Bachelorarbeit

Design & Development of a System-Level Debugging Tool

Merten Sach
319318

March 22, 2013

Technische Universität Berlin
Fakultät IV
Institut für Softwaretechnik und Theoretische Informatik
Professur Security in Telecommunications

Betreuender Hochschullehrer: Prof. Dr. Jean-Pierre Seifert
Betreuender Mitarbeiter: Dipl.-Inf. Michael Peter

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den March 22, 2013

Merten Christopher Sach

Zusammenfassung

Software bestimmt viele Aspekte unseres Lebens. Nicht nur unsere Kommunikation oder Infrastruktur hängt mittlerweile von fehlerfrei funktionierender Software ab, auch unsere Gesundheit wird immer häufiger der korrekten Funktion medizinischer Geräte unterstellt. Aus diesem Grund verbringen Entwickler einen großen Teil ihrer Arbeitszeit mit Testen und Fehlersuche. Allerdings zeigt die Erfahrung, dass Software Fehler enthält.

Häufig werden die Anwendungen auf Betriebssystemen ausgeführt. Damit ist die Fehlerfreiheit der Betriebssysteme wichtig für die Fehlerfreiheit der Anwendungen. Jedoch erschweren die Eigenschaften hardwarenaher Software die Fehlersuche. Die Ausführungsumgebung dieser Software ist eingeschränkt und instabil. Durch nichtdeterministisches Verhalten wird die Reproduktion von Fehlern aufwändig.

Debugger in virtuellen Maschinen bieten die Chance, die Fehlersuche in Betriebssystemen weniger arbeitaufwändig und fehleranfällig zu machen. In dieser Bachelorarbeit wird ein Debugger für die Systemebene entworfen und entwickelt. Der Debugger, genannt *deterministic QEMU*, kann SMP Betriebssystemkerne wie Linux und Fiasco ausführen.

Deterministic QEMU bietet nützliche Eigenschaften für Systementwickler in einem *iterativen Debugprozess*. Dies sind Methoden um Information aus dem Raum aller Programmzustände zu extrahieren. Dies sind Speicherhaltepunkte, die Verfolgung von Prozessen und Markierungen im Gastcode.

Um die Reproduktion von berüchtigten Heisenbugs zu ermöglichen, wird das Gastbetriebssystem deterministisch ausgeführt. Damit wird die Ordnung von Prozessen und Speicherzugriffen von mehreren Prozessoren konstant über mehrere Iterationen. Die von *deterministic QEMU* simulierten Uhren sind instruktionsgenau, da sie unabhängig vom Hostrechner und and ausgeführte Gastinstruktionen gebunden sind.

Zusätzslich bietet die in *deterministic QEMU* integrierte Skriptsprache *Lua* ein Interface für den Programmierer.

Die bereitgestellte API ermöglicht Zugriff auf alle Debugfunktionen, den Status aller CPUs sowie die Möglichkeit die Ausführung der virtuellen CPUs durch *deterministic QEMU* zu steuern.

Die Auswertung zeigt, dass der implementierte Debugger ein hilfreiches Werkzeug ist, um Fehler in Betriebssystemen ausfindig zu machen.

Abstract

Software controls many aspects of modern life. Our communication, infrastructure and, when it comes to medical equipment, even our lives depend on its correctness. Hence, developers spend a significant amount of their time on testing and debugging. However, experience shows programs are likely to contain errors.

These applications often run on top of an operating system. This makes the correctness of the operating system crucial for their correctness. Yet properties of all low level software make debugging of operating systems hard. Their execution environment is restricted and fragile. Non-deterministic behavior of the hardware platform makes the reproduction of bugs tedious.

Debuggers integrated in virtual machines provide the opportunity to make OS debugging less labor- and error-intensive. In this thesis a system-level debugging tool using the virtual machine QEMU is designed and developed. The tool called *deterministic QEMU* executes *x86* SMP production-grade operating system kernels such as Linux and Fiasco.

Deterministic QEMU implements several features that aid system programmers in an *iterative debugging* process. First these are basic information gathering facilities to dissect the huge state space. Memory tracking, process tracking, and guest annotation provide tools to extract relevant information.

To allow the reproduction of infamous interleaving-dependent *Heisenbugs* the guest OS is executed deterministically. Deterministic execution fixes the interleaving of processes and memory accesses from multiple cores. *Deterministic QEMU's* simulated clocks are instruction accurate because they are independent of the host system and based on executed guest instructions.

