# Efficient Packet Processing in User-Level OSes: A Study of UML

Younggyun Koh Calton Pu
College of Computing
Georgia Institute of Technology
Atlanta GA 30332-0280 USA
{young, calton}@cc.gatech.edu

Sapan Bhatia Charles Consel
LaBRI/INRIA
ENSEIRB, 1 Ave. de Dr. Albert Schweitzer
33402 Talence France
{bhatia, consel}@labri.fr

## ABSTRACT

*Network server consolidation has become popular through recent virtualization technology that builds secure, isolated network systems on shared hardware. One of the virtualization techniques used is that of User-level Operating Systems. (ULOSes) However, the isolation and security they bring comes at the price of performance, as virtualization introduces a number of overheads into the system. Such overheads can be surprisingly large, especially for complex OS modules like network protocol stacks. Our studies of the TCP/IP stack in User-mode Linux (UML), an implementation of a ULOS, attribute the resulting slow-downs to three main sources: the execution of privileged code, memory management across layers, and additional instructions to execute. To mitigate these bottlenecks, we present five optimization techniques, improving the network performance significantly, reducing packet processing latency by 60% and increasing network throughput by three folds. Furthermore, the network throughput of the improved ULOS is comparable to that of native Linux up to gigabit speeds.*

## 1. INTRODUCTION

Server consolidation is increasingly being considered as an important solution to implement complex internet services in untrusted environments. Server consolidation provides high server utilization with low hardware cost, increased manageability and easy expandability, while still maintaining strong isolation amongst individual services. [21]

System virtualization using user-level operating systems (ULOSes) [11][13][15][29][30] is a popular choice for implementing server consolidation. ULOSes are operating systems that run "over" other operating system kernels as user processes. A ULOS not only offers the advantages of typical virtual machine monitors (VMMs) [2][8][14][25][26] but also provides additional features such as easy installation, fine-grained configuration, and powerful diagnosis with the support of host OS tools.

On the negative side, ULOSes suffer from significant performance penalties for obvious reasons: running at user level, they must invoke the underlying host OS kernel to provide kernel services that cannot be emulated. While this indirection provides strong isolation, it introduces overhead considered to be unavoidable. From the server-consolidation point of view, it significantly increases the packet-processing it significantly increases the packet-processing overhead, reducing the maximum network throughput and increasing packet latency.

The first contribution of this paper is a study of the packet-processing overhead in a ULOS through a methodical analysis of packet processing in the TCP/IP stack. Our analysis divides the packet processing into five layers and identifies three main sources of overhead: privilege management, memory management, and additional software instructions. To reduce the overhead for each source, we propose five specific techniques: user-level signal masking, aggregated system calls, an address translation cache, shared socket buffers, and network stack specialization.

The second contribution is the application of optimization techniques to TCP/UDP packet processing in a case study of UML running on Linux. (denoted as UML+Linux) The case study shows that the performance of our optimized ULOS network protocol stack is comparable to that of native Linux. (within 5% for TCP and statistically the same for UDP, for network bandwidths up to gigabit speeds) However, significant research challenges remain, since the ULOS indirection overhead has not been completely masked in other metrics such as packet processing latency.

The rest of the paper is organized as follows. Section 2 describes current ULOS approaches. Section 3 outlines the ULOS performance problem and analyzes the sources of overhead. Section 4 describes the optimization techniques used in the UML+Linux case study to reduce the overhead. Section 5 presents the performance evaluation results. Section 6 outlines related work and Section 7 concludes the paper.

## 2. USER-LEVEL OPERATING SYSTEMS

A ULOS is a fully functional OS that runs as a process on a host OS. As a user process, a ULOS redefines its own core functionalities by using host OS interfaces such as system calls instead of the instruction set of the underlying processor. A ULOS provides a set of qualities that enables server consolidation.

**Resource allocation.** Each guest operating system is a user-level process. Resources such as CPU and memory are allocated according to the host OS' sharing and scheduling policy. Idle resources are allocated to busy processes to increase utilization, while maintaining fair sharing.

**Easy maintenance.** A virtual network server on a ULOS is easily migrated, paused, and recovered using traditional process migration and recovery techniques. System administrators can easily expand system capacity by adding more hardware and migrating the virtual servers to the new hardware.

**Strong isolation and reliability.** Since a ULOS is indeed a user-level process, isolation among guest ULOSes is naturally achieved by a host OS's process encapsulation. When a guest operating system is compromised, the fault is sandboxed and can not affect the host OS nor other guest ULOSes.

**Easy installation.** In contrast to type-I VMM approaches that need installation of VMMs on bare hardware, ULOS approaches install guest ULOSes on a host OS. This reduces hardware issues in installation, such as incompatibilities with the underlying hardware configuration.

