USENIX Association

Proceedings of the First Symposium on Networked Systems Design and Implementation

San Francisco, CA, USA
March 29–31, 2004

SWAP: A Scheduler With Automatic Process Dependency Detection

Haoqiang Zheng and Jason Nieh
Department of Computer Science
Columbia University
{hzheng,nieh}@cs.columbia.edu

Abstract

We have developed SWAP, a system that automatically detects process dependencies and accounts for such dependencies in scheduling. SWAP uses system call history to determine possible resource dependencies among processes in an automatic and fully transparent fashion. Because some dependencies cannot be precisely determined, SWAP associates confidence levels with dependency information that are dynamically adjusted using feedback from process blocking behavior. SWAP can schedule processes using this imprecise dependency information in a manner that is compatible with existing scheduling mechanisms and ensures that actual scheduling behavior corresponds to the desired scheduling policy in the presence of process dependencies. We have implemented SWAP in Linux and measured its effectiveness on microbenchmarks and real applications. Our results show that SWAP has low overhead, effectively solves the priority inversion problem and can provide substantial improvements in system performance in scheduling processes with dependencies.

1 Introduction

Modern applications often consist of a number of cooperating processes in order to achieve a higher degree of modularity, concurrency, and performance. Applications of this type span a broad range from high-performance scientific parallel applications to desktop graphical computing applications. Interactions among the cooperating processes often result in dependencies such that a certain process cannot continue executing until some other processes finish certain tasks. However, operating systems today often ignore process dependencies and schedule processes independently. This can result in poor system performance due to the actual scheduling behavior contradicting the desired scheduling policy.

Consider priority scheduling, the most common form of scheduling used today in commercial operating systems for general-purpose and real-time embedded systems. The basic priority scheduling algorithm is simple: given a set of processes with assigned priorities, run the process with the highest priority. However, when processes share resources, resource dependencies among processes can arise that prevent the scheduler from running the highest priority process, resulting in priority inversion [5]. For example, suppose there are three processes with high, medium, and low priority such that the high priority process is blocked waiting for a resource held by the low priority process. A priority scheduler would run the medium priority process, preventing the low priority process from running to release the resource, thereby preventing the high priority process from running as well. This situation is particularly problematic because the medium priority process could run and prevent the high priority process from running for an unbounded amount of time. Priority inversion can critically impact system performance, as demonstrated in the case of the NASA Mars Pathfinder Rover [12] when priority inversion caused repeated system resets and drastically limited its ability to communicate back to the Earth.

Because priority inversion can cause significant performance problems, much work has been done to address this issue [1, 5, 15, 17]. The general idea behind these approaches is to boost the priority of a low priority process holding the resource so that it can run and release the resource to get out of the way of a high priority process waiting on the resource to run. However, there are four important limitations that occur in practice with such approaches in the context of general-purpose operating systems. First, these approaches focus on mutex resources only and do not address other potential resource dependencies among processes. For instance, a high priority X window application can suffer priority inversion while waiting on the X server to process requests from other lower priority X applications without any dependencies on mutexes [9]. Second, these approaches typically assume that it is possible to precisely determine dependencies among processes and do not consider dependencies such as those involving UNIX signals and System V IPC semaphores where no such direct correlation exists. Third, these approaches generally assume that priorities are static whereas priorities are often adjusted dynamically by scheduling policies in modern operating systems. Fourth, implementing these approaches for a given resource in a commercial operating system can require adding detailed resource-specific usage information and numerous modifications to many parts of a complex operating system.

We have developed *SWAP*, a *S*cheduler *W*ith *A*utomatic process de*P*endency detection, to effectively account for process dependencies in scheduling in the context of general-purpose operating systems. Rather than focusing on process dependencies arising from mutex resources, SWAP's dependency detection mechanism tracks system call history to determine a much broader range of possible resource dependencies among processes, including those that arise from widely used interprocess communication mechanisms. Because some dependencies cannot be precisely determined, SWAP associates a confidence level with each dependency that is dynamically adjusted using feedback from process blocking behavior. SWAP introduces a general dependency-driven scheduling mechanism that can use imprecise dependency information to schedule processes to run that are determined to be blocking high priority processes. SWAP scheduling is compatible with existing scheduling mechanisms. It is more general than popular priority inheritance approaches [5, 15] and can be used with schedulers that dynamically adjust priorities or non-priority schedulers. Furthermore, SWAP automatically accounts for process dependencies in scheduling without any intervention by application developers or end users. We have implemented SWAP in Linux and measured its effectiveness on both microbenchmarks and real applications. Our experimental results demonstrate that SWAP operates with low overhead and provides substantial improvements in system performance when scheduling processes with dependencies.

This paper presents the design and implementation of SWAP. Section 2 discuses related work. Section 3 describes the SWAP automatic dependency detection mechanism. Section 4 describes SWAP dependency-driven scheduling. Section 5 presents performance results that quantitatively measure the effectiveness of a Linux SWAP implementation using both microbenchmarks and real application workloads. Finally, we present some concluding remarks.

2 Related Work

Lampson and Redell discussed the priority inversion problem more than two decades ago and introduced priority inheritance to address the problem [5]. Using priority inheritance, a process holding a resource inherits the highest priority of any higher priority processes blocked waiting on the resource so that it can run, release the resource, and get out of the way of the higher priority processes. Priority inheritance assumes that priorities are static while they are inherited because recalculating the inherited priority due to dynamic priority changes is too complex. Priority inheritance addresses the priority inversion problem assuming the resource dependency is known; it does not address the underlying issue of determining resource dependencies.

Sha, et. al. developed priority ceilings [15] to reduce blocking time due to priority inversion and avoid deadlock in real-time systems. However, priority ceilings assume that the resources required by all processes are known in advance before the execution of any process starts. This assumption holds for some real-time embedded systems, but does not hold for general-purpose systems. Other approaches such as preemption ceilings [1] can also be used in real-time embedded systems but also make assumptions about system operation that do not hold for general-purpose systems. Like priority inheritance, priority ceilings typically assume static priorities to minimize overhead and does not address the issue of determining resource dependencies among processes.

To address priority inversion in the presence of dynamic priorities, Clark developed DASA for explicitly scheduling real-time processes by grouping them based on their dependencies [3]. While the explicit scheduling model is similar to our scheduling approach, DASA needs to know the amount of time that each process needs to run before its deadline in order to schedule processes. While such process information may be available in some real-time embedded systems, this information is generally not known for processes in general-purpose systems. DASA also assumes that accurate dependency information is provided. It does not consider how such information can be obtained, and can fail with inaccurate dependency information.

Sommer discusses the importance of removing priority inversion in general-purpose operating systems and identifies the need to go beyond previous work which focused almost exclusively on the priority inversion problem for mutex resources [17]. Sommer notes the difficulty of addressing priority inversion for non-mutex resources when it is difficult if not impossible to determine precisely on which process a high priority dependent process is waiting. Sommer proposes a priority-inheritance approach for addressing priority inversion due to system calls, but only implemented and evaluated an algorithm for a single local procedure system call in Windows NT. Sommer did not address general interprocess communication mechanisms that can result in priority inversion and does not consider the impact of dynamic priority adjustments.