At last a *Lua* script engine is embedded into *Deterministic QEMU* to provide a scriptable interface to the debugger. The provided API contains access to all debug functions, state of all virtual CPUs, and facilities to control the execution of virtual CPUs by *Deterministic QEMU*.

The evaluation shows that the implemented debugger provides a valuable tool for OS debugging.

Contents

List of Figures

1 Introduction

Software is found in many areas of our lives today. It controls cellphones, cars, and also medical equipment. Even applications that were formerly implemented as applications specific analog and digital circuits are moved to software. This is made possible by cheap processors that provide performance comparable to older desktop computers. More performance allows the engineers to deploy full size operating systems instead of writing single purpose applications. It also enables to move more tasks to the software components.

A tremendous amount of these cheap processors are deployed in embedded systems, much more than desktop computers and servers. These devices in particular have high requirements on reliability. Programmers spend numerous man hours on testing and debugging besides other steps like specification, design, and programming. But still experience shows that errors exist in almost every software.

In some cases these errors can even be life-threatening. Every student learns about bad practice examples like the Therac-25[LT93] accidents. An electron accelerator that exposed a number of patients to a radiation overdose due to software errors in the 1980s. It took multiple investigations until the bugs were resolved.

Bugs can hide in the complex interaction between the operating system and the software components. The NASA Mars Pathfinder Team observed spurious system resets[1] after the space craft landed on Mars. Through extensive debugging on earth the team discovered a priority inversion. A priority inversion causes a low priority task to block a higher priority task which then misses its deadline.

These two examples are rather old but still relevant. Since then even more task have been consolidated on fewer devices and new functions have been added. This causes more complexity due to more and sometimes unforeseeable interactions between the components.

This shows that debug tools are more important than ever. Unlike discrete circuits software replaces, software is hard to analyse. To verify circuits a designer can attach a multimeter to interesting points and measure if the voltage matches the specification. For software it is not as easy because one cannot simply "probe" the software.

1.1 Iterative Debugging

Debug processes are ofter similar to each other. The programmer goes through a number of steps to explain the unwanted behavior. The following steps are inspired by the book "Why Programs Fail"[Zel09].

[1] http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_-Account.html

First he or she has to reproduce the problem. Reproducing the problem means recreating the environment at the time of the crash. This includes the same input values, the same system architecture, and software version. It helps to reduce the crashing environment to the smallest possible because this gives the programmer already a hint about likely and unlikely causes since it excludes environmental parameters that are not part of the problem.

In this prepared debugging environment the programmer is now ready to investigate the source of the crash. Starting from the program state at the time of crash the programmer works his way back to the original error. This will result in a chain of program states that link the source of the bug to its manifestation. To retrieve a complete chain of causes one has to find all invalid states going back from the crash.

In contrast to this search backwards in time, program time always travels forward. Therefore, the method of choice for most programmers is an *iterative debugging* process. When the programmer wants to look at a previous states he restarts the execution of the program from the start until it reaches the past state. This *iterative debugging* requires deterministic execution, otherwise the program can follow a different path in each iteration and miss the searched state.

The programmer, at this point, is faced with two problems. First the amount of state information is huge. Modern processors can execute billions of instructions per second. Many of these are not relevant for the bug under investigation. So the programmer has to identify the relevant instructions within a tremendous amount of uninteresting instructions. Second the chain of causes potentially contains many intermediate states. Therefore, precise bookkeeping is necessary to retrieve a complete picture of the behavior.

Fortunately there is a big number of tools that allow to "probe" user space applications on computer systems. Debuggers like *gdb* offer a great amount of additional information that makes it easy to follow the program flow and understand the behavior.

Eventually with the help of filtering the determined programmer will be able to extract the chain of causes. The successful search for the bug leaves the programmer with the task to correct the bug.

1.2 Operating System Debugging

As mentioned the bug has to emerge deterministically to use *iterative debugging*. A bug is deterministic if it does not depend on non-deterministic events and manifests itself identically in multiple program executions. Many single-threaded user space applications are inherently deterministic. However the situation is different for multi-threaded programs and operating systems.

In contrast to single-threaded user space programs the execution of an OS depends on non-deterministic events. For instance many systems rely on hardware timer interrupts to provide pre-emption opportunities for a scheduler. These hardware events are not implemented deterministically. The counter readings and signal timings are dependent on many unknown effects like power stabilization at startup and energy saving modes.

In addition operating systems control hardware devices. Data received from these devices can originate from external sources such as network packets received by a network interface. Depending on the content of these packets the program flow will change. For instance a new HTTP request will spawn a new web server thread. Depending on the content of the request a different file is accessed. Since the payload of the received package is random the program flow is non-deterministic. Unpredictable behavior introduced through interrupts and input data is called *input non-determinism*.