**Easy system diagnosis.** For diagnosing a ULOS, system administrators are able to use common tools, such as gdb and oprofile, installed in a host OS without requiring special VMM support or kernel patches.

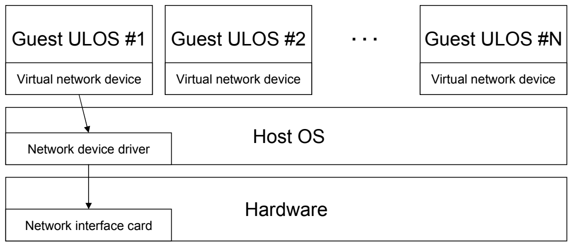

**Figure 2-1 User-level operating system architecture**

Despite these merits, the performance of a ULOS is limited by its architecture. In order to access hardware resources, a ULOS goes through a "thick" host OS layer, which slows down the overall system performance. More detailed overhead analysis of ULOSes is discussed in section 3.

## 2.1 ULOS Approaches

Because a ULOS runs in user mode, executing privileged instructions creates many challenges in ULOS implementations. Several approaches were proposed to overcome the lack of the privilege.

### *2.1.1 Binary translation*

VMware workstation [30] is a user-level VMM that enables unmodified guest OSes to run on commodity systems (e.g. Windows and Linux). It implements dynamic binary translation to run privileged instructions in user mode. It provides efficient on-the-fly binary translation because it translates only small sets of instructions that are not virtualizable.

On the other hand, LiLyVM [13] uses static code rewriting to implement a user-level VMM for Linux and NetBSD. When a guest operating system is complied, LiLyVM uses a special assembler that replaces the unvirtualizable instructions with instructions trapping into a VMM.

### *2.1.2 Direct OS ports*

Another approach for ULOSes is to port guest OSes for the host OS's interfaces. A ported ULOS understands specific host OS's interfaces (e.g. system calls) and uses them to emulate privileged operations.

Because a guest OS directly understands the host OS's interfaces, this approach removes a VMM layer between the guest OS and the host OS. The host OS meditates resource allocation among guest OSes.

## 2.2 User-Mode Linux

In this paper, we have chosen one of the direct OS port ULOSes, User-Mode Linux (UML), as our case study. UML is a port of the Linux kernel that runs as a user process on native Linux. It supports the full Linux API through the UML core.[1] When privileged kernel functions are invoked, the UML core calls the host Linux kernel to actually carry out these functions. The support for I/O devices such as network devices is provided through corresponding virtual devices.

## 3. PACKET PROCESSING IN UML

We measured the network performance of UML+Linux over a gigabit network. (Details of our experimental setup are in section 5.1.) In our preliminary experiments, UML+Linux exhibited considerably poor throughput and latency characteristics. Figure 3-1 compares the throughput of UML+Linux and native Linux, showing that Linux outperforms UML+Linux by 1.5 to 3 times. We also observed a 10 fold increase in packet processing time in UML+Linux.

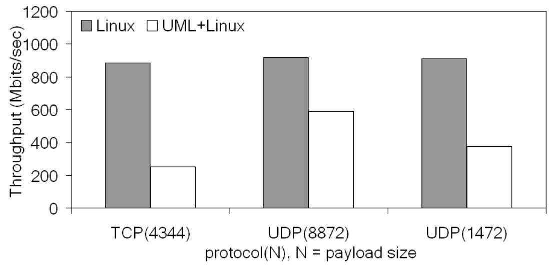

**Figure 3-1 Network throughput in Linux and UML+Linux**

[1] In this paper, the term "UML core" refers to the OS code that normally would run in kernel mode. Although the term "UML kernel" is the normal usage in the community, the slightly different term "UML core" avoids the overloading of the word kernel. In this paper the term kernel consistently refers to code that runs in kernel mode.

## 3.1 The Sources of Overhead in a ULOS

We analyze the overhead of a ULOS, dividing the sources of overhead into three main categories:

- Privilege management: a ULOS executes privileged instructions and provides kernel-level services by invoking the host kernel.
- Memory management: Extra memory copies to move data through the ULOS core and additional virtual address translations.
- Additional software instructions: More instructions to be executed due to the ULOS layer. (e.g., virtual I/O devices)

### *3.1.1 Privilege Management Issues*

A ULOS reuses the large majority of the kernel code of an existing OS. However, this simple reuse of the existing code raises an impedance mismatch between the code originally written as kernel code and its execution environment in user mode. In particular, the design of the kernel code depends on low-impedance base operations, assuming that executing privileged instructions and accessing kernel data structures are simple and cheap. This assumption does not hold in a ULOS. The increased cost of base operations entails a high-impedance design for the ULOS.

