A Virtual Instruction Set Interface for Operating System Kernels

John Criswell, Brent Monroe, and Vikram Adve

University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
Email: {criswell,bmmonroe,vadve}@cs.uiuc.edu

***Abstract*— In this paper, we propose and evaluate a virtual instruction set interface between an operating system (OS) kernel and a general purpose processor architecture. This interface is a set of operations added to a previously proposed virtual instruction set architecture called LLVA (Low Level Virtual Architecture) and can be implemented on existing processor hardware. The long-term benefits of such an interface include richer OS-information for hardware, greater flexibility in evolving hardware, and sophisticated analysis and optimization capabilities for kernel code, including across the application-kernel boundary transformations. Our interface design (LLVA-OS) contains several novel features for machine-independence and performance, including efficient saving and restoring of (hidden) native state, mechanisms for error recovery, and several primitive abstractions that expose semantic information to the underlying translator and hardware. We describe the design, a prototype implementation of LLVA-OS on x86, and our experience porting the Linux 2.4.22 kernel to LLVA-OS. We perform a performance evaluation of this kernel, identifying and explaining the root causes of key sources of virtualization overhead.**

I. Introduction

Modern operating system (OS) kernels today are compiled into machine code and use a set of low-level hardware instructions that allows the kernel to configure the OS response to hardware events and to manipulate program state. Because of these choices, substantial parts of the kernel are difficult to analyze, type-check or verify. For example, even basic but crucial properties like memory safety become difficult to enforce. Program analysis techniques for more sophisticated security properties (e.g., enforcing isolation between kernel extensions and the core kernel [1] or analyzing reference checks on sensitive kernel operations [2]) are also handicapped. First, they can only be applied to a limited extent because of the presence of inline assembly code. Second, they must be applied offline because it is difficult to analyze machine code, which means they cannot be used at key moments like loading a kernel extension or installing a privileged program. In practice, such compiler techniques are simply not applied for widely-used legacy systems like Linux or Windows.

An alternative approach that could ameliorate these drawbacks and enable novel security mechanisms is to compile kernel code to a rich, virtual instruction set and execute it using a *compiler-based* virtual machine. Such a virtual machine would incorporate a *translator* from virtual to native code and a run-time system that monitors and controls the execution of the kernel. To avoid the performance penalties of dynamic compilation, the translation does not need to happen online: it can be performed offline and cached on persistent storage.

In previous work, we designed a virtual instruction set called LLVA (Low Level Virtual Architecture) [3] that is low-level enough to support arbitrary languages (including C) but rich enough to enable sophisticated analyses and optimizations. LLVA provides computational, memory access, and control flow operations but lacks operations a kernel needs to configure hardware behavior and manipulate program state.

In this paper, we propose and evaluate a set of extensions to LLVA (called LLVA-OS) that provide an interface between the OS kernel and a general purpose processor architecture. An *Execution Engine* translates LLVA code to machine code and includes a library that implements the LLVA-OS operations. Together, LLVA and the Execution Engine define a virtual machine capable of hosting a complete, modern kernel. We emphasize, however, that incorporating techniques for memory safety or other security mechanisms is not the goal of this paper: those are subjects for future work. We believe LLVA-OS provides a powerful basis for adding such techniques but designing them requires significant further research. We also observe that kernel portability is not a primary goal of this work, even though it may be achieved as a side effect of the virtual instruction set design.

Specifically, the primary contributions of this paper are:

1) The design of LLVA-OS, including novel primitive mechanisms for supporting a kernel that are higher-level than traditional architectures.
2) A prototype implementation of LLVA-OS and a port of the Linux 2.4.22 kernel to LLVA-OS.
3) A preliminary evaluation of the prototype that shows the performance overhead of virtualization in terms of four root causes.

Our evaluation revealed where our design choices added virtualization overhead to the Linux kernel: context switching, data copies between user and kernel memory. read page faults, and signal handler dispatch. Our analysis explores how certain design decisions caused overhead in page faults and data copying and how better implementation could reduce context switching and data copying overheads. Our analysis also shows that many of these overheads do not severely affect overall performance for a small set of applications, and that, in most

cases, these overheads can be reduced with relatively simple changes to the LLVA-OS design or implementation.

Section II describes in more detail the system organization we assume in this work. Section III describes the design of the LLVA-OS interface. Section IV briefly describes our experience implementing LLVA-OS and porting a Linux kernel to it. Section V evaluates the performance overheads incurred by the kernel on the LLVA-OS prototype. Section VI compares LLVA-OS with previous approaches to virtualizing the OS-hardware interface, and Section VII concludes with a brief summary and directions for future work.

