Speculative Use of Idle Resources

Ph.D. Dissertation Proposal

Lars Eggert

larse@isi.edu

Computer Science Department

University of Southern California

Los Angeles, CA 90089-0781

USA

October 22, 2001

(Appendices added July 2, 2002)

Abstract

Even a fully loaded computer system, where the bottleneck resource is constantly busy, often has some idle capacities available on other resources. This proposal argues for using these idle capacities speculatively, increasing system performance for correct predictions. In such a system, all resources will ideally be constantly loaded with either regular foreground tasks, or speculative idle-time tasks.

The key contribution of this proposal is a model for non-interfering use of idle resource capacity, based on three principles: resource *prioritization* between regular foreground and idle-time use, *preemptability* of idle-time processing, and *isolation* of speculative side effects. Current operating systems fail to provide all three capabilities. Without new mechanisms, processing of speculative tasks can delay or even starve foreground processing, and result in a decreased foreground performance, instead of increasing it.

Under the proposed model, speculative tasks only execute using otherwise idle resource capacities; the model also shields foreground processing from the side effects of their presence in the system. Thus, speculation can no longer delay or interfere with foreground processing. Based on the model, a proof-of-concept design of network extensions for idle-time service can isolate foreground network packets from the presence of idle-time traffic to within 1-2% throughput.

The remainder of this thesis will focus on idle-time support for the network file system (NFS). Idle-time NFS requires an extension of the current mechanism to disk I/O and storage capacity, as well as new mechanisms to isolate speculative side effects based on virtualization of OS state.

Contents

Figures

1 Introduction

One of the main tasks of an operating system (OS) is resource management. Many computer systems have plenty of idle resource capacity, even under peak load. For any increasing workload, a small fraction of the total resources becomes the fully utilized system bottleneck, while other resources have idle capacity available. The workload determines the bottleneck resource: it may be the network link for a web server, while it could be the disk for a database system. Other non-bottleneck resources (RAM, CPU, etc.) may remain partially idle.

The focus of this proposal is a model to utilize such idle capacities for speculative tasks, without interfering with or delaying regular non-speculative use. The second part of this proposal applies the model to design idle-time mechanisms for network service. An experimental evaluation of a proof-of-concept implementation of the idle-time networking mechanisms shows them to isolate regular foreground traffic from the presence of speculative tasks to within 1-2% throughput.

Several studies investigate the resource utilization of systems. One reports an average of 50-70% of the total memory of a cluster of machines to be available [ACHARYA 1999], and approximately 15-30 minutes between memory use peaks on a single machine. It concludes, “dips in memory availability (…) are likely to lead to a perception of memory being short.” Other studies focus on CPU utilization [MUTKA 1991][MUTTKA 1987][WYCKOFF 1998] and report that around 70% of the monitored machines in a network where idle.

Idle capacities may be even larger than above studies suggest, due to their coarse metrics to determine idle times (e.g. “no user logged in”, “screen saver active”, “CPU load minimal”). Short, transient idle times may remain undetected due to quantization effects caused by these coarse idle metrics. Furthermore, none of these studies monitored multiple, different resources. In cases where the monitored resource of a system seems busy, other resources could have significant idle capacities. For example, a system with a 50% loaded CPU (“not idle” according to above coarse metrics) may still have significant idle disk capacity – in fact, it even has a considerable idle CPU capacity of 50%.

Idle resource capacity is wasted; it cannot be saved for later use. Scheduling useful work during idle resource periods could increase system efficiency. Ideally, all resources should be constantly

busy, executing either regular or speculative requests. A good speculative request is a likely future request issued by a regular process. Pre-execution of such a request – before it is issued explicitly – can hide its processing latency from the issuing process, resulting in a performance increase. Not all speculations are likely to be correct. Thus, it is important to shield other processing from their presence until a correct prediction is confirmed, and the result of the speculation can be made available to the system.

To avoid interference with regular use of a resource (and thus decreasing performance), speculation should be limited to periods where the resource is not busy – speculative resource use should only occur when resource capacity would have been idle in its absence. Ideally, the presence of speculative resource use in the system will thus have no impact on regular processing, neither preventing nor delaying it: it should be *non-interfering*. In the pathological case of a constantly busy resource, a speculative request for idle time use will starve forever. A conventional OS will strive to prevent starvation of any request, prohibiting idle-time use.

Such speculative resource use is already common in some areas: One area is microprocessors with support for speculative branch execution. Such processors use idle execution units and memory bandwidth to pre-fetch and speculatively execute likely future instructions, giving total priority to non-speculative processing. Hardware mechanisms preempt speculative use without affecting regular execution, and manage speculative state, keeping it isolated until commit time (or discard time, for mispredictions). The key design principles of these microprocessors are *prioritized* and *preempted* use of microprocessor resources, and complete *isolation* of the side effects of speculative execution.

The focus of this proposal is mechanisms to enable such non-interfering uses of idle resource capacity. It argues that the same design principles enabling speculative branch execution on microprocessors (*prioritization*, *preemptability*, and *isolation*) are necessary and sufficient to do the same at the OS-level. Current systems fail to provide these capabilities, because most of their resource schedulers do not offer prioritized, preempted access. Instead, a general-purpose OS employs simple and predictable resource schedulers, trying to provide fair service to all users and prevent starvation. On such systems, speculative *background* use of idle capacity can delay or even prevent regular *foreground* use, as the speculative workload increases.

A conventional OS also does not isolate all the side effects of one process from another. Without *isolation*, speculative state can interfere with regular execution. *Isolation* virtualizes the operating system state, to shield regular processing from the side effects of speculation.

One of the main contributions of this proposal is a model for non-interfering idle-time resource use that encompasses both I/O and storage resources. Another challenge is inter-resource interference, where idle-time processing on one resource delays regular use of another. Finally, integrated resource scheduling, where speculative tasks are scheduled depending on their idle-capacity requirements, is a possible optimization to spend idle capacities more effectively.

Later chapters of this proposal apply the model to network stack extensions supporting idle time use. With these extensions, routers and end systems differentiate between regular and idle-time transmissions: Idle-time packets are dropped or delayed in favor of regular best-effort packets, and only receive a diminished service. Experimental results show that these idle-time extensions can isolate foreground traffic from the presence of speculative transmissions to within 1-2% of obtainable throughput.

One example application that benefits from such idle-time resource use is prefetching of likely future FTP or web requests [TOUCH 1994][TOUCH 1995][PADMANABHAN 1996]. Conventional prefetchers must explicitly limit their speculative transmissions, to avoid excessive interference with regular network traffic. Idle-time networking enables aggressive prefetching without the possibility of interference with regular network traffic. Similarly, idle-capacity use of storage resources (such as memory or disk space) allows the prefetch cache to grow without affecting foreground storage use.

As mentioned above, the three principles establishing non-interfering idle-time use are *prioritization*, *preemptability*, and *isolation*. *Prioritization* guarantees that a waiting regular request will always receive service before any idle-time one. *Preemptability* describes the property of immediately suspending or aborting ongoing idle-time use if capacity is required to service an incoming regular request. Finally, *isolation* shields regular use from the side effects of speculative processing until they are committed (successful speculation) or discarded.

The proof-of-concept implementation for idle-time use presented in this proposal supports only *prioritized* and *preempted* network I/O and CPU use, *isolation* of speculative side effects is not

yet established. One application that requires additional mechanisms is idle-time use of the network file system (NFS). To support this, the remainder of this thesis research will investigate idle-time mechanisms for disk I/O and storage capacity, as well as integration of these new techniques with the existing idle-time network stack.

Investigation of mechanisms to establish *isolation* between regular NFS and idle-time NFS will be one focus area of this research. Another key issue is inter-resource interference between the disk, network, and CPU subsystems. While the current mechanisms control interference between the CPU and network stack, their effectiveness must be confirmed in the presence of speculative disk use. Finally, given that idle-time NFS depends on the availability of idle capacity on multiple resources, it is an effective testbed to experiment with integrated scheduling, to optimize what speculative tasks available idle capacity is allocated to.

Using idle resources for productive work is not a new idea. Several *remote execution* systems can detect idle (or under-utilized) remote machines, and include mechanisms to migrate part of the local workload onto these remote hosts. Other migration systems use idle remote memory instead of local secondary storage. A few key differences between these systems and this proposal exist: First, migration systems focus on distributing the workload for a single resource (typically CPU or memory) – other remote resources remain idle. The proposed system tries to utilize idle capacities on *all* resources. Second, migration systems typically do not issue speculative tasks; the migrated requests are part of the regular system workload. Third, migration systems depend on remote hosts to actively donate idle resources, while this proposal can use idle local resources in a number of ways: Simply turn them off to save power, donate them for remote use (supporting migration systems), or use them speculatively to increase local performance. Idle capacities in a migration system never benefit the local system.

Other techniques to increase local performance through local speculation also exist, such as file system optimization or read-ahead caching. However, all these systems process speculative work at the same priority as regular tasks. Thus, speculation can affect foreground performance, and speculative tasks must explicitly limit their aggressiveness to avoid decreasing non-speculative performance. This proposal, on the other hand, schedules speculative work at a lower priority than all other processing, and implicitly shields foreground tasks from the presence of speculative resource use.

The remainder of this proposal is organized as follows: Chapter 2 defines the idea of non-interfering idle-time use for resources in detail. It identifies the key principles to support such idle-time use, and discusses how current OS mechanisms fail to satisfy these requirements. A design for OS extensions supporting non-interfering idle-time use form the main part of that chapter, followed by a discussion of applications that benefit from idle-time use.

Chapter 3 discusses some key challenges with offering idle-time use of resource capacity, such as preemption overheads, cache pollution and speculative workload generation.

Chapter 4 applies the model to idle-time extensions for the network stack. It first presents experimental evidence that highlights how current OS mechanisms fail to provide differentiated network service, and identifies the key issues prohibiting such service under the current network model. The final part of that chapter presents a prototype implementation of idle-time extensions for the BSD network stack, and evaluates their effectiveness through a series of experiments.

Related work, such as real-time systems, idle-time execution and other speculative techniques form the main part of Chapter 5.

Finally, Chapter 6 presents the plan for the remainder of this thesis research, including a timeline. The focus will be on idle-time support for the network file system (NFS). NFS service requires idle-time support for CPU and network service (already existing), as well as idle-time support for disk I/O and storage. This dependency on multiple resources makes idle-time NFS a good candidate to study inter-resource interference, as well as investigate mechanisms for integrated scheduling.

2 A Model for Speculative Use of Idle Resources

This chapter defines the idea of non-interfering idle-time use for resources in detail. It then identifies the key principles required to support such idle-time use, and discusses how current OS mechanisms fail to follow these principles. The later sections of the chapter present OS scheduler extensions that enable non-interfering idle-time use. Finally, it presents several application areas that benefit from idle-time use.

2.1 Introduction

A typical computer system contains multiple resources, normally at least a CPU and some main memory. Usually, a system also has some persistent storage devices (e.g. disks), communication devices (e.g. network, modem), and user I/O devices (e.g. keyboard, display, audio).

The resource use of processes can be seen as a request/response stream, where processes generate *resource requests* to acquire processing capacities (e.g. “read this disk block” for a disk, “send this packet” for the network, or “run me” for the CPU). Resources process these requests in some order, and may generate *resource responses* (e.g. “here is the block you wanted” for a disk read request). Note that some requests may not trigger a response, such as a “run me” request for CPU capacity.

Resources can be categorized according to a number of criteria. One such categorization is according to sharing patterns, distinguishing *spatially* and *temporally-shared* devices.

Spatially-shared devices divide their capacity into allocation units, and can serve multiple processes concurrently. Processes must lease allocation units before use. Leased capacity is unavailable to others; leased capacity becomes available for reuse only after a process explicitly returns it. Storage capacity (e.g. disk space, memory swap space) is an example of a spatially-shared resource.

A second category of resources is *temporally-shared*. Unlike spatially-shared resources, such resources do not subdivide their capacity for concurrent use. Instead, a single process is leased the full resource capacity for a certain (usually fixed) period. I/O devices (e.g. network interfaces) and the CPU are examples of temporally-shared resources.

Systems may contain multiple, identical resources. For example, on a system with multiple, channel-bonded network interfaces, a request can use any available interface. Such a device bundle gains some characteristics of a spatially-shared resource, because its components can serve multiple requests simultaneously. Another example is a multiprocessor, where the individual CPUs execute in parallel.

Some physical devices combine aspects of temporally and spatially-shared resources. One example is a disk drive. Its storage capacity is spatially-shared (different disk blocks allocated to different processes), while its I/O capacity is temporally-shared: a drive only serves a single I/O request at a time. A mechanism to support idle-time use of a disk drive must consider both these dimensions.

User I/O devices (e.g. keyboard, audio) are a special subcategory of temporally-shared devices, for which idle-time may not be appropriate. Users explicitly control these devices, and the OS should not override these scheduling decisions. However, user I/O devices may share an I/O channel (e.g. USB) with other devices. Idle-time use of the shared channel capacity is possible if mechanisms treat user I/O requests as foreground use.

In some sense, it is possible to model spatially-shared resources as temporally-shared, by treating each allocation unit as a separate resource with an unlimited lease time. For example, instead of viewing disk storage capacity as a single spatially-shared resource, from another perspective each disk block is a separate temporally-shared resource with an infinite lease time. Thus, schedulers for such resources can be similar to schedulers for temporally-shared resource bundles.

Some OS resources are virtualized. A virtualized resource isolates its different users from one another. It also presents each user with a private, virtual resource that may be larger (or otherwise different) than the underlying physical device. Capacity of other resources is required to support a virtualized resource when overcommitted. The main example of a virtualized resource is virtual memory (VM). VM presents each process with an isolated address space, typically larger than the available physical memory, by paging seldom-used parts of address spaces to secondary storage. Virtualized resources themselves are users of other resource capacity. The simple model presented in this chapter is not sufficiently powerful to completely describe such behavior. It will be extended in the remainder of this thesis research to support this capability.

A single-tasking OS does not require resource scheduling. The single existing process can use all resources as needed; unused resources remain idle. This typically leads to a low overall system utilization. A multi-tasking OS can increase system utilization by running multiple processes concurrently. All processes share the CPU and other resources; the OS allocates a share of resource capacity to each process upon request. While this may increase execution time of a single process compared to the dedicated case, it improves overall system utilization. For example, when a compute-bound process and a disk-bound process run concurrently, each of them is able to use resource capacity the other left available. A disk-bound process will spend most of its time in a blocked state waiting for device operations to finish. The compute-bound process can thus utilize the unused CPU time of the disk-bound one. Likewise, the CPU-bound process will not require disk access often, so the disk-bound one can utilize the disk almost fully. The net effect is that the aggregate execution time of the two processes can be lower when they run concurrently using multi-tasking, compared to running them back-to-back on a dedicated machine. In a way, a multi-tasking OS is an example of using idle resource capacities for productive work. However, it fails to actively schedule requests depending on idle capacities.

Various systems try to use idle capacity, either speculatively or non-speculatively. One established technique is microprocessors with support for speculative execution (see Section 5.2.3.3). Such CPUs use idle execution units and memory bandwidth to pre-fetch and speculatively execute likely future instructions. They give total priority to non-speculative processing, and have hardware mechanisms to preempt speculative uses without affecting regular execution. Other hardware mechanisms manage speculative state, and keep it isolated from regular execution until commit time (or discard time, for mispredictions). Remote execution or storage systems (see Section 5.2.3.1) are another example of systems that try to utilize idle capacity.

A system supporting speculation must shield the side effects of speculative tasks from the rest of the system until a prediction has been verified as correct. Similarly, a system to support non-interfering use of idle resources must prioritize resource use. Speculative idle-time use requires both capabilities, based on principles defined in the next section.

2.2 Principles for Speculative Use

Speculative use of idle capacity depends on *prioritization* and *preemptability* between regular and idle-time use, as well as *isolation* of speculative side effects. This section will formally define these three principles as properties of two models. Informally, these principles are:

1. *Prioritization*: Never process idle-time requests while regular requests are waiting for service.

2. *Preemptability*: Immediately preempt active idle-time use to service incoming regular requests. Never preempt regular requests because of idle-time use.

3. *Isolation*: The side effects of a speculative request must remain hidden until they are committed or discarded.

A more formal definition requires a model of resource processing. In this model, a resource is a tuple $\langle R, Q, A, C, p, c \rangle$, where R is the base set of possible requests, $Q \subseteq R$ is a subset of requests waiting for service, $A \subseteq R$ is a subset of active requests currently being serviced. $C \in \Im$ is the capacity of the resource expressed as an integer, $c : R \to \Im$ is the capacity required to service a given request, again expressed as an integer, and $p : R \to \Im$ is a priority assignment, where higher priorities (integers) are served before others.

The *active capacity* $\overline{A}$ of a resource $\langle R, Q, A, C, p, c \rangle$ is defined as $\overline{A} = \sum \{c(a) | a \in A\}$. In addition, the *idle capacity* $\overline{I}$ of such a resource is defined as $\overline{I} = C - \overline{A}$.

A resource supports these operations:

- $enqueue(q)$ adds a new request $q \in R$, $q \notin Q$ to the set of waiting requests: $Q = Q \cap \{q\}$

- $start()$ picks a request $a \in Q$ for service and moves it from the waiting set to the active set: $Q = Q / \{a\}$ and $A = A \cap \{a\}$

- $finish(a)$ removes an active request $a \in A$ from the service set: $A = A / \{a\}$

A resource must satisfy the following axioms:

- A request cannot be both queued and under service: $Q \cup A = \varnothing$
- All possible requests must be satisfiable: $\max\{c(r) | r \in R\} \leq C$
- Active capacity cannot exceed resource capacity: $\overline{A} \leq C$
- Work conservation: $\overline{A} < C \wedge \exists q \in Q : c(q) \leq \bar{I} \Rightarrow start()$

Prioritization and *preemptability* are properties of resource scheduling, and establish idle-time capacity as a distinct service class, described in the model above. Regular performance does not decrease in the presence of idle-time load. However, *prioritization* and *preemptability* alone are not sufficient for speculative use: Visible idle-time state may influence and thus interfere with regular execution.

One example is a state inconsistency caused by an idle-time request that was preempted during an update operation. This could occur both for kernel state (e.g. corrupt device chain) and user state (e.g. interrupted write to a configuration file). Both may obviously affect regular use, as well as other ongoing speculations. Another, less obvious example is where the simple presence of idle-time state (even consistent state) can interfere with other processing (e.g. idle-time lock on a shared file). The *isolation* principle prevents this scenario. It hides *all* speculative state from regular processing, as well as shielding the state of one speculation from that of another. Only after an explicit commit operation does speculative state become visible. *Isolation* is a property of OS processing, and requires a more high-level model, defined in Section 2.2.3 below.

One shortcoming of this model of resource operation is the assumption that resources are *stateless*, i.e. that the capacities $c : R \rightarrow \Im$ of each request are fixed. This is not the case for all resources. One example of *stateful* resources is disk drives. Here, earlier requests can influence the cost of subsequent ones, by moving the disk arm towards (lower cost) or away from (higher cost) the location of the subsequent access. During the remainder of this thesis research, the model should be extended to describe stateful resources.

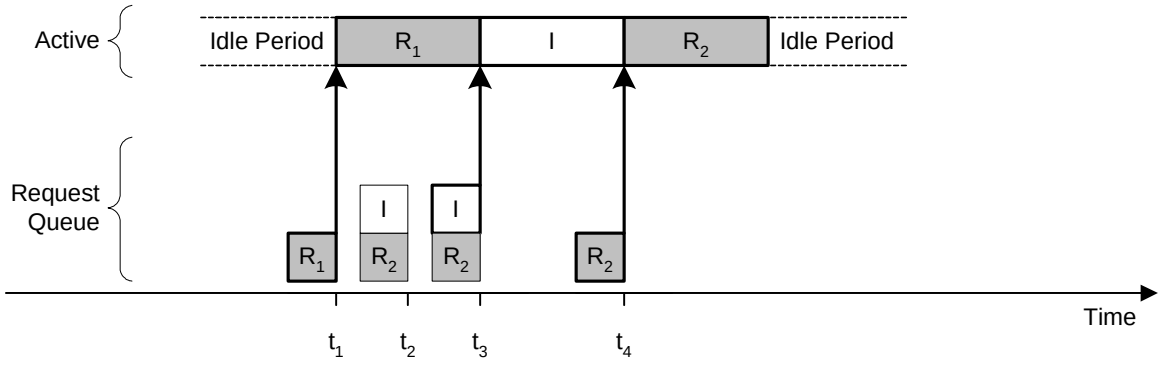

Figure 1: Scheduler for a temporally-shared resource without *prioritization*.

All resources in an OS with speculative use of resources must support *preemptability* and *prioritization*. Without idle-time use, a system has a single bottleneck resource at any time, and the scheduler of that resource controls overall system behavior. With idle-time use, the scenario changes: Its goal is it to fill available capacity with useful work, and keep all resources utilized at all times; thus, all loaded resources become bottlenecks. If some schedulers do not support idle-time use, foreground performance may decrease. Section 4.2 presents experimental results that illustrate how prioritized CPU scheduling is insufficient to provide prioritized network service.

The remainder of this section discusses the operation of resource schedulers and kernel processing, and required mechanisms to extend them for idle-time use.

2.2.1 Prioritization

This section illustrates prioritization for temporally and spatially-shared resources. *Prioritization* is a function of work queue management.

Prioritization: *Never process idle-time requests while regular requests are waiting for service.*

A resource $\langle R, Q, A, C, p, c \rangle$ supports *prioritization* if and only if its $start()$ operation picks a new request $a \in Q$ to start servicing such that $p(a) = \max\{p(q) | q \in Q\}$.