Steere, et. al. developed a feedback-based resource reservation scheduler that monitors the progress of applications to guide resource allocation [18]. The scheduler allocates resources based on reservation percentages instead of priorities to avoid explicit priority inversion. Symbiotic interfaces were introduced to monitor application progress to derive the appropriate assignment of scheduling parameters for different applications based on their resource requirements in the presence of application dependencies. However, applications need to be modified to explicitly use the interfaces for the system to monitor progress effectively.

Mach's scheduler handoff mechanism [2], Lottery scheduling's ticket transfers [19] and doors [8] are mechanisms whereby applications can deal with process dependencies by explicitly having one process give its allocated time to run to another process. However, these handoff mechanisms typically require applications to be modified to explicitly use them. Applications also need to identify and know which processes to run. These mechanisms are not designed to resolve priority inversion in general and do not resolve priority inversions due to dependencies that are not explicitly identified in advance.

Co-scheduling mechanisms have been developed to improve the performance of parallel applications in parallel computing environments [4, 10, 16]. These mechanisms try to schedule cooperating processes or threads belonging to the same parallel application to run concurrently. This reduces busy waiting and context switching overhead and improves the degree of parallelism that can be used by the application. Because many of these applications are written using parallel programming libraries, these libraries can be modified to implement co-scheduling. Co-scheduling mechanisms focuses on supporting fine-grained parallel applications. They typically do not support multi-application dependencies and do not address the problem of uniprocessor scheduling in the presence of process dependencies.

3 Automatic Dependency Detection

SWAP introduces a mechanism that automatically detects potential process dependencies by leveraging the control flow structure of commodity operating systems. In commodity operating systems such as Linux, process dependencies occur when two processes interact with each other via the interprocess communication and synchronization mechanisms provided by the operating system. These mechanisms are provided by the operating system as system calls. This suggests a simple idea that SWAP uses for detecting resource dependencies among processes: if a process is blocked because of a process dependency, determine the system call it was executing and use that information to determine what resource the process is waiting on and what processes might be holding the given resource.

Based on this idea, SWAP uses a simple resource model to represent process dependencies in a system. The model contains three components: resources, resource requesters, and resource providers. A resource requester is simply a process that is requesting a resource. A resource provider is a process that may be holding the requested resource and therefore can provide the resource by releasing it. If a certain resource is requested but is not available, the resource requesters will typically need to block until the resource is made available by the resource providers. SWAP uses this simple yet powerful model to represent almost all possible dependency relationships among processes. SWAP applies this model to operating system resources to determine dependencies resulting from interprocess communication and synchronization mechanisms.

An assumption made by SWAP is that resources are accessed via system calls. While this is true for many resources, one exception is the use of memory values for synchronization, most notably user space shared memory mutexes. User space mutexes may simply spin wait to synchronize access to protected resources. Since no system call is involved when accessing this kind of mutex, SWAP does not detect this kind of dependency relationship. However, thread library mutex implementations such as pthreads in Linux do not allow spin waiting indefinitely while waiting for a mutex. Instead, they allow spin waiting for only a time less than the context switch overhead then block. Given that context switch times in modern systems are no more than a few microseconds, the time spent busy waiting and the time not accounted for by SWAP is relatively small. For example, the Linux pthread mutex implementation will spin wait for only 50 CPU cycles before blocking [6]. SWAP focuses instead on process dependencies that can result in processes blocking for long periods of time.

3.1 SWAP Resource Model

In the SWAP resource model, each resource has a corresponding resource object identified by a tuple consisting of the resource type and the resource identifier. The resource identifier can consist of an arbitrary number of integers. The meaning of the resource identifier is specific to the type of resource. For example, a socket is a resource type and the inode number associated with this socket object is used as the resource specific identifier for sockets. SWAP associates with each resource object a list of resource requesters and a list of resource providers.

SWAP creates a resource object when a resource is accessed for the first time and deletes the object when there are no more processes using it, which is when it has no more resource requesters or providers. SWAP efficiently keeps track of resource objects by using a resource object hash table. A resource object is added to the hash table when it is created and removed when it is deleted. The hash key of a resource object is generated from its resource identifier. If a resource identifier consists of N integers, SWAP uses the modulus of the sum of all these integers and the hash table size as the hash key for this resource. Generating the hash key this way allows resource objects to be quickly added into or retrieved from the hash table. Separate chaining is used if resource identifiers hash to the same hash table entry, but such conflicts are infrequent.

SWAP associates a process as a resource requester for a resource object if the process blocks because it is requesting the respective resource. When a process blocks, SWAP needs to first determine what resource the process is requesting. Since resources are accessed via system calls, SWAP can determine the resource being requested by examining the system call parameters. For example, if a process requests data from a socket resource by using the system call `read(sock,...)`, we can identify which socket this process is accessing from the socket descriptor `sock`. When a process executes a system call, SWAP saves the parameters of the system call. If the process blocks, SWAP then identifies the resource being requested based on the saved system call parameters. Once the resource is identified, SWAP appends the process to the resource object's list of resource requesters. When a resource requester eventually runs and completes its system call, SWAP determines that its request has been fulfilled and removes the process from the requester list of the respective resource object.

To allow the resource requester to wake up and continue to run, a resource provider needs to run to provide the respective resource to the requester. To reduce the time the resource requester is blocked, we would like to schedule the resource provider as soon as possible. However, waking up the requester by providing the necessary resource is an action that will not happen until some time in the future. Knowing which process will provide the resource to wake up the requester is unfortunately difficult to determine before the wake up action actually occurs. The process that will provide the resource may not have even been created yet. Furthermore, there may be multiple processes that could serve as the provider for the given requester. For example, a process that is blocked on an IPC semaphore could be provided the semaphore by any process in the system that knows the corresponding IPC key. That is to say, any process existing in the system could in theory be the possible resource provider. In practice though, only a few processes will have the corresponding IPC key and hence the number of possible resource providers may be more than one but is likely to be small.

To identify resource providers, SWAP uses a history-based prediction model. The model is based on the observation that operating system resources are often accessed in a repeating pattern. For example, once a process opens a socket, it will usually make many calls to access the socket before having it closed. This behavior suggests that a process with a history of being a good resource provider is likely to be a future provider of this resource. SWAP therefore treats all past providers of a certain resource as potential future resource providers. SWAP first identifies these potential resource providers and then applies a feedback-based confidence evaluation using provider history to determine which potential providers will actually provide the necessary resource to a requester in the most expedient manner.