II. Background: VISC Architectures and LLVA

Our virtual instruction set extensions are based upon LLVA [3], shown in Fig. 1. This instruction set is implemented using a virtual machine that includes a translator (a code generator), a profile-guided optimizer, and a run-time library used by translated code. This virtual machine is called the Low Level Execution Engine (LLEE).

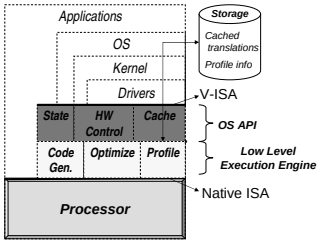

Fig. 1. System Organization with LLVA

The functionality provided by the virtual instruction set can be divided into two parts:

- *Computational interface*, i.e., the core instruction set;
- *Operating System interface*, or LLVA-OS;

The computational interface, defined in [3], is the core instruction set used by all LLVA software for computation, control flow and memory usage. It is a RISC-like, load-store instruction set with arithmetic, memory, condition testing, and control-flow instructions. Compared with traditional machine instruction sets, LLVA includes four novel features: a language-neutral type system suitable for program analysis, an infinite register set in static single assignment (SSA) form [4], an explicit control flow graph per function, and explicit distinctions between *code*, *stack*, *globals*, and *other* memory. These features allow extensive analysis and optimizations to be performed directly on LLVA code, either offline or online [3].

The OS interface, LLVA-OS, is the focus of this work. This interface provides a set of operations that is primarily used by the OS to control and manipulate architectural state.

III. Design of the OS Interface

Operating systems require two types of support at the OS-hardware interface. First, the OS needs to access specialized and privileged hardware components. Such operations include registering interrupt handlers, configuring the MMU, and performing I/O. Second, the OS needs to manipulate the state of itself and other programs, e.g. context switching, signal delivery, and process creation.

A. Design Goals for the OS Interface

To achieve our primary long term goal of designing an architecture that improves the security and reliability of system software, we designed LLVA-OS with the following goals:

- The LLVA-OS must be designed as a set of abstract (but primitive) operations, independent of a particular hardware ISA.
- The LLVA-OS must be OS neutral (like current native instruction sets) and should not constrain OS design choices.
- The LLVA-OS must be light weight and induce little performance overhead.

The first goal is beneficial to enhance kernel portability across some range of (comparable) architectures. Although it may be similar to a well-designed portability layer within the kernel, moving this into the virtual instruction set can expose greater semantic information about kernel operations to the underlying translator and even the hardware. For example, it can allow the translator to perform sophisticated memory safety checks for the kernel by monitoring page mappings, system calls, and state-saving and restoring operations. The second goal allows our work to be applied to a variety of operating systems used in a wide range of application domains. The third goal is important because our aim is to incorporate the virtual machine model below widely-used, legacy operating systems where any significant performance overhead would be considered unacceptable.

B. Structure of the Interface

We define LLVA-OS, semantically, as a set of functions, i.e., an API, using the syntax and type system of the core LLVA instruction set. We call these functions *intrinsic functions*, or intrinsics, because their semantics are predefined, like intrinsics in high-level languages. When compiling a call to such a function, the translator either generates inline native code for the call (effectively treating the function like a virtual instruction) or generates a call to a run-time library within the Execution Engine containing native code for the function.

Using function semantics for the interface, instead of defining it via explicit LLVA instructions, provides two primary benefits. First, uses of these functions appear to a compiler as ordinary (unanalyzable) function calls and therefore do not need to be recognized by any compiler pass except those responsible for translation to native code. Second, using functions makes the control flow of the operations more explicit, since most operations behave like a called function: some code is executed and then control is returned to the caller.

C. Virtual and Native System State

The state of an executing program in LLVA-OS can be defined at two levels: virtual state and native state. Virtual state is the system state as seen by external software, including the OS. The virtual state of an LLVA program includes:

- The set of all virtual registers for all currently active function activations.
- The implicit program counter indicating which virtual instruction to execute next.
- The contents of memory used by the current program.
- The current stack pointer indicating the bottom of the currently active stack.
- The current privilege mode of the processor (either privileged or unprivileged).
- A set of MMU control registers.
- A local interrupt enable/disable flag.

Native state is the state of the underlying physical processor and includes any processor state used by a translated program, such as general purpose, floating point, and control flow registers. It may also include the state of co-processors, such as MMUs or FPUs.

A key design choice we made in LLVA-OS is to expose the *existence* of native state while keeping the *contents* hidden. In particular, we provide functions that can save and restore native state without being able to inspect or manipulate the details of native state directly. This design choice is motivated and explained in Section III-D.

D. Manipulating Virtual and Native State

There are two broad classes of operations that one program may use to manipulate the state of *another, separately compiled, program*: (i) saving and restoring the entire state of a program, e.g., OS context switching routine, and (ii) directly inspecting or modifying the details of program state, e.g., by a debugger.