In multi-processor environments the execution becomes even more fragile. When data is shared between cores, by message passing or shared memory, it can change the behavior completely. A common case are locks that synchronize the execution between processors. Depending on the order of writes to the lock one or the other processor retrieves the lock. Since the program path depends on the interleaving of data transfers it is called *memory non-determinism*.

OSs provide the environment and hardware abstraction for all applications on a system. In addition it opens a safety net for crashes within the user space. If a program violates its permissions (e.g. forbidden access to memory mapped devices or jump to non-executable areas) the operating system takes over and terminates the process. For the OS kernel itself there is only a limited set of these safeguards. This property and the non-determinisms make debugging and testing of operating systems time consuming and arduous.

Some kernels offer limited integrated debugging capabilities. However, in many cases there is no possibility for the kernel to recover. If a kernel process corrupts kernel memory it will panic and dump state information or "just fail".

Virtual machines (VM) give opportunity to do system-level debugging similar to user space applications. Even when the operating system crashes without any state information the VM can still extract information. A debugger integrated into the virtual machine can inspect the machine's memory and access formerly inaccessible data, even from peripheral devices. Additionally VMs give the possibility for fine grained control that can be used to avoid or steer non-deterministic effects.

However most available VMs are optimized for speed and non-deterministic. Also the hardware models contain simplifications that allow faster implementation but only approximate the behavior compared to the real device. Still recent publications[MQB+08, KDC05, DLFC08, MCE+02] showed that debugging and logging in virtual machines is feasible.

1.3 Outline

This thesis focuses on the implementation of a system-level debugger based on the open source virtual machine QEMU (Quick EMUlator). The enhanced QEMU allows to debug common operating systems like Linux and Fiaso. Special care was put on the ability to perform iterative debugging of multi-core systems. This is supported by a deterministic debugging mode. A number of features like memory monitoring and execution control can be steered by an integrated Lua interpreter.

First Section 2 will give an overview over the challenges and possible approaches to implement a system-level debugger. Thereafter, I will describe the design and implementation of a system-level debugger based on QEMU. Section 4 evaluates the feasibility using a synthetic example. After the evaluation the implementation will be compared to existing projects and solutions before I go into limitations of the model. To sum up I will discuss possible future extension or improvements to the implementation.

2 Analysis

As described in the Introduction debugging, in particular of OS kernels, is challenging due to a number of reasons. This section will go through them and point to solutions. Some of which will be used in the scriptable and deterministic QEMU implementation presented in Section 3

2.1 Huge State Space

The most simple programs are user-space programs. They follow a sequential instruction stream and do not use parallel execution. The execution of the program is only transparently pre-empted by an operating system. Interrupts, I/O, or other constructs that lead to concurrency are hidden by the OS. Instead the programs use operating system facilities to interface with the outside world.

Even these simple application execute billions of instruction per second. Multiple nested function calls generate a lot of information on the stack. This and other events combined comprise the state space of the program. One can see that this state space grows very fast.

The debugging programmer is only interested in a subset of the information. In fact only states that are related to the chain of causes. But the complete state space contains a great amount of unrelated information. The programmer, therefore, wants to define a subset of information to be extracted from the execution. At best he can use the high level context from the source code like variable names to define this subset.

This information is not available in a binary file. The compiler hides this information in the compilation process. In binary code, local variables are only offsets to a frame pointer and often only shortly valid. However, the compiler can generate a supplementary file that makes this information available to the debugger. This information increases the executable file but does not include additional code. A common format is DWARF[1] that is understood by debuggers like *gdb*.

2.1.1 Source Level Debugging

The most convenient analysis technique for the programmer is done on source code level. All debug code is directly integrated into the program. This insertion into the program can either be done by hand or by the compiler. Of course this requires the source code to be available to the programmer. All of these methods modify the executed binary. This is not feasible in all situation as this section will discuss.

[1] http://dwarfstd.org/

Manual Instrumentation

In the most naive approach source level debugging requires no language support. Instructions that output information about the program state are added to the source code. These include *printf* calls in C or instructions that write to a log file. Because of the use of *printf* calls this is often called *printf debugging*. It is frequently used because it is convenient and requires no additional knowledge about debugging methods. However, it has some major disadvantages.

A big disadvantage is that it potentially overwrites important program information. This is the case if the program is in an invalid state when debug information is output. textitprintf uses the stack, therefore, a call to *printf* can overwrite information on an already corrupted stack.