An important factor leading to the high impedance of base operations in ULOS is frequent and expensive user/kernel boundary crossings that typical ULOS facilities (e.g., disabling interrupts) trigger to implement privileged operations. Boundary crossings are expensive; with a Pentium4 processor machine used in our experiments, the `getpid()` null system call requires more than 1000 cycles (around 0.37 µs) to complete.

### *3.1.2 Memory Management Issues*

Another major source of overhead in a ULOS is extra data copies between the added layers. Since the ULOS core inserts a layer between an application and the host kernel, a network packet from the application is copied twice for below layers, the ULOS core and the host kernel. Note that native Linux requires only one copy between the application and the kernel. Our measurement with MTU-sized UDP packets shows that the packet payload copy accounts for around 40% of the latency measured in the virtual network device.

The next overhead related to memory management is introduced by virtual address translation. While the virtual-to-physical address translation in a host OS leverages on fast hardware such as the translation look-ahead buffer (TLB) and hardware-supported page-table manipulation, the address translation in a ULOS is implemented entirely in software. This software implementation naturally suffers from the additional memory accesses for traversing page tables and inefficient error handling without hardware support for catching traps.

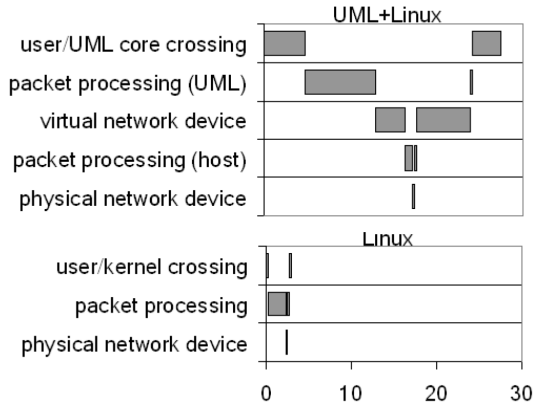

**Figure 3-2 Per-layer latency of UML+Linux and native Linux.** The x-axis represents latency in µseconds. We will use MTU-sized UDP packets for latency analysis.

### *3.1.3 Additional Software Instructions*

Since a ULOS adds an extra layer between applications and the host operating system, packet processing in a ULOS consumes more instructions. First, packets cross more protection boundaries and consume instructions at each crossing. Second, each packet goes through both the ULOS core and the host kernel, potentially causing extra context switches.

One way to reduce the extra overhead from the layered architecture is to specialize the code for a given context [17][23]. Particularly because the network packet processing has the tendency to have static parameters, techniques such as program specialization can help reduce the overhead [5][6].

## 3.2 Network Packet Processing Overhead

To illustrate the impact of the overhead on packet processing, we divide the network protocol stack into five layers. For concreteness, we use outgoing UDP packets in this analysis.

1. ***The user/UML core boundary crossing.*** An application invokes a `sendto()` system call. Control is transferred to the UML core.
2. ***Packet processing in the UML core.*** The UML core executes the usual steps in packet processing, including routing decisions, header filling, network queue processing, and packet forwarding.
3. ***The virtual network device.*** The UML core sends the packet to the virtual network device, which passes the packet to the host kernel. Control is transferred from the UML core to the host kernel.
4. ***Packet processing in the host kernel.*** The host kernel forwards the packets to an appropriate physical network device, e.g., an Ethernet bridge.
5. ***The physical network device.*** Finally, the physical network device driver sends out the network packets through the Network Interface Card (NIC).

We measured the time spent in each layer. The execution time at each layer is divided into two parts: (1) before invoking the lower layer, and (2) after returning from the lower layer. Figure 3-2 shows our experimental results for MTU-sized[2] packets in Linux and UML+Linux. Note that significant additional latency is introduced by UML in the top three layers.

## 4. Optimized Packet Processing

Despite the non-trivial overhead outlined in the previous section, we are able to alleviate those problems through a combination of system optimization techniques. First, we solve privilege management issues by reducing the need for frequent user/kernel boundary crossings (Section 4.1) and aggregated system calls (Section 4.2). Second, we introduce an address translation cache (Section 4.3) and shared socket buffers (Section 4.4) to resolve memory management issues. Third, we apply program specialization techniques to collapse software layers (Section 4.5) in the ULOS network stack, decreasing the total number of instructions executed. We have implemented these techniques in an experimental ULOS called Enhanced User-mode Linux (EUL).

In the following subsections, we describe these techniques and provide a component-level evaluation of performance gains for each technique.

