Piglet: A Low-Intrusion Vertical Operating System

Steve Muir, Jonathan Smith\textsuperscript{1}  
{sjmuir,jms}@dsl.cis.upenn.edu  
\textit{Distributed Systems Laboratory}  
University of Pennsylvania

Vertically-structured operating systems are a recent proposal for providing application-level resource management. While this has been accomplished, the issue of how the structure of the operating system affects application behaviour remains largely unexplored.

We introduce the concept of OS intrusion as a measure of the overheads that the operating system's policies and mechanisms impose upon applications. Vertical operating systems address policy intrusion; we analyse the costs of two forms of mechanism intrusion—interrupt handling and system call overhead—in the Linux kernel, and demonstrate their significance.

Piglet is a new architecture intended to provide the resource-management capabilities of a vertical OS, while reducing mechanism intrusion. Measurements of a prototype implementation show that Piglet achieves these goals; for example, round-trip times measured by the ping application are up to 20\% lower than for Linux.

1 Introduction

The primary purpose of a computer system is to run user applications; the operating system provides a supporting infrastructure for applications but does not directly carry out user tasks. The goal of the OS is to share system resources among multiple applications which may be unaware of and/or competing with each other. Hence the overhead imposed by the OS upon those applications must be minimised if the system is to be utilised most efficiently. In particular, the manner in which applications can access resources is critical to their performance.

Resource management can be decomposed into two somewhat independent parts: mechanism and policy. The mechanism defines the details of how a physical resource is shared amongst multiple applications e.g., a CPU page-table, while the policy defines how application requirements are satisfied by the resource e.g., page replacement policy.

1.1 Operating System Intrusion

Both the mechanism and policy of resource management have considerable impact upon an application's behaviour—we introduce the term OS intrusion to denote this effect. Inefficient mechanisms can impose considerable overhead upon applications, and inappropriate policies can have a significant negative effect on application performance. These two aspects of OS intrusion are referred to as mechanism intrusion and policy intrusion respectively.

A great deal of research has been carried out into reducing policy intrusion e.g., Harty and Cheriton's analysis of application-level paging [Harty92], Stonebraker's discussion of the impact of inappropriate OS policies upon database performance [Stonebraker81]. The vertically-structured OS (MIT's Exokernel [Engler95, Kaashoek97], the University of Cambridge's Nemesis [Leslie96]) has been proposed as a new architecture intended to reduce policy intrusion by efficiently supporting application-level resource management.

Mechanism intrusion has also been extensively studied. Mogul and Ramakrishnan [Mogul97] describe how an interrupt-driven OS can experience 'receive livelock' due to high interrupt load, and several operating systems have been designed which reduce the overhead of IPC e.g., the L3 and L4 microkernels [Liedtke93, Hartig97], Spring [Hamilton93], and EROS [Shapiro96].

1.2 The Piglet Architecture

We propose a new architecture, Piglet, which provides the same reduction of policy intrusion as the vertically-structured operating systems but also significantly reduces mechanism intrusion.

Piglet is based upon the principle of dedicating a system CPU to executing an instantiation of the OS. Thus the OS becomes an active entity within the system rather than a library of functions only executed as required by applications or devices. This fundamental change in the nature of the OS facilitates
the use of different control mechanisms with lower intrusion costs than those required in a conventional OS.

While conceptually similar to the microkernel architecture where application-level servers provide OS services, servers in such systems are still passive and hence the overhead of communication, specifically the context-switch required to activate the server, is still high. While innovative IPC mechanisms such as Spring’s doors [Hamilton93] reduce this overhead, it is still much higher than in Piglet where the OS is constantly active.

The increasing availability of multiprocessor systems, a trend which we expect will continue e.g. single-chip multiprocessors [Burger97], makes the Piglet architecture attractive. In particular, we believe that it is particularly applicable to those application domains where I/O comprises a large fraction of the workload and thus the cost of mechanism intrusion is significant. An example of such an application domain is the network appliance—web servers, firewalls, active network nodes, etc.

2 Reducing Policy Intrusion—the Vertical OS

In order to understand how the Piglet architecture relates to other operating system architectures it is illustrative to first briefly consider the structure of conventional and vertically-structured operating systems.

2.1 Conventional OS Structure

Figure 1 shows the structure of a conventional SMP OS e.g., UNIX [McKusick96], Windows NT [Solomon98]. n processors each run different application-level threads and interact with physical devices through the OS.

The heavy dashed line indicates the user/kernel privilege boundary—in a conventional OS this is coincident with the boundary between application and system functions, the latter being indicated by the darker grey shading, while the lightly-shaded area represents resource protection and multiplexing within the OS. Arrows represent control transfers: applications invoke system functions through traps; the OS uses interrupts to interact with applications, either by raising signals or causing a reschedule. Similarly, the OS controls devices using I/O commands; devices request that the OS execute a specific function by raising an IRQ.

The primary disadvantage of this structure is that the coincidence of the privilege boundary and application programming interface prevents applications from changing resource-management policies which are embedded within the OS.