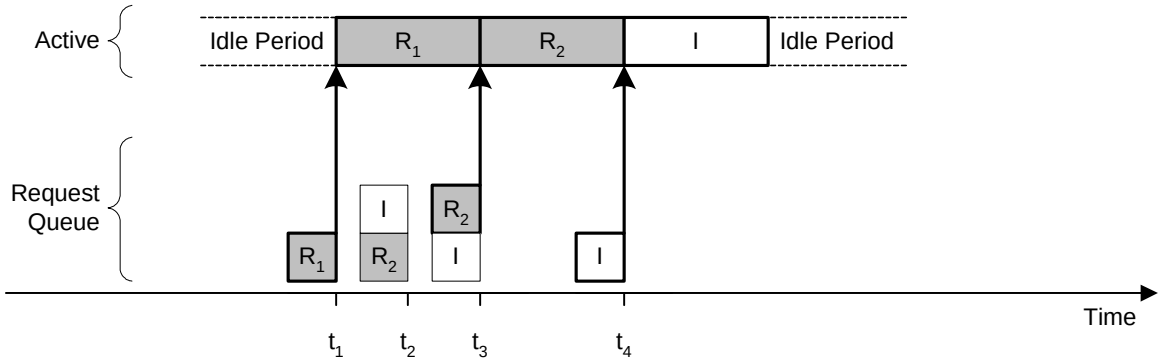

Figure 2: Scheduler for a temporally-shared resource with *prioritization*.

First, Figure 1 illustrates the operation of a FIFO scheduler for temporally-shared resources that does not support *prioritization*. Before time t_1, the resource is idle. At t_1, request regular R_1 arrives and the resource immediately starts processing it, ending the idle period. At t_2, idle-time request I and regular request R_2 arrive and are enqueued. At t_3, processing of R_1 finishes.

Here, the scheduler picks idle-time request I for processing, instead of regular request R_2. Thus, idle-time processing for I delays regular processing (R_2 must wait until t_4 before receiving service), violating the *prioritization* principle. A scheduler with support for prioritization would have picked R_2 over I at t_3 instead (see Figure 2).

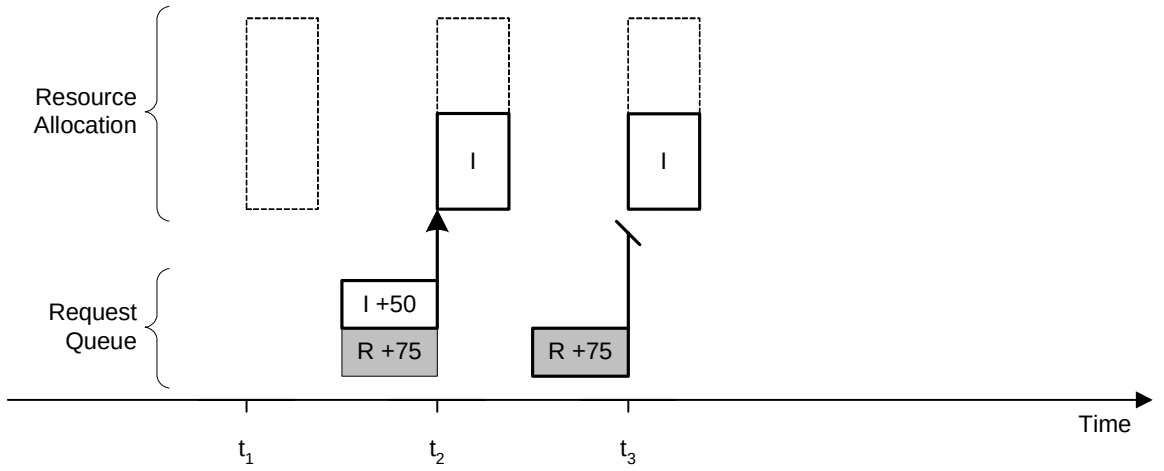

Figure 3: Scheduler for a spatially-shared resource without *prioritization*.

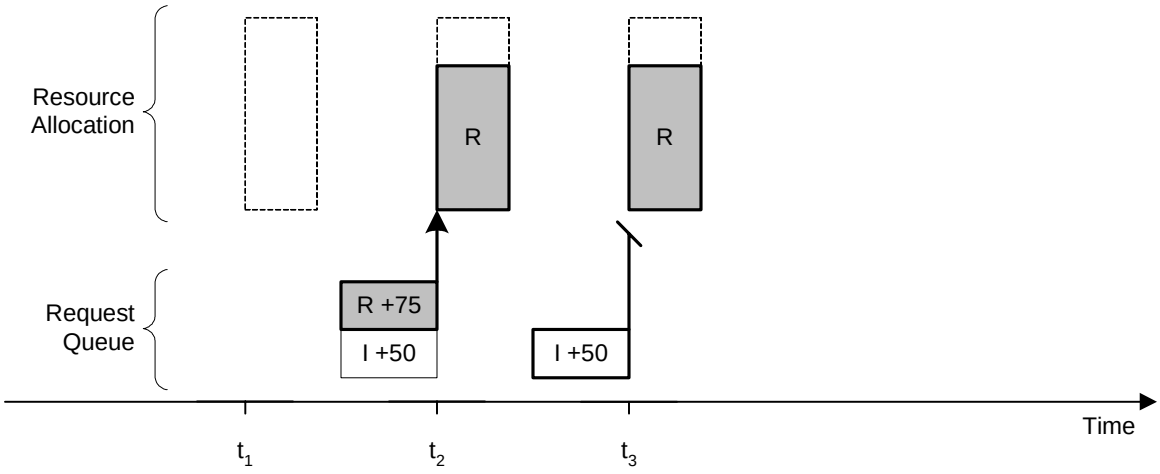

Figure 4: Scheduler for a spatially-shared resource with *prioritization*.

Prioritization is also critical when spatially-shared resources should support non-interfering idle-time use. Figure 3 displays such a scenario for a spatially-shared resource with 100 allocation units with a scheduler that does not support *prioritization*. Here, the resource is completely idle at t_1. At t_2, an idle-time request *I* for 50 units and a regular request *R* for 75 units arrive at the resource. By allocating the capacity for the idle-time request first, the scheduler causes the subsequent allocation of *R* at t_3 to fail due to insufficient capacity. This interferes with regular processing: the process issuing *R* may abort or be delayed.

A spatial scheduler with support for prioritization (see Figure 4) will schedule *R* before *I*. Even though *I* cannot be serviced at t_3 (again due to insufficient resources), this has no impact on regu-

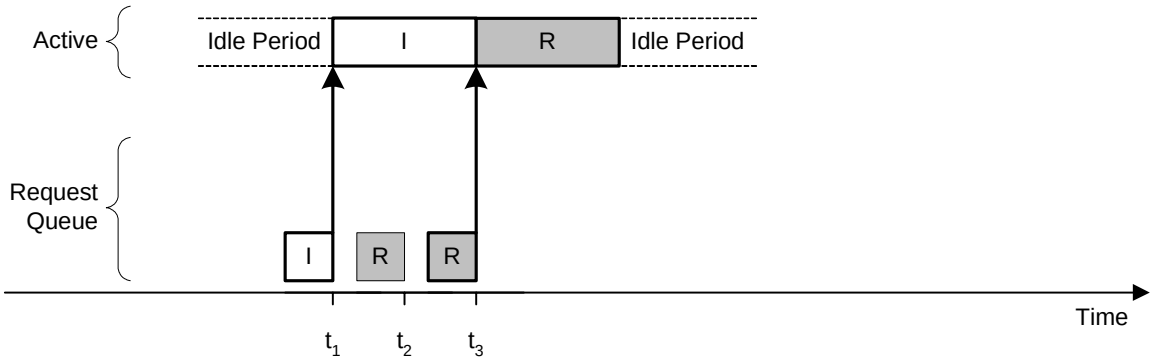

Figure 5: Scheduler for a temporally-shared resource without *preemptability*.

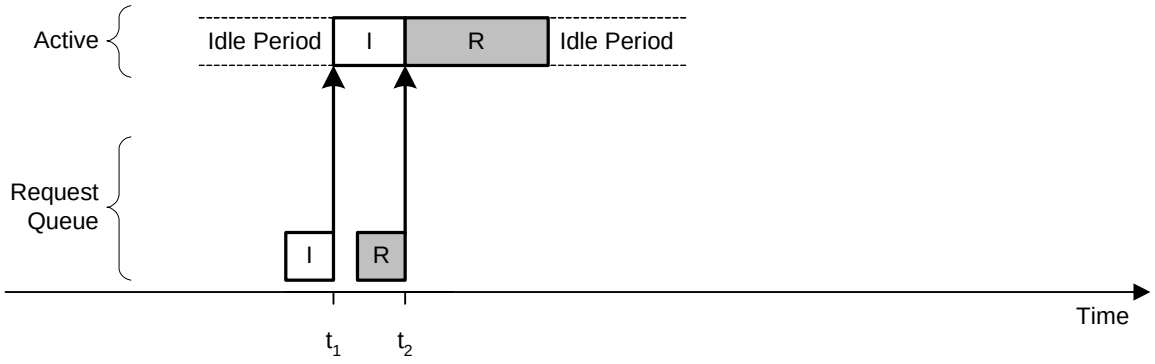

Figure 6: Scheduler for a temporally-shared resource with *preemptability*.

lar use.

2.2.2 Preemptability

Similar to *prioritization*, *preemptability* is the second key principle required for non-interfering idle-time use. This section will describe how schedulers for temporally and spatially-shared resource must operate to support *preemptability*.

Preemptability: *Immediately preempt active idle-time use to service incoming regular requests. Never preempt regular requests because of idle-time use.*

A resource $\langle R, Q, A, C, p, c \rangle$ supports preemptability if and only if it supports prioritization, and during its $enqueue(q)$ operation, if $\bar{I} < c(q)$ it picks a subset $F \subseteq A$ of strictly lower-priority active requests of sufficient capacity such that $\forall f \in F : p(f) < p(q)$ and $\bar{F} + \bar{I} \geq c(p)$, and then preempts these lower priority requests $\forall f \in F : finish(f)$ such that q is immediately started following the $enqueue(q)$ operation.

First, Figure 5 shows an example for a scheduler for a temporally-shared resource. At t_1, it starts processing idle-time request *I*. While *I* is being processed, regular request *R* arrives at t_2. However, the resource continues to process *I*, delaying execution of *R* until t_3, when *I* finishes.

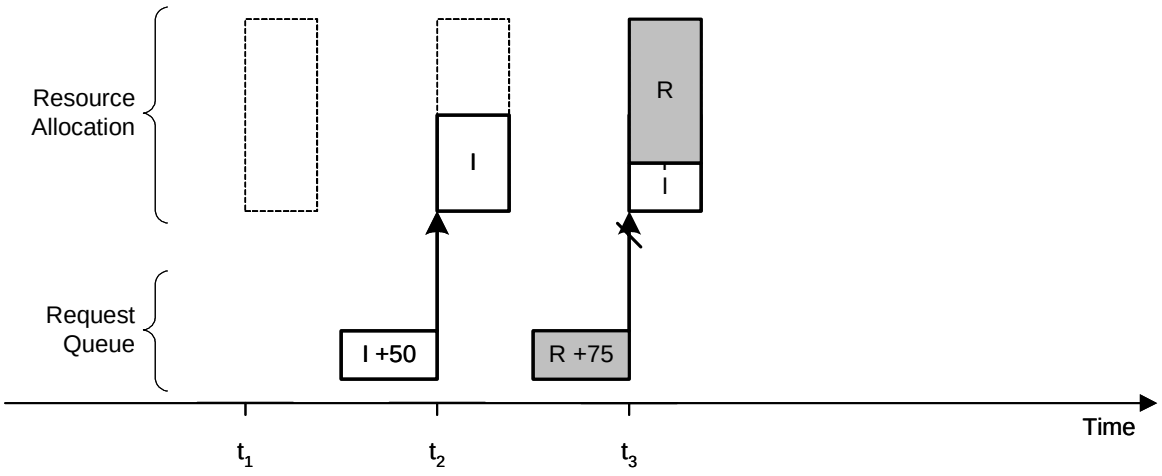

Figure 8: Scheduler for a spatially-shared resource with support for *preemptability*.

The scheduler in this scenario violates the *preemptability* principle, because it does not immediately yield the resource to the newly arriving regular request *R* at t_2. Figure 6 shows how a scheduler with support for preemptability operates in the same scenario: At t_2, it preempts (or aborts) the active request *I*, and immediately starts processing *R* instead. Thus, it causes no delay for regular processing – only for idle-time use, which is the correct behavior.

The next example illustrates how a scheduler for a spatially-shared resource supports *preemptability*, again for a resource with 100 allocation units. At time t_1 in Figure 7, the resource is completely idle. At t_2, idle-time request *I* for 50 units arrives, and the capacity is allocated. When a regular request *R* for 75 units arrives at t_3, it is declined due to lack of available capacity. This violates *preemptability*.

Instead of declining request R, a scheduler with support for preemptability must reclaim (part of) the capacity allocated to idle-time use whenever it has insufficient capacity for an incoming regular request. In Figure 8, the scheduler transparently reclaims 25 of the units allocated to idle-time use, so it can satisfy the regular request *R*.

Preemptability is a function of processing. Note that for most resources, preempting a request and/or switching to another one is not instantaneous. This preemption overhead is the largest challenge faced when supporting idle-time use. Section 3.2.2 below discusses this issue in more detail.

2.2.3 Isolation

The *isolation* principle states that all side effects of speculative execution must remain hidden, until the system has verified whether the speculation was successful. If so, an atomic operation makes the speculative state visible, otherwise it is discarded.

Isolation is the key principle that allows transparent speculative resource use. While *prioritization* and *preemptability* are sufficient to establish an idle-time resource class, they alone do not enable speculative use of that idle capacity. Without *isolation*, speculative state can interfere with regular processing. One example of such interference could happen when speculative execution leaves a system data structure in an inconsistent state, because regular use preempted it during modification of the structure.

The state of an operating system state forms a set S that incorporates all variables and structures visible to processes. Processes modify this state through a sequence of *state operations* $O = (o_1, \ldots, o_n)$, where each operation $o \in O$ is a function $o : S \to S$ that transforms the state. The order of the operations depends on process scheduling, and is not relevant. S_0 is the *initial state* before processing begins, S_1 is the state after the first operation was processed, and S_n is the *final state* after all n operations. The *intermediate state* after k operations is defined as $S_k = o_k(S_{k-1})$. The intermediary states for a given sequence of operations O and a starting state S_0 form a sequence $\overline{S} = (S_0, \ldots, S_n)$. For a pair of operations $(o_x, o_y) \in O \times O$, $o_x < o_y$ if $x < y$.

Given a sequence of operations O and a starting state S_0, speculative idle-time use in this model is a new sequence of *speculative operations* $I = (i_1, \ldots, i_m)$. They operate on an *extended OS state* $S' \supset S$, such that $i_x : S' \to S'$, and require an extended starting state $S'_0 \supset S_0$.

A *combined sequence of operations* is any sequence $C = O \cap I$, which retains the relative order of the operations of O and I, such that for all pairs of operations $(o_x, o_y) \in O \times O$, there exists a pair of operations $(c_r, c_s) \in C \times C$ such that $(o_x = c_r) \wedge (o_y = c_s) \wedge (o_x < c_r \Rightarrow o_y < c_s)$, and similarly for I.

Isolation in the context of the above model is defined as follows:

Isolation: *The side effects of a speculative request must remain hidden until they are committed; or must be discarded.*

Given a sequence of operations O, a starting state S_0, a sequence of speculative operations I and an extended starting state starting state S'_0, an OS supports isolation, if and only if the side effects of all speculative operations are limited to the extended state, such that $\forall i_x \in I : S_x = S_{x-1}$. In other words, all speculative operations $i_x \in I$ may only modify $S'_{x-1} \setminus S_{x-1}$.

This model supports pre-executing a regular operation, in which case a speculative operation $i \in I$ is a duplicate of a regular one ($\exists o \in O : i = o$) and $i < o$ in the combined sequence $C = O \cap I$. In this case, o is speculatively executed earlier in the sequence (as i), but the ef-

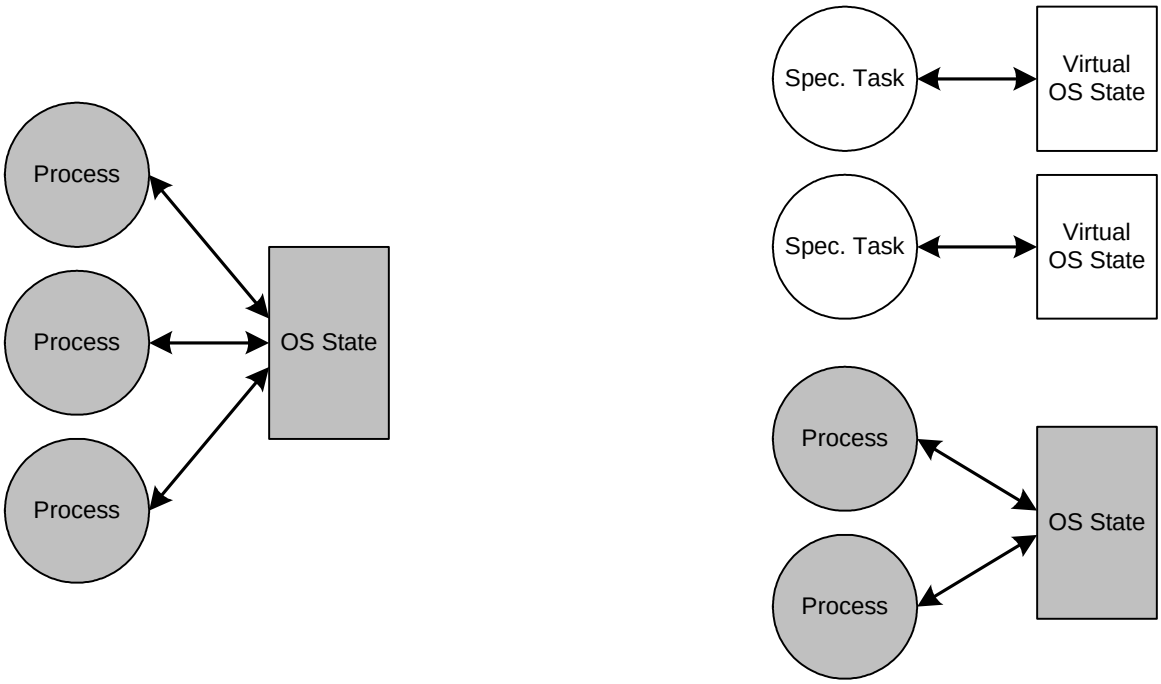

Figure 9: From shared OS state (left diagram) to *virtualized* OS state for speculative tasks (right diagram).

fects of the execution are kept in the extended state. During execution of o, they are moved from the extended into the regular part of the state space.

Isolation virtualizes the OS state: Instead of sharing the OS state between all regular and speculative tasks, each speculative task executes with its own *shadow copy* of the state. Regular processes still access and share the master copy of the state, as before. Figure 9 shows the current sharing situation on the left side, and the virtualized OS state on the right. Virtualized OS state is similar to the concept of virtual memory, where each process executes in a separate address space.

When a speculative task modifies OS state, a private *shadow copy* of the OS state is created (copy-on-write); these shadow copies are invisible to regular processing. The OS updates the shadow copies together with the master copy on regular use. Update conflicts with some shadow

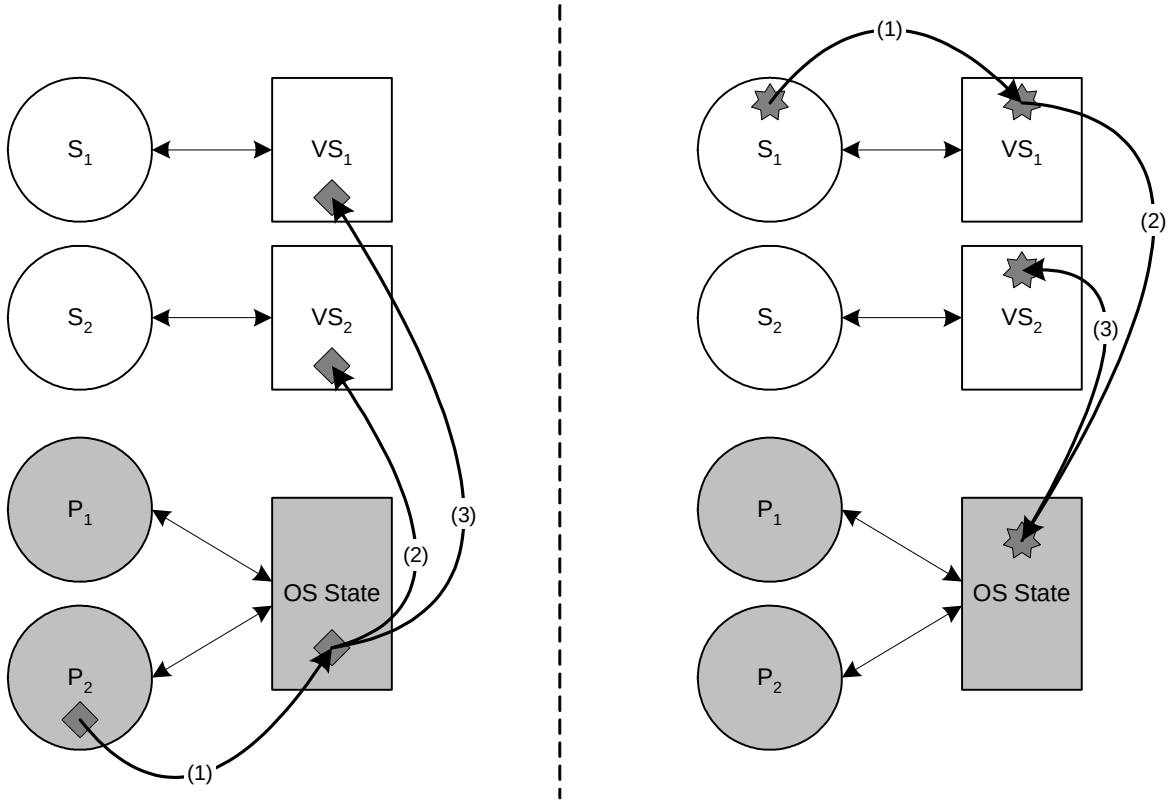

Figure 10: Non-speculative modification to the master OS state (left diagram), and commit operation of speculative state (right diagram).

copies cause speculative tasks that depend on them to abort (or enter recovery, if supported). For successful speculations, speculative state moves from the shadow copies into the master copy through an atomic operation. Remaining shadow copies are also updated during this commit operation.