### 4.1 User-Level Signal Masking (ULSM)

**Problem: UML Signal Overhead.** Disabling interrupts is a cheap synchronization mechanism in uniprocessors to share kernel data structures. Many critical sections in Linux are protected by `cli/sti` assembly instructions along with a few stack operations for saving/restoring the current interrupt flags.

Meanwhile, UML handles interrupts using process signals from virtual devices to the UML core (shown in Figure 4-1). Therefore, the UML core disables virtual interrupts by masking the signals using the `sigprocmask()` system call. Saving the interrupt state for nested critical sections is implemented by the same `sigprocmask()`, which returns the previous value of the signal mask. Consequently, disabling interrupts, cheap in the Linux kernel, becomes expensive in the UML core, because invoking a system call is costly.

**Solution: User-Level Signal Masking.** To avoid using `sigprocmask()`, EUL implements user-level signal masking. EUL removes the host system call by keeping the signal states in the user level (i.e., in the EUL core) rather than in the host Linux kernel. By toggling the interrupt state and keeping track of pending interrupts, the EUL core achieves the same synchronization without the host kernel intervention.

[2] The maximum transmission unit (MTU) size in this paper is 1500 bytes, unless specified otherwise.

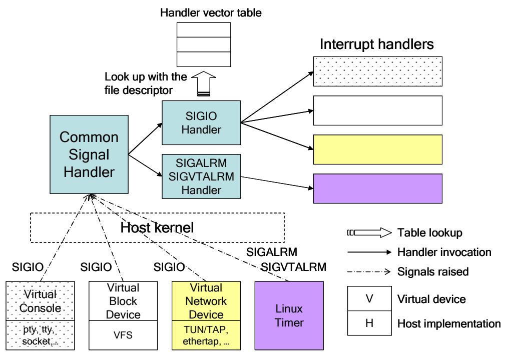

**Figure 4-1 Interrupt handling in UML+Linux.** Virtual devices raise SIGIO or SIGALRM signals for new events. The common signal handler in the UML core receives the signals and invokes appropriate handlers in turn.

For evaluation, we measured the number of system calls invoked by the UML and EUL core for a `sendto()` system call that sends a UDP packet. We show the results in Table 4-1.

**Table 4-1 Latency of a `sendto()` with a UDP packet.** The payload size of 1472 bytes fills one full MTU.

| Payload size | 1472 bytes | |
|---|---|---|
| Operating systems | UML+Linux | EUL+Linux |
| # of `sigprocmask()` | 20 | 0 |
| Time (μs) | 27.83 | 16.37 |
| 95% Confidence Interval (C.I.) | 0.34 | 0.30 |

For each packet, a `sendto()` in UML+Linux requires 20 `sigprocmask()` host system calls: four pairs of interrupt disabling/enabling in the protocol stack and one in a virtual device driver. EUL+Linux requires no system calls, resulting in 42% less elapsed time. Figure 4-2 illustrates the reduced overhead at the packet processing and virtual device layers.

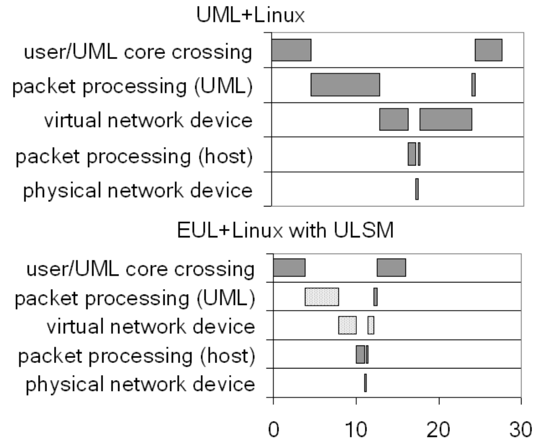

**Figure 4-2 Latency gains due to user-level signal masking**

Since we started the implementation of the EUL core (based on the UML core version 2.4.26), the original UML core has also evolved. The UML core version 2.6.12 includes a mechanism called soft interrupts [29], similar to EUL user-level signal masking.

## 4.2 Aggregated System Calls (AGSC)

**Problem: UML System Call Overhead.** The UML core implements UML system calls in five steps. First, the UML core process uses `wait()` for a system call from an application process to be intercepted by the UML core. Second, the UML core uses `ptrace()` to copy system call arguments to UML-core space. Third, the UML core executes the system call for the application. Fourth, the UML core copies the return value to the application space using `ptrace()`. Fifth, the UML core resumes the application by another `ptrace()` and goes to the first step—waiting. These steps add up to three `ptrace()` and one `wait()` host system calls.