2.2 Vertical OS Structure

The benefits of separating resource management mechanisms and policies have been clearly demonstrated for many different classes of resource. Application-specific management of virtual memory has been extensively researched: Appel and Li [Appel91] provide general considerations for implementing application-level VM primitives while Hand [Hand99] and Engler [Engler95] describe specific implementations.

User-level network protocols are another area which has been covered by many groups: Cornell’s U-Net [vonEicken95], and the Virtual Interface Architecture [VIA97] derived from it, provide applications with a direct interface to the network adapter; Thelkath et al. [Thelkath93], and Edwards and Muir [Edwards95] describe user-level implementations of network protocols.

Stonebraker [Stonebraker81] discusses how the services provided by a general-purpose OS (UNIX) are unsatisfactory for supporting a high-performance database management system. He comes to the conclusion that “A DBMS would prefer a small efficient operating system with only desired services... On the other hand, most general-purpose operating systems
offer all things to all people at much higher overhead.” Hence the need for alternative operating system structures.

Vertically-structured operating systems, such as MIT’s Exokernel [Engler95] and the University of Cambridge’s Nemesis [Leslie96], attempt to provide application-level resource management by restructuring the operating system as shown in Figure 2. As before, n processors each run application threads and interact with physical devices through the OS.

Such operating systems attempt to reduce policy intrusion by providing a degree of separation between resource management mechanisms and policies. The OS provides mechanisms—protection and multiplexing—to share resources between applications; applications use those resources according to their own application-specific policies (within the constraint imposed by the OS mechanisms).

In order to provide these properties the vertically-structured OS separates the privilege boundary and application programming interface. The interface presented by the OS at the privilege boundary is much lower-level, the OS only performing protection and multiplexing of resources. Applications can be programmed directly to these lower-level interfaces, or they can access resources through higher-level interfaces provided by a libOS [Kaashoek97].

By providing low-level interfaces to access resources vertical operating systems effectively reduce policy intrusion since an application is free to implement its own policies using those interfaces. However, the fact that the OS provides lower-level primitives means that a function which a conventional OS provides as a single system call may require the invocation of several primitives in a vertical OS.

As an example, a vertical OS typically provides packet-level operations on the network interface e.g., send an Ethernet frame, while a conventional OS provides system calls which may send many such packets. Thus, the cost of mechanism intrusions becomes particularly important in a vertical OS—if the overhead of invoking low-level primitives is too high then the benefits of reducing policy intrusion may be compromised.

On the other hand, the structure of a vertical OS may actually reduce mechanism intrusions, especially system calls, in certain circumstances. Because each application communicates with its own libOS, certain operations which must be protected in a conventional OS can be unprivileged in a libOS, and hence not require a privilege-crossing to invoke.

3 The Cost of Mechanism Intrusions

Mechanism intrusions are inherent in conventional and vertical OSes due to the passive nature of the OS. The OS is essentially a library of I/O and process management functions; if supporting multiple applications in a protected manner was not necessary then it could be implemented as such without protection mechanisms, as in MS-DOS.

A passive OS is protected from applications by privilege boundaries; applications run in user (unprivileged) mode and must switch to supervisor/kernel (privileged) mode to execute OS functions. The overhead of crossing the privilege boundary is an example of a mechanism intrusion—the structure of the OS imposes an additional invocation cost upon applications.

The passivity of the conventional OS also causes mechanism intrusions in the form of device interrupts. A device requiring OS service request that a device driver function be invoked; since it is unaware of the state of each CPU, it cannot depend on being ‘asked’ for its request with any guarantee of timeliness and hence must raise an interrupt to force some CPU to service that request. Whatever application was running on the CPU that was interrupted will have its execution suspended while the OS is invoked; in addition to the time taken to execute the interrupt handler the application’s CPU caches will be contaminated with OS code.
begin_irq: 2776480895
timeout: 2776481212
Evortex_interrupt: 2776482423
Etetif_rx: 2776484514
Lnetif_rx: 2776484660
Lvortex_interrupt: 2776485399
Emet_bh: 2776486586
Eip_rcv: 2776487198
Eicmp_rcv: 2776488110
Lip_rcv: 2776488770
Lnet_bh: 2776489162
end_irq: 2776497874

Figure 3: Example of contents of event log

3.1 Device Interrupts

A device interrupt occurring during the execution of an application affects the application in two primary ways: it consumes CPU cycles and contaminates the CPU’s caches. Both costs will be analysed in turn.

3.1.1 Measurement Methodology

In order to accurately measure the cost of OS intrusions the Linux kernel was instrumented to record the times at which various events occurred e.g. interrupts, entering and leaving relevant functions. These times were measured using the timestamp counter present on Intel’s Pentium and P6-class CPUs [Intel97]. Each such event was recorded by writing an event descriptor (the event name and the time the event occurred) into an array. The contents of the array can be read at a later time using the /proc filesystem.