Some virtual machines, e.g., the Java Virtual Machine (JVM) [5], encapsulate the saving and restoring of state entirely inside the VM. This yields a simple external interface but moves policy decisions into the VM. This model is inadequate for supporting arbitrary operating systems, without significant design changes to allow external control over such policies. Other virtual machines, such as the Transmeta [6] and DAISY [7] architectures based on binary translation, allow external software to save or restore virtual program state but hide native state entirely. This requires that they maintain a mapping at all times beween virtual and native state. Maintaining such a mapping can be expensive without significant restrictions on code generation because the mapping can change frequently, e.g., every time a register value is moved to or from the stack due to register spills or function calls.

We propose a novel solution based on two observations. First, operations of class (i) above are frequent and performance-sensitive while those of (ii) are typically not. Second, for class (i), an OS rarely needs to inspect or modify individual virtual registers in order to save and restore program state. We provide a limited method of allowing the OS to directly save and restore the native state of the processor to a memory buffer. The native state is visible to the OS only as an opaque array of bytes.

For operations of class (ii), the translator can reconstruct the virtual to native mapping lazily when required. This moves the mapping process out of the critical path.

We provide the functions in Table I to save and restore native state. Because floating point registers do not always need to be saved but integer registers usually do, we divide native processor state into two sets. The first set is the Integer State. It contains all native processor state used to contain non-floating point virtual registers and the virtual software's control flow. On most processors, this will include all general purpose registers in use by the program, the program counter, the stack pointer, and any control or status registers. The second set is the Floating Point (FP) State. This set contains all native state used to implement floating point operations. Usually, this is the CPU's floating point registers. Additional sets can be added for state such as virtual vector registers (usually represented by vector co-processors in native state). Native state not used by translated code, e.g. special support chips, can be accessed using I/O instructions (just as they are on native processors) using the I/O instructions in Table III.

E. Interrupts and Traps

LLVA-OS defines a set of interrupts and traps identified by number. These events are serviced by handler functions provided by the OS. LLVA-OS provides intrinsics that the OS uses to register handler functions for each interrupt or trap (these are omitted for space but are in [8]). On an interrupt or trap, a handler function is passed the interrupt or trap number and a pointer to an Interrupt Context, defined below. Page fault handlers are passed an additional address parameter.

When an interrupt or trap occurs, the processor transfers control to the Execution Engine, which saves the native state of the interrupted program (on the kernel stack) before invoking the interrupt or trap handler. Saving the entire Integer State would be sufficient but not always necessary. Many processors provide mechanisms, e.g., shadow registers, to reduce the amount of state saved on an interrupt.

To take advantage of such hardware, when available, we define the *Interrupt Context*: a buffer of memory reserved on the kernel stack capable of holding the complete Integer State when an interrupt or trap occurs. Only the subset of Integer State that will be overwritten by the trap handling code need be saved in the reserved memory by the Execution Engine. Any other part of Integer State masked by facilities such as shadow registers is left on the processor.

In cases where the complete Interrupt Context must be committed to memory, e.g., signal handler dispatch, we provide intrinsics that can commit all the Integer State inside of an Interrupt Context to memory. This allows us to lazily save interrupted program state when needed. Interrupt and trap handlers can also use the Interrupt Context to manipulate the state of an interrupted program. For example, an OS trap handler

TABLE I
FUNCTIONS FOR SAVING AND RESTORING NATIVE PROCESSOR STATE

Name	Description
llva.save.integer(void * buffer)	Save the Integer State of the native processor in to the memory pointed to by *buffer*.
llva.load.integer(void * buffer)	Load the integer state stored in *buffer* back on to the processor. Execution resumes at the instruction immediately following the llva.save.integer() instruction that saved the state.
llva.save.fp(void * buffer, int always)	Save the FP State of the native processor or FPU to the memory pointed to by *buffer*. If *always* is 0, state is only saved if it has changed since the last llva.load.fp(). Otherwise, save state unconditionally.
llva.load.fp(void * buffer)	Load the FP State of the native processor (or FPU) from a memory buffer previously used by llva.save.fp().

can push a function call frame on to an interrupted program's stack using llva.ipush.function(), forcing it to execute a signal handler when the trap handler finishes. Table II summarizes the various functions that manipulate the Interrupt Context.

F. System Calls

LLVA-OS provides a finite number of system calls identified by unique numbers. Similar to interrupt and trap handlers, the OS registers a system call handler function for each system call number with an LLVA-OS intrinsic function.

Software initiates a system call with the llva.syscall intrinsic (Table III). Semantically, this appears much like a function call. They only differ in that system calls switch to the privileged processing mode and call a function within the OS.

