Software Tools for Low-Level Software and Operating Systems Classes

Maxwell Walter
Technical University of Denmark
Email: maxw@dtu.dk

Sven Karlsson
Technical University of Denmark
Email: svea@dtu.dk

Abstract—Low-level software is a particularly challenging topic to learn. In this paper, we present a framework and an application that makes the task of understanding system level software and hardware simpler. Traditional analysis tools, such as profilers and debuggers, are either too difficult to use in these situations, or are simply not available.

Hardware interactions, which are typical of low level software, make the challenge of understanding even more complicated. The tools we have built provide a view into the workings of such systems in both a simulation environment and on real hardware. The motivation for our framework, and the visualization tools that use it, is the teaching of an undergraduate level operating systems course. We discovered while teaching this course that many students struggled with the low-level environment and concepts of the course. We decided then to develop a set of applications and utilities to help students understand the inner workings of the systems they were dealing with.

I. INTRODUCTION

Due to the changing nature of the computing landscape, computer science curriculum has been changing in recent years as introductory classes have focused more on high level languages and technologies. Despite this shift in curriculum focus, there is still a need for courses that focus on low-level software and hardware/software interactions. Such software systems would include operating system kernels, hypervisors, bare-metal applications, and software support for hardware accelerators. This is because, recent trends in both hardware and software are making low-level software classes more important. These trends such as the Internet of Things, the proliferation of custom hardware accelerators, the rise of many-core systems, and FPGA fabric integrated onto processor dies, has caused a resurgence of interest in operating systems development and research. This, in turn, has rekindled a need for classes in operating system programming as well as low-level software design.

In teaching a class like the type described above, where students develop their own rudimentary operating system kernel through a progressing series of programming assignments, we witnessed the challenges the students were faced with. Students were unfamiliar with command line based tools, due to only having taken previous classes using IDEs. Students struggled with important C concepts such as memory allocation and pointers as previous classes were taught in higher level languages such as Java or Python. This resulted in students having great difficulty understanding the internal workings of the software as they became bogged down by more fundamental concerns.

In addition to the difficulty posed by the opaque nature of low-level software, the interaction of such systems with hardware also pose unique challenges. In introductory level classes such as this students are not yet familiar with even simple architectural concepts such as interrupts and registers, the distinction between physical and virtual memory, or even how a microprocessor boots and initializes itself and executes code. As a result, teaching students about operating systems necessitates teaching simple architectural concepts, as the two are very inter-related.

To help solve these problems, and provide students with a more advanced platform on which to learn, we have attempted to provide the students in our class with the more advanced tools that they are used to, such as IDEs and profilers. However, we have discovered that many tools require access to information that is difficult if not impossible to obtain from an arbitrary running kernel. The frameworks, such as ptrace, that provide this information do not exist, and would be too complicated to create, for our simple kernel. This prompted us to develop our own framework, and tools using it, to support students.

Even then, this only provides a view into the operation of the system software state. The most important aspect of low-level software is how it interacts with the underlying hardware architecture, which is fundamentally different from user-level software. In order for students to gain insight into what their software is doing they also need information on those what the hardware is doing. This includes things like page tables, interrupts, traps, and signals, and access to physical memory. Our framework must also include ways to access this information in order to obtain the information we need.

In this paper, we present two versions of our framework. The first version we developed for use in a virtual execution environment, attempting to maintain architecture and execution environment independence. We then modified the framework to run on actual hardware, which required surprisingly few changes.

Our software framework was originally developed for use in a virtual execution environment, as virtual execution provides complete access to the underlying system and hardware internals. The framework sits externally to our simple kernel, and provides access to the information we need in an easy
way. We make use of a technique called introspection, which allows us to read the state of the data and variables that we are interested in [1], [2]. We have also modified several execution environments in order to expose useful aspects of the underlying architecture. Additionally, we can make use of HDL simulation environments, such as GHDL, combined with virtualization to incorporate FPGA hardware effects into the simulation.

While our software framework was designed to run in simulation by leveraging virtual execution environments for its operation, similar approaches can also be used on actual hardware. Our second iteration of our framework is for inspecting real hardware state. The hardware platform that we target is microprocessors with embedded FPGA fabric, such as the Xilinx Zynq. The Zynq has proven to be a popular platform in education as it pairs a common ARM platform with reconfigurable hardware [3]. On hardware, we make use of JTAG hardware combined with a small FPGA logic core to provide a subset of the level of access and resolution we have in our virtualized system.