SWAP associates a process as a potential resource provider for a resource object the first time the process executes a system call that makes it possible for the process to act as a resource provider. For example, if a process writes data to a socket resource, SWAP identifies this process as a resource provider for the socket. When a process executes a system call, SWAP determines the resource being provided by examining the system call parameters. Once the resource is identified, SWAP appends the process to the resource object's list of resource providers. Note that when SWAP adds a process to the resource provider list, it has only been identified as a potential resource provider. The potential provider must have provided the resource at least once to be identified as a resource provider, but just because it has provided the resource before does not necessarily mean it will provide the resource again.

SWAP uses feedback-based confidence evaluation to predict which process in a list of potential resource providers will provide the necessary resource most quickly to a resource requester. This is quantified by assigning a confidence value to each potential resource provider. A larger confidence value indicates that a provider is more likely to provide the resource quickly to a resource requester. SWAP adjusts the confidence values of potential providers based on feedback from their ability to provide the resource quickly to a requester. If a resource provider is run and it successfully provides the resource to a requester, SWAP will use that positive feedback to increase the confidence value of the provider. If a resource provider is run for a certain amount of time and it does not provide the resource to a requester, SWAP will use that negative feedback to decrease the confidence value of the provider.

We first describe more precisely how SWAP computes the confidence of each resource provider. Section 4 describes how SWAP uses the confidence of resource providers in scheduling to account for process dependencies. SWAP assigns an initial base confidence value K to a process when it is added to a resource object's provider list. SWAP adjusts the confidence value based on feedback within a range from 0 to $2K$. K can be configured on a per resource basis. If a resource provider successfully provides the resource to a requester, SWAP increments its confidence value by one. If a resource provider runs for T time quanta and does not provide the resource to a requester, SWAP decrements its confidence value by one. T can be configured on a per resource basis. A process on the resource provider list will not be considered as a potential resource provider if its confidence drops to zero.

Because it is possible to have cascading process dependencies, a resource provider P_1 for a resource requester P can further be blocked by another process P_2, which is the resource provider for P_1. As a result, P_2 can be indirectly considered as a resource provider for P. SWAP dynamically determines the confidence of an indirect provider as the product of the respective confidence values. Let $C(P, P_1, R)$ be the confidence of provider P_1 for resource R with requester P and $C(P_1, P_2, R_1)$ be the confidence of provider P_2 for resource R_1 with requester P_1. Then the indirect confidence $C(P, P_2, R)$ is computed as $C(P, P_1, R) * C(P_1, P_2, R_1)/K$. Since P_2 is not a direct resource provider for P, if P_2 is run as an indirect resource provider for P, the feedback from that experience is applied to the confidence of P_1. This ensures that a resource provider that is blocked and has multiple providers itself will not be unfairly favored by SWAP in selecting among direct resource providers based on confidence values.

A process will usually remain on the resource provider list until it either terminates or executes a system call that implicitly indicates that the process will no longer provide the resource. For example, a process that closes a socket would no longer be identified as a resource provider for the socket. SWAP does, however, provide a configurable parameter L that limits the number of resource providers associated with any resource object. SWAP groups the providers in three categories: high confidence providers, default confidence providers, and low confidence providers. When this parameter L is set, SWAP only keeps the L providers with highest confidence values. If a new resource provider needs to be added to the provider list and it already has L providers, the provider is added to the end of the default confidence provider list, and an existing provider is removed from the front of the lowest category provider list that is not empty. When there are many potential resource providers, this limit can result in a loss of past history information regarding low confidence providers, but reduces the history information maintained for a resource object.

3.2 SWAP Dependency Detection in Linux

To further clarify how the SWAP resource model can be generally and simply applied to automatically detecting process dependencies in general-purpose operating systems, we consider specifically how the model can be applied to the kinds of resources found in Linux. These resources include sockets, pipes and FIFOs, IPC message queues and semaphores, file locks, and signals. We discuss in detail how sockets, IPC semaphores, and file locks can be identified by SWAP and how the requesters and potential providers of these resources can be automatically detected. Pipes, FIFOs, IPC message queues and signals are supported in a similar fashion but are not discussed further here due to space constraints.

Sockets are duplex communication channels that can involve communication either within a machine or across machines. We only consider the former case since the SWAP resource model only addresses process dependencies within a machine. This includes both UNIX domain sockets and Internet sockets with the same local source and destination address. A socket has two endpoints and involves two processes, which we can refer to as a server and a client. SWAP considers each endpoint as a separate resource so that each socket has two peer resource objects associated with it. These objects can be created when a socket connection is established. For example, when a client calls `connect` and a server calls `accept` to establish a socket connection, SWAP creates a client socket resource object and a server resource object. All system calls that establish and access a socket provide the socket file descriptor as a parameter. SWAP can use the file descriptor to determine the corresponding inode number. SWAP then uses the inode number to identify a socket resource object.

SWAP determines the resource requesters and providers for socket resource objects based on the use of system calls to access sockets. A socket can be accessed using the following system calls: `read`, `write`, `readv`, `writev`, `send`, `sendto`, `sendmsg`, `recv`, `recvfrom`, `recvmsg`, `select`, `poll`, and `sendfile`. When a socket is accessed by a process, the process is added as a resource provider for its socket endpoint and the system call parameters are saved. The process will remain a resource provider for the resource object until it explicitly closes the socket or terminates. If the system call blocks, it means that the process is requesting the peer resource object on the other end of the socket. For example, when a client does a `read` on its endpoint and blocks, it is because the server has not done a `write` on its peer endpoint to make the necessary data available to the client. If the system call blocks, SWAP therefore adds the process as a resource requester for the peer resource object.

System V IPC semaphores provide an interprocess synchronization mechanism. SWAP associates a resource object with each IPC semaphore. An IPC semaphore object is created when a process calls `semget` to create a semaphore. `semget` returns a semaphore identifier which is a parameter used by all system calls that access the semaphore. SWAP therefore uses the semaphore identifier to identify the respective resource object. SWAP determines the resource requesters and providers for semaphores based on the use of the `semop` system call to access the resource. When a semaphore is accessed by a process, the process is added as a resource provider and the system call parameters are saved. The process will remain a resource provider for the resource object until the semaphore is explicitly destroyed or the process terminates. If the system call blocks, the calling process is added as a resource requester of the semaphore object.

File locks provide a file system synchronization mechanism between processes. Linux supports two kinds of file locking mechanisms, fcntl and flock. Since both of these mechanisms work in a similar way and can both provide reader-writer lock functionality, we just discuss how SWAP supports the flock mechanism. SWAP associates a resource object with each file lock. An flock object is created when a process calls `flock` to create a file lock for a file associated with the file descriptor parameter in `flock`. SWAP distinguishes between exclusive locks and shared locks and creates a different flock resource object for each case. Since `flock` is used for all operations on the file lock, the file descriptor is available for all file lock operations. SWAP can therefore use the file descriptor to determine the corresponding inode number. SWAP uses the inode number and a binary value indicating whether the file lock created is shared or exclusive to identify the respective resource object.