Figure 3 shows a fragment of the event log used to gather the measurements below. The labels Exxx and Lxxx indicate when execution enters and leaves function xxx, with the numerical value indicating the least-significant word of the timestamp counter (a 64-bit register). Similarly, the line irq9 indicates when the CPU entered the first-level interrupt handler. Finally, the begin_irq and end_irq labels are the pre- and post-interrupt timestamp readings recorded by the application being profiled.

The costs of an interrupt were measured using the code shown in Figure 4. The application spins in a tight-loop reading the timestamp counter until two successive values differ by a significant amount (as defined by the THRESHOLD constant). If we assume that loop is tight enough that the cost of executing it is negligible then the difference between those two values indicates the number of cycles consumed by the interrupt handler. A similar technique was used by Endo et al. [Endo96] to measure the time to process user-generated events in an interactive system.

The measurement system was a 200MHz Pentium Pro running RedHat Linux 5.1 with version 2.0.30 of the Linux kernel. The interrupts measured is an up-Complete interrupt generated by a 3Com 3e905 Fast Ethernet PCI card [3Com97] upon receiving a network packet. Cycle counts were converted to time periods by multiplying by the clock period, 5ns for the test machine.

3.1.2 Interrupt Period and Latency

Table 1 shows various time-based metrics calculated from the Linux kernel event log and application timestamp counter readings. The interrupt period, Period, represents the total application CPU time consumed by a single interrupt.

While the interrupt period is the simplest measure of OS intrusion it fails to take account of the useful work done by the OS within that period. Judging exactly what is ‘useful’ work is somewhat subjective but we consider it to be all code whose execution directly contributes to the progress of some application. This includes system call, device driver, and bottom-half functions (‘soft’ interrupts), but excludes first-level interrupt/trap handlers, context switches, etc.

As an example, Table 1 shows the total time spent in the device driver’s receive function and Linux’s network protocol stack processing the received packet (the Work column). The difference between this figure and the interrupt period gives us the overhead of an interrupt, shown in the Overhead column.

The overhead can be further subdivided into the time between the device signalling the interrupt and the handler being called by the OS (shown as Latency in Table 1), and the time taken to return from the interrupt context to the application (the difference between Overhead and Latency, not shown).

old=read_tsc();
while (1)
{
    new=read_tsc();
    if (new - old > THRESHOLD)
        break;
    old=new;
}

Figure 4: Measuring the cost of an interrupt
<table>
<thead>
<tr>
<th>Form of intrusion</th>
<th>Period</th>
<th>Time (μs)</th>
<th>Cache misses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Work</td>
<td>Overhead</td>
</tr>
<tr>
<td>Network interrupt (upComplete)</td>
<td>44.3</td>
<td>27.5</td>
<td>15.8</td>
</tr>
<tr>
<td>System call (sendto())</td>
<td>32.3</td>
<td>24.8</td>
<td>7.5</td>
</tr>
</tbody>
</table>

*Measured for timer interrupt
†Measured for getpid()

Table 1: Measured costs of OS intrusion

3.1.3 The Impact of Interrupts on CPU Caches

It is not possible to extend this method to simultaneously measure both the cycles consumed by the interrupt handler and the effect upon the CPU’s caches since the interrupt could occur between the two instructions necessary to read both the timestamp counter and the performance monitoring counter used to track cache misses. This problem could be avoided by disabling interrupts while reading the counters if measurement was done inside the OS rather than from an application. However, taking an interrupt when already executing in the OS has a different execution path than when in an application. Thus the number of cache misses was measured separately.

The most useful metrics for measuring the effect of an interrupt on the CPU’s caches are the number of the application’s cache lines replaced during handling of the interrupt. This is equivalent to the number of cache misses taken during execution of the interrupt handler, with one complication. Cache lines prefetched by the CPU can be accessed without a cache miss, thus introducing an error between the measured number of cache misses and the actual number of cache blocks replaced.

The code used to measure cache misses was similar to that used to measure interrupt handler period, as shown in Figure 4. Intel’s P6-class CPUs contain a pair of performance monitoring counters (PMCs) which can be configured to count occurrences of specific events. These were used to measure L1 D-cache and I-cache misses instead of reading the timestamp counter.

Before entering the loop a section of code is executed in order to flush OS data and code from the L1 caches. This consisted of a loop reading elements in an 8k array to flush the D-cache, and 8192 no-ops to flush the 8k I-cache. The application then spins in a loop until two successive reads of the I-cache miss counter are different. The D-cache counter need only be read after flushing the D-cache and again after exiting the loop since the loop code does not reference data in memory at all. Mean values from a sample of 1000 measurements are shown in Table 1.

What is clear from these results is that an interrupt has a non-negligible impact upon the application executing when it occurs, both in CPU cycles consumed and cache contamination. While the network interrupt is more complex than most other interrupts e.g., timers, and thus may not be representative of the time to execute those handlers, it is not unreasonable to assume that a better estimate lies somewhere between that figure of 40μs and the lower-bound of 15.8μs given by the overhead.