Our goal with our framework is to provide students with a way to view internal state of their system, both software and hardware, without having to add complex trace points or deal with initialization of hardware such as USART or ethernet PHYs. We also want to provide students with a way to examine the state of both hardware and software at the same time, in the same way.

II. MOTIVATION

Our work is due to a trend we noticed while teaching an undergraduate level operating systems class. The entry level classes, which we rely on to supply the background students need, had changed. They have been switching to high level languages which means concepts such as memory, pointers, and even header files have become unfamiliar to students. The entry level classes also use more advanced IDEs, such as Eclipse, meaning students are not familiar with command line operations, or possibly even what a compiler is.

Additionally, students have little to no knowledge of the underlying complexities of computer hardware. Operating systems require fairly intimate knowledge of the underlying architecture and so must be addressed the class. This, then, means the class has to cover a very large amount of diverse material.

As a result, we found that we need to either scale back the content of the class, or provide a more advanced environment to mitigate some of these issues. Scaling back the class content to some extent is required, as allowances must be made so that students can learn important C concepts such as memory allocation and pointers, and hardware concepts like interrupts and hardware memory management.

More difficult though is what to do about the lack of knowledge of the environment. Simply taking class time to explain what a shell is, how to navigate a directory, or how to use a command line debugger is not worthwhile as they are not contributing to the course learning objectives. We have instead decided to try and give students access to tools they are familiar with which will make working with the assignments easier.

We envision using and developing high level tools that students can use to gain an insight into how the software they are developing in the class works. Familiar tools like IDEs, profilers, and tracing software can help students gain valuable insight and intuitive understanding of how their software works. These tools will also, unlike their userspace based counterparts, provide insight into the underlying hardware by exposing interrupts, exceptions, and other architectural features.

A. Incorporating an IDE

The first series of assignment students are tasked with is to create a simple operating system. We provide basic boot code and students fill in the rest as the labs progress. This includes drawing text to the screen via hardware frame buffers, implementing a userspace and context switches, creating system calls, and finally creating multiple processes that interact via those system calls. We used the bochs emulator to run the software the students developed, which is convenient for this purpose as it has a built-in gdb server that can be used for debugging purposes.

Our first action was to provide students with access to an IDE, namely Eclipse, so that they have a familiar environment to work in. We created a project in Eclipse that was able to build the software and generate a bootable image, all from within the IDE. We also created a debug configuration within Eclipse that, when started, would launch the Bochs emulator with the bootable image that was built. Additionally, we created a gdb startup script that would connect to the gdb server in the running bochs instance. This allowed students to run and debug their operating system code, running in bochs, from within the Eclipse IDE.

This setup greatly enhanced the student’s ability to immediately participate in the software lab activities. They were able to work in a familiar environment, as the Eclipse platform was used in previous classes. They also did not have to learn the somewhat esoteric interface of a command line debugger.

We did have one problem with Eclipse, namely that it was not designed with such a use case in mind. This caused odd issues, such as the debugger stopping when source files were viewed outside of the debugging context. These issues were frustrating for students and demonstrated the need to tools specialized for this purpose.

Our Eclipse integration was also unable to provide us with extra insight into the hardware of the execution environment. We were forced by the setup to use a generic gdb target and the gdb server exposed by bochs. Neither of these tools are aware of any of the architectural features exposed by the virtual environment. As a result, if we wanted to add tracing of interrupts and traps, for instance, we could not do so using this method.
B. Extension to Other Tools

Due to our success with Eclipse, and after discovering its limitations, we decided to see what other tools we could incorporate into the class to make the laboratory exercises more understandable. Of particular interest were profiling tools such as valgrind or a tracing framework like DTrace or sysdig. These tools require a working userspace to function properly which meant we were not able to use them directly, however we took inspiration from them in order to create a framework from which we could create our own tools.

One of the most important aspects of the profiling and tracing tools is their ability to tap into certain parts of the system and extract information from them as the system is running. We sought to provide the same level of functionality with our framework, but without the need to have a working user space. Our first iteration of the framework was based on a virtual execution environment.

