Condor* [19]. Another category is *data migration systems*, which push data to remote machines that execute a common process, such as *SETI@home* [14], *Folding@home*, and *Genome@home* [17]. Other data migration systems exploit idle remote memory as secondary storage [21][26].

Most existing systems that try to exploit idle capacity do not establish background processing as a separate, lower-priority service class. Instead, they often treat idleness as a system-wide condition and use *ad hoc* schemes to detect it (*e.g.*, CPU utilization threshold, no user logged in, screen-saver active). During perceived idle periods, they simply add background tasks to the system's workload under the regular execution priority. For a limited class of applications and workloads, such as the previously mentioned "@home" projects, this coarse approach works surprisingly well. However, it is not a general-purpose mechanism suitable for arbitrary usage scenarios. The approach fails to take advantage of idle capacities that exist even during busy periods and can severely affect foreground performance. Furthermore, many of these approaches, including process and data migration systems, focus only on a single resource (usually the CPU), and are ineffective at utilizing idle capacities elsewhere in the system.

A third area of related work focuses on prioritized service for specific resources or services, such as network traffic or disk I/O. Section 3.2 presented an idletime network service. Several other techniques aim at establishing prioritized network service. The idea of marking packets according to their priority is present in the original Internet architecture and its extensions to provide differentiated service [5] as well as several link-layer technologies, such as the ATM *cell loss priority* bit or the Frame Relay *discard eligible* bit.

Other approaches address traffic prioritization at the transport layer. One system to preload web caches uses a simulated, connectionless datagram protocol (essentially UDP) together with low-priority forwarding [7]. *TCP Nice* [38], *TCP-LP* [15] and *MulTCP* [6] are modifications of the traditional TCP congestion control algorithm that can support connection priorities.

Various application-level mechanisms strive to provide support for a background service class. *MS Manners* [8] is an application-level service that monitors the progress of cooperating background applications and reactively adjusts their aggressiveness. Microsoft's *Background Intelligent Transfer Service* (BITS) [20] and the *Mozilla* web browser [11] include network transmission schedulers that support background web transmissions.

The mechanism of the idletime scheduler is similar to *anticipatory disk scheduling* [12]. That work defines "deceptive idleness" that can lead to a reduction in performance for a work-conserving disk scheduler when multiple processes issue bursty disk requests. Anticipatory scheduling overcomes this issue by injecting short periods of idleness to stimulate the formation of request queues that improve the effectiveness of the *disksort* algorithm. These idle periods prevent excessive seeks that significantly lower performance when two processes access different locations on the disk. The idletime scheduler is a more general solution supporting arbitrary resources that specifically focuses on supporting different service levels. It was the result of an independent research effort that recognizes and counteracts a similar effect.

7. CONCLUSION

Common workloads on many computer systems rarely utilize resources fully. Using this idle capacity for productive work – without interfering with the ongoing foreground work – can improve overall system efficiency and user-perceived performance. Current application- and kernel-level approaches to provide different levels of service are frequently resource-, application- or workload-specific, require widespread changes to the OS or applications, fail to utilize significant amounts of available capacity for background use, or do not sufficiently protect foreground tasks in the presence of high-volume background work.

The idletime scheduler addresses these limitations. It is a generic, resource- and workload-independent kernel mechanism based on relaxing the work conservation property for the background service class. By introducing controlled delays – *i.e.*, preemption intervals – during background use, the idletime scheduler limits the likelihood of preemption events that delay foreground tasks. This consequently reduces the performance impact of idletime use on concurrent foreground work.

Preemption intervals amortize the cost of switching from background to foreground use over a series of foreground requests. The length of a preemption interval is a parameter that allows tuning of the mechanism, trading a reduction in idletime performance for an increase in corresponding foreground performance. This allows the idletime scheduler to adapt to a wide variety of resources, workload scenarios, and user delay policies. Preemption intervals establish system-wide idletime service through localized modifications to a subset of the schedulers in an existing OS. This property allows the idletime scheduler to enable background execution of existing applications and services in a traditional OS, because its API and foreground service model remains unmodified.

This paper discussed challenges associated with idletime use in detail, including system architecture, properties of existing schedulers, preemption cost as the key factor delaying foreground tasks, and cache effects. It explained how new and existing services and applications benefit from the availability of idletime service. Examples include prefetching, precomputation, and caching, transparent replication of data, and scheduling of maintenance operations and system optimizations. Finally, it defined two metrics for successful idletime mechanisms: minimizing foreground delays, and maximizing background workload.