This is not the only effect of an interrupt upon the application however, since the number of cache lines evicted by the interrupt handler is also significant. On a system with 8k L1 caches and 32-byte cache lines (hence 256 lines in the cache), the fraction of the cache affected by the interrupt is almost one third for the D-cache. The I-cache impact is harder to estimate due to the inaccuracy of measurement (as discussed above) but is at least 15%. The effect of this cache contamination is to effectively increase the time penalty of taking an interrupt, since the application must now reload the caches.

Of course, the actual performance impact of interrupts upon applications depends upon the interrupt load upon the system. These measurements do show though that a system operating under a high interrupt load of several thousand or tens of thousands of interrupts per second would spend the majority of its time handling those interrupts. An example of this phenomenon, receive livelock due to network interrupts, was described by Mogul and Ramakrishnan [Mogul97] (see Section 6.2).

3.2 System Calls

The second form of intrusion which Piglet attempts to reduce is the overhead of invoking a system call. In both conventional and vertical operating systems system calls are the mechanism used by applications to invoke OS services. Since these services typically perform privileged operations e.g., device I/O, modifying OS state, the invocation mechanism must be able to guarantee that applications invoke the OS only through certain fixed access points rather than arbitrary points in the OS. This is typically done by
forcing the application to execute a trap or interrupt instruction which causes execution to jump to a fixed entry point in the OS. However, other mechanisms are also possible, such as the call-gate mechanism provided by the Intel x86 architecture.

While such a mechanism does guarantee that the application cannot execute arbitrary sequences of OS code, the price to pay is the overhead of making the privilege crossing. This overhead can be broken down into several costs:

- Executing the trap or interrupt instruction. This is extremely machine-dependent, but can be relatively high e.g., 48 clock cycles on the Intel Pentium Processor [Intel95] (higher if via a task gate).

- Saving application state. This is also machine-dependent since each CPU register must be written to some region of the process’s state maintained by the OS.

- Entering the OS. Depending upon the structure of the OS this may involve checking whether another processor is already executing within a critical region of the kernel, thus requiring that this CPU block.

- Dispatching the call to the correct function. This can be determined in several ways e.g., implicitly by the instruction which caused the trap, using the value of a specific register as a table index.

The fact that these steps must be taken before the OS begins to execute the function requested by the application means that the cost of these steps is essentially overhead. Minimising this overhead is important if applications frequently make system calls, which may well be the case in a vertical OS where a system call implements a lower-level primitive than in a conventional OS.

The second row of Table 1 describes the costs of making a sendto() system call in the Linux OS. The time costs were determined using the same instrumentation mechanism as described above, and the same quantities are presented (period, work, overhead, latency). Cache effects were determined in a similar manner to those for an interrupt i.e., flushing the L1 caches of OS code and data before the event of interest occurs, but in this case the trivial system call, getpid(), was invoked rather than waiting for an interrupt. By measuring the cache effects of the trivial system call we can isolate the overheads of the above steps.

These measurements show that the time-domain overheads (shown in the Overhead and Latency columns) of making a system call in Linux are approximately half those of taking an interrupt. This is most likely due to the fact that interrupt handlers incur all the costs of a system call but also have to communicate with the external interrupt controllers, thus incurring the cost of slow I/O operations.

What is most surprising about these measurements is the large number of D-cache misses incurred even by the trivial system call. Given the nature of the steps described above, and specifically the memory references involved, it appears that the Linux kernel’s system call invocation mechanism is somewhat poorly structured. However, one can still conclude that the time cost is sufficiently high that an application making many system calls will lose a considerable amount of time as overhead.

4 The Piglet Architecture

While conventional and vertical OS architectures differ fundamentally in their placement of the privilege boundary, they both use the same control transfer mechanisms i.e., traps and interrupts. The Piglet architecture offers a placement of the privilege boundary similar to vertical operating systems but uses different control transfer mechanisms, as shown in Figure 5. n – 1 CPUs run application threads while the nth runs the lightweight device kernel (LDK); applications interact with physical devices only through the LDK.

Piglet derives many of its primary properties from the fact that the OS is now an active rather than passive entity:

- Protection between applications and the OS is enforced by physical rather than logical (privilege) boundaries.

- Applications invoke OS services by sending a message to the LDK rather than executing service functions directly.

- The OS continually polls devices, eliminating the need for them to use interrupts in order to guarantee timely service. Consequently, application CPUs are never interrupted by devices.

- Elimination of interrupts significantly simplifies the design and implementation of the OS.

Applications running on Piglet thus incur very low mechanism intrusions: system calls are replaced by a simple protocol to post a message to a shared-memory queue, and device interrupts are eliminated.
4.1 The Lightweight Device Kernel

As stated above, the Piglet architecture is centred around a system CPU executing the lightweight device kernel. The behaviour of the LDK is conceptually very simple: it continuously polls every device and application for service requests i.e., operations which must be performed on behalf of the requester.

Although simple, the structure of the LDK is critical to the performance of Piglet. If Piglet is to offer quality-of-service guarantees to applications and devices then the distribution (mean, variance, best-case, etc.) of the polling period must be tightly controlled.