III. ANALYSIS FRAMEWORK

A pictorial representation of our virtual execution framework can be seen in figure 1, and it can be thought of in terms of three separate components. The first is a management and monitoring interface which provides a way for applications to interface with the execution environments. The second component is the execution environment itself that the software runs in. The final component is the system interface which provides the mechanism by which applications interact with the software that is running in the execution environment, and with the hardware that is being virtualized.

A. Execution Environment

The Execution Environment is the portion of the framework in which the software that is being examined runs. We have designed the framework to be largely agnostic to what sort of environment the software is actually running in. So in reality the execution environment is the interface to that environment, be it a virtualization platform, and emulator or real hardware. This allows us to use any of a number of different environments within our framework. Some environments we have investigated include virtualization platforms like QEMU, simulators such as Bochs and gem5, as well as actual hardware. This allows us to use different platforms depending on the needs of the class or assignment, while still providing instructors with a consistent interface on which to build their tools and assignments.

It is important that the framework abstract as much of the execution environment away as possible to provide a platform agnostic solution. There are many different types of assignments appropriate to different types of environments. Linux programming experiments can use QEMU, experiments on different architectures, such as our internally developed Tinuso, can use gem5, while hands-on assignments can be performed using real hardware. Flexibility is a useful asset when designing labs and courses.

B. Management Interface

The Management Interface provides an application with the ability to discover and control execution environments. Its purpose is to provide a consistent and convenient way to perform a number of necessary actions, including:

- **Create/Discover** new execution environments, which we will interchangeably call domains. The exact mechanism and capabilities of a domain depend on the capabilities of the execution environment in question. Virtualization platforms and simulators can be created with differing parameters such as memory size, CPU count and even different architectures. Real hardware is limited in its setup and can only be discovered, not created. Multiple domains can be created or opened at the same time, which allows for different types of system interactions, such as network communication or distributed filesystems.
- **Start, stop, and restart** the domains that have been created. These are the basic functions that are required to manage domains. These functions also allow for simulating things like random power loss or failure events, and provide a way to restart the system in the case of catastrophic failure, which can be quite common when initially developing kernel code. Providing an easy way for students to reset their activities can alleviate the stress of making mistakes.
- **Pause** the domain. The ability to pause the domain is critically important for analysis. Pausing gives applications the framework time to sample and inspect the system state in a more detailed manner. Without pausing, we may read an inconsistent internal state, as some part of it may have changed between when we started reading and when we finish. Pausing also gives an application the ability to modify the internal state of the kernel or software without having to worry about timing or synchronization issues.
C. System Interface

The System Interface is responsible for providing communication between applications utilizing the framework and the domains, in order to collect state information and report on system status. We categorize the communication with the running software into two broad types: implicit communication and explicit communication.

Implicit communication is where the application using the framework receives information from the domain about the state of the executing software without its knowledge or explicit action. One primary example of this is the ability of the software to read the contents of physical and virtual memory from any of the domains, through memory introspection. The software is not aware that this is occurring and is unable to perform any action in response. Other types of information that we are able to receive implicitly can include hardware events such as exceptions or traps, and operating system events such as system calls and user-space/kernel-space transitions. We can also extract information such as page tables and hardware performance counters.

Explicit communication is traditional program communication where the software intentionally sends a message to the external framework. This can be used by the software to indicate to the framework that a specific code path has been reached or a specific known event has occurred. It can also be used to transfer a large amount of state information to external applications. Data structures may be distributed across memory which could make reading them via memory introspection complicated, as multiple pointers must be tracked down and followed. Also, because it is initiated by the software running in the domain with its full knowledge, explicit communication can also be used without having to pause it, as the synchronization issues will be taken care of.

IV. Virtualization Implementation

Our framework is designed to be execution environment agnostic, however our first implementation was designed specifically with virtualization in mind. The interface we use to access the individual domains is not specific to any one virtual machine implementation. Currently we have implemented an interface for QEMU, which is a virtualization and emulation platform [4]. QEMU is able to simulate a number of architectures of interest including x86, AMD64, armhf, and AArch64, which makes it a convenient platform for education. It also supports multiple processing cores, and a number of projects are using QEMU for advanced system simulation that we could take advantage of [5], [6], [7].