Figure 10 gives an example of operations on the virtual OS state. The diagram on the left shows how an OS state change (1) by regular process P_1 results in immediate updates to the virtual states VS_1 and VS_2 belonging to speculative tasks S_1 and S_2, in steps (2) and (3).

The right diagram of Figure 10 shows how a speculative modification of VS_1 by S_1 in step (1) is atomically committed back to the master OS state (speculation successful) in step (2) and thus becomes visible to regular processes P_1 and P_2. Furthermore, the commit operation triggers an immediate update (3) to speculative state VS_2, as if a regular process had modified the master OS state.

The completed network stack extensions for idle-time use do not establish *isolation* yet – mechanisms to guarantee *isolation* will be investigated as part of the proposed thesis research (see Chapter 6).

2.3 Application of the Model to OS Extensions

One of the main tasks of an OS is resource management. To support a wide variety of applications, a general-purpose OS employs simple and predictable resource schedulers, trying to provide fair service to all users. Section 2.1 above presented three key principles for schedulers with support for non-interfering idle-time: *prioritization*, *preemptability,* and *isolation*. This section focuses on OS extensions to support these principles, as well as additional mechanisms required for effective speculative idle-time use.

2.3.1 Prioritization

Most current OS schedulers do not support prioritized access. One exception is typically the CPU scheduler, since the CPU has traditionally been the bottleneck resource of a system. Optimizing CPU allocation was thus an important factor to maximize overall system utilization for a particular workload.

UNIX systems use a *multilevel feedback queue*, a variant of a *round-robin* (RR) scheduler. It favors bursty processes, which do not fully utilize their allocated CPU quantum by raising their priority over time, and punishes compute-bound processes by lowering theirs. Most I/O-bound processes are bursty – they block during device operations – and thus achieve high CPU priorities. Commonly, the CPU scheduler offers the user processes some degree of control over their priorities. Non-privileged processes may lower their priority from the default, while increasing the priority is restricted to privileged processes. However, monopolizing the CPU through this mechanism is impossible; it merely adjusts the share of processing time and does not establish total priority.

Simple *first-in-first-out* (FIFO) schedulers organize access to most other resources, such as disk and network devices. While FIFOs by themselves do not assure fairness, they can do so in combination with a fairness-enforcing CPU scheduler (because a process cannot issue any resource requests without a CPU to run on). These other resource schedulers typically do not allow processes to influence their scheduling decisions.

Both FIFO and RR schedulers do not satisfy the *prioritization* principle; all requests receive equal service. Even a multilevel feedback queue only allows adjustment of shares, and does not prevent starvation. To support non-interfering idle-time use, resource schedulers must instead replace such scheduling disciplines with priority queue with two service classes, for regular and idle-time requests.

The CPU scheduler on many *POSIX*-compliant systems [POSIX 1993] already offers this capability. The *POSIX* CPU scheduler has three distinct priority classes for processes (real-time, regular and idle-time), each managed by its own multilevel-feedback queue. Processes in higher classes preempt any lower-class ones. Consequently, processes running under the *POSIX* idle-time scheduling class will not receive any CPU time while processes in higher classes are runnable. Starvation of lower-class processes occurs when higher-class load increases to saturation. Thus, the *POSIX* scheduler satisfies the *prioritization* principle. Experiments with the *POSIX* scheduler show that it can isolate regular use from idle-time requests to within 1%. Some other experimental CPU schedulers also support an idle-time processing class [FORD 1996] explicitly.

Other resources, most importantly disk and network, do not often offer different levels of service. Chapter 4 below presents an extension to the network stack that replaces its many FIFOs with suitable priority queues. A similar extension to the disk scheduler is under investigation.

2.3.2 Preemptability

Service *prioritization* alone, however, is not sufficient to guarantee non-interfering use of idle capacity. *Preemptability* is a second key requirement. Without it, a currently executing idle-time request would delay a newly arriving regular one, because the scheduler would let it run to completion – a form of *priority inversion* [LAMPSON 1980]. Instead, the scheduler must immediately abort or suspend idle-time use whenever a new regular request arrives. Thus, the priority queues proposed in the previous chapter to replace FIFOs must support *preemptability*.

Preemption cost is *the* key factor currently limiting the deployment of idle-time use. Mechanisms to minimize it are critical. Resources that frequently switch between different requests often have hardware support to minimize this overhead. One example is CPUs, which typically offer instructions to save and restore the register set. However, most other resources do not have hardware support for preemption. For example, interrupting a disk output request in the middle of a device write operation is often impossible, because disk controllers normally do not support preemption (though some are proposed [SPRUNT 1988]). This is an example of *physical* priority inversion. *Logical* priority inversion occurs whenever preempting a lower-priority operation is impossible, because a shared resource would remain in an inconsistent state.

Another factor that sets CPUs apart from other resources is that they typically serve a process for longer than a single resource request (i.e. a single instruction). A *time quantum* limits the maximum amount of time a CPU will service each process. It also minimizes the preemption overhead, by reducing the frequency of preemptions relative to instructions.

The idea of a time quantum may also apply to other resources with a high preemption cost. Disk drives are one example. A disk I/O request is typically uninterruptible, so the preemption cost of disk I/O is high (within the same order of magnitude as the service time). If the disk scheduler allocates itself to a service class for a certain time, preemption overhead decreases. For example, with a time quantum of 1 second for regular use, the disk scheduler would only switch to idle-

time use once every second, instead of possibly after each completed request (i.e. each few milliseconds). The drawback is of course that idle-time requests incur an additional delay.

Schedulers for spatially-shared resources must thus allow transparent use of idle allocation units. Resource capacity allocated for idle-time use must be reclaimable whenever it is required to fulfill a regular resource request. One example is allocation of disk space to processes. A simple *first-come-first-serve* scheduler prohibits the non-interfering use of idle resource units, because when it allocates available resource units for idle-time use, they become unavailable for regular use. A subsequent (large) request for allocation units from a regular process may thus fail, because idle-time use was not preempted. Section 2.2.2 above presented an example.

This has interesting consequences for users of idle-time spatial capacity, as previously allocated capacities can disappear upon reallocation to satisfy a regular request. One example of such a case is reclaiming disk blocks used for idle-time storage on a full disk when a regular process starts writing. This does not occur under the regular service model, where a process can rely on allocated resources to be available until it explicitly returns them. Consequently, applications using idle spatially-shared resource capacity must adapt to such situations, or at least gracefully abort. This is similar to mechanisms for establishing *preemptability* (see Section 2.3.2 above), where speculative state can disappear due to regular use.

2.3.3 Isolation

Isolation is the principle of hiding side effects of idle-time use from regular resource users. One such side effect is a decrease in performance, which the *prioritization* and *preemptability* principles already concentrate on. *Isolation* focuses on all other user-perceivable aspects of idle-time support.

Generally, the execution environment observed by regular processes in the presence of idle-time use must be identical to a scenario in which idle-time use is absent from the system. For example, using the regular file system to store idle-time data (even if ample disk space is available) is problematic, since the files would then be visible to regular processes, and could interfere with regular processing.

Ideally, the system would use idle resources to provide an isolated virtual execution environment, in which speculative tasks execute. This is similar to *jail* sandboxes [KAMP 2000] supported by some Unix variants that restrict the set of system calls that super-user processes can execute, to improve system security.

Isolation requires a different set of capabilities from the sandbox environment; namely, a virtualization of OS state. Each speculation would execute in a separate sandbox, completely separating their side effects from one another, and from the regular OS state. Only side effects of correct speculations become visible in the regular system.

Complete *isolation* requires the elimination of all shared state between sandboxes, a difficult goal, similar to the elimination of all covert channels between two parties (each piece of shared state presents a potential information leak). However, the current idle-time network subsystem does not yet provide *isolation*, but is already effective in shielding regular use from idle-time traffic (to within 1-2% of throughput, see Section 4.5).

Thus, while complete *isolation* is theoretically required to eliminate all possibilities for interference, it may be sufficient to virtualize a limited subset of OS state to establish *isolation* in most practical scenarios. A detailed investigation of mechanism to support *isolation* is part of the proposed thesis research.

2.4 Integrated Scheduling

Another challenge with speculative processing is optimization of idle-time use. Starting speculative processing of a task that requires multiple resources is problematic when some of them do not currently have idle capacities.

For example, assume idle-time process *A* requires some idle CPU and disk capacity, and idle-time task *B* requires idle CPU capacity and idle network bandwidth. Assume the disk is fully loaded with regular requests. Speculatively starting *A* in this scenario is not beneficial, since its completion is unlikely due to unavailable disk capacity. However, *B* may finish successfully, since its required resources are not fully loaded. In this scenario, picking *B* instead of *A* for speculative execution has a higher chance of filling idle capacity with useful work.

Thus, speculative idle-time use could benefit from integrated scheduling, where initiation of candidate idle-time tasks depends on the available idle capacity at each point in time. While such a mechanism is not required for correct speculative use, in can increase its effectiveness, by decreasing partial processing due to unavailable idle-time capacity.

Integrated scheduling requires candidate speculative tasks to disclose their planned resource use. While this is unacceptable for regular processing, speculative idle-time tasks must already include mechanisms to gracefully handle disappearing or corrupt speculative state, which regular process need not. Thus, resource usage hints are not the only difference between regular and idle-time processing. Furthermore, they only enable optimized speculative processing – they are not a required component.

The proposed research for the remainder of this thesis includes a proof-of-concept design for such a mechanism for integrated idle-time scheduling.

3 Discussion

This chapter discusses implications of speculative idle-time use. First, it investigates which applications could benefit from the availability of idle-time use. Then, it examines several potential issues with the proposed model.

3.1 Applications and Benefits

Speculative use of idle resource capacity achieves performance improvements through *latency hiding*. For temporally-shared resources (e.g. network bandwidth), the OS speculatively schedules probable future resource requests during idle-times. Thus, speculative requests execute *before* a process issues them, and the OS caches the results. For correct predictions, this approach avoids much of the latency associated with processing the request in a regular fashion – to the process, it appears to execute faster than it would on an unmodified system.

Idle, temporally-shared capacity permits the kernel to prefetch remote data. If the peer also supports idle-time use, pre-sending data for speculative storage is possible. This hides the execution delay of the operation for correct predictions.

Unused capacity of spatially-shared storage resources can also hide latencies. In this case, however, the performance improvement is a result of speculatively storing information that is costly to obtain. Any data item whose access involves a delay is a candidate for speculative storage.

Caches are the main application of idle spatially-shared capacity. A system's storage facilities form a hierarchy according to their access delays. Capacity at higher (faster) levels is typically costly and smaller, while capacity of lower (slower) levels is large. Caching data from lower levels of the hierarchy at higher levels improves performance, by reducing access delays. Swapping is the inverse of caching: It pushes data from higher into lower levels, to simulate larger virtual capacity at the higher level.

One major issue in caching is cache size. Caches reduce the available storage capacity for regular use. Excessively large caches can force the system to swap part of the working set of a process to lower storage hierarchies, and thus decrease overall performance. On the other hand, an extremely small cache size limits the hit rate, and thus the obtainable benefit. Ideally, a cache should

always use any idle capacity available at a certain level in the storage hierarchy, and grow or shrink according to the capacity use.

Using idle capacities for caching (instead of regular capacities, as is current practice) solves this issue. Aggressive caches can make maximal use of idle available capacities. The system will automatically shrink the cache and reclaim capacity for regular users. Optionally, the cache could participate in the reclaiming, by releasing infrequently used capacities to the system.

3.1.1 Network Service

One important group of applications focuses on improving user-perceived network service. Speculation can reduce both connection-open latencies and transmission times. The key idea here is to trade idle current bandwidth for a possible future latency reduction [TOUCH 1992]. Ideally, in a network with support for idle-time use, lower-priority packet processing will only occur when resources would have been idle in the absence of such traffic.

One well-known application of this idea is *web prefetching* [PADMANABHAN 1996]. It would greatly benefit from the availability of idle resources use: First, transmitting prefetched data using idle network resources completely shields regular network users from its presence. Consequently, the prefetcher no longer needs to limit its aggressiveness to prevent monopolizing the network bandwidth. Second, larger caches become possible by using idle storage space for the prefetched data.

Even without using idle storage space for caching prefetched information, current idle bandwidth can reduce future network latency, by *prefetching the means* [COHEN 2000]. This scheme does not prefetch any data, but instead pre-negotiates the means to transfer future data, such as pre-opening TCP connections, or pre-resolving DNS names. Speculative execution of these operations creates very little state compared to caching the data, so idle-time access to storage capacity may not be necessary.

Another technique described in [COHEN 2000] is *pre-warming* a TCP connection, by pre-sending a small amount of throwaway data over a pre-opened connection. This may pre-establish additional state in the end system and router caches, and thus further improve performance. Using idle network capacity for this purpose improves on the original proposal, by permitting a host to pre-

send probe packets without interfering with regular traffic. Thus, larger amount data can be present, allowing TCP to better estimate the RTT and congestion window for the connection. This may result in a better network throughput for later, non-speculative transmissions over the pre-warmed connection.

Pre-execution of two additional network operations during idle time (to "prefetch the means") could be effective. One is to speculatively initiate path MTU discoveries [MOGUL 1990] to likely future hosts. A PMTU discovery can add one or more round-trip-times (RTTs) to the connection-establishment delay. Hosts supporting PMTU discovery implicitly do so whenever a connection is opened (speculative or regular). Another operation is speculatively initiating IPsec key negotiations (IKE) [HARKINS 1998] with likely future peers, which also could also save several RTTs [HARKINS 1998]. With current proposals for opportunistic encryption [RICHARDSON 2001], IPsec negotiations may become much more frequent. Speculatively executing an IKE exchange during idle-time can reduce the user-perceived connection-open delays for successful predictions.

All previous applications required idle bandwidth to operate. However, even without idle bandwidth, a server system can use idle local resources to increase its network performance. Most servers (e.g., NFS, FTP and web) incur *packetization overhead* for each requested object by reading it from the disk (or disk buffer) and splitting it up into a packet chain before transmission. For static objects, caching the packets in idle storage[1] can reduce this overhead for repeated accesses [LEVER 2000]. As packetization overhead increases – for example, with IPsec processing – the potential for improvement becomes even greater.

Note that idle-time use of the network requires router support. However, the new service model is a simple extension of the current Internet service model, where routers (and hosts) treat packets equally according to a *best-effort* discipline [CLARK 1988]. Idle-time use does not change this fundamental model: The network may still reorder, drop, or duplicate packets. Idle-time networking is strictly a per-hop function of giving higher processing preference to certain packets. Chapter 4 discusses idle-time networking in more detail.

[1] Jon Postel. Private Communication. 1998.

3.1.2 Disk Service

All the applications for speculative use of idle resources described above mainly use idle network bandwidth, and to a lesser degree memory and CPU. Speculatively using idle disk capacity (both I/O bandwidth and disk space) has also the potential to improve system performance.

Most file systems already use *read-ahead* techniques to improve input performance [PATTERSON 1995]. A straightforward improvement would be to execute read-ahead prefetches (with are speculative by nature) with idle disk resources, and move the disk buffer caching them into unused physical memory. Prefetches would then no longer interfere with regular read operations, and large idle-memory disk buffer sizes would not limit memory availability for regular uses.

Another technique that would benefit from the availability of idle-time disk service is *disk block replication* [AKYUREK 1995]. This approach spreads replicas of frequently used disk blocks out over the entire disk. In effect, it moves the data closer to the disk arm, reducing arm movement and thus access times. One drawback of this scheme is that replicas decrease available disk space, and replica management uses disk bandwidth. Using idle disk space and bandwidth would mitigate these shortcomings.

The inverse of the previous scheme is to speculatively move the disk arm near spots of likely future accesses during idle time [KING 1990][MUMOLO 1999]. Unlike the disk block replicators, this approach does not transfer or cache any data, and the memory and disk subsystems need not support use of idle capacity. The drawback is that replication can have better prediction rates, because the likelihood of the arm being near the data increases with the replication factor.

Prefetching and caching file system meta-data is another technique to increase file system performance [MOLANO 1998]. As with many caches, choosing the correct size is critical for system performance. Using idle memory for the cache solves this problem, as the cache will automatically shrink as memory use by regular processes increases.

Many file systems must periodically checked for inconsistencies due to loss of power, etc. As part of an improvement to the *fast file system*, McKusick proposes a file system checker that continually monitors and fixes file systems for inconsistencies [MCKUSICK 1999]. Such a process would be a prime candidate for execution with idle CPU and disk resources. Similarly, adaptive techniques to optimize performance of *log-structured file systems* require periodic reorganization of

disk contents [MATTHEWS 1997]. Executing these tasks with idle resources could improve overall system performance by minimizing interference with regular use.

3.1.3 Application-Layer Uses

Application-layer uses for idle resources also exist. One such application is an improved *nice* utility to schedule periodic optional maintenance tasks in a system. Examples of such tasks are checking for viruses, defragmenting the file system, and auditing system security.

Non-optional system management tasks, typically run through *cron* [REZNICK 1993], also benefit from using idle-time resources. *Cron* runs specified tasks at certain times. Simply running *cron* using idle resources is not sufficient, because regular resource use could then prevent scheduled *cron* tasks to miss deadlines. *Deadline-bounded backgrounding* is an extension to *cron* that allows scheduling of tasks during time intervals. Many *cron* tasks are maintenance operations that do not need to run at fixed times, as long the system could guarantee they run within a certain time interval. For example, instead of scheduling a regular disk cleanup explicitly at 2am (because resources tend to be idle at that time of the day), the system would schedule an idle-time disk cleanup anytime between 1-2am. If the task did not run by 2am due to unavailable idle resources, it would then execute using resources regularly.

Under this model, foreground processing can be isolated from the presence of periodic background tasks by pushing those into idle periods before a deadline. If insufficient idle capacity is available before the deadline, the system switches a *cron* task over to foreground execution. Thus, in the fallback case, operation is similar to regular *cron*, while still isolating regular use when sufficient idle capacities are available.

3.2 Challenges

Several issues affect the feasibility and effectiveness of a mechanism to use idle resources speculatively. The most obvious is the distribution of idle times for a system's resources for a given workload. If idle capacities are rare (meaning resources are mostly fully utilized), the chance for performance improvements is low. The same is true if idle times are of short duration (before regular use continues). As mentioned in Chapter 1, however, ample idle capacity is usually available.

The remainder of this section discusses other issues affecting the ability to use idle resources speculatively in more detail.

3.2.1 Inter-Resource Interference

Most computer systems contain multiple resources, usually at least a CPU and some memory. While *prioritization* and *preemptability* extensions to all resource schedulers are necessary to establish non-interfering idle-time use, they alone are not sufficient. In a multi-resource system, interactions between resource schedulers (even prioritized, preempted ones) can cause interferences between regular and idle-time use.

Inter-resource interference occurs because processing inside most kernels is an intricate combination of queueing, timeouts, interrupts, and blocking and resuming processes. The upper half of a typical kernel implements the system call API, the lower half hardware-dependent drivers. The upper and lower halves communicate through a set of work queues. The rationale behind this processing scheme is optimization of resource utilization, not prioritized use. The CPU scheduler controls process access to the top half – processes that are blocked cannot issue resource requests. The lower half, however, executes asynchronously, driven by device interrupts and signals. Interrupt processing at the lower half has priority, and always preempts the upper half (or user space execution).

For example, when a process issues a resource request via a system call, the kernel simply enqueues it in the work queue of the appropriate resource and signals the resource scheduler to start processing. The resource scheduler immediately dequeues the request, hands it to the resource for processing, and then relinquishes control to the upper half. The upper half blocks the calling process until the lower half signals completion, and the CPU scheduler switches to another runnable process. When the resource is finished processing the request, it will raise an interrupt. The CPU starts executing the corresponding interrupt handler, preempting all other processing. The interrupt handler for a resource transfers any response data from the hardware to the input buffer, if the request returned any, and signals to the upper half that the request processing is complete.

This processing scheme raises several issues: First, consider the case of a finishing idle-time request, when the CPU is currently busy executing a regular process. In that case, interrupt process-

ing on behalf of an idle-time request preempts regular use of the CPU, and violates the *preemptability* principle: idle-time processing for one resource interrupts regular processing on another.

Second, most interrupt handlers do not relinquish control of the CPU at this point. Instead, they check whether the work queue holds additional requests, and start processing them. The rationale for this scheme was optimized resource utilization (waiting request are started without additional context switches). This can amplify the problem: the newly starting idle-time requests could again interrupt regular processing in the future.

In some cases, this processing scheme can also violate the *prioritization* principle. In the example above, assume the interrupted process was about to issue its own requests to the resource. The delay in relinquishing the CPU prevented it from issuing these regular requests, and resource scheduler fills this false idle capacity with more idle-time requests. Not only was the generation of regular resource requests delayed, they may also incur a preemption overhead, because the resource must switch from idle-time to regular use. Here, a problem with one resource scheduler creates ripples which cause additional interference for other resources.

Thus, resources need to cooperate to establish non-interference. The proof-of-concept design of networking extensions for idle time use (see Chapter 4) controls some inter-resource interference by controlling transmissions at the network layer.

3.2.2 Preemption Overhead

The largest challenge faced to support non-interfering idle-time use of resource capacity is *preemption overhead*. For most resources, aborting one request and switching to another involves some amount of work, and thus incurs a delay. For example, switching the CPU from one process to another requires a *context switch* (swap of the register set), before execution can continue. Even worse, some resources do not support preemption. In that case, eligible requests must wait until the currently executing one finishes, even if their higher priority should allow them immediate access to the resource. *Direct-memory-access* (DMA) devices (such as disk drives or network interfaces), which move data to and from memory without involving the CPU, fall into this category, because DMA transfers are usually non-preemptable. Delay due to non-preemptable service is another kind of *preemption overhead*.

