CS 497yyz Project Report: LLVA-emu

Misha Brukman Brian Gaeke
{brukman,gaeke}@uiuc.edu

December 12, 2003

1 Introduction and Background

The LLVM compiler infrastructure [11] employs an intermediate representation which is similar to a typed assembly language. LLVM's representation is fully typed, RISC-like, has data-flow information in Static Single-Assignment form built-in, and is powerful enough to express any C or C++ program, type-safe or otherwise. It is superior to typical bytecode and machine-code formats because it retains sophisticated control-flow and type information about the values used in each and every instruction of every function, thus making useful compiler analyses and optimizations cheaper to perform. The LLVM system includes static and dynamic (just-in-time) compilers, as well as a small and growing suite of program analysis tools and optimizations.

Traditional architectures use the hardware instruction set for dual purposes: first, as a language in which to express the semantics of software programs, and second, as a means for controlling the hardware. The thesis of the Low-Level Virtual Architecture (LLVA) [1] project is to decouple these two uses from one another, allowing software to be expressed in a semantically richer, more easily-manipulated format, and allowing for more powerful optimizations and whole-program analyses directly on compiled code.

The semantically rich format we use in LLVA, which is based on the LLVM compiler infrastructure's intermediate representation, can best be understood as a "virtual instruction set". This means that while its instructions are closely matched to those available in the underlying hardware, they may not correspond exactly to the instructions understood by the underlying hardware. These underlying instructions we call the "implementation instruction set." Between the two layers lives the translation layer, typically implemented in software. The structure of this two-layer architecture can be seen in Figure 1.

In this project, we have taken our next logical steps in this effort by (1) porting the entire Linux kernel to LLVA, and (2) engineering an environment in which a kernel can be run directly from its LLVM bytecode representation — essentially, a minimal, but complete, emulated computer system with LLVA as its native instruction set.[1] The emulator we have invented, `llva-emu`, executes

[1]We use "LLVA instruction set" and "LLVM bytecode" interchangeably in this paper; there

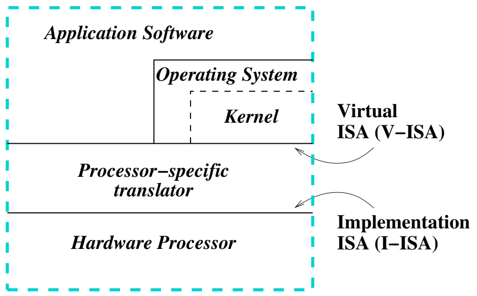

Figure 1: The LLVA execution manager with the virtual and implementation ISAs.

kernel code by translating programs "just-in-time" from the LLVM bytecode format to the processor's native instruction set. In Figure 1, you can conceive of `llva-emu` as representing the "Processor-specific translator" and "Hardware Processor" layers combined.

Previous work has addressed some of the same issues — that is, of transparently running programs compiled to LLVA; however, those efforts, including the Jello JIT, which evolved into the standard LLVM JIT, and thence into LLEE, have been defined to work only on user-level programs [1, 12]. The current project has extended this work to apply to arbitrary kernel code or standalone (e.g., embedded-system) code.

1.1 Details

1.1.1 OS Functionality

Operating-system kernels necessarily involve machine-dependent code, for manipulating hardware structures involved in process and memory management, for doing I/O in device drivers, etc. These machine-dependent functions require special handling in the LLVM representation, which is largely machine-independent. The structure of the LLVM bytecode permits the addition of so-called "intrinsic functions", each of which corresponds to a piece of functionality that must be handled idiosyncratically for each supported processor architecture. One of the outcomes of this project is the relatively small set of intrinsic functions that we needed in order to boot Linux to a point where it can handle interrupts and process I/O.

1.1.2 Environment

The environment in which we run our ported OS kernel, called `llva-emu`, can be understood as a simple whole-machine simulator similar to Virtutech's Simics or

do exist minor differences, but they are not relevant to this project.

Stanford's SimOS [14]. We built it by extending our existing execution engine for LLVA bytecode, based on a just-in-time compiler and bytecode loader, with simple emulated devices and facilities for supporting operating-system kernel code. By leveraging our existing user-level execution engine, we are able to use the host operating system's ordinary process mechanisms and APIs to provide the emulated LLVA machine with memory, emulated I/O devices, timer interrupts, and the like.

This implementation scheme is superficially similar to the User-Mode Linux project, but with some important differences: 1) the LLVA Linux kernel does not depend on the existence of a host operating system – only our *current implementation* of LLVA-emu does; 2) the LLVA Linux kernel is compiled to a different ISA (namely, the LLVA virtual ISA) than the host operating system kernel.