SWAP determines the resource requesters and providers for flock resource objects based on the use of system calls to access the resources. A shared flock resource object is accessed when `flock` is called with `LOCK_SH` and the respective file descriptor. When this happens, the calling process is added as a resource provider for the object. If the system call blocks, then some other process is holding the file lock exclusively, which means the process is a resource requester for the exclusive flock object. SWAP therefore adds the process as a resource requester for the exclusive flock resource object. An exclusive flock resource object is accessed when `flock` is called with `LOCK_EX` and the respective file descriptor. When this happens, the calling process is added as a resource provider for the object. If the system call blocks, then some other process is holding the file lock either shared or exclusively, which means the process is a resource requester for the both the exclusive and shared flock object. SWAP therefore adds the process as a resource requester for both flock resource objects. For both shared and exclusive flock resource objects, a process remains a resource provider for the object until it terminates or explicitly unlocks the flock by calling `flock` with `LOCK_UN`.

4 Dependency-Driven Scheduling

SWAP combines the information it has gathered from its dependency detection mechanism with a scheduling mechanism that is compatible with the existing scheduling framework of an operating system but accounts for process dependencies in determining which process to run. We describe briefly how a conventional scheduler determines which process to run and then discuss how SWAP augments that decision to account for process dependencies.

A conventional scheduler maintains a run queue of runnable processes and applies an algorithm to select a process from the run queue to run for a time quantum. Processes that block become not runnable and are removed from the run queue. Similarly, processes that wake up become runnable and are inserted into the run queue. With no loss of generality, we can view each scheduling decision as applying a priority function to the run queue that sorts the runnable processes in priority order and then selects the highest priority process to execute [14]. This priority model, where the priority of a process can change dynamically for each scheduling decision, can be used for any scheduling algorithm. To account for process dependencies, we would like the priority model to account for these dependencies in determining the priority of each process. Assuming the dependencies are known, this is relatively easy to do in the case of a static priority scheduler where each process is assigned a static priority value. In this case when a high priority process blocks waiting on a lower priority process to provide a resource, the the lower priority process can have its priority boosted by priority inheritance. The scheduler can then select a process to run based on the inherited priorities. However, priority inheritance is limited to static priority schedulers and does not work for more complex, dynamic priority functions.

To provide a dependency mechanism that works for all dynamic priority functions and therefore all scheduling algorithms, SWAP introduces the notion of a *virtual runnable* process. A virtual runnable process is a resource requester process that is blocked but has at least one runnable resource provider or virtual runnable resource provider that is not itself. Note that a resource requester could potentially also be listed as a provider for the resource, but is explicitly excluded from consideration when it is the resource requester. The definition is recursive in that a process's provider can also be virtual runnable. A process that is blocked and has no resource providers is not virtual runnable. A virtual runnable process can be viewed as the root of a tree of runnable and virtual runnable resource providers such that at least one process in the tree is runnable. SWAP makes a small change to the conventional scheduler model by leaving virtual runnable processes on the run queue to be considered in the scheduling decision in the same manner as all other runnable processes. If a virtual runnable process is selected by the scheduler to run, one of its resource providers is instead chosen to run in its place. SWAP selects a resource provider to run in place of a virtual runnable process using the confidence associated with each provider. The confidence of a runnable provider is just its confidence value. The confidence of a virtual runnable provider is its indirect confidence value as described in Section 3.1. If a virtual runnable provider is selected, the confidence values of its providers are examined recursively until a runnable process with the highest confidence value is selected. Once a virtual runnable requester process becomes runnable as a result of being provided the resource, it is simply considered in the scheduling decision like all other runnable processes.

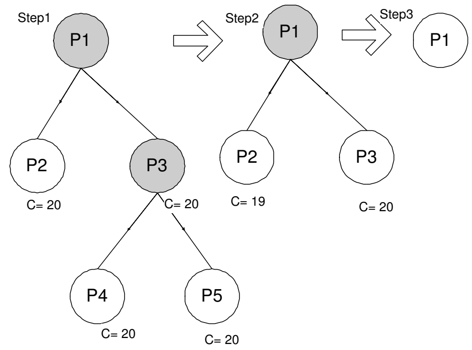

Figure 1: SWAP Scheduling Example

Figure 1 shows an example to further illustrate how SWAP dependency-driven scheduling chooses a resource provider based on confidence. Virtual runnable processes are shaded and runnable processes are not. Suppose at the time the scheduler is called, a virtual runnable process P1 is currently the highest priority process. In Step 1 in Figure 1, P1 is blocked because it is requesting resource R1 which has two resource providers P2 and P3. P3 is also blocked because of it is requesting resource R2 which has two providers P4 and P5. In this example, P4 needs to run for 40 ms to produce resource R2, and P3 needs to run for 20 ms to produce resource R1. P2 and P5 do not actually provide the respective resources this time. In this example, we assume the confidence quantum T described in Section 3.1 is 100 ms. SWAP needs to decide which process among P2, P4, and P5 is likely to wake up P1 early. It decides by using the confidence value associated with each provider. Assume there is no previous confidence history for these resources so that all confidence values are equal to a base confidence K of 20. In deciding which process to run in place of virtual runnable process P1, SWAP then determines P2's confidence $C(P1, P2, R1)$ as 20, P4's indirect confidence $C(P1, P4, R1)$ as $C(P1, P3, R1) * C(P3, P4, R2)/K = 20$, and P5's indirect confidence $C(P1, P5, R1)$ as $C(P1, P3, R1)*C(P3, P5, R2)/K = 20$. The first provider with the highest confidence value is P2, so it will be selected to run first. It will run for 100 ms and then receive negative feedback because it does not wake up P1 after one confidence time quantum is used. $C(P1, P2, R1)$ will then become 19. P4 becomes the first provider with the highest confidence, so it will be selected to run. After P4 runs for 40 ms, it provides the resource for P3, which wakes up P3, resulting in the new situation shown in Step 2 in Figure 1. At this time, P3 can be the selected to run for virtual runnable P1 since it has a higher confidence value than P2. It will continue to run for another 20 ms and eventually wake up P1, resulting in P1 being runnable as shown in Step 3 in Figure 1.

SWAP's use of virtual runnable processes provides a number of important benefits. By introducing a new virtual runnable state for processes, SWAP leverages an existing scheduler's native decision making mechanism to account for process dependencies in scheduling. SWAP implicitly uses an existing scheduler's dynamic priority function without needing to explicitly calculate process priorities which could be quite complex. SWAP does not need to be aware of the scheduling algorithm used by the scheduler and does not need to replicate the algorithm as part of its model in any way. As a result, SWAP can be integrated with existing schedulers with minimal scheduler changes and can be easily used with any scheduler algorithm, including commonly used dynamic priority schemes. By using a confidence model, SWAP allows a scheduler to account for process dependency information in scheduling even if such information is not precisely known.