Section 3 discussed the implementation of one variant of idletime scheduling that satisfy the principles defined in Section 2; Appendix A describes other variants. Section 3 then described the implementation of idletime scheduling for the disk and network schedulers of the FreeBSD OS and discussed its features and limitations. Changes to a single scheduler established idletime use in each case. Section 4 experimentally investigated the disk and network scheduler implementations for several different workloads. All scenarios modeled the worst case of an unlimited idletime workload and measured regular and idletime performance. In most of the worst-case scenarios, the idletime scheduler was effective in shielding regular tasks from concurrent idletime use, incurring a maximum of 10-15% performance impact. The experiments also verified that the length of the preemption interval is an effective control mechanism that allows trading a reduction in idletime performance for an increase in foreground performance. Although these experiments confirm the general effectiveness of the present idletime scheduler, specific aspects, such as setting the preemption interval or supporting advanced performance policies, can be improved. Section 5 discusses such extensions.

APPENDIX A: ALGORITHMIC VARIANTS

This appendix discusses the Moore machines of different algorithmic variants that all conform to the idletime principles described in Section 2. The first obvious observation is that the resource shall remain in state *I* as long as it remains idle (event *i*). Second, Section 2 identifies strict work conservation for foreground requests as a requirement for idletime scheduling based on preemption intervals. It requires that the resource must immediately transition to state *F* when the head of the queue contains a foreground request (event *f*).

Another constraint is that a timeout event *t* can happen only during the preemption interval (when in state *P*). Repeated timeouts are not useful, ruling out the transition $P \rightarrow P$ on *t*. Because event *t* implies that there is no foreground request present in the queue (*not f*), always transitioning to the idle state *I* on event *t* is the only useful choice. Furthermore, a preemption interval starts only after useful work, *i.e.*, following state *F* or state *B*. They never follow state *I*, ruling out the $I \rightarrow P$ transition.

The key criterion for the idletime scheduler is that a preemption interval occurs when switching from foreground to idletime work. This means that all paths from state *F* to state *B* must go through state *P* (the preemption interval), eliminating $F \rightarrow B$ and $F \rightarrow I$. Consequently, state *P* must follow state *F* for both events *b* and *i*. (Note that on event *b*, $F \rightarrow P$ does not consume *b*, signified by *b/b*.) Furthermore, entering state *B* requires event *b*. Thus $I \rightarrow B$ can occur only on event *b*.

The resource can only serve background requests that exist at the head of the queue. Therefore, entering state *B* requires event *b*. This rules out $B \rightarrow B$ on event *i*. Also, a transition $B \rightarrow I$ on *b/b* is not useful, because $I \rightarrow B$ on event *b* immediately follows. For event b in state B, the only two possibilities are therefore $B \rightarrow B$ or $B \rightarrow P$. Similarly, the only possibilities for event i in state B are $B \rightarrow I$ or (again) $B \rightarrow P$.

Figure 13 shows all four remaining state machine variants, with their differences highlighted using thicker arcs. The four variants all satisfy the idletime properties: they are prioritized, preempting, strictly work-conserving for foreground requests, and weakly work-conserving for idletime requests. Whenever the resource switches from foreground to idletime use, it incurs a preemption interval. In terms of the state machine, this means each path from F to B visits P.

However, several differences exist between the four variants. The top two variants remain in state B for a burst of b events, whereas the bottom two switch to state P and require a timeout on each b event. The difference is that the top two variants incur a single preemption interval before each burst of idletime requests: $b^{+} \rightarrow IB^{+}$. The bottom two instead incur a preemption interval before each idletime request in a burst: $(bt)^{+} \rightarrow I(BPI)^{+}$. In other words, the top two variants are strongly work-conserving for idletime requests once state B is reached and idletime use begins, whereas the bottom two variants are always weakly work-conserving for idletime requests.

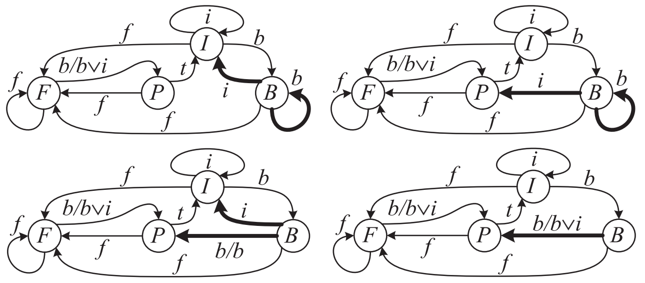

Figure 13. State transitions of four idletime variants.

Idletime performance of the two variants that are strongly work-conserving for the idletime workload is significantly higher than for the weakly work-conserving variants. On the other hand, the additional preemption intervals before each idletime request enforced by the weakly work-conserving variants increases the likelihood that arriving foreground requests find the resource idle. Consequently, they can further decrease foreground delays. Because the purpose of a preemption interval is to delay idletime use after foreground use, enforcing additional preemption intervals between idletime requests, when the last foreground request may have happened long ago, is not likely to be useful.