QEMU already provides read access to physical and virtual memory, but we required more information than what QEMU provides to track system state. In addition to reading memory, we added the ability to modify large portions of memory at once, which gives us the ability to alter the system state. This also provides us with a crude way to send messages to the executing software. Additionally, we modified QEMU to enable it to send simple messages to the framework in response to various system events.

Our modified QEMU is able to send messages on hardware and software interrupts and upon execution of the syscall and sysret instructions. We also included the ability to send messages in response to the specific x86 assembly instruction, xchg, when both operands are the same 8-bit register. This is a technique used in simulators like bochs to trigger a simulator breakpoint [8], which we make use of to trigger framework events. These modifications were sufficient for our assignments, but we should be able to expose any architectural feature of the virtualization software.

While the framework was designed to be as generic as possible to different architectures, tying specific methods of communication to the specific assembly instructions restricts the implementation to the x86 architecture. The reason our virtualization implementation is based on the x86 architecture because our previous programming assignments were also based on x86. There is nothing in our implementation specific to x86, however. For example, we can also modify QEMU to send messages on ARM swi and svc system call instructions.

A. Framework Communication

Figure 2 shows the four methods of communication in our framework that can be used by the domains to provide information to the monitoring application.

Virtual Hardware Communication: Virtual hardware communication is an explicit communication method where the executing system communicates with the framework via hardware provided by the virtual execution environment. Virtual hardware peripherals such as network cards and serial ports can provide bidirectional communication in the standard way. In addition, many virtual environments provide hardware pass-through mechanisms, including USB and PCI, giving the virtual execution environment access to real host hardware.
Using virtual hardware teaches students how to interact with hardware peripherals. Dealing with special registers, interrupt driven communication, and memory mapped I/O help students to understand that they are working with real and complex hardware behind the scenes. This does add complexity to the initial software though, as peripheral initialization code will then be required.

**Memory Introspection:** One of the most important methods an application has to obtain information from the domain is memory introspection. Memory introspection is an implicit communication mechanism that allows the application using the framework to read from and, in some cases, write to any address in physical or virtual memory. This gives the application the ability to sample any data structure in the executing software. Consistency is a concern for reading large structures and avoiding torn writes, but this can be mitigated through multi-sampling for slow changing data structures, or pausing for quickly-changing ones.

**xchg Instruction:** In addition to the above communication methods, it is convenient to have an explicit way to send lightweight messages from the executing software to framework applications. One reason for this is to enable communication very early in the boot process, before hardware and memory have been properly initialized. We can also use these lightweight messages to send information, such as addresses or short status information, to framework applications for display or further analysis.

While the ability to send simple messages is quite useful, an even more useful aspect of the xchg instruction is that we are able to pause the virtual machine in response to it. Some kernel data changes very quickly, making them very difficult to sample. Also, stack data structures can be difficult to locate externally from debug symbols, and can disappear quickly as the stack unwinds. Pausing the virtual machine in response to a specific event gives the framework an opportunity to fully sample the data it needs without having to worry about it changing or becoming stale.

**Hardware Events:** There are many hardware events generated by low-level software that are of interest when trying to understand how systems operate. Events and interrupts, system calls, and user-space/kernel-space transitions are crucially important to understanding how a system works and being able to follow the flow of execution. Using the xchg instruction to send a message on each of these events would be possible, but would require a large number of alterations to performance critical code. Instead, we chose to modify QEMU to send events to the monitor on all hardware and software interrupts, exceptions, and system call routines. Using these messages requires no alterations to any system code, and can be used in cases where the software could be unreliable. For example, in the presence of a double fault, explicit communication from the software may not be trustable as it may no longer be executing properly.

As with the xchg instruction events, the virtual machine can be configured to pause on a particular type of hardware event. This is especially useful for hardware events because it provides applications using the framework time to capture the state of the system in the face of critical errors. Without pausing, the kernel may destroy its state, or the system may reset before enough information can be obtained for analysis. Critical errors are highly expected when developing kernel code and this is a scenario that should be considered common.

**V. APPLICATION TO HARDWARE**

Running in a virtualized environment has many advantages for educational purposes. Virtual machines can be created and distributed to students to ensure that every student has the same platform to work from. Virtual execution environments provide a view and a level of introspection that is impossible to obtain in any other way. Virtual environments also don’t require additional expensive hardware to make use of whereas real hardware requires additional input/output devices and potentially more esoteric devices like programmers and hardware debuggers.