One issue with using the confidence model in scheduling is that it may unfairly favor processes with high confidence. If SWAP keeps selecting a given provider, its confidence will continue to rise if it never fails to provide the necessary resource. This behavior is consistent with SWAP's objective to run the process that can most quickly provide the resource for a virtual runnable process. However, there may be other providers that could also provide the resource just as quickly but are not selected by the confidence model because they were not initially given the chance to run. We have not seen this issue in practice for three reasons. First, the process that runs in place of a virtual runnable process is still charged by the underlying scheduler for the time it runs, so running it sooner results in it running less later. Second, a process that provides a resource typically only runs for a short duration until it provides the resource. A process selected using the confidence model also must provide the resource within a confidence quantum of T which is usually small, otherwise other resource providers will be selected. Third, the confidence model is only used when the resource requester blocks because the resource is not available and there are multiple potential providers. If a requester does not need to block or there is only one provider, the confidence model is not used.

While the SWAP approach provides important advantages, we also note that it can be limited by the decision making algorithm of an existing scheduler in the context of multiprocessors. To support scalable multiprocessor systems, schedulers typically associate a separate run queue with each CPU to avoid lock contention on a centralized run queue. Since CPU scheduling decisions are decoupled from one another, the process that is selected to run may not be the most optimal. In a similar manner, if a virtual runnable process is selected by the scheduler on a CPU, it may not be globally the best virtual runnable process to select. It is possible that the virtual runnable process selected is not the one with the highest priority across all CPUs according to the native scheduler's priority model. Other approaches could be used to select a globally more optimal virtual runnable process, but would incur other disadvantages. One could provide separately managed queues for virtual runnable processes, but this would require duplicating scheduler functionality and increasing complexity. If a single queue was used for virtual runnable processes, this could also impact scheduler scalability.

5 Experimental Results

We have implemented a SWAP prototype in Red Hat Linux 8.0 which runs the Linux 2.4.18-14 kernel. The SWAP automatic dependency detection mechanisms were designed in such a way that they can be implemented as a loadable kernel module that does not require any changes to the kernel. The SWAP dependency-driven scheduling mechanism was also largely implemented in the same kernel module, but it does require some changes to the kernel scheduler. These changes only involved adding about 15 lines of code to the kernel and were localized to only a couple of kernel source code files related to the kernel scheduler. As discussed in Section 3.1, SWAP provides three configurable parameters, the default maximum number of providers per resource L, the initial base confidence value K, and the confidence quantum T. In our SWAP implementation, the default values of L, K, and T were set to 20, 20, and 20 ms, respectively.

We have used our SWAP prototype implementation in Linux to evaluate its effectiveness in improving system performance in the presence of process dependencies. We compared Linux SWAP versus vanilla Redhat Linux 8.0 using both microbenchmarks and real client-server applications. Almost all of our measurements were performed on an IBM Netfinity 4500R server with a 933 MHz Intel PIII CPU, 512 MB RAM, and 100 Mbps Ethernet. We also report measurements obtained on the same machine configured with two CPUs enabled. We present some experimental data from measurements on four types of application workloads. Section 5.1 presents results using a client-server microbenchmark workload to measure the overhead of SWAP and illustrate its performance for different resource dependencies. Section 5.2 presents results using a multi-server microbenchmark to measure the effectiveness of SWAP when multiple processes can be run to resolve a resource dependency. Section 5.3 presents results using a thin-client computing server workload to measure the effectiveness of SWAP in a server environment supporting multiple user sessions. Section 5.4 presents results using a chat server workload to measure the effectiveness of SWAP in a multiprocessor server environment supporting many chat clients.

5.1 Client-server Microbenchmark

The client-server microbenchmark workload consisted of a simple client application and server application that are synchronized to start at the same time. Both the client and the server run in a loop of 100 iterations. In each iteration, the client waits for the server to perform a simple bubblesort computation on a 4K array and respond to the client via some method of communication, resulting in a dependency between client and server. We considered six common communication mechanisms between client and server:

- Socket (SOCK): Server computes and writes a 4 KB data buffer to a Unix domain socket. Client reads from the socket.
- Pipe/FIFO (PIPE): Server computes and writes a 4 KB data buffer to a pipe. Client reads the data from the pipe.
- IPC message queue (MSG): Server computes and sends a 4 KB data buffer via an IPC message queue. Client receives the data from the message queue.
- IPC semaphores (SEM): Two semaphores called *empty* and *full* are initialized to true and false, respectively. Server waits for *empty* to be true then computes and signals *full* when it completes. Client waits for *full* to be true then signals *empty*. Wait and signal are implemented using the `semop` system call.
- Signal (SIG): Server waits for a signal from the client, computes, and sends a signal to client when it completes its computation. Client waits until it receives the signal.
- File locking (FLOCK): Server uses `flock` to lock a file descriptor while it does its computation and unlocks when it is completed. Client uses `flock` to lock the same file and therefore must wait until server releases the lock.

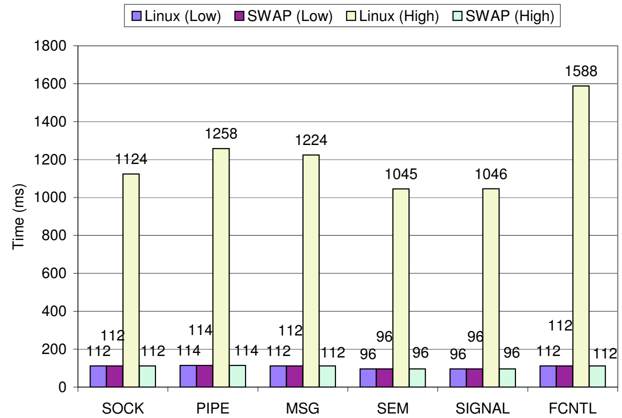

Figure 2: Client-server Microbenchmark Results

We measured the average time it took the client to complete an iteration of each of the six client-server microbenchmarks when using vanilla Linux versus SWAP. For this experiment, we assumed that the client is an important application and is therefore run as a real-time SCHED_FIFO process in Linux. All other processes are run using the default SCHED_-OTHER scheduling policy. In Linux, SCHED_FIFO processes are higher priority than SCHED_OTHER processes and are therefore scheduled to run before SCHED_OTHER processes. We measured the client iteration completion time when there were no other applications running on the system to further compare the overhead of SWAP with vanilla Linux. We then measured the client iteration completion time using vanilla Linux versus SWAP when ten other application processes were running at the same time as the client-server microbenchmark. The application processes were simple while loops imposing additional load on the system. This provides a measure of the performance of vanilla Linux versus SWAP on a loaded system in the presence of process dependencies.

Figure 2 shows the measurements for each of the six client-server microbenchmarks. For low system load, the measurements show that the client iteration completion time for each microbenchmark was roughly 100 ms for both vanilla Linux and SWAP. The client completed quickly in all cases, and the difference between the completion times using SWAP and vanilla Linux were negligible. Figure 3 shows the average total time of executing system calls in each iteration for the six microbenchmarks. This provides a more precise measure of the overhead of SWAP versus vanilla Linux. The extra system call overhead of SWAP ranges from 1.3 μs for the FLOCK microbenchmark to 3.8 μs for the SEM microbenchmark. Although the absolute processing time overhead for SWAP is smallest for the FLOCK microbenchmark, it is the largest percentage overhead of all the microbenchmarks because the overall system call cost of the FLOCK microbenchmark is small.