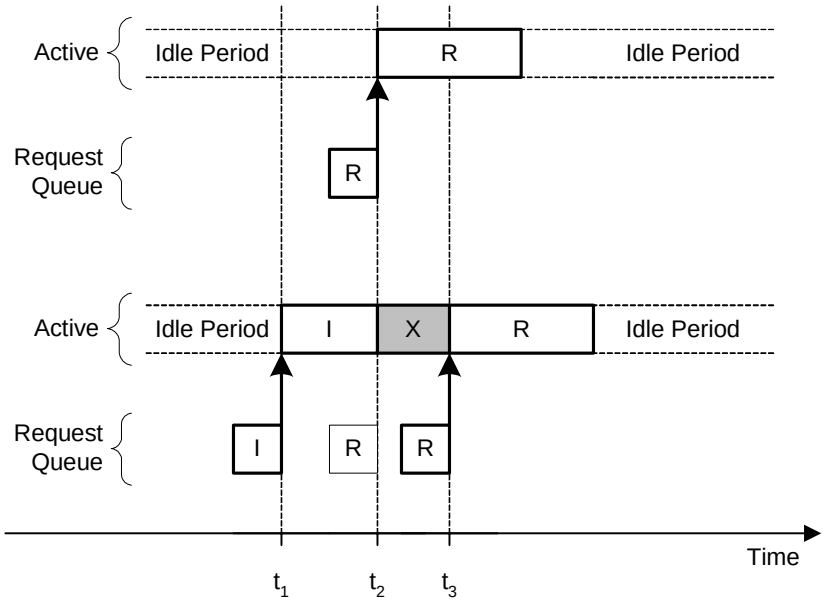

Figure 11: Preemption cost due to idle-time use (bottom diagram), compared against the basic case (top diagram).

Figure 11 illustrates this cost for a regular request *R* by comparing scheduling without idle-time use (top diagram) and in the presence of an idle-time request *I* (lower diagram). In the lower diagram, idle-time request *I* starts processing at t_1. At t_2, regular request *R* arrives at the resource. It immediately starts preempting *I*, but aborting request *I* incurs a preemption cost (depicted by *X*). Thus, *R* cannot start processing until t_3, whereas it could start as early as t_2 in the absence of *I* (top diagram).

While zero-cost preemption is feasible on CPUs (the designers have full control over the system hardware), it is almost impossible to achieve for an OS, simply because most hardware does not support it. Each time a resource switches from idle-time to regular use, a preemption cost incurs. Without idle-time use, the resource would have been unused, ready to immediately serve the new request. Thus, regular performance decreases.

Without zero-cost preemption overhead, non-interfering idle-time use is impossible. Thus, the key issue becomes minimizing preemption costs. The hope is that a larger performance increase due to the ability to speculatively use idle resources compensates for a small performance decrease due to preemption overheads, in the majority of cases. The pathological case is a workload with unlimited speculative load, where a regular request immediately follows each speculative request

as it starts executing, and all speculations fail. Thus, each request incurs the preemption overhead, and speculation never results in a performance increase.

While regular performance will truly suffer in the pathological case, performance for more realistic workloads may still be acceptable. Most resource use tends to be bursty; a number of back-to-back regular requests will interrupt idle-time use. Thus, the whole request chain only incurs a single extra preemption cost. Additionally, preemption cost varies greatly for different resources. A CPU context switch typically takes a few nanoseconds, while a disk request may take several milliseconds. It may be possible to disable speculative use for resources for which the aggregate preemption cost (i.e. the impact on regular performance) becomes too great, but continue to allow it for other resources.

Additionally, the resource can eliminate the preemption overhead in special cases where it can predict the next occurrence of a regular request. In such a case, it can stop idle-time processing ahead of time, to push the preemption overhead into the idle period. To an arriving regular request, the resource seems idle, and no preemption cost delays it. For certain periodic workloads, or resources that require prior reservation, such a scheme is possible.

3.2.3 Cache Pollution vs. Pre-Load Effect

Another issue with speculative use of idle capacities may be *cache pollution*: Many hardware and software caches exist in a typical computer system to speed up operation. They replicate frequently/recently used data in faster storage space, to reduce retrieval latency on future use. Because cache sizes are limited, speculative operations may create cache state that removes entries created by regular requests. This may increase the delay of a future regular resource request.

Thus, regular resource use can be slower due to the presence of speculation in the system – which violates the *isolation* principle. To prevent this effect, it may be necessary to disable caching during speculative processing [DOUGAN 1999] when regular caches grow large enough to prevent speculative storage. This will slow down idle-time use, but because it is not critical by definition, this performance decrease may be acceptable.

However, in other scenarios, leaving caching enabled during idle-time use may *increase* future regular requests. Some studies indicate that speculative use can *pre-load* caches with useful data

for later regular uses [KWAK 1999][PIERCE 1994]. In that case, the regular processing benefits from cached speculative state.

Thus, it can be beneficial to violate the *isolation* principle, and to allow pre-loading of caches. A mechanism to determine whether a speculation is likely to result in cache pollution or a pre-load could increase system performance.

3.2.4 Speculative Workload Generation

Effectiveness (hit-rate) of the speculatively executed requests is another issue with speculative resource use. A higher rate of correct predictions utilizes idle capacity better with useful work, improving system performance. Thus, generation of candidate requests for speculative execution with idle capacities is critical. Ideally, an OS would automatically identify probable future resource requests, and speculatively execute them with idle resources. That way, speculative execution is transparent to processes; application modifications to take advantage of the speculative execution facility are not required.

Unlike the instruction stream processed by the CPU, the stream of resource requests served by the OS is not static. The system cannot simply look ahead at the next requests in the stream, as a CPU can. In addition, a branch in the CPU's instruction stream results in only two possible future paths of execution, while the set of possible resource requests is typically much larger. Furthermore, conditional branch or indirect jump instructions are the only events that introduce ambiguity into an instruction stream, and prediction methods can specifically target those cases. In a given resource request stream, any request introduces ambiguity – the next request is unknown until issued.

Automatically deducing good candidate requests for speculative execution is thus a much harder problem than identifying likely future instructions in the instruction stream. In some sense, it is comparable to predicting branches in self-modifying code. Simple general strategies, such as randomly picking one branch path, which on average is correct half the time for the instruction stream, are thus not effective. This proposal thus assumes that processes explicitly generate resource requests for speculative execution. The rationale behind this scheme is that processes should have better information for an informed decision on their likely next resource requests than the OS.

However, automatic deduction of candidate requests is effective for a limited subset of speculative idle-time use (i.e. prefetching disk blocks) [CHANG 1999]. It may be possible to extend this scheme to other common scenarios, to improve performance for applications that do not explicitly support idle-time use.

3.2.5 Miscellaneous Issues

Priority inversion [LAMPSON 1980] happens when a higher-priority process must wait for the completion of a lower-priority one that holds a required resource. The higher-priority process could block indefinitely while a third intermediate-priority process prevents the lower-priority job from finishing its resource use. However, only two priority classes (regular and idle-time) exist in this proposal, so priority inversion cannot occur for temporally-shared resources. For spatially-shared resource, the proposed system aborts resource use by the lower-priority process, and reallocates the capacity, avoiding priority inversion.

Speculative execution happens in a much more volatile environment than regular execution. The OS may delay speculative requests indefinitely, and it may abort them at any time. Furthermore, speculatively stored data may disappear when a regular user reclaims the space. This means that users of idle-time resource capacities (processes in the process-driven approach, the kernel itself in the kernel-driven approach) must gracefully handle a wider variety of error conditions than regular users.

Finally, increased power consumption and mechanical wear-and-tear may be issues encountered with using idle-time speculatively. Because idle-time use will ideally constantly utilize all resources, the power requirements of a system, as well as mechanical wear-and-tear on moving parts (e.g. disk drives) may be increased. This is especially important for mobile devices, which operate under stricter parameters than their stationary counterparts do.

4 Idle-Time Networking

A network service for idle-time use must follow the *prioritization*, *preemptability*, and *isolation* principles defined above. Thus, it should treat regular and idle-time packets differently; transmitting packets queued at a router in order of decreasing priority, and dropping lower-priority packets from a full queue when higher-priority packets arrive. This chapter focuses on extending the end system for such a network model, and assumes network support is present.

After defining the ITN model in the next section, experimental results presented in Section 4.2 show that current OS mechanisms are not effective in establishing such different service levels for network traffic. The event-driven, asynchronous nature of network stack processing interferes with attempts to use CPU-scheduler-based mechanisms as offered by current systems to control network send behavior.

Observations gained during an analysis of network stack operation form the basis of a design to support ITN, comprising of a simple set of extensions to the current BSD network stack. These modifications concentrate on the sender's network layer; transport protocols and socket API remain unchanged. Section 4.4 describes these extensions in more detail.

The final section of this chapter evaluates a proof-of-concept implementation of these mechanisms in the BSD network stack. Experimental results suggest that the proposed extensions are effective in establishing ITN service: Higher-priority senders can achieve 97-99% of the throughput in the basic case, effectively isolating them from the presence of concurrent lower-priority traffic.

4.1 Idle-Time Network Model

The idle-time networking (ITN) model used throughout this chapter is a simple extension of the current Internet service model, where routers (and hosts) treat packets equally according to a *best-effort* discipline [CLARK 1988]. Note that ITN does not change this fundamental principle: The network may still reorder, drop, or duplicate packets. ITN is strictly a per-hop function of giving higher processing preference to certain packets. In the ITN model, packets belong to either of two classes: *foreground* (FG) or idle-time *background* (BG) traffic. Ideally, BG packet processing will

only occur when resources would have been idle in its absence. Under real conditions (non-interruptible packet transmissions, non-zero-cost queue operations), complete *isolation* of FG traffic is difficult to achieve.

Router support for ITN is straightforward: A router will always forward all FG packets in its queue before any BG packets, and it will drop BG packets from a full queue to make room for arriving FG ones. In other words, ITN replaces a router's FIFO queue with a two-layer priority queue. FG packets continue to experience best-effort service, while BG packets see sub-best-effort (i.e. *least-effort*) service. This is not a new idea: The original IP specification [POSTEL 1981] contains support for a precedence field in the datagram header to indicate dropping and forwarding priorities.

More recently, some of the proposed extensions to support differentiated services in the Internet [BLAKE 1998] are similar to the idea of ITN: *Expedited forwarding* (EF) [JACOBSON 1999] redefines a value in the IP *type-of-service* field to mark some packets with a higher forwarding priority. It also suggests configuring a rate limit for expedited packets, in order to prevent starvation of lower-priority traffic. While EF focuses on providing virtual leased lines with a fraction of the capacity of the physical link, in the absence of a configured rate limit for expedited traffic it becomes one possible implementation of ITN: Expedited packets belong to the FG class, and regular packets belong to the BG class.

ITN can also be seen as a combination of two other proposals from the differentiated services community: One is marking packets as *in* or *out* at routers [CLARK 1998], indicating whether they are in compliance with their assigned traffic class. During congestion, packets marked as out are give drop preference (similar to ATM's *cell-loss-priority* bit [ATM 1999] or frame relay's *discard-eligible* bit [THIBODEAU 1998]). The other proposal is a scheme where routers forward packets in strict order of priority [GUPTA 1997]. Together, these proposals can implement ITN by giving drop preference and lower forwarding priority to BG packets.

In a previous paper, we have investigated the idea of ITN service at the application layer, by distinguishing between FG and BG web transactions [EGGERT 1999]. The LSAM project [TOUCH 1998] built on this idea and used speculative background multicasting of web transactions to preload self-organizing, distributed caches with popular content.

The network stack of an ITN end system must implement the same outbound and inbound processing mechanisms as ITN routers. However, while routers only need to concern themselves with prioritizing packets during forwarding, ITN support for end systems is more complex. Routers operate at the network layer, while packet processing on end systems covers the whole depth of the protocol stack. Thus, end systems need to satisfy additional requirements to support end-to-end ITN. Processes need CPU time and possibly other resources to send and receive packets. Thus, simply replacing the FIFO of a network interface with a priority queue – which enables ITN on routers – is not sufficient: Other resources participating in packet processing may be the bottleneck, and dominate system behavior.

4.2 Idle-Time Networking with Current OS Mechanisms

A fully loaded computer system has usually a single bottleneck resource at a time, depending on its workload. Traditionally, that resource has been the CPU, but on many network servers, the network interface may be the bottleneck resource.

Having a well-known bottleneck allows optimizations for that particular resource to control system behavior. One example is the UNIX multilevel feedback queue for CPU scheduling, which originated on time-sharing systems where CPU time was scarce and needed to be carefully controlled. Other examples are recent proposals for providing multiple levels of network service [BLAKE 1998][CLARK 1998]. Both cases strive to improve control over the bottleneck resource, to optimize system behavior.

With speculative use of idle resources, the picture changes. No longer is one resource the deciding factor for overall system behavior. Ideally, processing of speculative requests fills all existing idle capacities. Schedulers interact in a fully loaded system. All resource schedulers in the OS must differentiate between regular and idle-time uses. Otherwise, unmodified schedulers will counteract the scheduling decisions of extended ones.

One example of such a scheduler interaction happens in network scheduling. A CPU scheduler with support for idle-time use (*POSIX*) cannot establish idle-time network service with a regular unmodified network stack, as experiments in the remainder of this section show.

Two simple CPU-based backgrounding mechanisms available on current systems include running the idle-time sender at *nice* or *POSIX* idle-time priorities. Experimental results show that both these mechanisms are ineffective in establishing idle-time network service.

4.2.1 Experimental Setup

In the experiments below, two copies of the same benchmark process run in parallel on a single host. The process is network-bound; it simply tries to send as much pre-generated random data to a second machine as possible. At the end of the experiment, the process reports the amount of data successfully sent. One of the two benchmark processes is the regular *foreground* (FG) sender, the other one the idle-time *background* (BG) sender.

As a metric for the effectiveness of support for idle-time use, we compare the throughput of the FG sender in the presence of a BG sender against the basic case (no BG sender present). Better mechanisms will yield higher FG throughputs. With an optimal schedule, the FG sender should reach 100% throughput, and not observe any change in transmission latency.

Each benchmark process uses three TCP or UDP connections to send its traffic, because a single TCP connection cannot easily overload an isolated network link due to TCP's congestion control algorithm. When sending with TCP, the benchmark blocks until one or more connections become writeable, sends a block of data over and starts over. When using UDP, it sends one message over each descriptor until the send call fails with an indication that the outbound device queue is full. It then sleeps for 10ms, and starts over. This emulates the sending behavior of the *ping* utility in "flood" mode, and generates enough traffic to saturate the 100Mbps link used in these experiments.

Another variable is the *intensity* of the FG sender, which controls how large a fraction of its CPU time quantum a benchmark process spends in the previously described sending loops. For a fraction of 0.1, for example, the process will only try to send traffic for 10% of its allocated time quantum. On BSD systems, the default quantum is 100ms, meaning the benchmark will generate send bursts of 10ms before sleeping for 90ms. The BG sender always sends at full intensity to simulate the worst-case situation for FG senders.

For each combination of transport protocol (TCP and UDP) and send intensity (full: intensity = 1 and light: intensity = 0.1), the experiment is run for 1 minute. Figure 12 and Figure 13 show mean normalized FG throughputs (against the throughput of a solitary sender) with 95% confidence intervals over a series of 10 iterations.

The figures below only show FG throughputs, as the performance of BG senders is not critical (by definition). An optimal backgrounding mechanism would allow FG throughput to reach 100%. Fair OS schedulers that cannot differentiate between FG and BG use would result in a FG throughput of 50% (the other 50% goes to BG traffic). Finally, under a completely ineffective backgrounding mechanism, BG traffic could starve FG packets, resulting in 0% FG throughput.

The sending host (running the two load-generators) and receiving host are two identical FreeBSD 4.2-RELEASE machines with 300Mhz Pentium II processors. They are located on an isolated, switched, full-duplex 100Mbps Ethernet. This setup is network-bound; one machine can satiate the link with a CPU load of 55%.

4.2.2 Full Foreground Load

In the first experiment, the FG sender sends TCP traffic at full intensity to the receiver. The left diagram in Figure 12 shows the measured and normalized FG throughput rates together with 95% confidence intervals (narrow white bars overlaying the wider gray bars). In this scenario, the link should not have any idle capacity, and no BG transmissions should occur. Achieving a full 100% throughput in this scenario means a FG was successful in monopolizing the link.

With a BG TCP sender (left bars in the left graph of Figure 12), neither the *POSIX* nor the *nice* backgrounding mechanism can establish idle-time network service. FG throughput reaches only 50%, indicating that some other scheduler is fairly splitting up network capacity between FG and BG, not the CPU scheduler. An optimal backgrounding mechanism would allow FG throughput to reach 100% here.

For a UDP BG sender (right bars in left graph of Figure 12), this experiment demonstrates the worst-case scenario: a BG sender without rate-control can virtually shut down FG service for all three cases. FG throughputs are around 5% across the board. An effective backgrounding mechanism must adapt to this scenario; both CPU-based schedulers fail to do so.

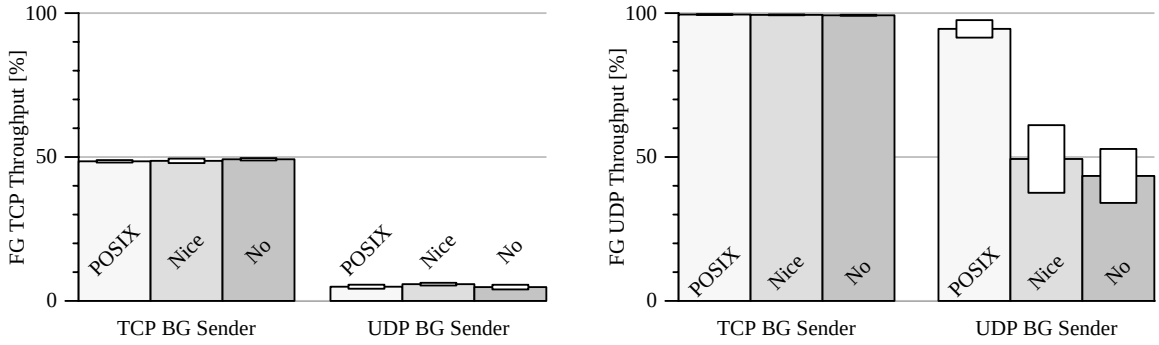

Figure 12: Normalized mean throughput of a FG sender under unlimited load in the basic case (*No*) and with two backgrounding mechanisms (*Nice* and *POSIX*), using TCP (left graph) and UDP (right graph) with 95% confidence intervals.

The right graph in Figure 12 shows the same experiment, but with a FG UDP sender. Here, the FG UDP senders achieve 100% throughput when running concurrently with a TCP BG sender (left bars in the right graph of Figure 12). However, this is not due to effective backgrounding, since the basic case also achieves 100%. The aggressiveness of an unlimited UDP sender manages to starve concurrent TCP traffic: this case is the inverse of the worst-case scenario discussed above.

If both the FG and BG sender use UDP (right bars in the right graph in Figure 12), the *POSIX* scheduler noticeably outperforms *nice* (90% throughput versus 50%). This scenario is the only one where one of the two CPU-based mechanisms is effective in achieving some service discrimination. The reason is that kernel processing for UDP send operations is synchronous (does not happen during interrupt handling), and CPU schedulers have some indirect control over network scheduling. However, FG performance is around 95% (with some variation) – idle-time use decreases FG service by about 5% here.

4.2.3 Light Foreground Load

In the second set of experiments, the FG sender is only active for 10% of its time quantum (= 10ms) and thus generates bursty traffic. In this scenario, the network link has idle capacity available, and BG traffic should be transmitted during its idle times. When a FG sender achieves 100% throughput in this scenario, it was successful in sending the same amount of as with no BG traffic present.

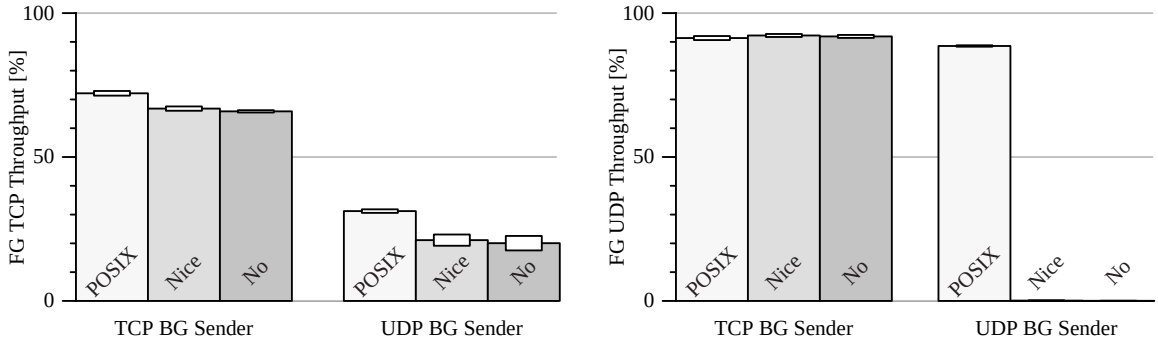

Figure 13: Normalized mean throughput of a bursty FG sender in the basic case (*No*) and with two backgrounding mechanisms (*Nice* and *POSIX*), using TCP (left graph) and UDP (right graph) with 95% confidence intervals.

When the FG sender uses TCP to transmit its bursts (left graph of Figure 13), the *POSIX* backgrounder offers small FG performance improvements (5-10%) over the basic case for both TCP and UDP BG senders, while the *nice* mechanism is ineffective. However, FG throughputs only reach 30% with BG UDP and 70% with BG TCP senders – the presence of BG traffic decreases FG performance by 30-70%. Clearly, the *POSIX* scheduler is not an effective backgrounding mechanism in this scenario.

With a FG UDP sender running concurrently with BG TCP traffic (left bars in the right graph of Figure 13), none of the backgrounding mechanisms is more effective than the basic case. FG throughputs still reach about 90%, but this again is due to aggressiveness of the UDP sender implementation, not due to backgrounding mechanisms.