2 Design & implementation

Our work on this project can be broken down into several tasks:

1. Porting Linux to LLVA
2. Compiling and linking the entire Linux kernel using the LLVM C front-end and compilation system
3. Implementing operating-system support "intrinsics" in LLVA-emu
4. Implementing virtual emulated devices in LLVA-emu

2.1 Porting Linux to LLVA

This task involved implementing all the architecture-specific hooks and header files which are common to all architectures. These are the files kept in the subdirectories `linux/arch/asm-llvm` and `linux/arch/llvm` of the Linux source tree. We accomplished this task by following two general strategies:

1. We removed architecture-specific and processor-specific inline assembly code from the kernel, and replaced it, only wherever it was necessary to do so, with LLVA intrinsic functions.
2. We replaced architecture-specific inline assembly code with C equivalents in some cases. We typically accomplished this by adapting portions of Linux's existing Intel x86, IBM S/390, and MIPS implementations of architecture-specific hooks.

It is important to understand that LLVA's intrinsic functions are external to the Linux kernel and are implemented by our emulator for the LLVA. Because we have a just-in-time compilation framework, and because these intrinsic functions are specially identified as such to the framework, they can be implemented using actual function calls, or expanded directly into sequences of instructions by the code generator.

2.2 Compiling Linux to LLVM bytecode

Initially, the majority of our work went into compiling the Linux kernel with the LLVM C front-end while simultaneously porting architecture-specific code in Linux to use LLVA intrinsic functions. One of the setbacks, but also benefits for our LLVM compiler infrastructure was the fact that Linux source code exercised our compiler in ways we hadn't yet tested, and so we discovered bugs in our C front-end and in our optimizers that prevented compilation. As a side benefit from this project, we now have a more robust compiler infrastructure and are now able to compile, link, run interprocedural optimizations, and generate code for the entire Linux kernel under the LLVM system, preserving full type and data-flow information in the final output.

Our implementation of LLVA-emu uses an execution engine based on the just-in-time compilation model in LLVM, using the X86 code generator for native execution. We used the standard LLVM just-in-time bytecode loader to load the kernel, which allows us to load from disk and generate code only for those portions of the kernel which are actually executed, and not before they are actually needed.

2.3 Intrinsic functions

As we stated above, our architecture's virtual instruction set is closely related to the LLVM compiler's intermediate representation. Although that representation is sufficient to represent arbitrary user-level code, it does not specify how system-level code can interface directly with the hardware.

Our solution to porting system code is to define a set of *intrinsic* functions which encapsulate the functionality that an OS kernel would expect from the architecture that it is running on, but expressed in such a way that makes them easy to implement on a variety of architectures. Put another way, we found that a clean way to add functionality to our LLVA definition was as follows: rather than adding new instructions to read or write architecture-specific state, we added intrinsic functions. These look like ordinary function calls in the LLVM bytecode, but their names contain a special "llvm." tag, and they can be intercepted by the runtime code generator and translated into real function calls, or sequences of arbitrary machine code, as necessary.

The intrinsic functions we added are of two sorts: a higher-level sort, which tries harder to abstract away details of the hardware which are not very interesting even to the operating system, such as the details of how low-level interrupt handlers are structured, and a lower-level sort, which preserves architecture-specific device I/O functionality, at the expense of abstraction.

Examples of the higher-level sort include:

1. `registerInterruptHandler`(*func*) - This intrinsic function changes the processor's current interrupt handling mechanism so that future interrupts will be handled by a call to *func*. *func* is passed an integer ID number which discriminates among the different possible types of interrupts that the hardware understands; this ID number is implementation-specific.

2. `timerInit`() - This intrinsic function enables the on-board timer chip and causes timer interrupts to be generated at the default frequency; they will be delivered to the currently-registered interrupt handler if one exists.

An example of the lower-level sort is `byteOutput`(*port*, *byte*). This intrinsic function causes *byte* to be output to I/O port number *port*. The result of doing so, and the assignment of port numbers to devices, are both implementation-specific. Input can be accomplished by a similar `byteInput`() intrinsic function.

One of the ways in which these intrinsic functions can be considered as multiple levels of hardware abstraction is as follows: if you assigned a port number to the timer chip, and defined how this timer chip should operate in terms of byte input and output operations, it would be possible to implement `timerInit`() in terms of `byteInput` and `byteOutput` instructions. Different kernels may find different levels of abstraction more appropriate. In Linux, for example, the higher-level intrinsics have been completely sufficient, but the lower-level ones may be more useful for back-porting device drivers.