The extra overhead associated with SWAP is due to the costs of intercepting the necessary system calls for each microbenchmark, examining their parameters, determining which if any processes should be added or removed from the SWAP resource requester and provider lists, and performing dependency driven scheduling when necessary. The difference in SWAP overhead for the different microbenchmarks is due to different numbers of SWAP operations being done in each case. For each iteration of the FLOCK microbenchmark, there are two context switches, resulting in extra SWAP scheduling overhead since it needs to schedule virtual runnable processes. In addition for each iteration, SWAP intercepts two system calls and manages two resource objects for the file lock, one corresponding to exclusive lock access and the other corresponding to shared lock access. For these two system calls, SWAP needs to check the status of resource providers and requesters a total of three times. For each iteration of the SEM microbenchmark, there are also two context switches, resulting in extra SWAP scheduling overhead since it needs to schedule virtual runnable processes. In addition for each iteration, SWAP intercepts four system calls and manages two resource objects corresponding to two IPC semaphores used by the SEM microbenchmark. For these system calls, SWAP needs to check the status of resource providers and requesters a total of eight times. The higher number of resource provider and requester operations for the SEM microbenchmark results in the higher SWAP processing costs compared to the FLOCK microbenchmark. While these microbenchmarks were designed to exercise those system calls that SWAP intercepts, note that SWAP only intercepts a small minority of system calls and only the intercepted system calls incur any additional overhead.

While Figure 3 shows the average overhead due to system calls across all iterations of the microbenchmarks, the cost of a system call can be larger the first time it is called when using SWAP due to extra memory allocation that may occur to create resource objects, resource providers, and resource requesters. For each resource, SWAP creates a 64 byte resource object. When a process is first added as a resource requester for a resource, SWAP creates a 32 byte resource requester object. When a process is first added as a resource provider for a resource, SWAP creates a 32 byte resource provider object. The memory overhead of SWAP grows with the number of resource objects, providers, and requesters that need to be allocated. However, since the associated SWAP objects are all small, and the resulting memory overhead imposed by SWAP ends up also being small.

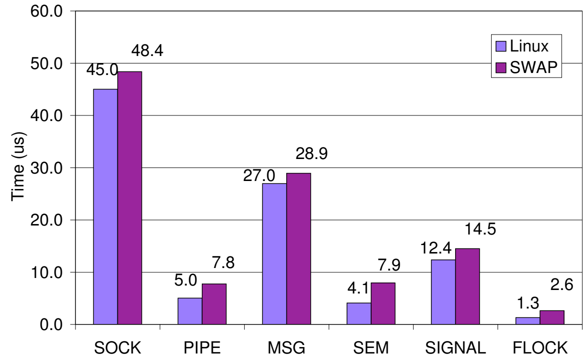

Figure 3: SWAP Overhead

For high system load, Figure 2 shows that the client iteration completion time for each microbenchmark was an order of magnitude better using SWAP versus vanilla Linux. Despite the fact that the client was the highest priority process in the system, the client iteration completion times using vanilla Linux ballooned to over 1 second, roughly ten times worse than for low system load. The problem is that the client depends on the server, which runs at the same default priority as the other processes in the system. Since Linux schedules processes independently, it does not account for the dependency between client and server, resulting in the high priority client process not being able to run. In contrast, the client iteration completion times when using SWAP remained almost the same for both high and low system load at roughly 100 ms for all of the microbenchmarks. The client performance using SWAP for high system load is roughly ten times better than vanilla Linux for high system load and essentially the same as vanilla Linux for low system load. SWAP automatically identifies the dependencies between client and server processes for each microbenchmark and correctly runs the server process ahead of other processes when the high priority client process depends on it.

5.2 Multi-server Microbenchmark

The multi-server microbenchmark workload consisted of a simple client application and five server applications that are started at the same time. The microbenchmark is similar to the SEM microbenchmark described in Section 5.1 with two differences. First, since each server increments the semaphore and there are multiple servers running, the client will only need to wait until one of the servers increments the semaphore before it can run and decrement the semaphore. Second, each of the servers may do a different number of bubblesort computations, resulting in the server processing taking different amounts of time. For this experiment, the five servers repeated the bubblesort computation 2, 5, 5, 10, and 10 times, respectively. As a result, the servers vary in terms of the amount of processing time required before the semaphore is incremented.

We measured the time for the client to complete each of the first 15 loop iterations when using vanilla Linux versus SWAP. For SWAP, we considered the impact of different confidence feedback intervals by using two different intervals, 20 ms and 200 ms. For this experiment, we assumed that the client is an important application and is therefore run as a real-time SCHED_FIFO process in Linux. All other processes in the system are run using the default SCHED_OTHER scheduling policy. We measured the client iteration time when no other applications were running on the system as a baseline performance measure on vanilla Linux and SWAP. We then measured the client completion time using vanilla Linux versus SWAP when ten other application processes were running at the same time as the client-server microbenchmark. The application processes were simple while loops imposing additional load on the system. This provides a measure of the performance of vanilla Linux versus SWAP on a loaded system in the presence of process dependencies.

Figure 4 shows the multi-server microbenchmark measurements. It shows the measured client iteration completion time for each iteration using vanilla Linux and SWAP for both low system load and high system load. In this figure, SWAP-20 is used to denote the measurements done with a SWAP confidence feedback interval of 20 ms and SWAP-200 is used to denote the measurements done with a SWAP confidence feedback interval of 200 ms.

For low system load, Figure 4 shows that client iteration time is roughly the same at 1 second when using SWAP or vanilla Linux for the first iteration. However, the client iteration time when using SWAP is much better than when using vanilla Linux for subsequent iterations. While the client iteration time remains at roughly 1 second for all iterations when using vanilla Linux, the client iteration time drops to about 200 ms when using SWAP, with the iteration time dropping faster using SWAP-200 versus SWAP-20. Of the five servers running, the server running the bubblesort computation twice increments

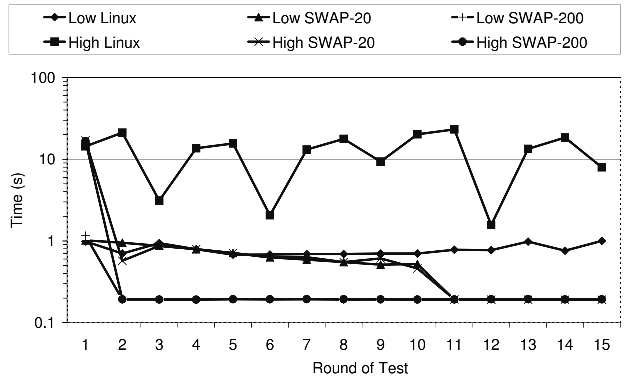

Figure 4: Multi-server Microbenchmark Results

the semaphore the fastest at roughly 200 ms. As a result, the client completes its iteration the fastest when this server is scheduled to run instead of the other servers. The results in Figure 4 show that SWAP eventually finds the fastest server to run to resolve the process dependency between the high priority client and the servers.