Three main factors contribute to the reduction of the polling period. Firstly, the complexity of the LDK is drastically reduced by the elimination of device interrupts (except for exceptional cases e.g., hardware failure), since algorithms do not have to be designed to cope with the possibility of an interrupt occurring at unpredictable times. Secondly, the use of lock-free synchronisation in the shared-memory queues used to communicate with applications allows the LDK itself to be lock-free. Finally, the use of runtime code generation to specialise the polling loop and other critical functions e.g., packet filters, packet header generators, reduces the cost of invoking those functions.

4.2 Application Invocation of OS Services

Since the LDK and applications run on different CPUs applications cannot directly affect the LDK, instead communicating through lock-free shared-memory queues (the lock-free protocol used is a variant on those described by Michael and Scott [Michael96]). Most system services are invoked by the application posting a request to the appropriate queue; the exceptions are described below (Section 4.2.1).

Execution of OS services by the LDK on behalf of the application rather than by the application itself has two primary benefits. Firstly, applications do not need to cross the privilege boundary to post a request to the shared-memory queue. Protection is instead enforced by physical separation and validity-checking of requests by the LDK. Secondly, invoking a system service is asynchronous rather than synchronous. Once the application has posted the request it continues executing concurrently with processing of the request.

Both these benefits reduce the overhead of invoking a system service: in addition to eliminating the cost of the privilege crossing, the application also performs less work since it only has to post a message to the LDK rather than execute the service function itself. However, the asynchronous nature of service invocations means that it is generally not possible for the invoker to determine what the outcome will be. This is a significant difference from the conventional OS model whereby all system calls return a value to indicate success or failure. Piglet therefore uses a different programming model for service invocation:

- Failure to complete a service request is an exceptional event; the common case is that the request is successful. Thus Piglet uses an exception mechanism to indicate a failed service request.
- A service request cannot fail due to lack of resources. Applications must provide the LDK with all the resources necessary to execute the service request e.g., memory pages, network buffers.
- Applications can annotate a service request to indicate that failure should not raise an exception; this is equivalent to discarding the return value of a system call invocation.
- If the application cannot proceed further until it knows that the service request has completed successfully it can annotate the request to raise a
successful completion' exception, then wait for an exception.

The last construction allows applications to construct synchronous service invocations from asynchronous primitives. In particular, library operating systems could present synchronous APIs such that applications need not be aware of the asynchronous nature of Piglet's underlying primitive operations.

4.2.1 Directly-Invoked Services

Although most system services are invoked as described above, some must be invoked in the conventional manner i.e., by the application crossing the privilege boundary and executing OS code. Broadly speaking, such services are those which directly affect local CPU state, must be privileged, and cannot be efficiently executed by a remote CPU e.g., TLB management.

4.2.2 Application-LDK Communication

As stated above, the shared-memory queues via which applications communicate with the LDK use lock-free synchronisation to support concurrent access. Lock-free synchronisation is important primarily so that applications cannot cause the LDK to block.

The algorithm used in the current implementation of the LDK supports multiple concurrent writers and a single reader and so can be used as-is for the application-to-LDK queues; for queues used in the reverse direction the application must impose its own scheme for controlling concurrent access to the queue.

4.3 Exceptions and Asynchronous Scheduling

Exceptions in the Piglet architecture are functionally equivalent to UNIX signals. They serve two main purposes: notification of service request failure and notification of events specified by the application. Both needs are dictated by the asynchronous nature of a service request invocation: it cannot return a value to indicate success or failure, nor can it block the application. An application which needs to wait for an event does so by notifying the LDK which events should cause it to be unblocked and then forcing a reschedule.

Scheduling is also handled differently in the Piglet architecture than in conventional and vertical operating systems. Rather than invoking a scheduler function every time a CPU is rescheduled the LDK instead continuously updates the scheduler state. For each thread currently running on a CPU the LDK generates a specialised reschedule function which switches contexts directly to the next thread to be executed on that CPU (as in Synthesis [Massalin92]). This function can be changed concurrently with the thread’s execution as other threads are blocked and restarted.

5 Evaluation of the Piglet Architecture

In order to experimentally evaluate key components of the Piglet architecture a hybrid Piglet/Linux kernel was constructed. Adding the functionality of Piglet to an existing Linux kernel allowed for evaluation of that functionality without having to build supporting OS infrastructure from scratch.

The key aspects of Piglet that need to be evaluated experimentally are its reduction of OS intrusion and efficient support for application-level resource management. In order to do so a user-level ICMP network protocol stack was implemented and used to compare the cost of packet processing with those of a conventional OS.

Two different types of comparison were made between the Piglet user-level protocol stack and the Linux kernel protocol stack. Application-level performance was compared using the ping application to measure round-trip latency. A more detailed analysis of the overheads of packet transmit and receive was performed using the instrumentation described in Section 3.1.1.

5.0.1 Evaluation Platform

All tests were conducted using the Piglet/Linux hybrid kernel derived from Linux version 2.0.30 but updated with current drivers for the network cards used. The test machine was a dual 200MHz Pentium Pro with 96MB of EDO DRAM connected to an identical machine via an isolated 100Mb/s Ethernet hub using 3Com 3c905 NICs.