**Solution: Aggregated Host System Calls.** The EUL core avoids the repeated callings of `ptrace()` by expanding the scope of tracing facility slightly. We modified the `wait()` routine in the host kernel. The expanded `wait()` carries out the whole parameter manipulation functions described above. This way, the EUL core crosses the user/kernel boundary only once for each system call, compared with four times in the UML core. In Table 4-2, we show the improvement of the elapsed time for `getpid()` and `sendto()`.

**Table 4-2 Overhead of ULOS system calls**

| Syscalls | `getpid()` | | `sendto()` | |
|---|---|---|---|---|
| Operating systems | UML+ Linux | EUL+ Linux | UML+ Linux | EUL+ Linux |
| Time (µs) | 6.83 | 4.47 | 16.37 | 15.08 |
| 95% C.I. | 0.07 | 0.05 | 0.30 | 0.28 |

## 4.3 Address Translation Cache (ATCA)

**Problem: Address Translation Overhead.** While virtual-to-physical address translation in Linux leverages on hardware, the UML core implements the address translation in software. This software implementation suffers from the additional memory accesses required for traversing page tables. Also, without hardware support for catching traps, the error handling in UML gets inefficient because a segmentation fault signal must be intercepted by the UML core when an unmapped address is referenced. For the protection from accessing the wrong address, the UML core utilizes `sigsetjmp()` and `longjmp()`. The cost of using `sigsetjmp()` for error protection and walking through page tables has an adverse impact on the network performance. As a concrete example, `sendto()` has five arguments, two of which are the address pointers that cause address translations.

**Solution: Address Translation Cache.** To speed up the address translation, we added an address translation cache (ATC) to EUL. ATC is a software version of the TLB (Translation Look-ahead Buffer). The prefix of a translated address is stored in a hash table for future reference. This hash table simplifies the virtual-to-physical address translation. In the `sendto()` example, ATC reduces the overhead of two address translations. Figure 4-3 shows that ATC reduces the latency for the user/UML core boundary crossing and packet processing layers, where copying the destination address and payload requires address translations.

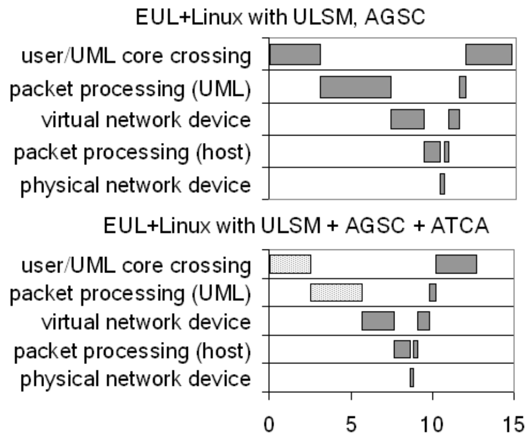

**Figure 4-3 Latency gains due to ATCA**

## 4.4 Shared Socket Buffers (SSKB)

**Problem: Additional Copy across Layers.** For applications to send data over network, the Linux kernel copies the packet content once, from the application buffer to kernel space. On the other hand, the UML core copies the packet twice, once from the user buffer to the UML core, then another time from the UML core into the host kernel.

**Solution: Shared Socket Buffers.** For the implementation of zero-copy between the EUL core and the host kernel, we use a technique similar to fbuf [12]. When an application sends a packet, the packet content is copied into special memory regions shared between the EUL core and the host kernel. The EUL virtual network device passes the identifier of the shared memory region to the host kernel. ZTAP device (for Zero-copy TUN/TAP) in the host kernel locates the shared memory address from the identifier and creates a socket buffer using the address without copying. Then, the new socket buffer is delivered to a network device.

We measured the packet transfer time spent in a virtual network device of UML and EUL. For the experiments with UDP packets, ZTAP in EUL reduces the elapsed time significantly as shown in Table 4-3. EUL packet transfer time using ZTAP is about 60% of UML.

**Table 4-3 Elapsed time of a MTU-sized UDP packet transfer in virtual network devices**

| Virtual network devices | TUN/TAP | ZTAP |
|---|---|---|
| Elapsed time (µs) | 2.90 | 1.68 |
| 95% C.I. | 0.06 | 0.05 |

## 4.5 Network Stack Specialization (NSSP)

To reduce the number of CPU instructions spent in the multi-layered network protocol stack, we use program specialization [5][6][17]. Program specialization has been acknowledged as a powerful technique for optimizing operating system code for a given execution context. Network protocol stack code particularly provides good opportunities for program specialization [5][6], as network parameters, such as IP addresses and port numbers of peers and socket options, tend to be static once a network connection is established.