Unlike current designs, an LLVA processor knows the semantic difference between a system call and an instruction trap and can determine the system call arguments. This extra information allows the same software to work on different processors with different system call dispatch mechanisms, enables the hardware to accelerate system call dispatch by selecting the best method for the program, and provides the Execution Engine and external compiler tools the ability to easily identify and modify system calls within software.

G. Recovery from Hardware Faults

Some operating systems use the MMU to efficiently catch errors at runtime, e.g., detecting bad pointers passed to the write system call [9]. The OS typically allows the page fault handler to adjust the program counter of the interrupted program so that it executes exception-handling assembly code immediately after the page fault handler exits. In an LLVA-supported kernel, the OS cannot directly change the program counter (virtual or native), or write assembly-level fault handling code. Another mechanism for fault recovery is needed.

We observe that recovering from kernel hardware faults is similar to exception handling in high level languages. The LLVA instruction set [10] provides two instructions (invoke and unwind) to support such exceptions. We adapted these instructions to support kernel fault recovery in LLVA-OS.

The invoke intrinsic (described in Table III) is used within the kernel when calling a routine that may fault; the return value can be tested to branch to an exception handler block. Invokes may be nested, i.e., multiple invoke frames may exist on the stack at a time.

If the called function faults, the trap handler in the OS calls llva.iunwind. This function unwinds control flow back to the closest invoke stack frame (discarding all intervening stack frames) and causes the invoke intrinsic to return 1. The only difference between the original unwind instruction and llva.iunwind is that the latter unwinds the control flow in an Interrupt Context (to return control to the kernel context that was executing when the fault occurred) while the former unwinds control flow in the current context.

This method uses general, primitive operations and is OS neutral. However, its performance depends on efficient code generation of invoke by the translator [10]. Our current version of LLEE has implementation and design deficiencies as described in Section V-B.

To address these, we added a combined invoke/memcpy intrinsic named llva.invokememcpy. This intrinsic uses efficient native instructions for data copying and always returns the number of bytes successfully copied (even if control flow was unwound by llva.iunwind).

H. Brief Overview of Virtual Memory and I/O

Our current LLVA-OS design assumes a hardware page table with no explicit memory regions. Intrinsics for changing the page table pointer and analyzing the cause of page faults are described in Table III. We have yet to design intrinsics for abstracting the page table format; that is left as future work.

The I/O functions, used to communicate with I/O devices and support chips, are briefly described in Table III.

IV. PROTOTYPE IMPLEMENTATION

While designing LLVA-OS, we implemented a prototype Execution Engine and ported the Linux 2.4.22 kernel to our virtual instruction set. This essentially worked as a port of Linux to a new instruction set. It took approximately one-person-year of effort to design and implement LLVA-OS and to port Linux to it.

Our Execution Engine is implemented as a native code library written in C and x86 assembly. It can be linked to an OS kernel once the kernel has been compiled to native code. It provides all of the functionality described in Section III and does not depend on any OS services.

Compilation to the LLVA instruction set is done using LLVM [10]. Since the LLVM compiler currently requires OS services to run, all code generation is performed ahead of time.

To port the Linux kernel, we removed all inline assembly code in i386 Linux and replaced it with portable C code or C code that used LLVA-OS.

TABLE II

FUNCTIONS FOR MANIPULATING THE INTERRUPT CONTEXT

Name	Description
llva.icontext.save (void * icp, void * isp)	Save the Interrupt Context *icp* into the memory pointed to by *isp* as Integer State.
llva.icontext.load (void * icp, void * isp)	Load the Integer State *isp* into the Interrupt Context pointed to by *icp*.
llva.icontext.get.stackp (void * icp)	Return the stack pointer saved in the Interrupt Context.
llva.icontext.set.stackp (void * icp)	Set the stack pointer saved in the Interrupt Context.
llva.ipush.function (void * icp, int (*f)(...), ...)	Modify the state in the Interrupt Context *icp* so that function *f* has been called with the given arguments. Used in signal handler dispatch.
llva.icontext.init (void * icp, void * stackp)	Create a new Interrupt Context on the stack pointed to by *stackp*. It is initialized to the same values as the Interrupt Context *icp*.
llva.was.privileged (void * icp)	Return 1 if the Interrupt Context *icp* was running in privileged mode. Return 0 otherwise.