5.1 Measurement of Round-Trip Latency

Measurement of Piglet’s effectiveness in reducing system call overhead was performed using the ping application. ping was chosen because of its simplicity, allowing the impact of system-calls to be easily isolated. Several modifications were made to the program:
ping muffin2 -f -c 100000 -s 64 --use-tsc 200 --histogram > /dev/null

where: muffin2 name of target machine
-f flood-ping mode
-c 100000 send 100000 packets
-s 64 add payload of 64 bytes
--use-tsc 200 use timestamp counter for timing, 200 cycles per $\mu$s
--histogram generate a histogram of timing frequencies
>/dev/null redirect output to /dev/null

Figure 6: Command used to measure round-trip latencies.

![Histogram](image)

Figure 7: Distribution of round-trip times as a function of payload size

<table>
<thead>
<tr>
<th>Payload/bytes</th>
<th>Modal RTT/$\mu$s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linux</td>
</tr>
<tr>
<td>64</td>
<td>199</td>
</tr>
<tr>
<td>256</td>
<td>233</td>
</tr>
<tr>
<td>1024</td>
<td>468</td>
</tr>
</tbody>
</table>

Table 2: Mean round-trip times

- Support for the Piglet user-space network interface was added. This entailed replacing socket calls with the equivalent calls to the Piglet user-space library.
- The CPU’s timestamp counter was used for measuring round-trip time (RTT) since this can be read directly from user-space, while `gettimeofday()` requires a system-call invocation.
- A profiling option was added to log the value in the timestamp counter at various points in execution.
- A histogram of RTT distribution can be generated.

ping was invoked as shown in Figure 6, with the addition of the `--piglet` parameter when running the Piglet tests. Three different payload sizes were used, and the experiment was repeated several times to ensure that results were representative. The histograms generated are shown in Figure 7, with the label `piglet-XXX` attached to the histogram for Piglet with XXX payload bytes, and similarly for `linux-XXX`.

The histograms show the observed frequency against RTT for the given payload size. The position and shape of the histograms are more significant than the frequency values themselves. Modal RTTs i.e., the position of the most-frequently observed RTT (highest column in the histogram) are shown in Table 2 as an aid to comparison, with the leftmost histograms having smallest RTTs. Taller, narrower histograms indicate smaller variance in the distribution of round-trip times.

The most significant point to take from Figure 7 is that the round-trip times for the Piglet system are lower than those for the Linux system by approxi-
mately 38\(\mu\)s for 64-byte payloads, and up to 55\(\mu\)s for 1024-byte payloads. There is some dependence upon payload size because the Piglet user-space protocol stack performs DMA of the packet payload (not headers) directly from application memory while Linux copies into a kernel buffer from which DMA is performed; the cost of copying is negligible for 64-byte payloads however so 38\(\mu\)s can be taken as a representative figure. The semantic implications of performing DMA from application memory will be addressed in future work.

5.2 Comparison of System-Call Overheads

In order to perform a more detailed analysis we used the event logs generated by the instrumentation described in Section 3.1.1. Representative sections of the log for a single round-trip were extracted and displayed graphically in Figures 8 and 9 for Linux and Piglet respectively.

The graphical representation shows the relative value of the cycle counter corresponding to various events in the sending and reception of a single packet. The counter value when the top-level function used to send the echo packet is called is used as the zero point; the \texttt{got\_packet} event indicates when the reply packet was received by the application.

The unshaded boxes to the right of the time axis represent functions executed in the application, while the shaded boxes to the left represent kernel functions. Stacked boxes represent nested function calls. Each dotted line indicates that the event with the given label occurred at the specified time. For the Linux trace certain events have been omitted since they do are not relevant to the analysis; the complete Linux trace is given in Appendix A.

The Piglet trace represents the execution of the LDK processor on the leftmost line. Since the Piglet trace is more concise every event is represented in Figure 9.

What is most obvious from these traces is the difference in the time taken to send a packet i.e., from \texttt{begin\_send} to \texttt{end\_send}. While Linux takes \(\approx6300\) cycles, Piglet only requires 230. This major difference is due to the application in Piglet only having to post a message to the shared-memory queue rather than executing the system call itself.

The latency between \texttt{begin\_send} and the packet having been passed to the network interface (\texttt{Lboomerang\_tx}) is also much lower in Piglet—1400 cycles as opposed to 5000. This corresponds to a contribution of an extra 18\(\mu\)s (with a 200MHz CPU clock) to the RTT. One factor in the low latency in

![Figure 8: Execution trace of Linux ping](image_url)
Piglet is that the delay between the application posting the message and the LDK processing that message is only 170 cycles.