The network code specialized for the given context contains fewer instructions and branches by:

- Eliminating the mapping between the file descriptor and the kernel-level socket structure.
- Avoiding the interpretation of socket options
- Avoiding making routing decisions for every `sendto()`
- Inlining layered functions

We use specialization templates generated by the Tempo C specializer [17] to implement specialized `sendto()` [5] in the EUL core. The specialized TCP protocol stack template is filled with the values of IP addresses and port numbers when a TCP connection is established. In the UDP case, we assume that a socket tends to send UDP packets to the same end point. (e.g., in multimedia applications) The template is filled with the process id, the socket file descriptor, and the address of the sock structure. If these values change, the specialized code is invalidated and the EUL core switches back to generic code. The following table shows the gains from network specialization.

**Table 4-4 Specialization impact on UDP processing**

| Network stack | EUL UDP | Specialized EUL UDP |
|---|---|---|
| Elapsed time (µs) | 3.23 | 2.83 |
| 95% C.I. | 0.04 | 0.02 |

## 5. PERFORMANCE EVALUATION

## 5.1 Experimental Setup

We conducted our experiments on machines that have a Pentium4 3.06 GHz processor with a 512 KB L2 cache, 533MHz front-side bus, 1 GB of main memory and a gigabit network adapter card. We used the Linux kernel version 2.4.26 for the host kernel and its corresponding UML core, patched by host and guest modifications from the UML source tree. [29]

Our packet processing latency was averaged over 200 runs. We also present the 95% confidence intervals for latency measurements. We use the ttcp tool for measuring the maximum network throughput. Each machine is connected to a gigabit switch.

We show experimental results for four systems: native Linux, UML+Linux, EUL+Linux, and XenLinux+Xen. XenLinux results are added to compare with other virtualization approaches. (XenLinux version 2.6.11 and Xen 2.0.7)

## 5.2 Packet Processing Latency

Table 5-1 shows total packet processing latency for outgoing and incoming MTU-sized UDP packets. EUL+Linux shows less than half the overhead of UML+Linux for both cases.

**Table 5-1 UDP packet processing latency**

| UDP packets | UML+Linux | EUL+Linux | Reduction |
|---|---|---|---|
| Outgoing | 27.47µs | 11.85µs | 57% |
| Incoming | 40.62µs | 18.17µs | 55% |

## 5.3 Sensitivity to Packet Size

### *5.3.1 Latency as a Function of Packet Size*

Figure 5-1 shows the elapsed time of the `sendto()` for various packet sizes. Compared with UML+Linux, the latency for small packets in EUL+Linux is lower by about 60%. For large ones that are fragmented, the slope of EUL+Linux curve is less steep than UML+Linux, since EUL has significantly reduced the overhead. For large packets, EUL+Linux incurs only about three folds the overhead of native Linux, compared with about ten folds of UML+Linux.

### *5.3.2 Throughput as a Function of Packet Size*

Figure 5-2 shows UDP throughput over a gigabit network. Due to the reduced overhead in the EUL core, EUL+Linux outperforms UML+Linux by around three times. For the large-sized packets, the combined optimizations allow EUL+Linux to match the throughput of native Linux even in a gigabit network because the fixed cost per packet is amortized over more bytes.

The elapsed time of `sendto()` for MTU-sized UDP packets is 11.85 microseconds in EUL+Linux. Hence, theoretically, the maximum throughput we can get is 947.7 Mbps, which is larger than the maximum network throughput (around 916 Mbps) of native Linux. For UDP packets with 1024-byte payload, the `sendto()` takes 11.97 microseconds, which limits the maximum throughput to 653 Mbps. The values in Figure 5-3 confirm these calculations.

Figure 5-3 and 5-4 show the results of the same optimization techniques applied to TCP protocol stack. We see the same trend as UDP, although the maximum throughput of EUL+Linux remains about 5% less than that of native Linux.