The second difference between the variants is their behavior when idletime use is bursty, *i.e.*, when i events occur between b events. The left two variants in Figure 13 immediately enter the idle state I when event i is encountered during idletime use in state B: $(bi)^{+} \rightarrow I(BI)^{+}$. The right two variants enter a preemption interval P instead and require additional timeouts: $(bit)^{+} \rightarrow I(BPI)^{+}$.

Extra preemption intervals between two idletime bursts therefore decrease idletime performance and, in turn, reduce foreground delays. As before, however, enforcing additional preemption intervals between bursts of idletime use, without occurring foreground use, is not likely to be useful.

Therefore, the top left variant was chosen for implementation and experimental evaluation (Sections 3 and 4). It maximizes idletime performance by avoiding preemption intervals between successive idletime requests and between bursts of idletime requests. Because maximizing idletime use was the secondary objective of a successful idletime mechanism (prevention of foreground delays is the primary objective), this variant appeared most functional.

ACKNOWLEDGMENTS

The authors would like to thank the members of T-Troup, especially Yu-Shun Wang, Amy Hughes, Venkata Pingali, Purushotham Kamath and Josh Train, for their feedback during all stages of this research. Ted Faber, John Heidemann, Greg Finn and Christos Papadopoulos have also provided valuable comments.

Partly supported by the Defense Advanced Research Projects Agency (DARPA) through FBI contract #J-FBI-95-185 entitled "Large-Scale Active Middleware" and through Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreements number F30602-98-1-0200 entitled "X-Bone" and number F30602-01-2-0529 entitled "DynaBone." The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, the FBI, or the U.S. Government.

REFERENCES

[1] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, Vol. 18, No. 3, August 2000, pp. 197-228.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. Proc. *AFIPS Joint Computer Conference*, Atlantic City, NJ, USA, April 18-20, 1967, pages 483-485.

[3] John Bruno, Eran Gabber, Banu Özden, and Abraham Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. Proc. *USENIX Annual Technical Conference*, New Orleans, LA, USA, June 15-19, 1998, pp. 235-246.

[4] Kenjiro Cho. A Framework for Alternate Queuing: Towards Traffic Management by PC-UNIX Based Routers. Proc. *USENIX Annual Technical Conference*, New Orleans, LA, USA, June 15-19, 1998, pp. 247-258.

[5] David Clark and Wenjia Fang. Explicit Allocation of Best-Effort Packet Delivery Service. *IEEE/ACM Transactions on Networking*, Vol. 6, August 1998, pp. 362-373.

[6] Jon Crowcroft and Philippe Oechslin. Differentiated End-to-End Internet Services using a Weighted Proportional Fair Sharing TCP. *ACM SIGCOMM Computer Communication Review*, Vol. 28, No. 3, July 1998, pp. 53-67.

[7] Brian D. Davison and Vincenzo Liberatore. Pushing Politely: Improving Web Responsiveness One Packet at a Time. *Performance Evaluation Review*, Vol. 28, No. 2, September 2000, pages 43-49.

[8] John R. Douceur and William J. Bolosky. Progress-based regulation of low-importance processes. Proc. *ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island Resort, SC, USA, December 12-15, 1999, pp. 247-260.

[9] Lars Eggert and Joseph D. Touch. End-System Support for Idletime Networking. *ISI Technical Report ISI-TR-559*, USC Information Sciences Institute, May 2001.

[10] Lars Eggert. Background Use of Idle Resource Capacity. *Ph.D. Thesis*, Department of Computer Science, University of Southern California, 941 W 37th Pl, Los Angeles, CA 90089, USA, May 2004.

[11] Darin Fisher. Mozilla Link Prefetching FAQ. October 14, 2002.

[12] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. Proc. *ACM Symposium on Operating Systems Principles (SOSP)*, October 21-24, 2001, Chateau Lake Louise, Banff, Alberta, Canada, pp. 117-130.

[13] Van Jacobson, Robert Braden and Dave Borman. TCP Extensions for High Performance. *RFC 1323*, May 1992.

[14] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb and Matt Lebofsky. SETI@home: Massively Distributed Computing for SETI. *IEEE Computing in Science and Engineering*, Vol. 3, No. 1, January/February 2001, pp. 78-83.

[15] Aleksandar Kuzmanovic and Edward W. Knightly. TCP-LP: A Distributed Algorithm for Low Priority Data Transfer. Proc. *IEEE INFOCOM*, San Francisco, CA, USA, April 2003, pp. 1691-1701.

[16] Butler Lampson and David Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, Vol. 23, No. 2, February 1980, pp. 105-117.

[17] Stefan M. Larson, Christopher D. Snow, Michael Shirts and Vijay S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*, Horizon Press, 2002.