While a small part of this difference, the reduced time spent in boomerang\textsubscript{rx} is due to an optimisation in the Piglet network driver (not stalling the NIC’s DMA engine if it is known to be already stalled), the biggest factor is the structure of Linux’s protocol stack. Because Linux implements a much higher-level interface (BSD sockets) it executes multiple levels of functions (\texttt{sys\_sendto}\textemdash the generic socket layer; \texttt{inet\_sendmsg}\textemdash the Internet domain socket family; \texttt{raw\_sendto}\textemdash the Internet domain raw socket; \texttt{ip\_build\_xmit}\textemdash build IP headers for the packet) before finally passing the packet to the network interface. The substantial overhead of these multiple layers of abstraction contribute the largest amount to the system call cost in Linux.

While not affecting the round-trip time, the traces also provide an example of mechanism intrusion due to device interrupts. In Linux the first \texttt{vortex\_interrupt} event is caused by the NIC raising an interrupt to inform the OS that the packet has been sent; in Piglet the application does not suffer this intrusion because the LDK polls the NIC to determine when packet transmission is complete.

After sending the packet, the \texttt{ping} application calls \texttt{select()} to block until the reply is received. Since the time when the kernel enters the corresponding top-level function is not recorded, it has been estimated using the measurements from Section 3.2. Notification that the packet has been received and transferred into memory by the NIC occurs in the second \texttt{vortex\_interrupt} event for Linux, and the boomerang\textsubscript{rx} event for Piglet. The time between that event and \texttt{select()} returning to the application is greater in Piglet because the LDK must send an inter-processor interrupt (IPI) to the Linux kernel to indicate that a packet was received.

The \texttt{select()} call could be removed in both the Linux and Piglet tests. For Linux it is unnecessary since the \texttt{recvfrom()} function blocks if no packet is available; if the call to \texttt{select()} is removed then the Linux RTT decreases by \(\approx 20\mu s\). In the Piglet environment it should be replaced with operations that use the Piglet exception and scheduling mechanisms (see Section 4.3); that should also lead to a reduction in RTT.

Finally, once returning from \texttt{select()} the application receives the packet; in Linux this entails making a \texttt{recvfrom()} system call, while in Piglet it removes a packet descriptor from a shared-memory queue. The time difference here (\texttt{end\_select} to \texttt{got\_packet}, \(21.0\mu s\) vs. \(3.5\mu s\)—a difference of 17.5\(\mu s\)) again demonstrates the costs of system call overhead and multiple levels of abstraction.

This analysis demonstrates two key points. Firstly, the overhead of a system call, 1500 cycles, is significant—approximately 25% of the total cycles executed for \texttt{sendto()}. Secondly, implementing general-purpose interfaces in the OS leads to inefficiency—the central argument used by proponents of vertical operating systems.

### 6 Related Work

While the generalised concept of OS intrusion is somewhat new, both policy and mechanism intrusion have been investigated to some degree by previous researchers. A number of OS projects which attempt to reduce policy intrusion have been designed and im-
implemented, and specific examples of mechanism intrusion have been studied by various groups.

6.1 Vertical Operating Systems

As mentioned earlier, MIT's Exokernel [Engle95, Kaashoek97] and the University of Cambridge's Nemesis [Leslie96] are the best-known examples of vertical operating systems, but other groups have also designed operating systems which address the same goals e.g., Stanford’s Cache kernel [Cheriton94], Columbia’s Synthesis OS [Massalin92], and the University of Washington’s SPIN OS [Bershad95].

Exokernel and Nemesis both adhere closely to the vertical OS model described earlier, providing low-level OS primitives upon which applications can construct their own resource management policies. However, beyond that similarity they differ considerably. The Exokernel supports the idea of application-level extension of the OS, thus allowing applications to download code into the OS itself in a safe and protected manner.

Nemesis, on the other hand, restricts the kernel to the minimal functions necessary to support application-level resource management. Particularly relevant to Piglet is its handling of interrupts, interrupt handlers being split into a minimal handler function and a schedulable region which performs the bulk of the work. This model is used to reduce QoS crosstalk caused by spending long periods of time executing interrupt handlers.

The Cache kernel also provides applications with a set of low-level primitives which support application-level resource management. However, rather than the OS implementing protection and multiplexing, the Cache kernel instead exports a minimal core set of objects (address spaces, threads, and kernels) upon which applications build services. It is also interesting from the Piglet perspective because of its heavy use of non-blocking synchronisation.

Massalin’s Synthesis OS made extensive use of run-time code generation to generate specialised OS functions. These functions were used for various tasks, including scheduling, interrupt handling, and system call invocation.

Finally, like the Exokernel and Synthesis, SPIN also uses run-time code generation to support OS extensibility. However, this extensibility, rather than the provision of low-level mechanisms, is used as the principle method of supporting application-level resource management. For example, applications can download code fragments to be invoked by the OS when particular events occur e.g., packet reception.

6.2 Mechanism Intrusion

Mogul and Ramakrishnan [Mogul97] describe how a conventional OS can become ‘livelocked’ due to continuously being interrupted by a network interface card. While the solution they propose, temporarily disabling interrupts and switching to a polling mechanism, is very similar to the Piglet architecture it differs significantly in that their OS switches dynamically between the two schemes while Piglet always uses polling.