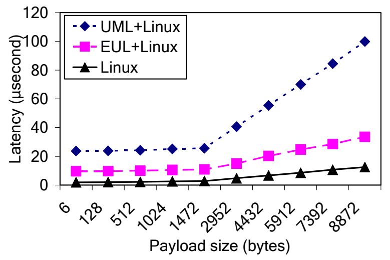

**Figure 5-1 UDP latency with outgoing packets**

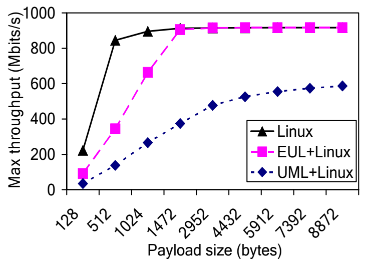

**Figure 5-2 Maximum UDP throughput**

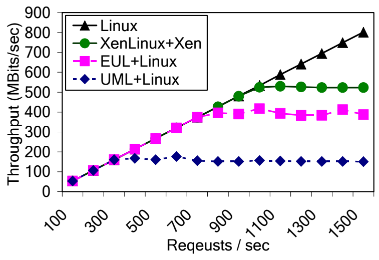

**Figure 5-5 Http server throughput**

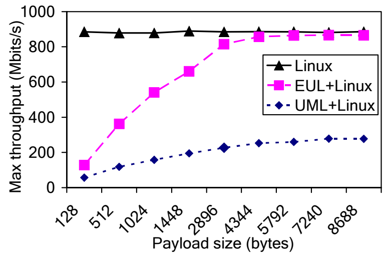

**Figure 5-3 Max. TCP sending throughput**

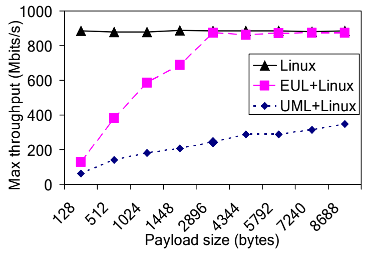

**Figure 5-4 Max. TCP receiving throughput**

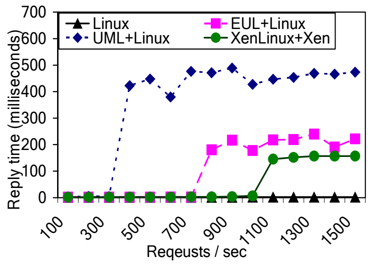

**Figure 5-6 Http server reply time**

### 5.4 HTTP Server Benchmarks

In addition to the micro-benchmarks, we compare the performance of HTTP servers (apache 1.3) using the httperf benchmark [19]. Two httperf clients connected to a gigabit network send requests for 32KB-sized documents to the HTTP server at a constant rate.

Figure 5-5 shows the throughput of the HTTP server for each setup. The server on UML+Linux can process a maximum of 300 requests/sec, while the one on EUL+Linux 700 requests/sec. (The CPU usage of the machines reaches 100% at the saturation)

Figure 5-6 shows that the reply time rapidly increases once the server is saturated. Instead of an exponential growth of input queue and response time, the graph shows a long but constant response time at saturation. This is due to a timeout mechanism in httperf clients, which limits the server load. Figure 5-6 also shows that the server on EUL+Linux has a lower response time (by half) compared with the server on UML+Linux during overload.

## 6. RELATED WORK

Virtual machine (VM) techniques have been explored in earlier efforts. One of the earliest was IBM VM/370, which used virtualization to support legacy binaries [14]. Recent efforts [2][8][26] with type-I VMMs use para-virtualization to improve system performance. User-level VMs were also introduced. Bochs project [27] emulates a number of different x86 processor environments on commodity OSes. Other ULOS approaches were discussed in an earlier section.

UMLinux [7][15] and UML are direct OS port ULOSes. King et al. [15] describe the overhead associated with UMLinux; frequent host context switches, protecting kernel space, and switching between applications. They reduce frequent host context switches by moving some of the UMLinux functionalities into the host kernel. Our aggregated system calls follow a similar approach. The other two sources of overhead described [15] were removed by SKAS host patch [29] in recent releases of UML. Our work can be considered a refinement of UMLinux, both in terms of overhead source analysis and additional optimization techniques.

PlanetlabOS [3] enables a number of users to share the same hardware, providing a virtualized user-level working environment to each user. However, PlanetlabOS, implemented by using Linux vserver [28] and SILK [4], is not a fully virtualized OS because its network subsystem is not virtualized.

Many research efforts have focused on achieving efficient packet processing through collapsing layers. Integrated layer processing (ILP) [1][10] increases performance by reducing redundant copying and buffering. Synthesis kernel [23] collapses layers by in-line code substitution and applying factoring invariants for further optimizations.

In addition to ILP, several schemes have been proposed to move data between layers without copying. Chu [9] describes a zero-copy TCP protocol stack implementation for Solaris using page remapping and copy-on-write. Fbufs [12] uses shared memory space to move data between different address space domains. Our shared socket buffers use the same idea of fbuf.

## 7. CONCLUSIONS

System virtualization efforts have created new opportunities in network server consolidation that builds secure and isolated network systems on shared hardware. Compared with type-I VMMs that run "under" normal operating systems, ULOSes that run "over" host operating systems have been considered too slow due to performance penalties.