When both the FG and the BG senders use UDP, BG traffic completely starves FG transmissions in the basic case and with the *nice* scheduler (right bars in the right graph of Figure 13). The *POSIX* scheduler manages to noticeably increase FG throughputs (to 90%), as it did in the UDP vs. UDP case under full load, described in the previous section. Again, this is due to synchronous UDP send operations that can be indirectly controlled through CPU priorities.

These results indicate that schedulers without support for idle-time can hinder the effectiveness of overall idle-time use, even if some of the other schedulers do support it. Thus, all resource schedulers must be extended for global support of idle resources. A previous study [NIEH 1993] has

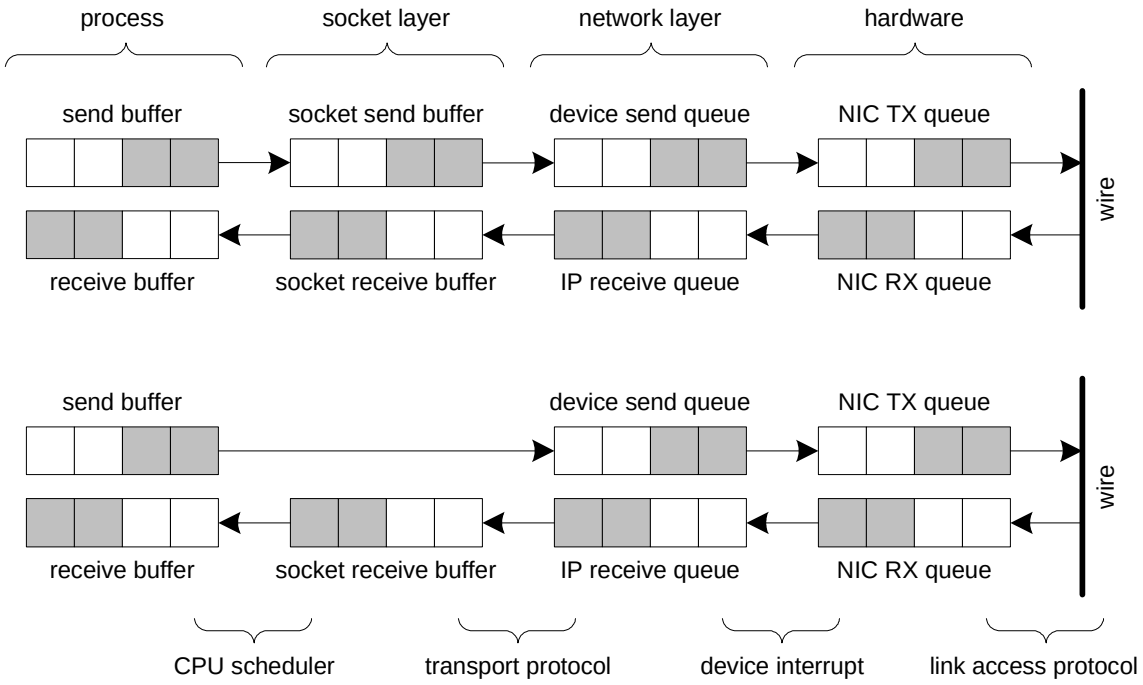

Figure 14: Queueing at different layers in the network stack for TCP (top) and UDP (bottom) processing.

presented similar results for a scenario where raising the CPU priority of a multimedia player did not produce the intended effect of a smoother stream playback.

4.3 Conventional Network Stack Processing

The experiments in Section 4.2 above have shown that current OS mechanisms (*nice* and *POSIX* scheduling) are not sufficient to establish ITN. This section will analyze the reasons of this failure by tracing the path of outgoing and incoming data through the BSD network stack, and pinpoint issues that inhibit ITN. Figure 15 and Figure 16 give a (simplified) view of the flow of execution inside the network stack during outbound and inbound processing, while Figure 14 shows the data flow through its various buffers. This analysis forms the basis of the OS modifications discussed in the next section.

On BSD systems, user-level processing cannot interrupt kernel processing; processing of kernel events has total priority. Inside the kernel, different events have different *interrupt priority levels* (IPLs). Thus, processing of one kernel event (which may have interrupted user-level processing)

can again be interrupted by a higher-priority event. User-level processing will only proceed after the kernel has processed both events (and no others occurred). Events at lower levels of the kernel (e.g. device drivers) have usually higher IPLs than events at higher levels of the kernel.

4.3.1 Outbound Network Processing

All user-level socket output flows through the `sosend()` function in the kernel down into the kernel (see Figure 15). Depending on the socket protocol and domain, it then calls the appropriate transport-layer output function through a dispatch table. For the Internet protocols, those are `udp_output()` and `tcp_output()`.

TCP sockets must maintain a copy of the user data so TCP's recovery algorithm can retransmit the contents of lost packets. Every socket contains a send buffer (`so_snd`) for that purpose. If the send buffer is full, `sosend()` will block the sending process until the buffer drains. When the send buffer has enough space available, `sosend()` appends a copy of the user data to it, and then calls the transport-layer output function `tcp_output()`. Inside `tcp_output()`, the protocol checks if it may send a segment for the respective connection, according to its congestion control algorithm and timeout rules. If so, `tcp_output()` calls the network-layer output function `ip_output()`; if not, the system call is complete and process execution continues after the write system call.

When a process writes on a UDP socket descriptor, `sosend()` does not buffer any data. UDP as a simple, unreliable datagram protocol does not offer protection from packet losses. Instead, `sosend()` immediately calls the transport-layer function `udp_output()`, which in turn simply calls the network-layer output function, `ip_output()`.

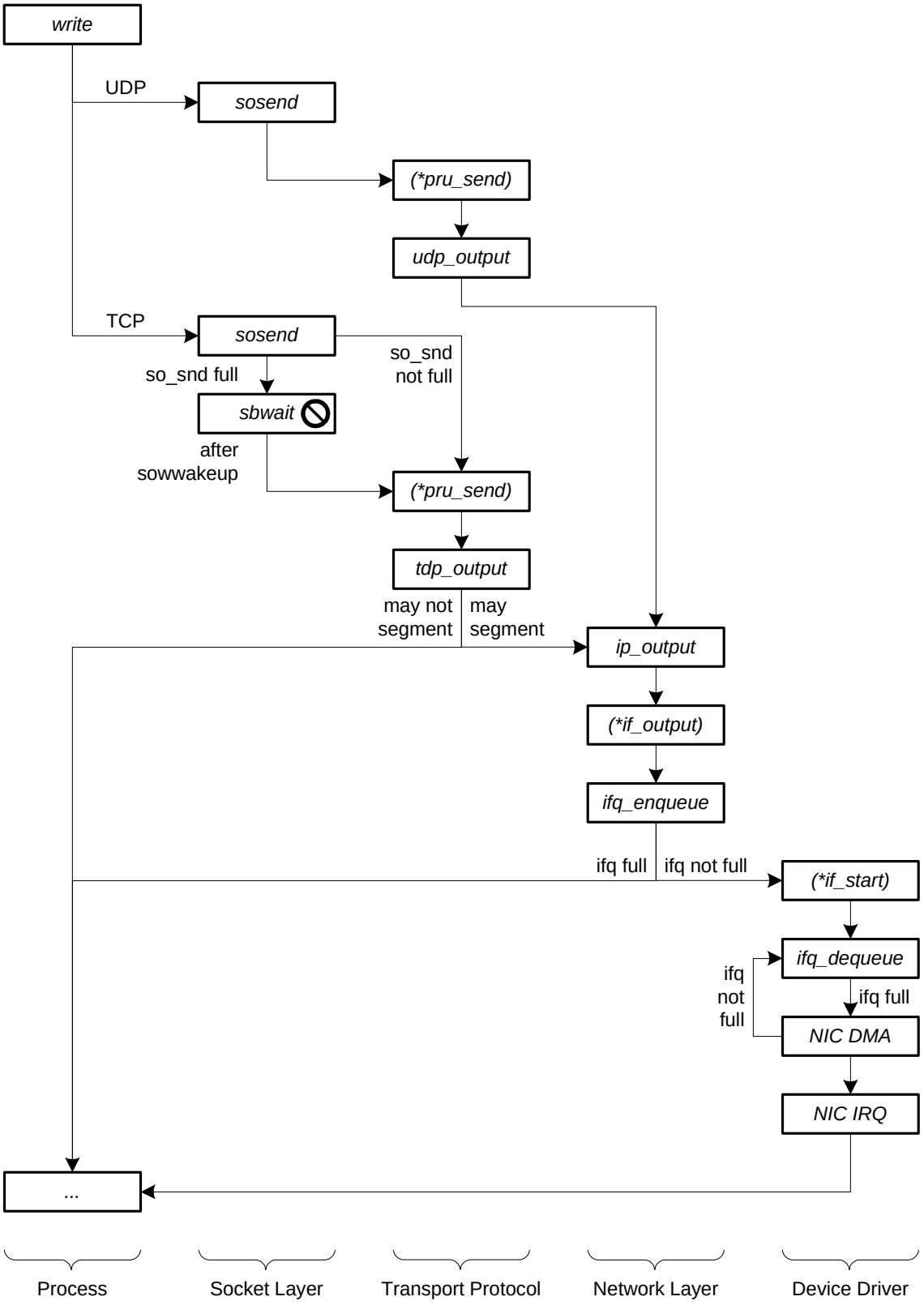

Figure 15: Network stack outbound processing.

Network layer processing for both UDP and TCP is identical. At first, `ip_output()` performs a route lookup, and then tries to enqueue the data in the device queue of the outgoing interface for that route. If the device queue is full, `ip_output()` drops the packet and the write system call is complete. If `ip_output()` was successful in enqueueing the packet, it calls the output function of the outgoing network interface (`*if_output`). This function, in turn, checks if the hardware is ready to transmit data, and if so, dequeues a packet from the device queue and starts transmission (`*if_start`). If not, it will simply return. After transmission starts, the driver will repeatedly dequeue and transmit packets until the device queue is empty (or the hardware's send buffer is full). It is important to note that the driver code runs at one of the highest interrupt priority levels (most interrupts are blocked), and so usually cannot be interrupted until the device queue is drained completely.

4.3.2 Inbound Network Processing

Inbound network stack processing starts with the physical reception of a packet by the network device (see Figure 16). The device will signal the availability of data to the kernel by issuing a device interrupt, which is handled by the device driver's interrupt routine. It copies the data from the device memory into main memory. The input routine of the driver then enqueues the data into the correct protocol receive queue. All IP data demultiplexes into the incoming IP queue (`ipintrq`) and a software interrupt signals data arrival to the upper half of the kernel. If `ipintrq` is full, the driver drops the data. At this point, processing loops back to the driver's interrupt handler. While more packets are ready to be transferred from the device memory, the driver will continue to demultiplex and enqueue them for reception by higher-level protocols. Again, since the driver runs at a high IPL, it will not exit this loop until the device receive buffer is empty.

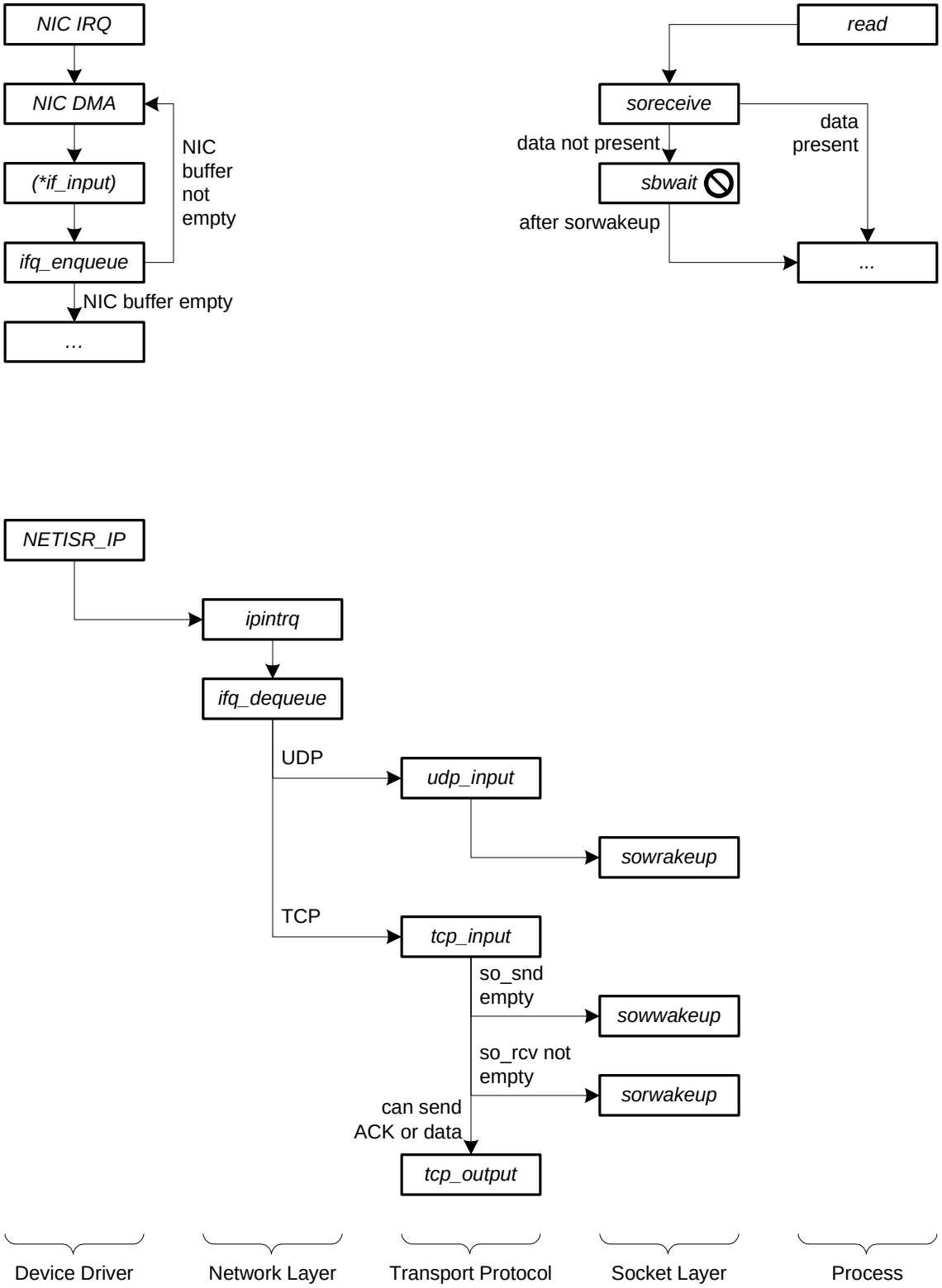

Figure 16: Network stack inbound processing.

When a software interrupt signaling IP packet reception occurs, the `ipintr()` handler loops over all packets in the IP incoming queue and calls `ip_input()` for each one. That function discards corrupted packets, dispatches packet forwarding (if needed) and manages fragment reassembly. For a packet destined for the local host, it calls the transport-layer input routine, based on the packet's protocol field. If dequeueing a UDP packet, `ip_input()` dispatches the packet to `udp_input()`, which appends the data to the receive buffer of the corresponding socket, and unblocks processes blocked to read data (`sorwakeup`). When `ip_input()` dequeues a TCP packet from `ipintrq`, it passes it to `tcp_input()`. As part of TCP protocol processing, `tcp_input()` may trigger sending new TCP packets (data and/or ACK) by calling `tcp_output()`, and wake up processes waiting to enqueue more data into the send buffer (`sowwakeup`). Data flows out of `tcp_input()` along the same path it does for UDP: the routine copies it into the receiving socket buffer, and waiting processes are unblocked (`sorwakeup`).

Whenever a process reads from a socket, `soreceive()` checks if enough data is present in the socket receive buffer to satisfy the read request. If so, it copies it to the process buffer and returns. If not, it blocks execution until the transport layer signals the arrival of more data through `sorwakeup`.

4.3.3 Discussion

For a TCP sender, the kernel buffers data in the socket send buffer, which the transport layer drains according to TCP's congestion control and timeout rules. The write call succeeds after the data enters the socket buffer, and the process continues execution. Either timeouts (in-kernel timer firing) or ACK receptions (device interrupt) trigger TCP packet sends.

Both of these events happen independently from CPU scheduling. The handlers for both events run at higher IPL than user-level processes, and will thus interrupt process execution. This means that a process may not even be running when the kernel sends packets on its behalf.

An ITN mechanism based on a modified CPU scheduler (like nice and *POSIX*) cannot hope to regulate network transmissions in this feedback system. It only controls which candidate process can access the socket queues to enqueue or dequeue data, not the timing of the transmission of that data.

For a UDP sender, data will usually go directly into the outbound device queue. It may seem that if the CPU scheduler enforced strict priorities, UDP data sent by a lower-priority process could never interfere with that of a higher-priority one, because the priority CPU scheduler would never allow the lower-priority sender to execute.

The experimental results in Section 4.2 demonstrate that while the *POSIX* scheduler can prioritize FG traffic to some degree (10% performance decrease), it is only able to do so when both FG and BG traffic is UDP. Whenever FG TCP traffic competes with BG traffic, it is not effective in prioritizing service.

The reason lies in the way typical UDP senders are implemented: In essence, UDP senders limit their send rate by blocking for a period of time when the send system call indicates a full device queue. (If this never happens, the outgoing link speed is higher than the data rate of the sender.) If the device queue fills up before the time quantum of a process runs out, it will sleep, causing the CPU to context-switch to another process. Even under the *POSIX* scheduler, if a higher-priority process voluntarily sleeps, lower-priority ones may run.

As noted above, the lower half of the kernel runs at IPL asynchronously from scheduled events in its upper half. This means that when the new process starts its time quantum, the driver has usually drained some packets from the device send queue and more data can be enqueued. So a BG sender scheduled when a FG sender starts sleeping (due to a full device queue) can usually send at least some packets before the queue fills up again, and it in turn sleeps.

Another issue with current OS processing is its focus on data reception: Network interrupt handlers suspend all other processing. Most systems follow an *eager receiver* model, and give highest priority to capture and storage of packets, second highest priority to protocol processing, and lowest priority to user-space process execution. The rationale is that receive buffer space is limited, and data must be moved off the device into main memory to prevent data loss due to buffer overruns.

Without hardware support, arriving idle-time data can interfere with any regular processing, because the system must first complete the reception before it can determine if it is regular or idle-time data, and drop or process it accordingly. In the worst case, a flood of idle-time data can drive a system into *receive livelock* [DRUSCHEL 1996][MOGUL 1997], where the system is loaded with

handling packet reception such that no other processing can occur. Again, this problem is worse for drivers that start scheduling additional requests (if present in the work queue), instead of relinquishing control.

Hardware support could allow the network interface to filter out the unwanted requests, and eliminate interrupt processing. However, hardware support and prioritized interrupts are major changes to current systems. Other techniques may offer some of their obtainable benefits with fewer modifications to the UNIX architecture. *Lazy receiver processing (LRP)* [DRUSCHEL 1996], for example, is a modification to the UNIX network stack. It charges resource use for network processing to the appropriate process, discards excess data early, and schedules (most) processing of inbound traffic at the process priority of the receiver. The basic idea is to minimize interrupt-level processing, instead of executing all link-, network-, and transport-layer processing on each packet reception. A host with support for LRP simply demultiplexes the packet stream (into per-socket work queues) at the driver level during interrupt processing. All higher-level receive processing is scheduled lazily, when the process issues the corresponding system call.

LRP (or a similar scheme) could benefit idle-time networking, by reducing the impact that BG packet receptions have on concurrent FG processing.

4.4 OS Extensions for Idle-Time Networking

The key issue with the two CPU-scheduler-based candidate mechanisms to implement ITN is the event-driven nature of kernel network processing. Nearly all network routines – with the notable exception of UDP sends – happen asynchronously with user mode execution: device interrupts trigger packet transmissions and receptions. Packet receptions trigger incoming transport protocol processing, which in turn may unblock processes waiting for data reception on a socket. For TCP, packet receptions (and to a lesser degree, kernel timeouts) trigger packet sends. In a sense, the network stack is an event-based system, where event priorities are equivalents to the IPL of the corresponding handlers. As demonstrated by the experimental results in Section 4.2, the previously examined CPU-scheduler backgrounding mechanisms have only very limited impact in such a system. A second issue is the use of FIFOs for all kernel queues. The processing order of a FIFO queue is identical to the enqueue order, which may cause a queue's consumers to process

earlier arriving BG data before FG (e.g. a FIFO device queue may send BG data before FG data, because it was enqueued earlier). This must not occur in a system supporting ITN.

4.4.1 Design Goals

The network stack is a complicated system, and many applications rely on its API (socket interface) and service semantics. Therefore, it is important to avoid fundamental changes to the network stack. Additionally, much effort went into designing and fine-tuning the Internet's transport protocols. OS extensions for ITN must not modify these transport protocols, to avoid incompatibilities with current standards. It is also impractical to change all network drivers to support ITN, so hardware-dependent driver code must not change for ITN extensions. Note that part of the driver code is common to all devices of the same family; these routines could be safe to modify. In addition, for end-to-end ITN, routers in the network must distinguish between FG and BG packets, as described above. The focus of this chapter lies thus on host extensions; it assumes network support for ITN is available and the network handles packets according to their service marks.

In summary, a design for OS extensions for ITN must be a simple extension of the current socket layer, must not modify the transport layer, and must not require changes to the hardware-dependent parts of device drivers – consequently, they must mainly extend the network layer.

4.4.2 Design

One issue identified earlier in this section was the use of FIFOs for all queues in the network stack. To support ITN, two-level priority queues must replace most FIFOs in the network stack. Part of the KAME IPv6/IPsec package [JINMEI 1998] for BSD is the ALTQ framework [CHO 1998] of alternate queueing disciplines. ALTQ replaces the outgoing standard FIFO queues of device drivers with configurable queueing disciplines, including priority queues. We have extended ALTQ to the inbound protocol queues (mostly `ipintrq`) and to drop lower-priority packets for higher-priority ones when the queue is full. ALTQ filters put marked packets into a lower-priority traffic class, for both outgoing device and incoming protocol queues.