TABLE III

SYSTEM CALL, INVOKE, MMU, AND I/O FUNCTIONS

Name	Description
int llva.syscall (int sysnum, ...)	Request OS service by calling the system call handler associated with number *sysnum*.
int llva.invoke (int * ret, int (*f)(...), ...)	Call function *f* with the specified arguments. If control flow is unwound before *f* returns, then return 1; otherwise, return 0. Place the return value of *f* into the memory pointed to by *ret*.
int llva.invokememcpy (void * to, void * from, int count)	Copy *count* bytes from *from* to *to*. Return the number of bytes successfully copied before a fault occurs.
llva.iunwind (void * icp)	Modify the state in the Interrupt Context pointed to by *icp* so that control flow is unwound to the innermost frame in which an invoke was executed.
void llva.load.pgtable (void * pgtable)	Load the page table pointed to by *pg*.
void * llva.save.pgtable (void)	Return the page table currently used by the MMU.
void llva.flush.tlbs (int global)	Flush the TLBs. If *global* is 1, remove global TLB entries.
void llva.flush.tlb (void * addr)	Flush any TLBs that are contain virtual address *addr*.
int llva.mm.protection (void * icp)	Return 1 if the memory fault was caused by a protection violation.
int llva.mm.access.type (void * icp)	Returns 1 if memory access was a read; 0 if it was a write.
int llva.mm.was.absent (void * icp)	Returns 1 if the memory access faulted due to translation with an unmapped page.
int llva.ioread (void * ioaddress)	Reads a value from the I/O address space.
void llva.iowrite (int value, void * ioaddress)	Writes a value into the I/O address space.

During the first phase of development, we continued to compile the kernel with GCC and linked the Execution Engine library into the kernel. This allowed us to port the kernel incrementally while retaining full kernel functionality. This kernel (called the LLVA GCC kernel below) has been successfully booted to multi-user mode on a Dell OptiPlex, works well enough to benchmark performance, and is capable of running many applications, including the thttpd web server [11], GCC, and most of the standard UNIX utilities.

We have also successfully compiled the LLVA-ported kernel with the LLVM compiler [10], demonstrating that the kernel can be completely expressed in the LLVA instruction set. This kernel also boots into multi-user mode on real hardware.

V. PRELIMINARY PERFORMANCE EVALUATION

We performed a preliminary performance evaluation on our prototype to identify key sources of overhead present in our current design. To do this, we benchmarked the LLVA GCC and native i386 Linux kernels. We ran all of our tests on a Dell OptiPlex with a 550 MHz. Pentium 3 processor and 384 MB of RAM. Since both kernels are compiled by the same compiler (GCC) and execute on identical hardware, the difference between them is solely due to the use of LLVA-OS as the target architecture in the former. The use of this interface produces multiple sources of overhead (described below) because a kernel on LLVA-OS uses *no native assembly code*, whereas the original kernel uses assembly code in a number of different ways. It is important to note that the use of the interface alone (with no additional hardware or translation layer optimizations) does not improve performance in any way, except for a few incidental cases mentioned below.

We do not compare the LLVA kernel compiled with LLVM to the above two kernels in this work. Doing so compares the quality of code generated by the LLVM and GCC compilers which, while useful long term, does not help us identify sources of overhead in the kernel.

We used three different types of benchmarks to study the performance impact of LLVA-OS. Nano benchmarks test primitive kernel functions, e.g., system call and trap latency, that typically use only a few LLVA-OS intrinsics. These overheads can be classified according to one of the four causes below. Micro benchmarks measure the performance of high-level kernel operations directly used by applications, such as specific system calls. Finally, macro benchmarks are entire applications and measure the performance impact of the abstractions on overall system performance.

Unless stated otherwise, all benchmarks use the HBench-OS framekwork [12] and present the average measurement of 100 iterations.

TABLE IV
NANONBENCHMARKS. TIMES IN μS. SOME NUMBERS FROM [8].

Operation	Native	LLVA	% Ovhd	LLVA-OS Intrinsics
System Call Entry	.589	.632	7.30	llva.syscall
Trap Entry	.555	.450	-18.92	Internal to LLEE
Read Page Fault	1.105	1.565	41.63	llva.mm.protection, llva.mm.access.type, llva.mm.was.absent
Kernel-User 1KB memcpy	.690	.790	14.45	llva.invokememcpy
User-Kernel 1KB strcpy	.639	.777	21.64	invoke

A. Sources of Overhead

There are four distinct causes of overhead when a kernel uses the LLVA-OS virtual architecture on a particular processor, compared with an identical kernel directly executing on the same hardware:

1) The native kernel used assembly code to increase performance and the same operation on LLVA must be written using C code, e.g. IP Checksum code on x86 can exploit the processor status flags to check for overflow, saving a compare instruction.
2) The native kernel used assembly code for an operation (for performance or hardware access) and LLVA-OS provides an equivalent function, but the function is not implemented efficiently.
3) The native kernel used assembly code to exploit a hardware feature and this feature is less effectively used in the LLVA-OS design, i.e., a mismatch between LLVA-OS design and hardware.
4) The native kernel exploited OS information for optimizing an operation and this optimization is less effective with LLVA-OS, i.e., a mismatch between LLVA-OS design and kernel design.