Despite these advantages, running on real hardware is critically important for learning low-level software and operating systems. Virtual environments, regardless of how accurate they are, are still just a software implementation and can differ in important ways. Virtual environments are often more forgiving, and the initialization of the various hardware components will differ, sometimes significantly. For these reasons we have taken the ideas from our software framework, originally developed with virtual environments in mind, and applied to real hardware.

The platform we have chosen for our hardware implementation is the Xilinx Zynq, which combines an ARM Cortex-A9 with a Xilinx FPGA, providing both software and hardware programmability. The FPGA is attached to the memory bus of the ARM processor and hardware developed in the FPGA show up in the processor’s memory space. The Zynq is ideal as an educational platform because it gives students a familiar base on which to start. The ARM Cortex-A9 is capable of running Linux and is supported by the well known Debian distribution. Additionally, having an FPGA integrated into the SoC provides students with the opportunity to learn how to develop hardware, and how to interact with it from a software perspective.

The ideas from our software framework proved to be relatively simple to apply to actual hardware. Following is a description of how each of the components of the software framework corresponds to our hardware oriented framework.

**A. Execution Environment**

The execution environment is simply the hardware itself. We are primarily interested in the Linux operating system, which is well supported on ARM and by Xilinx on the Zynq. For classroom use we would provide the students with the initial FPGA image, the bootloader image as well as the Linux Kernel supplied by Xilinx.

One modification we have make to the hardware platform to support our framework is the addition of an interrupt block in the FPGA. The FPGA has a number of interrupt signals...
that go from the programmable logic to the processor, and we make use of these to provide a stable timer interrupt to the processor. This provides the framework with a way to sample the state of the operating system at regular, known, intervals.

The timer block in the FPGA simply provides a stable interrupt to the processor. In order to take advantage of that, and to use it to provide information to the framework, we have developed a very simple Linux kernel module. The module simply registers a handler for the interrupt, which then provides the framework with a known location for a breakpoint.

B. Management Interface

The JTAG implementation in the Zynq ARM is supported by the OpenOCD Project, an open source JTAG compatible application and library. OpenOCD provides our framework with a low-level and powerful interface to the hardware platform that we can use to implement our various interfaces. Being open source also means that the software is freely available to students who can both use and learn from it.

To work on real hardware, we need a management interface that can support, at a minimum, pausing the execution environment in order to extract information. This capability is provided by the built-in JTAG support of both the ARM processor and the FPGA. Using JTAG, we are able to pause, reset, start and stop the execution environment in a way that is similar in functionality to the virtualized environments.

We are limited in our ability to create domains. Unlike the virtual environment where we can create an arbitrary number of domains with arbitrary configurations, with real hardware we are limited to what is connected via JTAG. This leads to the framework being able to only enumerate existing real hardware domains, not create new ones.

C. System Interface

Running on actual hardware is quite limiting compared to running in a virtualized environment when considering what information is available. While memory introspection is available via the JTAG interface, the other methods of communication are not in the same way. We are not able to send events on hardware interrupts, system calls, or the xchg instructions, which were very useful techniques for tracing software execution in the virtualized environment.

We can, however, mitigate these problems to a limited extent. Instead of receiving an event on every system call, for example, we can set a breakpoint on the system calls that we are interested in using OpenOCD. Then we are able to extract information from the specific system call or calls that we are interested in. The same method can be used for hardware events and xchg instruction messages by placing breakpoints in locations of interest.

It is worth noting, however, that due to the inconvenience of identifying all locations of interest and setting breakpoints at them, real hardware is not the best solution for exploratory investigation. In cases where the goal is basic system understanding it is best to run in a virtual environment where more information is available. Real hardware is best for cases where hardware interactions are of interest or there is a limited scope of investigation. Additionally, real hardware is obviously more accurate in its execution, and makes a good verification for the development work performed in a virtual system.

D. Limitations

Targeting actual hardware allows us to run real applications on real hardware, however this does have several draw-backs when compared to running in a virtualized environment. The primary limitation is the decreased visibility we have into the system that we are monitoring. In a virtualized environment we have access to nearly every aspect of the system as we have complete access to the execution environment itself. We can, and did in the case of QEMU, modify the execution environment to provide us with information from nearly any point in the system. For real hardware we are limited to what is exposed by the physical hardware itself, in this case over JTAG.