The service level of the network stack must not decrease for ITN-unaware applications – the kernel must not send their packets as BG by default. Only ITN-aware applications may use the new

service class, by explicitly indicating this to the kernel. The socket layer offers socket options to set user-configurable options on a per-descriptor basis. Thus, the only socket-layer change needed is a new socket option (`SO_BACKGROUND`) that indicates that the network stack should treat all traffic from or to a socket as low-priority BG traffic. Note that this scheme is the inverse of other proposals for packet marking that use marks to increase the service level (e.g., expedited forwarding). Without proper policing mechanisms, these schemes become problematic – nothing keeps processes from marking all their packets as high-priority, and thus receiving better than best-effort service. The proposed marking scheme for ITN avoids these complications by only allowing applications to lower their service level.

Because of the event-based nature of the lower half of the kernel, drivers will transmit packets as soon as they enter their device queue (a transmitter activation follows each enqueue operation). Because the driver code executes at a higher IPL than the network layer, it typically sends the packet before another one can be enqueued. Consequently, the network layer must verify if BG packets may be sent at a particular time before it enqueues them into the device queue. The key idea is that the host should never send BG packets to any destinations when a FG sender is using the same outgoing interface. Instead, the network layer should drop these BG packets, signaling an *out-of-buffers* (`ENOBUFS`) error condition. UDP senders must already be prepared to handle this error condition (it occurs when the device queue fills up), and TCP will take the packet drop as an indication of congestion and lower the rate of the BG sender.

There are several possible methods to determine if an interface is in use by a FG process before enqueueing a BG packet into a device queue. The simplest one is to check if a FG protocol control block (PCB) exists that uses the same outgoing interface. While this simple approach is effective, it is also too restrictive: A single FG TCP connection prohibits any BG traffic from being sent – even when it is idle. A more effective identifier of active senders would not only check for the presence of a PCB for an outgoing interface, but also use additional means to determine if the PCB is an active user of the interface. For example, it could check if the corresponding socket had any queued data in its send buffer, which would indicate an active sender. The prototype implementation evaluated in the next section uses this technique.

Active UDP senders are more difficult to identify. Unlike TCP, UDP does not buffer any data at the socket layer (all UDP socket send buffers are always empty), so the check described for TCP

in the previous paragraph is not effective. Furthermore, UDP writes are non-blocking; they either succeed in enqueueing data into the device queue or fail and return to the user process with an error code. No kernel state exists that allows determining precisely if a UDP sender is active or not at any given time. The current design for ITN thus uses the following heuristic to check for active UDP senders: For each UDP PCB, the network layer will check if the corresponding process is sleeping or not. A sleeping process indicates (paradoxically) an active UDP sender. This heuristic depends on the common structure of implementing UDP clients, which send until they fill the device queue or run out of data, then sleep to enforce a send rate limit. (See the next section.)

The design for ITN in this section is clearly a proof-of-concept, and practical reasons argued for minimal modifications to the current network stack. A completely redesigned network stack with support for ITN from the ground up would be an equally possible solution, but the extent of such an effort is outside the scope of this project.

Several performance issues exist with the current design. One is that the decision to enforce ITN at the network level causes BG packets to go through socket and transport layer processing, only to be dropped when FG senders use the same outgoing interface. Enforcing ITN at a higher layer would not incur this performance hit. For more compute-intensive future transport protocols (e.g. encrypted or tunneled flows), this may prove problematic. A second performance issue is the per-BG-packet overhead of looking up the PCB for a packet and determining if FG senders (PCBs) exist for the same interface. The current implementation adds list of users (pointers to PCBs) to each interface to limit the impact of this search. Schemes that are more complex may further mitigate this overhead, but are outside the scope of the initial implementation.

Detecting active UDP senders (to protect FG UDP traffic from BG interference) at the network layer is difficult, due to lack of information. The kernel can gain information about TCP connections and their corresponding processes from internal state. For UDP senders, no such state exists at the kernel level; UDP senders manage it inside the application. One future possibility could be to extend UDP to utilize to queue data at socket send buffer, and to drain it as the interface queue empties. This would allow the TCP technique to check for active senders to extend to UDP senders.

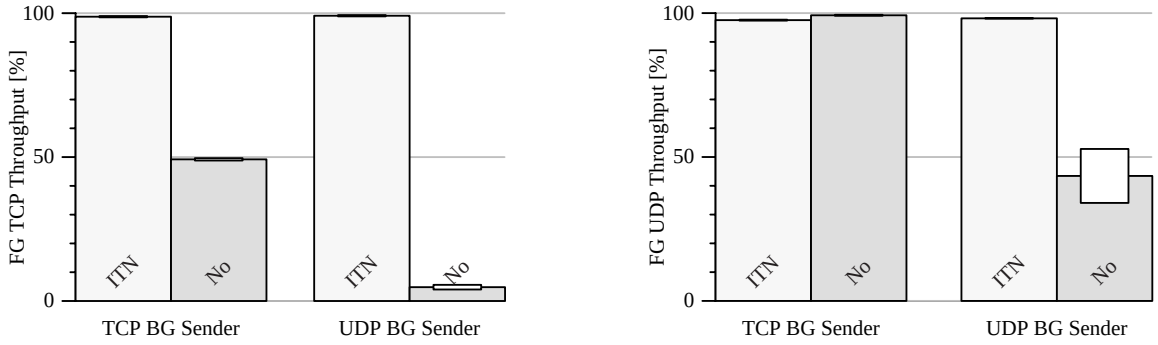

Figure 17: Normalized mean throughput of a FG sender under unlimited load in the basic case (*No*) and with the *ITN* backgrounding mechanism, using TCP (left graph) and UDP (right graph) with 95% confidence intervals.

4.5 Experimental Evaluation

To evaluate the effectiveness of the ITN mechanism designed in the previous section, we repeat earlier experiments (see Section 4.2) with the new ITN backgrounding technique. The experimental setup is unchanged, except that the BG senders are now using the new network ITN backgrounding method.

4.5.1 Full Foreground Load

The left graph in Figure 17 shows how a fully loaded TCP FG sender behaves under BG load generated by TCP or UDP senders that use the ITN backgrounder. In both cases, FG throughput reaches about 99% of the maximum. An optimal backgrounding mechanism would achieve the full 100%; the proposed ITN mechanism gets very close.

The right graph of Figure 17 displays the result for a UDP FG sender. Again, FG throughput under full load reaches 97-99% for both UDP and TCP BG traffic. With TCP BG traffic, this is no improvement over the basic case, because the aggressive UDP FG traffic can already monopolize the link. In fact, throughput is 1-2% lower, maybe due to processing overhead of the ITN mechanism. With a UDP BG sender, however, the ITN mechanism is again very effective, increasing FG throughput to 99%.

Under all these full-load scenarios, the ITN backgrounder is effective in isolating the FG traffic from the presence of any BG traffic in the system.

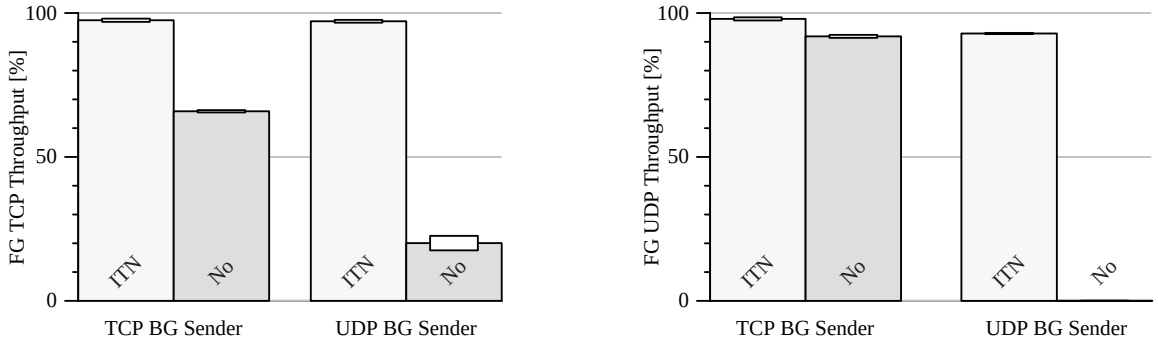

Figure 18: Normalized mean throughput of a bursty FG sender in the basic case (*No*) and with the *ITN* backgrounding mechanism, using TCP (left graph) and UDP (right graph) with 95% confidence intervals.

4.5.2 Light Foreground Load

The next set of experiments looks at the performance of a bursty FG sender using the ITN backgrounder. For a TCP FG sender (left graph in Figure 18), the new mechanism improves FG throughput between 35-80% to 99% total for both TCP and UDP BG traffic. Thus, it effectively isolates FG traffic from the presence of BG senders.

For a UDP FG sender (right graph in Figure 18), the ITN backgrounding method also increases throughput to 99% for both TCP and UDB BG traffic. With a TCP BG sender, this is a minor improvement of 5% over the basic case. Again, this is because the UDP sender is already aggressive enough to reach high throughput. With a BG UDP sender, the performance increase is around 90% – bursty FG UDP traffic was almost denied service in the basic case, now its performance is close to optimal.

4.5.3 Discussion

The experimental results presented in this section show that the proposed ITN extensions are effective in isolating FG traffic from the presence of BG traffic. In all the investigated scenarios, FG performance reaches 97-99% of the basic case (where no BG traffic is present), effectively isolating FG packets from the presence of BG traffic. While the benchmark framework used for these experiments is flexible, the current load-generating processes are very simple. This was a deliberate choice, to factor out secondary system interactions from the results. Future experiments

should investigate the behavior of the ITN backgrounder under real workloads, such as supporting a web server with FG and BG service classes.

Another shortcoming of these experiments is that they only measured FG throughput, not latency. For full *isolation*, not only must FG senders reach the same throughput they would in the absence of BG traffic, they also should not see any increase in transmission latency. Future experiments should investigate this further.

As mentioned above, the benchmark processes used during the experiments throughout this paper are deliberately simple. The next set of experiments should investigate the behavior of the current ITN design under a realistic workload, such as supporting a web server with FG and BG service classes. Another interesting set of experiments would compare the packet-scale send behavior of an ITN-enabled host with BG traffic against a basic one without (under the same workload) to investigate how close to the ideal behavior the current mechanisms are.

5 Related Work

Related work falls into two broad categories: First, systems that prioritize resource use, such as hard and soft real-time systems. The second category comprises of various speculative techniques, such as prefetching and caching, or remote execution systems. The remainder of this section contrasts these systems with this proposal.

5.1 Real-Time Systems

Correctness of the computation in a real-time system not only depends on its logical correctness, but also on the time at which results are produced. Missing the timing constraints (deadlines) of the computation results in a critical failure.

The system defined above is commonly referred to as a *hard real-time* system, where missed deadlines are equivalent to a total system failure. *Soft real-time* systems relax the latter restriction: Missed deadlines are undesirable, but not catastrophic. In both cases, construction of a schedule such that all tasks meet their deadlines is critically important. The spectrum of existing real-time systems – from hard real-time to soft "multimedia" real-time – is vast.

5.1.1 Examples

The *Spring Kernel* [STANKOVIC 1991] supports real-time execution on multiprocessor machines, guaranteeing absolute predictability based on *worst-case execution times*. One processor of the system is dedicated to execution of the system kernel; the rest is available to execute user processes. One unique feature of *Spring* is its planning-based approach to resource scheduling, which eliminates blocking from the system, but depends on detailed description of the resource use of all application programs. Spring offers predictable memory accesses by preloading and locking physical pages, and by saving and restoring the *translation look-aside buffer* during context switches (a related technique is also effective for traditional systems [BALA 1994]).

Nemesis [LESLIE 1996] is a vertically structured OS, where a microkernel implements only minimal task switching functionality. Shared libraries provide the bulk of in-kernel services offered by a traditional OS at the application level. Thus, most processing on behalf of user processes is subject to scheduling by the microkernel, and is correctly charged to the processes on whose behalf it

occurs. Unlike *Spring*, *Nemesis* does not support hard real-time processes, and processes are not required to specify their resource requirements in advance. Instead, it focuses on providing a consistent quality-of-service environment for multimedia (soft real-time) applications through a QoS manager. It notifies applications of changes to the service allocation, and expects them to adapt. Among other things, it signals to applications whether an increase in their resource share is due to a (temporary) increase in idle capacity, or to an actual change in the service allocation. Thus, *Nemesis* supports some notion of processing with idle capacity.

Eclipse/BSD [BRUNO 1998][BRUNO 1999] is similar to *Nemesis* in that it focuses on providing soft real-time service targeted at multimedia applications. Unlike the former, it requires explicit resource reservations through hierarchical CPU, disk, and network schedulers. *Real-Time Mach* [TOKUDA 1990] is a microkernel-based OS similar to *Eclipse/BSD*, but with support for hard real-time processes. Again, applications explicitly notify the system of their resource requirements through reservations.

AQUA [LAKSHMAN 1998] is a kernel-level framework that allows cooperating processes to dynamically negotiate their CPU and network requirements with the kernel. If a resource becomes congested, *AQUA* notifies affected processes to allow service adaptation. *Omega* [NAHRSTEDT 1996] is an end system framework that supports soft real-time scheduling of CPU, memory, and network I/O to provide end-to-end quality-of-service. *Omega* is similar to *AQUA*; processes dynamically negotiate their resource requirements with the system.

Scout [MOSBERGER 1996] is a communication-oriented OS based on the abstraction of data paths. *Scout* allocates threads to active paths according to a variety of schedulers, to vary the service model of the system. Idle-time execution in *Scout* would require the addition of idle-time paths combined with a thread scheduler supporting two service classes.

5.1.2 Discussion

All the real-time systems mentioned in the previous section differ in one or more of the following key characteristics from a traditional, general-purpose OS: *predictability*, *resource requirement specifications*, and *admission control*.

One key difference is *predictability*, which requires time bounds on all resource operations and scheduling overheads. Without such bounds, processing deadline guarantees are impossible. Narrow bounds are desirable for higher system utilization. Defining time bounds on operations is difficult and usually hardware dependent – for example, the maximum time of a disk read operation depends on the disk drive model.

Predictability is not required for non-interfering idle-time use as proposed here, although it might lower some preemption costs. With a known service time for a request, a scheduler may let an idle-time request finish instead of preempting it when a regular request arrives. If the time-to finish of the idle-time request is less than the preemption cost, this might decrease interference with regular use.

A second difference between regular and real-time systems is *resource requirement specifications*. Processes must disclose their future resource use to the system. In the basic case of a dedicated system, a programmer statically verifies that the system can satisfy all resource requirements of the various processes, and compiles a fixed schedule controlling resource use. Naturally, such a system will not support dynamic process creation, and is too limited for general-purpose use. More advanced real-time systems allow processes to explicitly disclose their planned resource use and deadlines at run-time, and can automatically generate resource schedules based on these reservations. In both cases (explicit or automatic schedule generation), the workload of the system must be periodic. The resource requirements of dynamic workloads are difficult to predict, and their worst-case resource use may be unbounded.

For this proposal, resource requirement specifications are not required for regular processes. If speculative tasks choose to specify their resource requirements, the system could optimize speculation by not allocating any idle-time capacity to tasks that require capacity on a resource that is fully loaded. However, this is an optional optimization of the speculation mechanism, and not a required condition.

When a new real-time process is created, the system verifies its execution feasibility dynamically, and rejects the process if execution would over-commit its resources. This *admission control* is the third key component of a real-time system. A general-purpose OS does not need to perform this operation, because is neither offers resource reservations nor fixed deadlines. Dedicated real-

time systems do not require admission control, because their workloads are static, with externally proven deadline guarantees.

With resource requirement specifications, more advanced systems can automatically generate schedules for periodic workloads. Such schedules require prioritized resource access. This aspect of real-time systems is very similar to the current proposal, which also requires resource *prioritization*: Many of the prioritized schedulers proposed for real-time systems can implement idle-time use for a given resource. However, the key difference is that real-time systems give preferential treatment to some resource requests, to meet specified or implied service goals.

The current proposal, on the other hand, is the inverse: Some resource requests receive less-than-default service. It is simple to convert a mechanism for the former into one for the latter (raise the default priority, use explicit notification to lower it) for a single resource. Thus, priority schedulers for real-time systems can implement idle-time use for a single temporally-shared resource. However, *prioritization* is not sufficient to establish non-interfering idle-time use; *preemptability* and *isolation* are also required, both of which real-time systems do not provide. It is, for example, acceptable for an RTOS to continue processing a lower-priority request when a higher-priority one arrives, as long as it misses no deadlines. It may in fact be advantageous to avoid preemption, to increase resource utilization. *Isolation* is a concept that has no equivalent in an RTOS; side effects of execution at different priorities are always globally visible.

Furthermore, real-time systems focus on scheduling temporally-shared resources. Idle-capacity use of spatially-shared resources is a key requirement of this proposal that an RTOS does not address.

Thus, while an RTOS can offer one requirement for speculative idle-capacity use (*prioritization*), two others are unsupported (*preemptability* and *isolation*). Furthermore, real-time execution requires *predictability*, *resource requirement specification* and *admission control*, all of which are unnecessary for idle-time resource use.

5.2 Speculative Uses of Idle Capacity

A wide variety of systems tries to use idle capacity speculatively. This section discusses three areas in which such uses of idle-times are common: prefetching and caching, optimization and maintenance tasks, and idle-time processing.

5.2.1 Prefetching and Caching

The storage facilities of a system form a hierarchy according to their access times and transfer speeds. *Prefetching* is a technique to retrieve a data item before access (hiding the access time). *Caching* is a related technique to replicate data in unused capacity at a higher level in the storage hierarchy. Prefetching requires caching to store the prefetched items, while caching is effective without prefetching, by simply replicating recently used data in faster storage.

Prefetching and caching are important techniques to speed up execution. The goal is to interleave I/O activity with computation, and to prefetch data prior to use, hiding I/O latency. Several proposals focus on prefetching, using different compile-time, run-time, and speculative techniques.

One approach uses idle CPU cycles while a process is blocked to speculatively continue execution of a shadow copy of the same process, to generate prefetching hints to speed up future I/O [CHANG 1999]. The shadow copy executes in a sandbox environment that prevents side effects to become visible to the original process. The authors claim 30-70% reductions in execution times for various benchmarks.

Another approach for *out-of-core* computations (where large datasets must reside on secondary storage) uses compiler techniques to automatically insert prefetch instructions into the application code [MOWRY 1996]. Experiments show that the technique is successful in hiding between 50-98% of the I/O latency, speeding up execution by a factor of 2-3. Similar techniques are also effective for memory cache prefetches [MOWRY 1998][OZAWA 1995].

Finally, another mechanism allows applications to disclose future disk accesses to the OS explicitly by passing hints [PATTERSON 1995]. The authors report a performance increase of up to 42% for some applications.

All three approaches strive to identify idle resource times to schedule their prefetches. The first system does so automatically, by running the hint generator when the process is blocked. However, due to the absence of prioritized resource access, the hint generator, as well as the prefetches can still interfere with CPU and disk use by other processes. The last two systems are even more limited, because they rely on application-level strategies to identify idle times. The mechanisms presented in this proposal could improve and simplify all these systems, by shielding regular resource use from the presence of the prefetches, as well as caching speculative data in idle storage capacity.

5.2.1.1 Effects on System Caches

Most operating systems contain caches at many levels in the processing hierarchy (memory cache, disk buffer, ARP, HTTP, etc). Speculative operations can modify cache contents, affecting regular performance. The disk block hint generator described above [CHANG 1999] is a system that explicitly tries to pre-load the disk cache with useful data, to increase performance.

However, performance decreases due to speculative uses can also occur. One study reports a decrease in regular performance when memory pages are speculatively cleared. Most operating systems clear memory pages before they allocate them to processes, to prevent security holes. However, page-clearing at allocation time severely affects performance. Clearing pages in the kernel's idle loop, so that pre-cleared pages are available at allocation time [DOUGAN 1999], may alleviate this problem. However, the authors report that memory cache pollution due to page-clearing limited the overall performance gain: Useful application-related state was flushed to cache page-clearing state, and application performance thus decreases (even though page allocations were faster). The obvious fix is to disable cache replacement during the idle time operation. While this decreases the performance of the page-clearer, it retains application state in the cache, and thus improves performance.

Other studies, however, find that leaving caches enabled during speculative execution can have a beneficial effect on overall performance, due to a pre-load effect. One such example is the prefetchers discussed above, which pre-load the disk cache with useful information. Another study investigates the memory cache behavior of a processor with support for multithreading [KWAK 1999], and finds that hit rates increase for related threads that exhibit locality-of-reference, while they decrease for unrelated threads. A third study monitors the execution behavior of specula-

tively executing processes [PIERCE 1994]. The authors report that while data references increase with speculative execution, data misses increase only moderately; and the prefetch effect more than offsets the performance impact, resulting in improved performance overall.

With extensive idle-time use of resources, as proposed in this paper, cache pollution is a major issue. To guarantee *isolation*, extensions to suspend cache replacements may be required for many of the system caches.

5.2.2 Optimization and Maintenance

Most operating systems regularly perform optimization and maintenance operations, to detect system problems and improve performance. Many of these operations are lengthy and not critical, and several proposed techniques try to schedule them during idle times. Examples of such tasks include disk block replication [AKYUREK 1995], speculative disk arm movement [KING 1990][MUMOLO 1999], prefetching and caching file system meta-data [MOLANO 1998] and file system consistency checking and reorganization [MCKUSICK 1999][MATTHEWS 1997]. Section 3.1.2 above discussed these in more detail.

Some network-related techniques to improve performance, like web prefetching [PADMANABHAN 1996], are similar to the disk ones. Unlike using local resources, network transmissions usually require an expensive connection setup. Thus, caching and pre-warming of connections [COHEN 2000], PMTU probing and IPsec key negotiations are techniques that do not apply to local resources. Speculative execution of these phases can hide much of the latency associated with connection setup. Section 3.1.1 above presented these in more detail.