In a similar vein to the analysis in Section 3.1, Mogul and Borg [Mogul91] evaluated the effect of context switches on cache performance. Dougan et al. [Dougan99] discuss possible modifications to Linux kernel in order to reduce the impact of the OS on cache performance. In particular, they recommend bypassing the cache in certain portions of the kernel e.g., when zeroing memory pages in the idle task, so as not to pollute the cache.

Several OS projects have made significant efforts to reduce the cost of inter-process communication e.g., the L3 and L4 microkernels [Liedtke93, Hartig97], the Spring microkernel [Hamilton93], and the EROS kernel [Shapiro96]. All four systems adopt the microkernel approach wherein services are provided by application-level servers rather than the OS itself, and thus high-performance IPC is essential.

While L3 and L4 are perhaps canonical examples of fast IPC-based microkernels, Spring and EROS introduce their own innovations. Spring introduces the concept of doors, a mechanism for supporting fast cross-address space invocations. A door is conceptually similar to a system call entry point in a conventional OS, in that it defines a fixed access point by which an untrusted client may invoke a specified function. However, doors are defined by applications and passed as capabilities to clients. EROS is based upon a capability architecture which provides strong safety and security properties while maintaining low IPC costs.

7 Conclusions

We have introduced the term OS intrusion to describe the effect that the OS has upon application behaviour, and a new operating system architecture, Piglet, which reduces that effect. Piglet builds upon existing work on vertical operating systems, which eliminate policy intrusion, to also greatly reduce mechanism intrusion.

Measurement of the cost of mechanism intrusion in a conventional OS shows that in an application which is subject to a high frequency of intrusion, ei-
ther interrupts or system calls, a considerable amount of time will be lost as overhead. While Mogul and Ramakrishnan [Mogul97] describe a worst-case scenario, we believe that the common case may also be an area of concern e.g., for high-performance web servers on high-bandwidth networks. If we assume that MTUs remain constant at approximately 1500 bytes, then a load of only 100 Mb/s corresponds to ≈8000 packets per second. Loads approaching a gigabit per second would cause mechanism intrusion to become a significant overhead. We believe the Piglet architecture to be particularly attractive in such application domains.

While Engler et al. describe in [Engler95] how Aegis, the first implementation of their Exokernel architecture, was heavily optimised to reduce the costs of invoking low-level primitives, Kaashoek et al. report in [Kaashoek97] that such optimisations are not necessary to leverage the most benefit from the Exokernel architecture. While we agree with this statement we also believe that in classes of application such as described above i.e., where I/O comprises the bulk of the workload, there is benefit to be had from reducing mechanism intrusion.

One aspect of Piglet which must be investigated further is scalability i.e., how many clients (applications and devices) the LDK can service while still offering satisfactory QoS guarantees. By virtue of the simplicity of the kernel and through the use of run-time code generation techniques to optimise the polling loop we expect Piglet to scale in a manner superior to interrupt-driven systems.

Conversely, the applicability of Piglet in small-scale multiprocessor, especially dual-processor, systems must also be addressed. While symmetric multiprocessing allows applications to directly utilise every CPU in a system we believe that Piglet’s superior support for high interrupt loads makes it more attractive in certain applications e.g., high-performance, low-cost network appliances such as web servers, firewalls, and active-network nodes. We intend to explore these specific application classes in depth.

References

[3Com97] 3Com Corporation; “3C90x Network Interface Cards Technical Reference”, 3Com Part Number 09-1163-000 (December 1997), pp.7-6.

[Aappel91] A. W. Appel and K. Li; “Virtual memory primitives for user programs”, 4th International Conference on Architecture Support for Programming Lan-


A Linux ping Execution Trace

begin_pinger: 1328188896
begin_send: 1328190439
Esys_sendto: 1328191174
Einset_sendmsg: 1328191857
Eraw_sendto: 1328192100
Eip_build_xmit: 1328192463
Eboomerang_tx: 1328194362
Lboomerang_tx: 1328195466
Lip_build_xmit: 1328195607
Lraw_sendto: 1328195709
Linet_sendmsg: 1328195919
Lsys_sendto: 1328196051
end_send: 1328196717
Eviewport_interrupt: 1328198649
vortex_down_complete: 1328199105
Lviewport_interrupt: 1328200575
begin onSelect: 1328212147
EviewportInterrupt: 1328216275
vortex_up_complete: 1328216717
Enetif_rx: 1328218164
Lnetif_rx: 1328218346
Lviewport_interrupt: 1328219074
Enet_bh: 1328220307
Eip_rcv: 1328220967
Eicmp_rcv: 1328222086
Licmp_rcv: 1328222720
Lip_rcv: 1328223124
Lnet_bh: 1328223209
end_select: 1328225838
Esys_recvfrom: 1328226744
Einet_recvmsg: 1328227308
Eraw_recvmsg: 1328227560
Lraw_recvmsg: 1328228682
Linet_recvmsg: 1328228841
Lsys_recvfrom: 1328229369
got_packet: 1328230030