Thus, an OS running on LLVA can be ported to another LLVA-compatible machine without having to change instruction sets. This also preserves the semantics of the functionality of reading and writing architected state: it is correct to conservatively assume that a call to an external function (such as one that is only available in the JIT compiler) to have arbitrary side-effects and hence not optimize code across the boundaries of that call, thus making it atomic.

It is important to point out that we have *not* attempted to build a system where Linux can be compiled once and run on any arbitrary machine. LLVA does not abstract away all possible combinations of device configurations that may be present in a machine. This is to say that if you compile Linux for an LLVA machine with a PCI bus, and you try to load it on a machine with a VME bus, the LLVA layer will not solve the problem of not having the right driver for you.

2.4 Virtual emulated devices

The virtual emulated hardware provided by LLVA-emu is relatively minimal but just complete enough to run Linux on. That is, it provides a CPU (with the LLVA instruction set), memory, a serial console, and a timer chip that generates periodic timer interrupts.

A virtual disk device is planned, but not yet implemented. It should be possible to boot all the way to user-level by using a RAM disk instead of a hard disk, so the virtual disk device is not crucial even for our long-term plans.

3 Evaluation & results

Our goal was to boot Linux to the point that it calculates the BogoMIPS value for the current system, which is a calibration of the delay loop and a rough estimate of the speed of the processor on which Linux is running.

We have achieved that goal. Here's an example session using LLVA-emu with Linux compiled to bytecode:

```
% llva-emu vmlinux
Linux version 2.4.22 (brukman@zion.cs.uiuc.edu) (gcc version 3.4-llvm
20030827 (experimental)) #46 Wed Dec 10 17:56:53 CST 2003
LLVA-emu command line:  ""
Kernel command line:
WARNING: trap_init() not yet implemented for llvm
Calibrating delay loop...  6173.49 BogoMIPS
```

4 Related Work

Virtual machines of different kinds have been widely used in many software systems, including operating systems (OS), language implementations, and OS and hardware emulators. These uses do not define a Virtual ISA at the hardware level, and therefore do not directly benefit processor design (though they may influence it).

We know of four previous examples of virtual instruction set computer (VISC) architectures: the IBM System/38 and AS/400 family [4], the DAISY project at IBM Research [7], Smith et al.'s proposal for Codesigned Virtual Machines in the Strata project [15], and Transmeta's Crusoe family of processors [10, 5]. All of these distinguish the virtual and physical ISAs as a fundamental processor design technique. To our knowledge, however, none except the IBM S/38 and AS/400 have designed a virtual instruction set for use in such architectures.

The IBM AS/400, building on early ideas in the S/38, defined a Machine Interface (MI) that was very high-level, abstract and hardware-independent (e.g., it had no registers or storage locations). It was the sole interface for all application software and for much of OS/400. Their MI was targeted at a particular operating system (the OS/400), it was designed to be implemented using complex operating system and database services and not just a translator, and was designed to best support a particular workload class, e.g., commercial database-driven workloads. It also had a far more complex instruction set than ours (or any CISC processors), including string manipulation operations, and "object" manipulation operations for 15 classes of objects (e.g., programs and files).

In contrast, our V-ISA is philosophically closer to modern processor instruction sets in being a minimal, orthogonal, load/store architecture; it is OS-independent and requires no software other than a translator; and it is designed to support modern static and dynamic optimization techniques for general-purpose software.

DAISY [7] developed a dynamic translation scheme for emulating multiple existing hardware instruction sets (PowerPC, Intel IA-32, and S/390) on a VLIW processor. They developed a novel translation scheme with global VLIW scheduling fast enough for online use, and hardware extensions to assist the translation. Their translator operated on a page granularity. Both the DAISY and Transmeta translators are stored entirely in ROM on-chip.

Transmeta's Crusoe uses a dynamic translation scheme to emulate Intel IA-32 instructions on a VLIW hardware processor [10]. The hardware includes important supporting mechanisms such as shadowed registers and a gated store buffer for speculation and rollback recovery on exceptions, and alias detection hardware in the load/store pipeline. Their translator, called Code Morphing Software (CMS), exploits these hardware mechanisms to reorder instructions aggressively in the presence of precise exceptions, memory dependences, and self-modifying code (as well as memory-mapped I/O) [5]. They use a trace-driven reoptimization scheme to optimize frequently executed dynamic sequences of code.

Smith et al. in the Strata project have recently but perhaps most clearly articulated the potential benefits of VISC processor designs, particularly the benefits of co-designing the translator and a hardware processor with an implementation-dependent ISA [15]. They describe a number of examples illustrating the flexibility hardware designers could derive from this strategy. They have also developed several hardware mechanisms that could be valuable for implementing such architectures, including relational profiling [8], a microarchitecture with a hierarchical register file for instruction-level distributed processing [9], and hardware support for working set analysis [6]. They do not propose a specific choice of V-ISA, but suggest that one choice would be to use Java VM as the V-ISA.