Over JTAG we have the ability to start and stop the processor, read and write to arbitrary locations in physical and virtual memory, and set breakpoints. Additional features we found useful in the virtualized environment, such as receiving messages on hardware events or system calls must be build on top of these. Other features, such as sending messages in response to certain instructions are simply not feasible. As a result, in exchange for the realism of running on real hardware, we trade off the ability to obtain a more complete and nuanced view of the system.

Another significant limitation is the complexity of dealing with real hardware. Whereas the virtual environment can be run, entirely self contained, under any common operating system, hardware is more limited. First, the hardware itself is required, the price of which can be a burden for classes. Hardware JTAG support is required as is OpenOCD, which can be non-trivial to set up on non-Linux operating systems. This can be problematic as the goal is to help students learn, not to place more barriers in their way. As a result, much more support for the teachers and assistants is typically required.

One final limitation we encountered is that of speed. Reading and writing to memory from the virtualized environment is nearly imperceptibly fast while we found that reading a single value from our hardware platform took nearly 500ms in may cases. This is likely a result of a number of things particular to our hardware platform. The JTAG interface on the board that we used, the Zybo, is not a direct JTAG connection. The JTAG interface is first converted to a serial interface which is then connected through USB-to-serial converted that is connected to the host. This daisy chained interface added latency. OpenOCD then communicates via the serial interface and opens up a socket connection that applications can make use of. A dedicated, hardware, JTAG connection would likely alleviate many of the speed issues we encountered and will be the subject of future exploration, though Andrus et.al. found similar speed limitations due to hardware debugging when teaching an operating systems class using Android [9].
for clocking data out to the LED panel. This would help
effects like gamma correction, to the right. We plan to extend
should look like based solely on color values, minus advanced
FPGA blockram. We also show a rendering of what result
the contents of the memory in the FPGA, extracted from the
calls, extracted from the running kernel, are shown alongside
example can be seen in figure 4. The values from the kernel
is happening with the system in real-time by displaying the
values from both the kernel call as well as the resulting
memory values. To present this information, we start by taking
inspiration from FPGA simulation tools, such as Vivado and
ModelSim. These tools display the simulation results as signal
values over time and also provide a convenient mechanism to
navigate the results by zooming and panning through time.

While the visualization provided by the simulation tools is
useful for FPGA simulation results, it is not entirely sufficient
for the complex data structures typically found in operating
systems and embedded applications. Many structures cannot
be properly represented just by their values; they simply
contain too much information to display in an understandable
way.

Instead, we provide a simplified view in the signal display
and a graphical representation in a separate portion of the
display. When the user clicks on a signal at a given mo-
ment in time, the graphical representation will be updated
to reflect the item that was clicked on. This does have the
disadvantage that any new data structure would need a new
graphical representation, which may be beyond the capabilities
of students. However, when incorporated into new software
labs and exercises this is not a serious issue and can be
prepared for ahead of time.

A simple GUI demonstrating this idea for the LED grid
element can be seen in figure 3. In this assignment, students
are asked to implement a driver interface to an LED panel that
is mapped into the processor memory. The kernel interface is
a simple \( x, y \), limited to a 5x5 grid, coordinate pair and a
\( (r, g, b) \) color triplet.

We wanted to provide students with a way to see what
is happening with the system in real-time by displaying the
values from both the kernel call as well as the resulting
memory values. To present this information, we start by taking
inspiration from FPGA simulation tools, such as Vivado and
ModelSim. These tools display the simulation results as signal
values over time and also provide a convenient mechanism to
navigate the results by zooming and panning through time.

While the visualization provided by the simulation tools is
useful for FPGA simulation results, it is not entirely sufficient
for the complex data structures typically found in operating
systems and embedded applications. Many structures cannot
be properly represented just by their values; they simply
contain too much information to display in an understandable
way.

Instead, we provide a simplified view in the signal display
and a graphical representation in a separate portion of the
display. When the user clicks on a signal at a given mo-
ment in time, the graphical representation will be updated
to reflect the item that was clicked on. This does have the
disadvantage that any new data structure would need a new
graphical representation, which may be beyond the capabilities
of students. However, when incorporated into new software
labs and exercises this is not a serious issue and can be
prepared for ahead of time.