In this paper, we analyzed the overheads of layered network protocol in a ULOS and proposed five optimization techniques: user-level signal masking, aggregated system calls, an address translation cache, shared socket buffers, and network stack specialization. Using these optimization techniques, we implemented a ULOS called Enhanced User-mode Linux (EUL).

Our measurements show considerable improvements for network performance over EUL (60% reduction in latency and 300% improvement in network throughput). Perhaps most importantly, the throughput achieved by EUL+Linux in a gigabit network is similar to native Linux, at 5% less for TCP and statistically the same for UDP. These results show that a ULOS can achieve high network throughput and become a serious alternative for network server consolidation.

## ACKNOWLEDGEMENTS

We thank Yasushi Shinjo and Hideki Eiraku for their insightful comments and help with the evaluation and anonymous UML subscribers for giving us the better understanding of the UML core. We would also like to thank the LCN reviewers, Lenin Singaravelu, and Chris Mapes for helping us improve this paper.

## REFERENCES

[1] M. Abbott and L. Peterson, Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, vol. 1, pp. 600-10, 1993.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, and R. Neugebauer. Xen and the art of virtualization. In *Proc. of the ACM SOSP*. Oct. 2003.

[3] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proc. of the NSDI*. San Francisco, CA, 2004.

[4] A. Bavier, T. Voigt, M.Wawrzoniak, and L. Peterson. SILK: Scout Paths in the Linux Kernel. Technical Report 2002–009, Uppsala University, Sweden, February 2002.

[5] S. Bhatia, C. Consel, A. Le Meur, and C. Pu. Automatic Specialization of Protocol Stacks in OS Kernels. In *Proc. of the 29th IEEE Conference on Local Computer Networks*, Tampa, Florida, November 2004.

[6] S. Bhatia, C. Consel, and C. Pu. Remote Customization of Systems Code for Embedded Devices, In *Proc. of the Fourth ACM International Conference on Embedded Software*, Pisa, Italy, September 2004.

[7] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proc. of the IEEE Symposium on High Assurance System Engineering*, pages 95–105, October 2001.

[8] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proc. of the ACM SOSP*, 1997.

[9] J. Chu. Zero-copy TCP in Solaris. In *Proc. of the USENIX Annual Technical Conference*, San Diego, CA, 1996.

[10] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. of the SIGCOMM*, Philadelphia, Pennsylvania, Sept. 1990.

[11] J. Dike. A user-mode port of the Linux kernel. In *5$^{th}$ Annual Linux Showcase & Conference*, Oakland, CA, 2001.

[12] P. Druschel and L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. of the SOSP*, 1993.

[13] H. Eiraku and Y. Shinjo. Running BSD Kernels as User Processes by Partial Emulation and Rewriting of Machine Instructions. In *Proc. of the USENIX BSDCon*, 2003.

[14] R. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pages 34-45, June 1974.

[15] S. King, G. Dunlap, P. Chen. Operating System Support for Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[16] J. Liedtke. On Micro-Kernel Construction. In *Proceedings of the 15th ACM SOSP*, December 1995.

[17] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, and P. Wagle, C. Consel, G. Muller, and R. Marlet. Specialization Tools and Techniques for Systematic Optimization of System Software. *ACM Transactions on Computer Systems*, Vol. 19, No. 2, pp 217-251, May 2001.

[18] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proc. of the USENIX Annual Technical Conference*, pages 279-294, Berkeley, 1996.

[19] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. Performance Evaluation Review, Volume 26, Number 3, December 1998, 31-37.

[20] D. Mosberger, L. Peterson, P. Bridges, and S. O'Malley. Analysis of Techniques to Improve Protocol Processing Latency. In *Proc. of the ACM SIGCOMM*, pages 73-84, Stanford, California, August 1996.

[21] D. Price and A. Tucker. Operating System Support for Consolidating Commercial Workloads. In *Proc. of the 18th LISA Conference*, Atlanta, GA, 2004.

[22] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Proc. of the SOSP*, December 1995.

[23] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis Kernel. *Computing Systems* 1, pp. 11-32, Winter 1988.

[24] J. Sugerman, G. Venkitachalam, B. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proc. of the USENIX Annual Technical Conference*, Boston, MA, 2001.

[25] C. Waldspurger. Memory and Resource Management in VMWare ESX Server. In *Proc. of the OSDI*, 2002.

[26] A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proc. of the OSDI*, Boston, MA, December 2002.

[27] Bochs: The Cross Platform IA-32 Emulator. (http://bochs.sourceforge.net/)

[28] Linux VServers Project. (http://linux-vserver.org/)

[29] User-Mode Linux kernel. (http://user-mode-linux.sourceforge.net/)

[30] VMware: Virtual Infrastructure Software. (www.vmware.com)