These sources of overhead are important to distinguish because the solution for each is different. For example, the first two sources above have relatively simple solutions: the first by adding a new intrinsic function to LLVA; the second simply by tuning the implementation. Mismatches between LLVA-OS design and either hardware features or OS algorithms, however, are more difficult to address and require either a change in LLVA-OS or in the design of the OS itself.

B. Nanobenchmarks

We used the benchmarking software from [8] to measure the overhead for a subset of primitive kernel operations (Table IV). These tests are based on HBench-OS tests [12] and use specialized system calls in the kernel to invoke the desired nano-benchmark feature. Many other primitive operations, particularly synchronization and bit manipulation primitives, are presented in [8].

As seen in Table IV, only a few primitive operations incur significant overhead on LLVA-OS. Note that these operations are extremely short (microseconds). Small inefficiencies produce large relative overhead, but their impact on higher-level kernel operations and applications is usually far smaller.

We improve performance for trap entry. The i386 kernel must support traps from Virtual 8086 mode. The LLVA kernel does not, yielding simpler trap entry code.

The slight increase in system call latency is because the native kernel saves seven fewer registers by knowing that they are not modified by system call handlers [8]. Some of the overhead is also due to some assembly code (which initiates signal dispatch and scheduling after every system call) being re-written in C. The overhead is partially due to a mismatch between LLVA-OS design and Linux design and partially due to the inability to write hand-tuned assembly code. A read page fault has relative overhead of 42% (but the absolute difference is tiny) [8] and is a mismatch between LLVA-OS and hardware. The native Linux page fault handler uses a single bitwise operation on a register to determine information about the page fault; the LLVA kernel must use two LLVA-OS intrinsics to determine the same information [8].

The overhead for copying data between user and kernel memory stems from several sources. Both the invoke instruction (used by strcpy) and llva.invokememcpy (used by memcpy) save a minimal amount of state so that llva.iunwind can unwind control flow. The invoke instruction adds function call overhead since it can only invoke code at function granularity. Finally, the invoked strcpy function is implemented in C. The i386 kernel suffers none of these overheads.

Reference [8] found that, for a small number of processes, context switching latency doubled with the LLVA kernel due to LLVA-OS saving more registers than the original context switching code. However, once the system reaches 60 active processes, the overhead of selecting the next process to run is greater than the LLVA-OS overhead due to a linear time scan of the run queue [8], [9]. For a large number of processes, LLVA-OS only adds 1-2 microseconds of overhead.

C. Microbenchmarks

We used the HBench-OS benchmark suite [12] to measure the latency of various high level kernel operations. We also used a test developed by [8] that measures the latency of opening and closing a file repeatedly. Tables V and VI present the results.

All the system calls listed in Tables V and VI incur the system call entry overhead in Table IV. All of the I/O system calls incur the memcpy overhead, and any system call with pathname inputs incurs the strcpy overhead.

Signal handler dispatch shows the greatest overhead (48%). We know that some of the overhead comes from system call overhead (the re-written dispatch code mentioned previously) and some comes from saving the application's state in kernel memory instead of on the user space stack. We suspect that the greatest amount of overhead, though, comes from saving and restoring the FP State on every signal handler dispatch. If true, we will need to revise the FP State intrinsic design to remove the overhead.

TABLE V
HIGH LEVEL KERNEL OPERATIONS. TIMES ARE IN μS.

Operation	Program	Native	LLVA	% Ovhd
open / close	N/A	4.43	5.00	12.93
fork and exit	lat_proc null	268.25	279.19	4.08
fork and exec	lat_proc simple static	1026	1100	7.25
Signal Handler Install	lat_sig install	1.36	1.52	12.06
Signal Handler Dispatch	lat_sig handle	3.00	4.44	48.18

TABLE VI
READ BANDWIDTH FOR 4 MB FILE (BW_FILE_RD AND BW_PIPE).

Buffer Size (KB)	File Native (MB/s)	File LLVA (MB/s)	Decrease (%)	Pipe Native (MB/s)	Pipe LLVA (MB/s)	Decrease (%)
4	276.27	271.47	1.74	264.17	200.84	23.97
16	179.71	177.70	1.12	206.31	170.48	17.37
64	183.20	182.57	0.34	212.59	177.85	16.34
256	111.71	112.60	-0.80	190.14	163.82	13.84
1024	84.37	86.39	-2.39	133.49	118.68	11.09
4096	86.16	86.37	-0.24	120.28	108.92	9.44

We believe the signal handler install overhead is partially due to the system call latency and partially due to the overhead from copying data from user space to kernel space (on our Linux system, signal() is implemented by a library routine that calls sigaction(), which reads from several user space buffers).