A simple GUI demonstrating this idea for the LED grid
element can be seen in figure 4. The values from the kernel
calls, extracted from the running kernel, are shown alongside
the contents of the memory in the FPGA, extracted from the
FPGA blockram. We also show a rendering of what result
should look like based solely on color values, minus advanced
effects like gamma correction, to the right. We plan to extend
this view to support more information, such as the signals
for clocking data out to the LED panel. This would help

trouble shoot problems, like the bottom-most values not being
displayed in figure 3.

VII. Related Work

Virtualization is a powerful tool and has been applied many
times and in many ways to computer science education. Kl-
emente [10] describes various ways in which virtualization has
been applied to classroom use. He considers three basic dimen-
sions of virtualization in education. The first is infrastructure
virtualization to consolidate and simplify the administration
of educational infrastructure. The second is teaching about
the operation and administration of virtualization technologies.
The third dimension is the use of virtualization in educational
activities themselves.

Along this third dimension, virtualization has been applied
many times, but typically as a platform to work in, rather than
as a technology to build on. Stewart et.al. discuss developing
a virtualization platform to create a network environment on
modest hardware, with the goal being the enabling of more
advanced networking and administration studied [11]. Nieh
and Vaill used virtualization as a way to provide all students,
even remote ones, with the same platform on which to com-
plete assignments using the Linux kernel [12]. Others have
focused on teaching operating systems with specific hardware
configurations by running them in a virtualized environment
and measuring things like memory performance, cpu load, and
pagefile usage [13], [14].

All of these works involve using virtualization as a way
to run operating systems, either for multiple different hard-
ware configurations or to provide isolation for exploration. In
contrast, we are examining using virtualization as a way to
enhance tools we provide to students, providing them with
better insight into the assignments.

While it does not seem common in operating systems educa-
tion, virtualization, and specifically virtual machine introspec-
tion, has been applied to other areas. Virtualization has been
used extensively in the field of intrusion detection to analyze
the internal state of an operating system without disturbing
it [15], [16]. Johnson et.al. use virtual machine introspection
to allow debugging of the entire system stack, from the
kernel through user-space servers to high-level web services,
simultaneously [17]. We use similar techniques, but with the
goal of obtaining simplifying the information provided and
presenting it in an accessible way.

Finally, we draw a great deal of inspiration from a number of
analysis tools such as DTrace, systrace and valgrind [18], [19],
[20]. These tools require a complete and working userspace
and, in the case of tracing tools, a fairly complex kernel
implementation to support tracepoint injection. Our goal is to
create versions of these tools that can be used external to the
running system so that an existing user-space is not required.
We also seek to remove some of the kernel requirements, such
as code injection points, to alleviate some of the burden of
initial development.

Fig. 3. Experimental hardware setup.
VIII. CONCLUSIONS

In this paper, we have presented a framework we have developed that is intended to make introductory classes in operating systems easier and more understandable for students. Our motivation for creating this framework was our desire to provide students with more advanced tools, such as profilers and tracing frameworks. These tools allow the students to look into the software and hardware of the systems they are developing to increase their intuition as to how the software and hardware work. These tools are typically unavailable for operating system kernel development as the features they require, such as trace point injection locations or working userspace and POSIX system calls, are not available.

We have described our framework, the actions, and types of communication that it supports. We have also described our initial implementation of our framework in the context of a virtual execution environment. This is because we find virtual execution is ideal for software laboratory settings due to the level of access virtualization provides to the software executing within it, and the hardware that the software is running on. Despite this, virtual execution is not able to fully reproduce the experience of running on real hardware, due to its nature as a software simulation.

To handle hardware aspects, and to provide students with experience on real hardware, we have created a version of our framework targeting the Xilinx Zynq. Additionally, we describe a sample assignment that makes use of it to display information from both hardware and software in nearly real-time in the same display. Assisting students by providing them with more advanced analysis and development tools makes the process of learning complex topics easier, and low-level software is challenging in that respect. Our framework, developed specifically with our assignments in mind, but applicable across a wide range of problems, helps to address these shortcomings.

REFERENCES