It takes the client about 1 second to complete an iteration using vanilla Linux because the default SCHED_OTHER scheduling policy used in the Linux 2.4.18-14 kernel is basically a round-robin scheduling algorithm with a time quantum of 150 ms. The Linux scheduler therefore will end up running each of the five servers in round-robin order until one of the servers increments the semaphore allowing the high-priority client to run and complete an iteration. The fastest server needs to run for 200 ms to increment the semaphore. Therefore, depending on when the fastest server is run in round-robin order, it could take between 800 ms to 1400 ms until the semaphore is incremented and the client can run, which is consistent with the results shown in Figure 4.

On the other hand, using SWAP the client only takes 200 ms to complete an iteration because SWAP's confidence feedback model identifies the fastest server after the first iteration because that server is the one that increments the semaphore and allows the client to run. In subsequent iterations, SWAP gives that server preference to run, resulting in lower client iteration time. Figure 4 also shows that SWAP with a feedback interval of 200 ms will reach the optimal level faster than SWAP with a feedback interval of 20 ms. This is because the confidence value is adjusted one unit for each feedback, which means each positive feedback will make the process run 1 quantum more ahead of the other processes. The larger the quantum, the more benefit a process will receive from a positive feedback. In this sense, it is desirable for the confidence quantum to be as large as possible. However, if the quantum is too large, the dependency-driven scheduler will behave in a FIFO manner, which can result in longer response times. In this case, configuring the confidence time quantum to be 200 ms works well because it is the time needed for the fastest provider to produce the desired resource.

For high system load, Figure 4 shows that client iteration time is roughly the same at 10 seconds when using SWAP or vanilla Linux for the first iteration. However, the client iteration time when using SWAP is significantly better than when using vanilla Linux for subsequent iterations. The reasons for SWAP's better performance for high system load are the same as for low system load, except that the difference between SWAP and vanilla Linux is magnified by the load on the system.

5.3 Thin-client Computing Server

The thin-client computing server workload consisted of VNC 3.3.3 thin-client computing sessions running MPEG video players. VNC [13] is a popular thin-client system in which applications are run on the server and display updates are then sent to a remote client. Each session is a complete desktop computing environment. We considered two different VNC sessions:

- MPEG play: The VNC session ran the Berkeley MPEG video player [11] which displayed a locally stored 5.36 MB MPEG1 video clip with 834 352x240 pixel video frames.
- Netscape: The VNC session ran a Netscape 4.79 Communicator and downloaded and displayed a Javascript-controlled sequence of 54 web pages from a web server. The web server was a Micron Client Pro with a 450 MHz Intel PII, 128 MB RAM, and 100 Mbps Ethernet, running Microsoft Windows NT 4.0 Server SP6a and Internet Information Server 3.0.

We measured the time it took for the respective application in one VNC session to complete when using vanilla Linux versus SWAP. For this experiment, we assumed that the application measured, either the video player or web browser, is important and is therefore run as a real-time SCHED_FIFO process in Linux. All other processes in the system are run using the default SCHED_OTHER scheduling policy. We measured the respective video player and web browser completion times when there were no other applications running on the system to provide a baseline performance measure of each application running on vanilla Linux and SWAP. We then measured each application completion time using vanilla Linux versus SWAP with 50 other VNC sessions running at the same time, each session running the video player application.

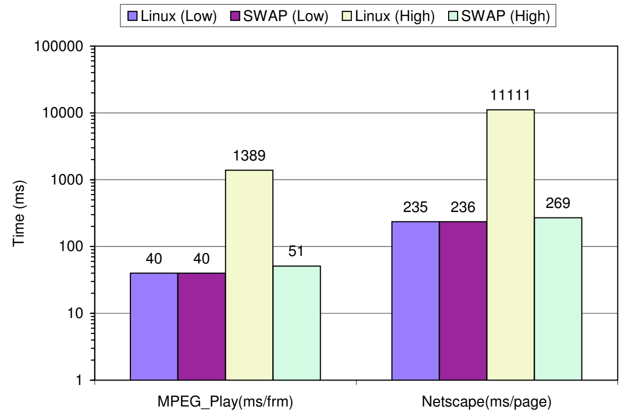

Figure 5: Thin-client Computing Server Benchmark Results

Figure 5 shows the thin-client computing server measurements for the VNC session running MPEG play and the VNC session running Netscape. For low system load, the measurements show that the client completion time for both real applications was roughly the same. The overhead caused by SWAP is less than 0.5%. For high system load, the measurements show that the client completion time for each application was an order of magnitude better using SWAP versus vanilla Linux. Despite the fact that the client was the highest priority process in the system, the video playback rate of MPEG play was only 0.72 frm/s when using vanilla Linux, which means that each video frame took on average 1389 ms to be processed and displayed. In the same situation, it took Netscape more than 11 seconds to download a single web page. In both cases the performance was unacceptable. The problem is that MPEG play and Netscape, as graphics-intensive applications, depend on the X Server to render the video frames and web pages. Since the X server was run at the same default priority as all the other VNC sessions in the system, this will effectively make all the clients depending on it run at low priority also.

On the other hand, Figure 5 shows the performance of both MPEG play and Netscape remained to be satisfactory even under very high system load when using SWAP . This further proves the effectiveness of the automatic dependency detection mechanism and dependency driven scheduler used by SWAP. The small difference between using SWAP for high and low system load can be explained by two factors. First, Linux still doesn't support features like a preemptive kernel which is important to real-time applications. Second, since access to resources such as memory and disk are not scheduled by the CPU scheduler, our SWAP CPU scheduling implementation does not solve performance degradation problems caused by accessing these resources.

5.4 Volano Chat Server

The Chat server workload consisted of VolanoMark 2.1.2 [7], an industry standard Java chat server benchmark configured in accordance with the rules of the Volano Report. VolanoMark creates a large number of threads and network connections, resulting in frequent scheduling and potentially many interprocess dependencies. It creates client connections in groups of 20 and measure how long it takes the clients to take turns broadcasting their messages to the group. It reports the average number of messages transferred by the server per second. For this experiment, all processes were run using the default SCHED_OTHER scheduling policy. We assumed that the chat clients are important and are therefore run as at a higher priority by running them with `nice -20` with all other applications run at the default priority. We measured the VolanoMark performance when there were no other applications running on the system to provide a baseline performance measure on vanilla Linux and SWAP. We then measured the VolanoMark performance with different levels of additional system load. The system load was generated using a simple CPU-bound application. To produce different system load levels, we ran different numbers of instances of the CPU-bound application. We ran VolanoMark using the dual-CPU server configuration. This provides a measures of the performance of vanilla Linux versus SWAP with a resource-intensive server application running on a loaded multiprocessor in the presence of process dependencies. VolanoMark was run using Sun's Java 2 Platform 1.4.0 for Linux which maps Java threads to Linux kernel threads in a one-to-one manner.