D. Macrobenchmarks

We ran two application-level benchmarks to evaluate the overall performance impact LLVA-OS has on two types of common applications: web servers and compilation.

First, we ran the WebStone benchmark [13] on the thttpd web server [11] (Table VII). WebStone measures the bandwidth that a web server can provide to a variable number of clients. Our test retrieves files of size 0.5KB to 5MB. The thttpd web server uses a single process and thread to handle I/O requests from web clients [11].

The LLVA kernel decreases the bandwidth of the server by less than 7%. We are still trying to determine the cause of the decrease; we have, however, determined that data copying overhead is not the primary cause, unlike an earlier version of the system that had higher overheads [8].

We also benchmarked a build of OpenSSH 4.2p1 [14] using the time command. A complete build took 176.78 seconds on the native kernel and 178.56 seconds on the LLVA kernel, yielding a negligible 1.01% overhead. The times are elapsed time (the LLVA kernel does not correctly report per process user or system time). The low overhead is because compilation is primarily a user-level, CPU-bound task.

TABLE VII
WEB SERVER BANDWIDTH MEASURED IN MEGABITS/SECOND.

# Clients	Native	LLVA	% Decrease
1	81.21	76.16	6.22
4	72.21	68.76	4.78

VI. RELATED WORK

There are several classes of related work, including (a) OS support in virtualized processor architectures (or "codesigned virtual machines" [15]); (b) hardware abstractions in previous operating systems; (c) virtual machine monitors; and (d) operating systems that exploit language-based virtual machines or safe languages.

Three previous system architectures have used an organization with separate virtual and native instruction sets: the Transmeta Crusoe and its successors [6], the IBM DAISY project [7], and the IBM S/38 and AS/400 families of systems [16]. Both Transmeta and DAISY emulated an existing, "legacy" hardware ISA (x86 and PowerPC, respectively) as their virtual instruction set on a completely hidden VLIW processor. Both allowed existing operating systems written for the legacy instruction sets to be run with virtually no changes. Unlike LLVA-OS, these systems aim to implement legacy instruction sets on novel hardware, but do not provide any benefits to the OS.

The IBM AS/400 (building on early ideas in the S/38) defined a high-level, hardware-independent interface called the Machine Interface (MI) that was the sole interface for all application software and for much of OS/400. The major difference from our work is that their MI was effectively a part of the operating system (OS/400); significant components of OS/400 ran below the MI and were required to implement it. In their approach, a particular OS is organized to enable many or all of the benefits listed in Section I. LLVA-OS is OS-neutral and aims to provide similar benefits to *existing* operating systems without a major reorganization of the OS.

Many modern operating systems include some design features to separate machine-independent and machine-dependent code, and at least a few do this by using an explicit architecture abstraction layer to define an interface to the hardware. Two such examples include the Windows Hardware Abstraction Layer (HAL) [17] and the Choices nanokernel [18]. These layers are integral parts of the OS and only achieve greater machine-independence and portability. In contrast, our OS interface is an integral part of the (virtual) processor architecture and yet provides richer primitives than a traditional architecture for supporting an OS.

The Alpha processor's PALCode layer [19] provides an abstraction layer similar to LLVA-OS. PALCode is special privileged code running below the OS that can execute special instructions and access special registers, e.g. the TLB. PALCode routines, like LLVA-OS intrinsics, act like additional instructions in the processor's instruction set, have privileged access to hardware, and provide a higher level interface to the processor compared with traditional instruction sets. PALCode's goals, however, differ significantly from ours. PALCode is co-designed with the OS and moves *OS specific*

functions into the abstraction layer. It does not hide the processor's native ISA from the OS. Because it is OS-defined, it is unrecognizable by external compilers. In contrast, LLVA-OS hides the entire native ISA from software, provides OS-neutral abstractions, and has well-defined semantics. PALCode is logically part of the OS, whereas LLVA-OS is part of the processor architecture.

Virtual machine monitors such as VMWare [20], Denali [21] and Xen [22] virtualize hardware resources to enable sharing and isolation for multiple instances of operating systems. These systems are orthogonal to our work, which virtualizes the instruction set interface for a single instance of an OS.

Several operating systems, such as JX [23] and Singularity [24], are written largely in safe languages (Java and C# respectively) and compiled into typed bytecode languages implemented via a virtual machine. These systems use language safety features to provide increased reliability and component isolation. These benefits, however, require profound OS redesign and the use of safe languages. In contrast, we do not (yet) provide a safe execution environment but focus on hardware abstraction, and we enable existing operating systems to be ported to our interface with relatively little effort.

Other systems, including JavaOS [25], J-Kernel [26], JRes [27], KaffeOS [28] and the Java extension defined in JSR-121 [29], have incorporated OS-like features into or as a layer on top of a JVM. These systems aim to provide OS services to Java applications but are not designed to be general-purpose operating systems.