Again, all these techniques could execute speculatively using idle resources, instead of heuristically limiting them in the hopes of minimizing their impact on regular processing.

5.2.3 Idle-Time Execution

All the previous techniques used idle *local* resources speculatively. Several other systems try to use idle *remote* resources for productive work. One category of such systems is *process migration* systems (*cycle harvesters*), which push local processes to idle remote machines for faster execu-

tion. Another category is *data migration* systems, which push data to remote machines that execute a common process.

One major difference to this proposal is that these systems concentrate on detecting remote availability and then utilizing idle capacity for a single resource only; other resources are only used as an implicit side effect. The proposed system, on the other hand, strives to utilize idle times of all resources independently of one another.

Another difference is that these systems do not prioritize between idle-time and regular processing. Thus, they treat idleness as a system-wide Boolean condition, while the proposed system supports utilization of partially idle resources. While many systems (especially real-time systems, see Section 5.1 above) support high-priority resource access, few others offer the notion of idle-time background use. One of the few that does is a hierarchical CPU scheduler, where arbitrary threads can act as schedulers for other threads by donating part of their allocated CPU time [FORD 1996]. One such scheduler explicitly supports *background* CPU use, similar to the *POSIX* idle-time scheduler [POSIX 1993].

5.2.3.1 Process Migration

Cycle harvesters schedule computations on a network of workstations, hoping to exploit idle remote resources to speed up local jobs. Historically, they have focused on utilizing remote CPU cycles (hence the name) and only utilized other remote resources indirectly. Cycle harvesting is especially effective for parallelizable jobs that can utilize multiple remote machines at once. However, even sequential jobs can benefit from remote idle-time execution, where they do not have to compete for resources with other active processes.

The *Sprite System* [DOUGLIS 1991], *Condor* [LIZTKOW 1988], *Batrun* [TANDIARY 1996], *DAWGS* [CLARK 1992] and the *V System* [THEIMER 1985] are cycle harvesters that support process remigration, when a remote host under idle-time use becomes unavailable. *Butler* [NICHOLS 1987], a component of the *Andrew* system, is a transparent remote process execution facility that does not provide process migration, but simply terminates remote processes when a remote machine becomes unavailable.

While cycle harvesters are similar in spirit to speculative idle-time use proposed here – both approaches aim at reclaiming wasted capacity for useful work – several key differences exist. Cycle harvesters are often application-level or middleware solutions running on top of a conventional OS without prioritized processing. Most of their shortcomings, such as *migration overhead* and *idle-time detection*, are artifacts of that design.

Without prioritized resource use, cycle harvesters cannot effectively utilize machines with partially idle resources (bursty local workloads). Since migrated processes run at the same priority as regular ones on the remote machine, any migrated process can severely decrease regular performance on a remote machine. Thus, most harvesters only reclaim cycles from remote machines that are fully idle. The system presented in this proposal, however, supports *prioritization* and *preemptability*, and can utilize partial idle capacity.

Another consequence of the lack of *prioritization* is high *migration costs*. Whenever a remote machine becomes unavailable for idle-time use, all remote processes on it must be re-migrated or terminated. Re-migration is a costly operation and will decrease regular performance of the remote machine during the migration period. Terminations are faster but still not instant, because the system must roll back to invalidate local state created by the terminated remote process. Additionally, partial work completed by terminated processes is lost.

With process migration systems, the finest-grained operation corresponds to the migration of a remote process. In the proposed system, on the other hand, an operation is a single resource request (e.g. sending a packet, reading a disk block). Thus, the overhead of aborting idle-time use in the proposed system is smaller, because the granularity of operations on the idle resource is finer-grained (e.g. wait for disk read to finish vs. re-migrating an entire process).

High migration costs further reduce the chance for utilizing idle resources: For bursty remote workloads with short idle-times, a cycle harvester could enter a state of thrashing, where all idle periods are spent migrating process to and from the machine, and no forward progress was made. Because the exact distribution of remote idle times is usually unknown, most cycle harvesters employ coarse heuristics and/or predictors [WYCKOFF 1998][GOLDING 1995] to find likely long idle periods. These techniques are effective in utilizing long, periodic idle periods (e.g. night hours), but fail to detect shorter, transient idle times due to quantization. They can thus fail to utilize some existing idle capacities of their target resource. The proposed system does not require

such heuristics, since *prioritization* inherently establishes different service levels, and can utilize all present idle capacities (of all resources).

5.2.3.2 Data Migration

Unlike cycle harvesters, which push both code and data to an idle remote machine for execution, *data migration* systems only move data to idle peers for processing or storage. All remote machines participating in such a distributed system already run a copy of the same client process. Process migration systems offer more flexibility in remote idle-capacity processing, but data migration systems are simpler, can be platform-independent and have smaller preemption costs.

One popular subclass of such systems are application-level clients for distributed computation projects, such as cryptographic code breaking or searching for large prime numbers [HAYES 1998] – or even extraterrestrial life [KORPELA 2001]. In these systems, all participants run the same client, and servers only migrate replicas of the data to be processed.

Other systems use unused remote memory as secondary storage, instead of a local disk [MINNICH 1989][FEELEY 1995][KOUSSIH 1999][MARKATOS 1996][NARTEN 1992]. This can improve performance, because access times for remote memory over a local area network can be an order of magnitude lower than access times for local disk space.

As with process migration systems, the idle-time mechanisms proposed in this paper can improve data migration systems by processing migrated data and communicating with remote peers during idle time.

5.2.3.3 Speculative Execution in Hardware

The proposed idea of using idle system resources speculatively is similar to some features found on modern microprocessors. A CPU with a *superscalar architecture* has multiple execution units; allowing it to execute multiple instructions per clock tick, further increasing performance. However, it cannot provide unlimited speedups, because speculation requires a continuous instruction stream. Conditional branch instructions and indirect jumps create problems for these systems, since they introduce ambiguities into the instruction stream that cannot be resolved until after the branch instruction has executed. Thus, execution units may remain idle until the CPU determines whether or not to follow a branch.

Instead, modern CPUs will use *speculative execution* to process likely future instructions. When the memory bus and some execution units are idle, the processor will speculatively prefetch, decode and execute likely future instructions.

Speculatively execution of instructions never decreases the execution speed of non-speculative processing, due to prioritized, preempted CPU resources (bus bandwidth, execution units). The CPU gives total priority to non-speculative instructions and immediately preempts any speculative processing for non-speculative execution. Hardware mechanisms eliminate preemption cost.

Furthermore, all side effects of a completed speculative instruction remain hidden until the processor can verify the prediction. For correctly predicted instructions, side effects become visible, while the CPU discards them for mispredictions. CPUs have hardware mechanisms to efficiently manage speculative state, and discard or commit operations do not delay regular processing.

Prioritized, preempted resource use, together with *isolation* of speculative side effects result in a worst-case performance that is identical to a CPU without speculation, even with constant mispredictions. For correct predictions, however, processor performance is improved. Because they implement the same three principles as the proposed system, CPUs with speculative execution are most closely related to it.

Processors designs supporting *simultaneous multi-threading* (SMT) interleave execution of instructions of multiple threads, to increase processor utilization compared to more traditional schemes that only exploit *instruction-level parallelism* (ILP). One speculative technique for SMT processors uses idle thread contexts to execute the less-likely branch of a predicted fork [WALLACE 1998]. The authors report a 14-23% average speedup for single program performance on an SMT with eight thread contexts, for programs with a high branch misprediction rate. These results may indicate that sharing idle-time capacity among multiple speculative tasks may also increase performance for the proposed system.

The *Address Resolution Buffer* (ARB) [FRANKLIN 1996] and the related decentralized *Speculative Versioning Cache* (SVC) [GOPAL 1998] allow reordering memory-referencing instructions, to better exploit instruction-level parallelism. Traditional processors enforce a total order between memory references, while the ARB enforces total order only among references to the same address. The ARB also supports speculative loads/stores, dynamically unresolved loads/stores and

memory renaming. The latter capabilities are similar to techniques required to support idle-capacity use of storage resources.

5.2.3.4 Speculative Execution in Software

Speculative execution has also been a part of some software systems, such as compilers or interpreters for programming languages. One example is a mechanism that speculatively interprets program branches in the *BaLinda* Lisp dialect, and assigns resources to speculative threads proportional to their relative likelihood [YEE 1993].

A related compile-time technique speculatively executes some method calls of Java programs using idle multiprocessor capacity [CHEN 1998]. For such methods, a speculative thread continues execution after the method's return point, using a predicted result value. The mechanism relies in part on properties of the Java virtual machine to shield threads from one another. The authors report significant speedups (up to a factor of 3) for data-parallel applications; only minor gains for control-flow-dependent programs.

Speculative execution has also been proposed in the area of information agents [BARISH 2000] and decision flow optimization [HULL 2000]. These approaches focus on generating good subtasks for speculative execution, but do not address the issue of executing them with idle capacities. This proposal, on the other hand, focuses on the OS extensions required for non-interfering idle-time use, but does not address generation of speculative subtasks. In that respect, the two mechanisms complement one another.

6 Plan

Chapter 2 introduced a model for speculative use of idle resource capacity, and identified *prioritization*, *preemptability* and *isolation* as the three key principles to establish non-interfering speculation. Without them, speculative idle-time use will interfere with regular use of resources. Based on the model, Chapter 4 presented a proof-of-concept design of idle-time extensions for the BSD network stack, and presented experimental evidence that the current mechanisms shield regular use from the presence of idle-time traffic in the system to within 1-2%.

The current proof-of-concept implementation supports *prioritization* and *preemptability* for the CPU scheduler (based on POSIX priority classes) and network I/O. Extensions for non-interfering idle-time use for other resources (notably disk I/O and storage capacity, and memory capacity) are not part of the implementation yet. The current mechanisms control inter-resource interference between the CPU and network stack, but may need extensions once idle-time support for other resources exists. In addition, isolation of speculative side effects does not exist yet, and integrated scheduling to optimize speculation is thus not yet available.

The remainder of this thesis research will address these limitations, by extending the current idle-time mechanisms to support idle-time use of the *network file system* (NFS). NFS [SANDBERG 1985] is a distributed file system that allows hosts to seamlessly mount part of a remote file system and present it as if it were part of the local file system. This allows transparent remote file access; processes are unaware of the physical location of the data they access. Idle-time NFS is a combination of idle-time mechanisms for the local disk and file system (at the server), and idle-

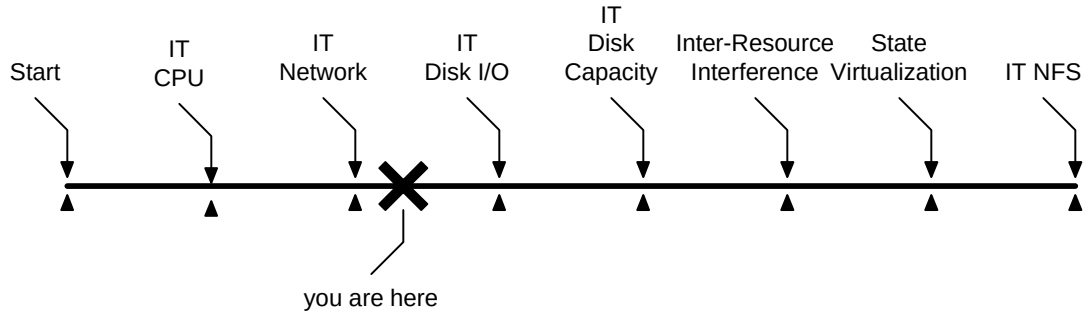

Figure 19: Phases of idle-time (IT) NFS implementation.

time networking mechanisms to speculatively access remote data.

Other methods for remote data access (e.g. FTP, web) were considered as candidate applications to extend and verify the idle-time model presented here. Idle-time NFS is a more interesting example compared to those others protocols: NFS as a file system supports a much wider variety of operations (read, write, seek, etc.) compared to FTP and web transactions, which are mostly read-only data retrievals. Tighter integration with the kernel offers a higher chance of interference with regular use – compared to the application-level candidates – and thus a better test case.

Support for idle-time NFS requires extensions to the current mechanisms that address its current limitations, and will thus verify the applicability of the idle-time model presented in this proposal:

- Server-side support for idle-time NFS requires new techniques for prioritized, preempted disk I/O and storage.
- Virtualization of OS state to establish *isolation* should be investigated, to prevent speculation from interfering with regular use.
- Inter-resource interference of the new idle-time disk service and the existing network and CPU extensions must be prevented, verifying the applicability of the current model.
- Idle-time NFS requires idle CPU, network and disk capacities and is thus a scenario to investigate mechanisms for integrated scheduling.
- Extensions of the idle-time model to support virtualized or stateful resources are required.

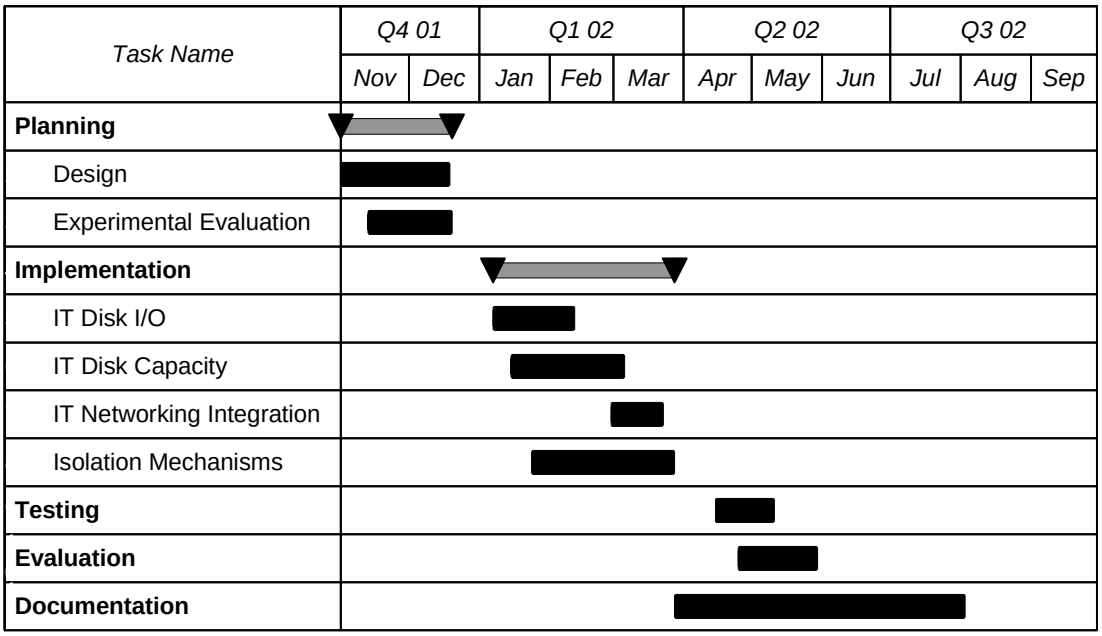

Figure 20: Timeline of the proposed research for idle-time (IT) NFS.

Figure 19 illustrates the planned research towards idle-time (IT) NFS, and indicates the current state of the mechanisms.

One part of the experimental evaluation of new mechanisms for idle-time NFS is similar to those presented in Section 4.5. It will compare throughput and latency of different-size NFS read and write operations in the basic case (without idle-time use) against the performance of the same requests under varying idle-time loads. Idle-time loads will both vary in intensity and in their target resources (e.g. disk-only, disk-and-CPU, etc.) Effective mechanisms for idle-time NFS will shield regular NFS traffic from the presence of idle-time use under all intensities and target resources.

While the previously described experiments are sufficient to determine to which degree new idle-time NFS mechanisms support *prioritization* and preemption, they do not measure whether *isolation* is established. *Isolation* is a condition, not a performance function. Part of the research to establish *isolation* must focus on defining an adequate metric for it.

Finally, Figure 20 shows the planned timeline for the completion of the proposed research.

7 Appendix: Extended Research Plan

The plan for the remainder of this thesis research is to support idle-time use of the network file system (NFS), as well as extend the underlying resource model. Both tasks will be detailed below.

First, the current resource model does not describe speculation costs and benefits, nor stateful resources (where prior operations can change the overhead of future ones) or virtual resources (that are users of other resources themselves). I plan to extend the model to describe these properties.

The major part of the thesis effort will be spent on the second part: extending the network file system (NFS) to support idle-time use. This work builds on the existing idle-time networking mechanisms. A major new component are mechanisms handling idle-time disk I/O and storage management.

One application for idle-time NFS are web caches, which replicate recently/frequently accessed web objects locally. Web caches aim at reducing both network and server load and page access times. Their effectiveness (aggregate hit rate) depends on two factors: cache size and per-entry hit rates. Increasing either of the two can increase the performance of the cache.

If remote idle disk capacity can be used as storage for the web cache – without interfering with regular use on the remote system – the cache size could grow by an order of magnitude or more, compared to storing it on local disk. As mentioned above, this can increase the aggregate hit rate of the cache, especially in scenarios where per-entry hit rates are low.

Data stored in idle-time capacity can disappear when part of the capacity is reclaimed for regular, non-idle-time use. Web cache entries are speculative by nature, and web caches already contain mechanisms (i.e. transparent re-requests) for missing data. Thus, they are well fitted for idle-time storage.

Support for idle-time NFS requires these components, which will be discussed in detail in the paragraphs below:

1. Extended idle-time networking (~ 1 month)

2. New idle-time support for disk I/O (~ 2 months)

3. New idle-time support for disk storage (~ 3 months)

The first step (1) are minor extensions to the existing idle-time network service, that should be completed within 1 month. A modified send mechanism (postponing a send attempt vs. simply dropping idle-time packets when the network is busy) may allow a higher idle-time throughput while still keeping regular traffic shielded. The current mechanisms, while effective in shielding regular traffic from the presence of idle-time use, too restrictively limit idle-time transmissions over a partially idle network. This part of the research will focus on TCP for NFS transmissions, UDP will not be investigated. The modified mechanisms will be evaluated in the same benchmark framework used during the proposal experiments.

The second step (2) implements idle-time use of disk I/O bandwidth. The current SCAN disk scheduler will be extended to support prioritized use, taking unique disk properties into account (seek time dominates, extra accesses during one arm sweep incur only minor overhead). Again, the new disk scheduler will be evaluated in the existing benchmark framework, both locally and together with the networking mechanisms. This part is estimated to require 2 months.

The final task addresses idle-time use of storage capacity. Tasks (1) and (2) focus on shielding regular processing from the presence of idle-time use (through prioritization and preemptability). Task (3) focuses on isolation mechanisms for disk storage capacity. Disk blocks that are unused by the regular file system will be used as storage capacity for a shadow file system for idle-time use. One major subtask is a mechanism for reclaiming blocks under idle-time use for regular storage, when needed. The estimated time for completing this component is 3 months.

8 Appendix: Related Work – Concurrency Control

Sections 2.2.3 and 2.3.3 in this thesis proposal presented the principle of isolation, which requires that the side effects of speculative requests must remain hidden until they are committed or discarded by the entity that issued the speculation (depends on the workload generation method).

The isolation principle virtualizes the operating system (OS) state. In an unmodified OS, all processing operates on the same system state, transforming it over time. In the presence of isolation, this can lead to incorrect processing, if the side effects (state modifications) of speculations become visible to regular processing.

One example of such a conflict is a speculative listen on a port. If a regular process tries to perform the same operation at a later time, the OS must deny this request, since the port number is already in use.

This is what isolation prevents. Since all speculations execute on virtual OS state (copy-on-write variant), the OS state seen by regular processing remains unchanged, and the execution behavior remains unchanged from the basic case (no speculation present in the system).

If the system deems a speculation to be unsuccessful, the virtual state associated with it can be discarded. However, the result of successful speculations (i.e. the side effects of the speculation) should be merged into the regular OS state. To prevent incorrect processing, this merge must be performed as an atomic operation with regard to other processing (regular and speculations).

Furthermore, conflicts between the regular and virtual OS state can arise, when regular processing modified the same pieces of state as a speculation did. This is similar to processing of concurrent transactions in a database system, where the same data item may be involved in multiple transactions.

8.1 Concurrency Control in Database Systems

Transactions in database systems are atomic operations on the contents (state) of a database. Allowing multiple transactions to execute concurrently increases performance, but requires mechanisms maintain database correctness.

Correctness depends on two conditions: integrity (defined through a set of constraints on the contents) and serializability. The latter requires database state changes to be equivalent to some serial execution of the given set of transactions.

A wide variety of mechanisms for concurrency control have been proposed [BERNSTEIN 1981][BHARGAVA 1999][KOHLER 1981][THOMASIAN 1998]. They can be roughly divided into three groups: locking, timestamps and rollback. Each of these groups will be briefly outlined below.

8.1.1 Locking

One scheme to address concurrency control is locking all data items required for a transaction. When a data item is already locked (by another concurrent transaction), a transaction can either wait, abort itself or preempt the other transaction. This is a pessimistic scheme, since the locking overhead is incurred even when transactions do not conflict.

One issue with this scheme is deadlock (circular lock dependencies among multiple transactions), which can be remedied with various solutions, (e.g. two-phase locking, ordered locks).

8.1.2 Timestamps

Another mechanism for concurrency control are timestamps on operations, which specify a fixed, serial processing order for all operations, guaranteeing consistency. Globally synchronized clocks are required. When conflicts arise, they are strictly resolved in timestamp-order.

Some timestamp schemes use implicit locking to maintain consistency, hile others are based on voting mechanisms, which trade overhead for central locks for communication overhead (which can be less in some decentralized systems).

8.1.3 Rollback

Rollback concurrency control schemes differ from the previously described classes in that no conflict prevention scheme is in effect during transaction processing. Instead, this scheme handles conflicts during commit time, by rolling back all state changes, and then either aborting or restarting.

Rollback schemes are optimistic in that the basic assumption is that conflicts will be rare, and infrequent concurrency control during commit time is more efficient than employing an a priori scheme on every transaction.