Figure 6 shows the performance of VolanoMark for different levels of system load. These results were obtained on the dual-CPU configuration of the server. The system load is equal to the number of additional CPU-bound applications running at the same time. For no additional system load, vanilla Linux averaged 4396 messages per second on VolanoMark test while SWAP averaged 4483. The measurements show that VolanoMark performs roughly the same for both vanilla Linux and SWAP, with the performance of SWAP slightly better. This can be explained by the fact that the Volano clients frequently call `sched_yield`, which allows the CPU scheduler to decide which client should run next. Because SWAP is aware of the dependency relationships among clients, SWAP can make a better decision than vanilla Linux regarding which client should run next.

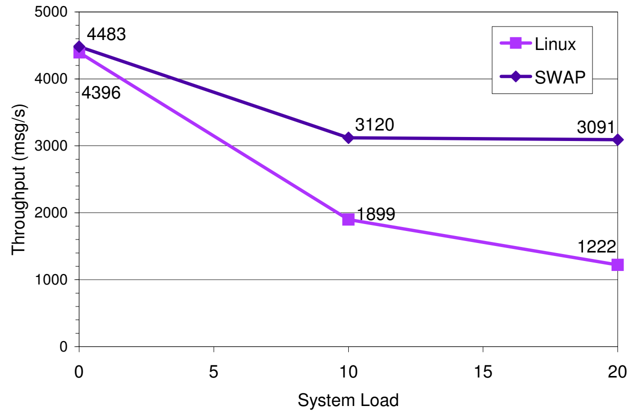

Figure 6: Volano Chat Server Benchmark Results

The performance of vanilla Linux and SWAP diverge significantly with additional system load. At a system load of 20, SWAP provides VolanoMark performance that is more than 2.5 times better than vanilla Linux. Although SWAP does perform much better than vanilla Linux, both systems show degradation in the performance of VolanoMark at higher system load. At a system load of 20, SWAP performance is about 70 percent of the maximum performance while vanilla Linux performance is less than 30 percent of the maximum performance. The degradation in performance on SWAP can be explained because of its reliance on the CPU scheduler to select among runnable and virtual runnable processes and the multiprocessor scheduling algorithm used in the Linux 2.4.18-14 kernel. For SWAP to deliver the best performance, high priority virtual runnable processes should always be scheduled before lower priority runnable processes. However, the Linux scheduler does not necessarily schedule in this manner on a multiprocessor. The Linux scheduler employs a separate run queue for each CPU and partitions processes among the run queues based on the number of runnable processes in each queue. It does not take into account the relative priority of processes in determining how to assign processes to run queues. As a result, for a two-CPU machine, the scheduler can end up assigning high priority processes to one CPU and lower priority processes to another. With SWAP, this can result in high priority virtual runnable processes competing for the same CPU even though lower priority processes are being run on the other CPU. As a result, some high priority virtual runnable processes end up having to wait in one CPU run queue when there are other lower priority CPU-bound applications which end up running on the other CPU.

Since Linux schedules processes independently, it does not account for the dependencies between client and server, resulting in high priority Volano clients not being able to run in the presence of other CPU-bound applications. Linux either delivers poor performance for these clients or places the burden on users to tune the performance of their applications by identifying process dependencies and explicitly raising the priority of all interdependent processes. SWAP instead relieves users of the burden of attempting to compensate for scheduler limitations. Our results show that SWAP automatically identifies the dynamic dependencies among processes and correctly accounts for them in scheduling to deliver better scheduling behavior and system performance.

6 Conclusions

Our experiences with SWAP and experimental results in the context of a general-purpose operating system demonstrate that SWAP is able to effectively and automatically detect process dependencies and accounts for these dependencies in scheduling. We show that SWAP effectively uses system call history to handle process dependencies such as those resulting from interprocess communication and synchronization mechanisms which have not been previously addressed. We also show that SWAP's confidence feedback model is effective in finding the fastest way to resolve process dependencies when multiple potential dependencies exist.

These characteristics of SWAP result in significant improvements in system performance when running applications with process dependencies. Our experimental results show that SWAP can provide more than an order of magnitude improvement in performance versus the popular Linux operating system when running microbenchmarks and real applications on a heavily loaded system. We show that SWAP can be integrated with existing scheduling mechanisms and operate effectively with schedulers that dynamically adjust priorities. Furthermore, our results show that SWAP achieves these benefits with very modest overhead and without any application modifications or any intervention by application developers or end users.

7 Acknowledgments

This work was supported in part by NSF grant EIA-0071954, an NSF CAREER Award, and an IBM SUR Award.

References

[1] Ted Baker. Stack-Based Scheduling of Real-Time Processes. *Real-Time Systems*, 3(1), March 1991.

[2] David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, 1990.

[3] Raymond K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990.

[4] Dror G. Feitelson and Larry Rudolph. Coscheduling Based on Run-Time Identification of Activity Working Sets. *International Journal of Parallel Programming*, 23(2):136–160, April 1995.

[5] Butler W. Lampson and David D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[6] Xavier Leroy. The LinuxThreads Library. Now a part of the glibc GNU C library.

[7] Volano LLC. Volanomark Benchmark. *http://www.volano.com/benchmarks.html*.

[8] Jim Mauro and Richard McDougall. *Solaris Internals: Core Kernel Architecture*. Prentice Hall PTR, first edition, 2000.

[9] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 35–48, Lancaster, U.K., 1993.

[10] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. *International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[11] R. Patel, K. Smith, and B. Liu. MPEG Video in Software: Representation, Transmission, and Playback. In *Proc. High-Speed Networking and Multimedia Computing*, San Jose, California, February 8-10 1994.

[12] Glenn E Reeves. What Really Happened on Mars Rover Pathfinder. *The Risks Digest*, 19, 1997.

[13] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[14] Manfred Ruschitzka and Robert S. Fabry. A Unifying Approach to Scheduling. *Communications of the ACM*, 20(7):469–477, July 1977.

[15] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[16] Patrick G. Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A. Chien. Dynamic Coscheduling on Workstation Clusters. *Lecture Notes in Computer Science*, 1459:231–257, 1998.

[17] S. Sommer. Removing Priority Inversion from an Operating System. In *Proceedings of Nineteenth Australasian Computer Science*, 1996.

[18] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the 3rd Symposium on Operating Systans Design and Implementation (OSDI-99)*, pages 145–158, Berkeley, CA, February 22–25 1999. Usenix Association.

[19] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-share Resource Management*. PhD thesis, Massachusetts Institute of Technology, 1995.