VII. Summary and Future Work

We have designed and implemented a virtual instruction set interface between OS kernels and ordinary hardware. We have successfully ported the Linux kernel to this interface, proving that it is capable of supporting a real world, multi-user OS.

Preliminary measurements indicate that performance is reasonable for some aspects of kernel execution, but improvement can still be made for context switching, signal dispatch, read page faults, and copying data between user and kernel memory.

In the near future, we will continue to tune our implementation to eliminate overhead. We also plan to enhance the LLEE so that it can do online code generation, profiling, and caching of translations. Longer term, we plan to continue exploring the performance and security possibilities afforded by our architecture.

References

[1] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 2003, pp. 207–222.

[2] V. Ganapathy, T. Jaeger, and S. Jha, "Automatic placement of authorization hooks in the linux security modules framework," in *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*. New York, NY, USA: ACM Press, 2005, pp. 330–339.

[3] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke, "LLVA: A Low-Level Virtual Instruction Set Architecture," in *Proc. ACM/IEEE Int'l Symp. on Microarchitecture (MICRO)*, San Diego, CA, Dec. 2003, pp. 205–216.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, pp. 13(4):451–490, October 1991.

[5] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Reading, MA: Addison-Wesley, 1997.

[6] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing Software: Using speculation, recovery and adaptive retranslation to address real-life challenges," in *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, San Francisco, CA, Mar 2003.

[7] K. Ebcioglu and E. R. Altman, "DAISY: Dynamic compilation for 100% architectural compatibility," in *Proc. Int'l Conf. on Computer Architecture (ISCA)*, 1997, pp. 26–37.

[8] B. M. Monroe, "Measuring and improving the performance of Linux on a virtual instruction set architecture," Master's thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Urbana, IL, Dec 2005.

[9] D. P. Bovet and M. Cesati, *Understanding the LINUX Kernel*, 2nd ed. Sebastopol, CA: O'Reilly, 2003.

[10] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, San Jose, Mar 2004.

[11] J. Poskanze, "thttpd - tiny/turbo/throttling http server," 2000. [Online]. Available: http://www.acme.com/software/thttpd

[12] A. Brown, "A decompositional approach to computer system performance," Ph.D. dissertation, Harvard College, April 1997.

[13] Mindcraft, "Webstone: The benchmark for webservers," 2002. [Online]. Available: http://www.mindcraft.com/webstone

[14] The OpenBSD Project, "Openssh," 2006. [Online]. Available: http://www.openssh.com

[15] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. 500 Sansome Street, Suite 400, San Francisco CA, 94111: Morgan Kaufmann, June 2005.

[16] B. E. Clark and M. J. Corrigan, "Application System/400 performance characteristics," *IBM Systems Journal*, vol. 28, no. 3, pp. 407–423, 1989.

[17] M. E. Russinovich and D. A. Solomon, *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Redmond, WA, USA: Microsoft Press, 2004.

[18] R. Campbell, N. Islam, P. Madany, and D. Raila, "Designing and implementing Choices: An object-oriented system in C++," *Communications of the ACM*, vol. 36, no. 9, pp. 36(9):117–126, 1993.

[19] Compaq Computer Corporation, *Alpha Architecture Handbook*. Compaq Computer Corporation, 1998.

[20] VMWare, "VMWare," 2006. [Online]. Available: http://www.vmware.com

[21] A. Whitaker, M. Shaw, and S. D. Gribble, "Scale and performance in the denali isolation kernel," in *Proc. Fifth Symp. on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, Dec 2002.

[22] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the art of virtualization," in *Proc. ACM Symp. on Operating Systems Principles*, October 2003.

[23] M. Golm, M. Felser, C. Wawersich, and J. Kleinoder, "The JX Operating System," in *Proc. USENIX Annual Technical Conference*, Monterey, CA, June 2002, pp. 45–58.

[24] G. C. Hunt and J. R. Larus, "Singularity Design Motivation (Singularity Technical Report 1)," Microsoft Research, Redmond, WA, Tech. Rep. MSR-TR-2004-105, Dec 2004.

[25] T. Saulpaugh and C. Mirho, *Inside the JavaOS Operating System*. Addison-Wesley, 1999.

[26] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken, "Implementing multiple protection domains in Java," in *Proc. 1998 USENIX Annual Technical Conference*, June 1998.

[27] G. Czajkowski and T. von Eicken, "JRes: A resource accounting interface for Java," in *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, 1998, pp. 21–35.

[28] G. Back and W. C. Hsieh, "The KaffeOS Java runtime system," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 4, pp. 583–630, 2005.

[29] Java Community Process, "JSR 121," 2003. [Online]. Available: http://jcp.org/jsr/detail/121.jsp