Previous authors have developed Typed Assembly Languages [13, 2] with goals that generally differ significantly from ours. Their goals are to enable compilation from strongly typed high-level languages to typed assembly language, enabling sound (type-preserving) program transformations, and to support program safety checking. Their type systems are higher-level than ours, because they attempt to propagate significant type information from source programs. In comparison, our V-ISA uses a much simpler, low-level type system aimed at capturing the common low-level representations and operations used to implement computations from high-level languages. It is also designed to to support arbitrary non-type-safe code efficiently, including operating system and kernel code.

Binary translation has been widely used to provide binary compatibility for legacy code. For example, the FX!32 tool uses a combination of online interpretation and offline profile-guided translation to execute Intel IA-32 code on Alpha processors [3]. Unlike such systems, a VISC architecture makes binary translation *an essential part of the design strategy*, using it for all codes, not just legacy codes.

4.1 Separation of Work

We split the coding of the LLVA emulator evenly amongst ourselves, and parallelized our work on finely-grained chunks that were independent of each other

– timing and console driver. We employed pair-programming and eXtreme Programming techniques on the initial part of getting the Linux kernel compiled under the LLVM C front-end, replacing X86/MIPS assembly sequences with C code and LLVM-specific intrinsic functions.

5 Conclusions

Our proof of concept shows that it is possible to run an operating system (at least to some extent) on a virtual architecture via dynamic run-time translation. This is due to our abstraction of architecture-specific services with our intrinsic functions.

This is a good starting point for our continued work for a complete definition and design of our virtual architecture and porting the entire operating system Linux (and possibly, others) to our virtual architecture.

Clearly, we were able to do this because of the fact that Linux has been ported to so many different platforms that the authors took the time to separate the platform-specific code into appropriate places in the source distribution to make it easier to port to new architectures, such as our LLVA.

5.1 Limitations and Future Work

We have not performed a *complete* port of Linux to LLVA. A lot of work remains to completely boot Linux to the point where it can run user-level applications. Many design issues loom ahead, such as abstraction of the memory management features of the processor, and porting additional device drivers to work with LLVA intrinsics. However, our initial experience suggests that the problems of porting drivers to the new interface will be relatively easily solved; the key is to define the right intrinsics to use, and the right layer of abstraction, for each new bit of functionality.

Once we have successfully abstracted the memory-management hardware and implemented a few additional device drivers, we should be able to boot Linux all the way to multi-user mode. This having been done, we will then attempt to "transplant" the LLVA execution environment onto bare hardware, replacing our virtual emulated devices with real devices implemented in hardware. The remaining bits of LLVA-emu in this situation would correspond to the just-in-time translator and the support for operating-system intrinsic functions. This is our long-term project.

References

[1] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. A Low-level Virtual Instruction Set Architecture. In *36th International Symposium on Microarchitecture (MICRO)*, pages 205–216, San Diego, CA, Dec 2003.

[2] J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound TAL for back-end optimization. In *PLDI*, San Diego, CA, Jun 2003.

[3] A. Chernoff, et al. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, 1998.

[4] B. E. Clark and M. J. Corrigan. Application system/400 performance characteristics. *IBM Systems Journal*, 28(3):407–423, 1989.

[5] J. C. Dehnert, et al. The Transmeta Code Morphing Software: Using speculation, recovery and adaptive retranslation to address real-life challenges. In *Proc. 1st IEEE/ACM Symp. Code Generation and Optimization*, San Francisco, CA, Mar 2003.

[6] A. S. Dhodapkar and J. E. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *ISCA*, Alaska, May 2002.

[7] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.

[8] T. H. Heil and J. E. Smith. Relational profiling: enabling thread-level parallelism in virtual machines. In *MICRO*, pages 281–290, Monterey, CA, Dec 2000.

[9] H.-S. Kim and J. E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *ISCA*, Alaska, May 2002.

[10] A. Klaiber. The Technology Behind Crusoe Processors, 2000.

[11] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[12] C. Lattner, M. Brukman, and B. Gaeke. Jello: a retargetable Just-In-Time compiler for LLVM bytecode, Dec 2002.

[13] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *TOPLAS*, 21(3):528–569, May 1999.

[14] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.

[15] J. E. Smith, T. Heil, S. Sastry, and T. Bezenek. Achieving high performance via co-designed virtual machines. In *International Workshop on Innovative Architecture (IWIA)*, 1999.