8.2 Discussion

There are two ways in which the proposed mechanism for speculative execution could benefit from database concurrency-control techniques. First, OS processes can be seen as database transactions, and the entire processing model could be mapped. Second, such techniques could improve the critical operation that maintains the isolation principle – merging of virtual state.

8.2.1 OS Processes as Database Applications

At some level, process execution in an OS and transaction processing in a database system are similar: Both allow multiple, concurrent entities (processes and transactions) to perform operations on shared state. However, concurrency control mechanisms for database systems may not directly apply to OS, due to a few key differences.

State conflicts in OS processing are relatively rare; first because processes usually spend a good part of their time in user-space (processing private, unshared state). Second, multiprocessors were rare, and systems had thus only one active physical thread of execution (even though simulating multiple threads of control through CPU scheduling). Thus, the OS could lock state through blocking interrupts, a fast operation. This is changing towards lock-based schemes as multiprocessors are becoming more common [LEHEY 2001][SCHIMMEL 1994], because interrupt blocking is limited to single CPUs. The locking overhead (compared to blocking interrupts) is compensated for by allowing more than one CPU to execute kernel code concurrently, and locks are placed on carefully (and manually) identified pieces of the kernel state.

Another issue is that concurrency-control mechanisms in databases must be general enough for a wide variety of dynamic application domains. On the other hand, the uses for such mechanisms in an OS are well-known and static, so simplified special-case mechanisms are worth deploying (e.g. for the process lists, device queues, etc.)

Furthermore, in the model proposed here, speculations have a lower priority than regular processing and are preemptable. While some concurrency control mechanism support similar prioritized models (e.g. for real-time databases) [HARITSA 1992][LINDSTROM 2000][YU 1994], they are not immediately applicable to prioritize speculations. (See the discussion on real-time systems in Section 5.1 of the proposal, similar arguments apply here.)

In databases, one correctness criterion is the existence of a serialized execution of the same transactions. The valid execution order of a set of OS operations in the presence of speculations is much more constrained: The order of regular operations on OS state must be unchanged from the basic case when speculations are present, and the intermediary OS states must also be identical. (See Section 2.2.3 for details.) Database mechanisms enforcing conventional serializability may not satisfy these stricter requirements.

8.2.2 Concurrency Control for State Merging

The atomic merge operation after a successful speculation is another place where concurrency-control ideas from databases may apply.

This state merge is a strictly confined operation. First, only two sets of data are involved (regular and speculative). It is rare that two successful speculations finish at the same time, and they can be committed in any order in that case.

Second, regular state has priority over speculative state: If a piece of regular state has changed during the speculation, the merge cannot be completed, and the result of the speculation must be discarded.

Third, even if a speculation runs to completion, it is not automatically successful - continued regular processing can change the usefulness of the speculation during its execution.

This makes optimistic, rollback-based ideas unsuitable for this operation. Such mechanisms would merge speculative state before the speculation ends (assuming absence of conflicts and success on termination). Rollbacks are triggered on conflicts, which causes regular processing delays whenever speculation fails.

Timestamp-based mechanisms are also not well suited to this scenario. Timestamps provide a serial execution order for transactions. However, timestamps do not capture the constraints of the state merge (regular state always overrides speculative state) well.

Lock-based mechanisms, on the other hand, are very applicable. A single lock for the whole state is the simplest solution. In effect, this preempts regular use for the duration of the merge operation, and will thus decrease regular performance. However, since this overhead is only incurred for successful speculations - which potentially improve regular performance - the locking overhead may be compensated by the speculation gain.

Using multiple locks for different parts of the OS state could further minimize the locking cost. For example, if a speculation has only changed state in the "network" part of the OS state, it would only need to acquire the "network" lock - regular processing that does not involve the "network" state could continue during the merge. This maps well to a copy-on-write approach for speculative state management, where different pieces of state can be locked to allow merging speculative revisions.

In the extreme case, each data item in the OS state would offer a separate lock. Clearly, this is infeasible due to the space overhead. An adequate mechanism will probably utilize multiple locks for logically separate parts of the state space.

8.2.3 Concurrency Control for Speculative Use

The database processing model is more general, but also more complex than the one proposed for an OS with speculative use. For the remainder of the thesis research, a simple copy-on-write variant is under investigation to manage virtual OS state for speculations.

When a speculation starts, no virtual state is associated with it until it starts performing write operations on OS state. Whenever is about to perform a write operation to a data item, the system atomically copies that data item (or a larger piece of state containing the item, such as a page), and then executes the write on the copied item. Read operations read from virtual state, if it exists for a given data item, and from regular OS state otherwise.

This scheme achieves the goal of isolating regular processing from the side effects of speculations - speculative writes modify only copies. It does not provide the reverse (i.e. speculations do see the side effects of regular processing), but this is not required.

It is a variant of traditional copy-on-write schemes [RASHID 1988][BRUSTOLONI 1996] because of the state merge operation required for successful speculations, which is unique to this scenario. At the end of a successful speculation, the system tries to merge the virtual state created by the speculation with the regular state existing at that time. Since data items can change in both virtual and regular state (when concurrent regular processing writes to the same data item), the system must detect these write conflicts, and abort (or restart) the speculation. Mechanisms to support this include checksums and access timestamps.

The details of the proposed mechanisms will be investigated as part of the thesis research.

8.3 Conclusion

While processing in general databases and operating systems is very similar at a high level, the operations required to support speculative use of idle resources have unique properties that mechanisms proposed for database systems databases either cannot support, or only support with additional overheads due to their generality. However, some of their more basic concurrency-control techniques (such as lock-based schemes) can be modified to support this scenario. A customized concurrency-control mechanism for isolation of speculative side-effects, based on a variant of copy-on-write and locking, has been outlined and will be further investigated in the remainder of this thesis.

Bibliography

[ACHARYA 1999] Anurag Acharya and Sanjeev Setia. Availability and Utility of Idle Memory in Workstation Clusters. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Atlanta, GA, USA, May 1999, pp. 35-46.

[AKYUREK 1995] Sedat Akyurek and Kenneth Salem. Adaptive Block Rearrangement. *ACM Transactions on Computer Systems*, Vol. 13, No. 2, May 1995, pp. 89-121.

[ATM 1999] ATM Forum. ATM Forum Traffic Management Specification Version 4.1. *AF-TM-0121.000*, March 1999.

[BALA 1994] Kavita Bala, M. Frans Kaashoek and William E. Weihl. Software Prefetching for Translation Lookaside Buffers. *Proc. 1st USENIX Symposium on Operating System Design and Implementation (OSDI)*, Monterey, CA, USA, November 14-17, 1994, pp 243-254.

[BARISH 2000] Greg Barish, Craig A. Knoblock and Steven Minton. Speculative Execution for Information Agents. *Proc. 17th National Conference on Artificial Intelligence (AAAI)*, Austin, TX, USA, August 2000.

[BERNSTEIN 1981] Philip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 185-221.

[BHARGAVA 1999] Bharat Bhargava. Concurrency Control in Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, No. 1, January/February 1999, pp. 3-16.

[BLAKE 1998] Steven Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An Architecture for Differentiated Services. *RFC 2475*, December 1998.

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

[BRUNO 1999] John Bruno, José Brustoloni, Eran Gabber, Banu Özden, and Abraham Silberschatz. Retrofitting Quality of Service into a Time-Sharing Operating System. *Proc. USENIX 1999 Annual Technical Conference*, Monterey, CA, USA, June 1999, pp. 15-26.

[BRUSTOLONI 1996] Jose Brustoloni and Peter. Steenkiste. Effects of Buffering Semantics on I/O Performance. *Proc. 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996, pp 277-291.

[CHANG 1999] Fay Chang and Garth A. Gibson. Automatic I/O Hint Generation through Speculative Execution. *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, USA, February 1999, pp. 1-14.

[CHEN 1998] Mike Chen and Kunle Olukotun. Exploiting Method-Level Parallelism in Single-Threaded Java Programs. *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1998, pp. 176-184.

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

[CLARK 1988] David Clark. The Design Philosophy of the DARPA Internet Protocols. *Computer Communication Review*, Vol. 18, No. 4, 1988, pp. 106-114.

[CLARK 1992] Henry Clark and Bruce McMillin. DAWGS - A Distributed Compute Server Utilizing Idle Workstations. *Journal of Parallel and Distributed Computing*, Vol. 14, 1992, pp 175-186.

[CLARK 1998] 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.

[COHEN 2000] Edith Cohen and Haim Kaplan. Prefetching the Means for Document Transfer: A New Approach for Reducing Web Latency. *Proc. 19th IEEE INFOCOM*, Tel Aviv, Israel, March 2000, pp. 854-863.

[DOUGAN 1999] Cort Dougan, Paul Mackerras and Victor Yodaiken. Optimizing the Idle Task and Other MMU Tricks. *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999, pp. 229-237.

[DOUGLIS 1991] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience (SPE)*, Vol. 21, No. 8, 1991, pp. 757-785.

[DRUSCHEL 1996] Peter Druschel and Gaurav Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. *Proc. 2nd USENIX Symposium on Operating System Design and Implementation (OSDI)*, Seattle, WA, USA, October 1996, pp. 261-275.

[EGGERT 1999] Lars Eggert and John Heidemann. Application-Level Differentiated Services for Web Servers. *World Wide Web Journal*, Volume 3, Issue 2, 1999, pp. 133-142.

[FEELEY 1995] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy and Chandroman A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. *Proc. 15th ACM Symposium on Operating System Principles (SOSP)*, Copper Mountain, CO, USA, December 1995, pp. 201-212.

[FORD 1996] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, USA, October 1996, pp. 91-105.

[FRANKLIN 1996] Manoj Franklin and Gurindar S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, Vol. 45, No. 5, May 1996, pp. 552-571.

[GOLDING 1995] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Proc. USENIX Technical Conference*, January 1995, pp. 201-212.

[GOPAL 1998] Sridhar Gopal, T.N. Vijaykumar, James E. Smith and Gurindar S. Sohi. Speculative Versioning Cache. *Proc. 4th Symposium on High-Performance Computer Architecture*, February 1998, pp. 195-205.

[GUPTA 1997] Alok Gupta, Dale O. Stahl and Andrew B. Whinston. Priority Pricing of Integrated Services Networks. *Internet Economics*, L. W. McKnight and J. P. Bailey (editors), MIT Press, 1997, pp. 323-352.

[HARITSA 1992] Jayant R. Haritsa, Michael J. Carey and Miron Livny. Data Access Scheduling in Firm Real-Time Databases. *Real-Time Systems*, Vol. 4, No. 3, 1992, pp. 203-241.

[HARKINS 1998] Dan Harkins and D. Carrell. The Internet Key Exchange (IKE). *RFC 2409*, November 1998.

[HAYES 1998] Brian Hayes. Collective Wisdom. *American Scientist*, Vol. 86. No. 2, March-April 1998, pp. 118-122.

[HULL 2000] Richard Hull, Francois Llirbat, Bharat Kumar, Ganz Zhou, Gouzhu Dong and Jianwen Su. Optimization Techniques for Data-Intensive Decision Flows. *Proc. 16th International Conference on Data Engineering (ICDE)*, San Diego, CA, USA, March 2000, pp 281-292.

[JACOBSON 1999] Van Jacobson, Kathleen Nichols and Kedarnath Poduri. An Expedited Forwarding PHB. *RFC 2598*, June 1999.

[JINMEI 1998] Tatuya Jinmei, Kazu Yamamoto, Jun-ichiro Hagino, Munechika Sumikawa, Yoshinou Inoue, Kazushi Sugyo and Soichi Sakane. An Overview of the KAME Network Software: Design and Implementation of the Advanced Internetworking Platform. *Proc. 9th Annual Conference of the Internet Society (INET'99)*, San Jose, CA, USA, 1998.

[KAMP 2000] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the Omnipotent Root. *Proc. System Administration and Networking Conference (SANE)*, Maastricht, The Netherlands, May 2000.

[KING 1990] Richard P. King. Disk Arm Movement in Anticipation of Future Requests. *ACM Transactions on Computer Systems*, Vol. 8, No. 3, 1990, pp. 214-229.

[KOHLER 1981] Walter H. Kohler. A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. *ACM Computing Surveys*, Vol. 13, No.2, June 1981, pp. 149-183.

[KORPELA 2001] 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.

[KOUSSIH 1999] Samir Koussih, Anurag Acharya and Sanjeev Setia. Dodo: A User-level System for Exploiting Idle Memory in Workstation Clusters. *Proc. 8th IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, Redondo Beach, CA, USA, August 1999, pp. 301-308.

[KWAK 1999] Hantak Kwak, Ben Lee, Ali R. Hurson, Suk-Han Yoon and Woo-Jong Hahn. Effects of Multithreading on Cache Performance. *IEEE Transactions on Computers*, Vol. 48, No. 2, February 1999, pp. 176-184.

[LAKSHMAN 1998] K. Lakshman, Raj Yavatkar and Raphael Finkel. Integrated CPU and Network-I/O QoS Management In An Endsystem. *Computer Communications*, Vol. 21, No. 4, April 1998, pp. 325-333.

[LAMPSON 1980] 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.

[LEHEY 2001] Greg Lehey. Improving the FreeBSD SMP Implementation. *FREENIX Track: 2001 USENIX Annual Technical Conference*, Boston, MA, USA, June 25-30.

[LESLIE 1996] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications (JSAC)*, Vol. 14, No. 7, September 1996, pp. 1280-1297.

[LEVER 2000] Chuck Lever, Marius Aamodt Eriksen and Stephen P. Molloy. An analysis of the TUX web server. *CITI Technical Report 00-8*, Center for Information Technology Integration, University of Michigan, November 16, 2000.

[LINDSTROM 2000] Jan Lindstrom and Kimmo Raatikainen. Using Importance of Transactions and Optimistic Concurrency Control in Firm Real-Time Databases. *Proc. 7th International Conference on Real-Time Systems and Applications (RTCSA '00)*, Cheju Island, South Korea, December 12-14, 2000.

[LIZTKOW 1988] Michael J. Liztkow, Miron Livny and Matt W. Mutka. Condor - A Hunter of Idle Workstations. *Proc. 8th International Conference on Distributed Computing Systems*, San Jose, CA, USA, June 1988, pp. 104-111.

[MARKATOS 1996] Evangelos P. Markatos and George Dramitinos. Implementation of a Reliable Remote Memory Pager. *Proc. 1996 USENIX Annual Technical Conference*, Berkeley, CA, USA, January 1996, pp. 177-190.

[MATTHEWS 1997] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang and Thomas E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint Malo, France, October 5-8, 1997, pp. 238-251.

[MCKUSICK 1999] Marshall Kirk McKusick and Gregory R. Granger. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. *Proc. Freenix Track of the 1999 USENIX Annual Technical Conference*, Monterey, CA, USA, June 6-11, 1999, pp. 1-17.

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

[MOGUL 1990] Jeffrey C. Mogul and Stephen Deering. Path MTU Discovery. *RFC 1191*, November 1990.

[MOGUL 1997] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, Vol. 15, No. 3, August 1997, pp. 217-252.

[MOLANO 1998] Anastasio Molano, Ragunathan Rajkumar and Kanaka Juvva. Dynamic Disk Bandwidth Management and Metadata Pre-fetching in a Real-Time Filesystem. *Proc. 10th IEEE Euromicro Workshop on Real-Time Systems*, Berlin, Germany, June 17-1, 1998, pp. 203-213.

[MOSBERGER 1996] David Mosberger and Larry L. Peterson. Making Paths Explicit in the Scout Operating System. *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, USA, October 1996, pp. 153-168.

[MOWRY 1996] Todd C. Mowry, Angela K. Demke and Orran Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, USA, October 1996, pp. 3-17.

[MOWRY 1998] Todd C. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *ACM Transactions on Computer Systems*, Vol. 16, No. 1, February 1998, pp. 55-92.

[MUMOLO 1999] Enzo Mumolo. Prediction of Disk Arm Movements in Anticipation of Future Requests. *Proc. 7th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, College Park, MD, USA, October 24-28, 1999, pp. 305-312.

[MUTTKA 1987] Matt W. Mutka and Miron Livny. Profiling Workstations' Available Capacity For Remote Execution. *Proc. 12th IFIP WG 7.3 Symposium on Computer Performance*, Brussels, Belgium, December 1987, pp. 529-544.

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

[NAHRSTEDT 1996] 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.

[NARTEN 1992] Thomas Narten and Raj Yavatkar. Remote Memory as a Resource in Distributed Systems. *Proc. 3rd IEEE Workshop on Operating Systems*, Key Biscane, FL, USA, April 23-24, 1992, pp. 132-136.

[NICHOLS 1987] David A. Nichols. Using Idle Workstations in a Shared Computing Environment. Proc. 11th ACM Symposium on Operating Systems Principles (SOSP), Austin, TX, USA, November 1987, pp. 5-12.

[NIEH 1993] Jason Nieh, James G. Hanko, J. Duane Northcutt and Gerald A. Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. *Proc. 4th Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Lancaster, UK, November 1993.

[OZAWA 1995] Toshihiro Ozawa, Yasunori Kimura and Shin'ichiro Nishizaki. Cache Miss Heuristics and Preloading Techniques for General-Purpose Programs. *Proc. 28th ACM International Symposium on Microarchitecture (MICRO)*, Ann Arbor, MI, USA, November 1995, pp. 243-248.

[PADMANABHAN 1996] 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.

[PATTERSON 1995] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky and Jim Zelenka. Informed Prefetching and Caching. *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, USA, December 3-6, 1995, pp. 79-95.

[PIERCE 1994] Jim Pierce and Trevor Mudge. The Effect of Speculative Execution on Cache Performance. *Proc. 8th Parallel Processing Symposium*, April 1994, pp. 172-179.

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

[POSTEL 1981] Jon Postel. DARPA Internet Protocol Specification. *RFC 791*, September 1981.

[RASHID 1988] Richard F. Rashid, Avadis Tevanian, Michael Young, David B. Golub, Robert V. Baron, David L. Black, William J. Bolosky, Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, Vol. 37, No. 8, 1988, pp. 896-907.

[REZNICK 1993] Larry Reznick. Using cron and crontab. *Sys Admin: The Journal for UNIX Systems Administrators*, Vol. 2, No. 4, July/August 1993, pp. 29-34.

[RICHARDSON 2001] Michael Richardson, D. Hugh Redelmeier and Henry Spencer. A method for doing opportunistic encryption with IKE. *Work In Progress* (`draft-richardson-ipsec-opportunistic-02.txt`), September 2001.

[SANDBERG 1985] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. *Proc. USENIX Summer Technical Conference*, Portland, OR, USA, June 1985, pp. 119-130.

[SCHIMMEL 1994] Curt Schimmel. UNIX Systems for Modern Architectures. Addison-Wesley, 1994.

[SPRUNT 1988] Brinkley Sprunt, David Kirk and Lui Sha. Priority-Driven, Preemptive I/O Controllers for Real-Time Systems. *Proc. IEEE 15th Annual International Symposium on Computer Architecture*, May/June 1988, pp. 152-159.

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

[TANDIARY 1996] Fredy Tandiary, Suraj C. Kothari, Ashish Dixit, and E.Walter Anderson. Batrun: Utilizing Idle Workstations for Large-scale Computing. *IEEE Parallel and Distributed Technology*, Vol. 4, No. 2, 1996, pp. 41-48.

[THEIMER 1985] Marvin M. Theimer, Keith A. Lantz and David R. Cheriton. Preemptable Remote Execution Facilities for the V-System. *Proc. 10th ACM Symposium on Operating Systems Principles (SOSP)*, December 1985, pp. 2-12; published as *Operating Systems Review*, Vol. 19, No. 5.

[THIBODEAU 1998] Jan Thibodeau (editor). The Basic Guide to Frame Relay Networking. Frame Relay Forum, Fremont, CA, USA, 1998.

[THOMASIAN 1998] Alexander Thomasian. Concurrency Control: Methods, Performance and Analysis. *ACM Computing Surveys*, Vol. 30, No. 1, March 1998, pp. 70-119.

[TOKUDA 1990] Hideyuki Tokuda, Tatsuo Nakajima and Prithvi Rao. Real-Time Mach: Towards a Predictable Real-Time System. *Proc USENIX Mach Workshop*, Burlington, VT, USA, October 1990, pp. 73-82.

[TOUCH 1992] Joseph D. Touch. Mirage: A Model for Latency in Communication. *Ph.D. Dissertation*, MS-CIS-92-42, DSL-11, Department of Computer and Information Science, University of Pennsylvania, January 1992.

[TOUCH 1994] Joseph D. Touch. Defining High Speed Protocols: Five Challenges and an Example That Survives the Challenges. *IEEE Journal on Selected Areas in Communications (JSAC)*, Vol. 13, No. 5, June 1995, pp. 828-835.

[TOUCH 1995] Joseph D. Touch and David J. Farber. An Experiment in Latency Reduction. *Proc. IEEE INFOCOM*, Toronto, Canada, June 1994, pp. 175-181.

[TOUCH 1998] Joseph D. Touch and Amy S. Hughes. The LSAM Proxy Cache - a Multicast Distributed Virtual Cache. Computer Networks and ISDN Systems, Vol. 30, No. 22-23, November 1998, pp. 2245-2252.

[WALLACE 1998] Steven Wallace, Brad Calder and Dean M. Tullsen. Threaded Multiple Path Execution. *Proc. 25th ACM Symposium on Computer Architecture*, June/July 1998, pp. 238-249.

[WYCKOFF 1998] 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.

[YEE 1993] Jenn-Jong Yee, Ming-Dong Feng and Chung-Kwong Yuen. Speculative Processing Mechanisms in a Parallel Lisp Machine: BIDDLE. *Proc. 26th Hawaii International Conference on System Sciences*, Vol. 1, January 1993, pp. 457-465.

[YU 1994] Philip S. Yu, Kun-Lung Wu, Kwei-Jay Lin and Sang H. Son. On Real-Time Databases: Concurrency Control and Scheduling. *Proc. IEEE, Special Issue on Real-Time Systems*, January 1994, pp. 140-157.