[18] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Bar-ham, 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* (JSAC), Vol. 14, No. 7, September 1996, pp. 1280-1297.

[19] Michael J. Liztkow, Miron Livny and Matt W. Mutka. Condor – A Hunter of Idle Workstations. Proc. *International Conference on Distributed Computing Systems (ICDCS*), San Jose, CA, USA, June 13-17, 1988, pp. 104-111.

[20] Microsoft Corporation. Background Intelligent Transfer Service. *Microsoft Windows Server Technical Article*, August 2002.

[21] Ronald G. Minnich and David J. Farber. The Mether system: A distributed shared memory for SunOS 4.0. Proc. *Summer USENIX Conference*, Baltimore, MY, USA, June 12-16, 1989, pp. 51-60.

[22] David Mosberger and Larry L. Peterson. Making Paths Explicit in the Scout Operating System. Proc. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, USA, October 28-31, 1996, pp. 153-168.

[23] Matt W. Mutka and Miron Livny. Profiling Workstations' Available Capacity For Remote Execution. Proc. *IFIP WG 7.3 Symposium on Computer Performance*, Brussels, Belgium, December 7-9, 1987, pp. 529-544.

[24] Matt W. Mutka and Miron Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, Vol. 12, 1991, pp. 269-284.

[25] Klara Nahrstedt, and Jonathan M. Smith. Design, Implementation and Experiences with the OMEGA End-point Architecture. *IEEE Journal on Selected Areas in Communications* (JSAC), Vol. 17, No. 7, September 1996, pp. 1263-1279.

[26] Thomas Narten and Raj Yavatkar. Remote Memory as a Resource in Distributed Systems. Proc. *IEEE Workshop on Operating Systems*, Key Biscane, FL, USA, April 23-24, 1992, pp. 132-136.

[27] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve World-Wide Web latency. *ACM SIGCOMM Computer Communication Review*, Vol. 27, No. 3, 1996, pp. 22-36.

[28] POSIX 1003.1b-1993. Portable Operating System Interface (POSIX) Part 1: System Application Program Interface Amendment 1: Realtime Extension [C Language], 1993.

[29] Jon Postel. Discard Protocol. *RFC 863*, May 1983.

[30] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, Vol. 27, No. 1, January 1997 pp. 31-41.

[31] Stanislav Shalunov and Benjamin Teitelbaum. QBone Scavenger Service (QBSS) Definition. *Internet2 Technical Report*, March 16, 2001.

[32] John A. Stankovic and Krithi Ramamritham. The Spring Kernel: A New Paradigm for Realtime Systems. *IEEE Software*, Vol. 8, No. 4, May 1991, pp. 62-72.

[33] Marvin M. Theimer, Keith A. Lantz and David R. Cheriton. Preemptable Remote Execution Facilities for the V-System. Proc. *ACM Symposium on Operating Systems Principles (SOSP)*, Orcas Island, WA, USA, December 1985, pp. 2-12.

[34] Hideyuki Tokuda, Tatsuo Nakajima and Prithvi Rao. Real-time Mach: Towards a Predictable Realtime System. Proc. *USENIX Mach Symposium*, Burlington, VT, USA, October 4-5, 1990, pp. 73-82.

[35] Joseph D. Touch. Parallel Communication. Proc. *IEEE INFOCOM*, San Francisco, CA, USA, March 28 - April 1, 1993, pp. 506-512.

[36] Joseph D. Touch and David J. Farber. An Experiment in Latency Reduction. Proc. *IEEE INFOCOM*, Toronto, Canada, June 12-16, 1994, pp. 175-181.

[37] George Varghese and Anthony Lauck. Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility. *IEEE/ACM Transactions on Networking*, Vol. 5, No. 6, 1997, pp. 824-834.

[38] Arun Venkataramani, Ravi Kokku and Mike Dahlin. TCP Nice: A Mechanism for Background Transfers. Proc. *Symposium on Operating Systems Design and Implementation (OSDI)*, December 9-11, 2002, Boston, MA, USA.

[39] Carl A. Waldspurger and William E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. *Technical Memorandum MIT/LCS/TM-528*, MIT Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, June 1995.

[40] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt. Scheduling Algorithms for Modern Disk Drives. Proc. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Nashville, TN, USA, May 16-20, 1994, pp. 241-251.

[41] Peter Wyckoff, Theodore Johnson and Karpjoo Jeong. Finding Idle Periods on Networks of Workstations. *Technical Report TR1998-761*, Computer Science Department, New York University, March 1998.

[42] Marko Zec and Miljenko Mikuc. Real-Time IP Network Simulation at Gigabit Data Rates. Proc. *International Conference on Telecommunications (ConTEL)*, Zagreb, Croatia, June 11-13, 2003.