Language-Supported Analysis

Analysing tools that are integrated with the language can offer more support to the programmer. The language runtime has full context information and can find and interpret information like global variables, called functions, and exceptions.

Eiffel[2] in particular is worth mentioning. During the development programmers think in post- and pre-conditions which are also written down in software specification and used in a design-by-contract workflow. The Eiffel programming language allows to specify post- and pre-conditions for functions and classes. These, together with variants and invariants for loops, are checked at runtime. Other languages offer similar capabilities in the form of assertions.

Other languages offer more simple predicate checks. A `assert` is the most common and present in many languages. The `assert` statement carries a predicate that is evaluated at run-time. The program is aborted if the predicate is evaluated to a false value.

As with *printf* these checks add code to the program. However, it gives a good way to find bugs before they trigger a crash.

Compiler-Supported Analysis

A compiler that translates the source code to machine code can also add analysis methods to the program.

The compiler defines conventions for the stack layout and function calling. The stack in particular was the cause for many known bugs in the past. Buffer overflows on the stack can overwrite return addresses. This can be used by malicious attacker to disturb services or execute code.

To protect from these kind of situations the compiler can add methods that check the integrity of the stack. For instance stack canaries can be used. These are known values that are pushed on the stack and checked at runtime for tampering.

All source level methods have disadvantages in common. First they add additional code to the program. This introduces run-time overhead and can change the behavior of

[2] http://docs.eiffel.com/book

the program. Second it requires access to the program source code. At last all methods generate extra instructions, that execute checks or output information. These actions are tied to these points, monitoring of memory locations is not possible

2.1.2 OS Support

Since the debug code is not included in the debugged binary an external debug program is required. This program is called the *debugger*. The OS has to give an external debugger an interface to control the debugged program. Also it has to offer methods to gather information at run-time.

Information Gathering

A specific state in the programs state space is often identified by the memory access or a specific executed instruction. For example the programmer wants to inspect the instruction at which a memory location is written. A low latency method is required to check for this specific memory access. Otherwise the program would have to be stopped and inspected after each instruction that accesses memory was executed. The processor would spend more time in the debugger than in the program doing actual work.

Common debuggers know two types of these conditions. *Breakpoints* and *watchpoints*. Breakpoints select a point in the code section of the program. The execution is interrupted when this instruction is executed. Watchpoint on the other hand monitor locations in the memory for accesses. Again, when this location is accessed the execution is interrupted and the control is handed to the debugger.

Hardware Breakpoints Break- and watchpoints implemented in hardware provide a zero latency solution. If implemented efficiently the condition checks add no extra delay to the processors critical path. Hence it introduces no delay if the condition is not fulfilled. Because the condition check is implemented in hardware, and longer checks mean longer critical paths, only basic checks are implemented. Examples are read/write access to memory locations, the execution of an instruction at an address, and accesses to I/O addresses.

The *x86* architecture[Int13, Chapter 17] implements this hardware support. Four debug registers are provided to define breakpoint addresses. A debug control register allows to further specify the condition such as task local or global breakpoints. When a breakpoint is hit a *debug exception* is generated. A debug status register gives additional information about the condition causing the debug exception. This gives the handler the opportunity to retrieve more information.

Software Breakpoints The above example shows that the capabilities of hardware condition checking are limited. The *x86* architecture only allows four concurrent breakpoints. Special instructions allow more breakpoints. These instructions are *trapping instructions*. If they are executed within a program the processor executes an exception handler. But every trapping instruction can be used for debugging.

These instructions can be inserted at run-time. An attached debugger can overwrite instructions before execution or at runtime. So if the programmer wants to stop the program at a specific instruction the tool overwrites this instruction. However, this puts an additional requirement on the trapping instruction. The trapping instruction must not be longer than the shortest instruction of the instruction set. If the trapping instruction is longer than the replaced instruction the subsequent instruction is overwritten. A jump to this instruction will jump into the second part of the trap instruction. When the processor executes the inserted instruction an exception is generated and the control is handed over to the debugging tool.

The x86 architecture implements a special trapping software interrupt `INT3` instruction. This instruction is only one byte long while all other `INT` instructions are longer.

The architecture offers also a technique that allow single stepping. If the `TF` bit in the `FLAG` register is set a trap is generated after the execution of each instruction. Single stepping allows the programmer to look at the effect of each single instruction. Therefore, reducing the execution speed to a level that enables the human to follow the execution.

Monitoring Pages

As discussed the x86 debug registers can only monitor four virtual addresses at once. Software breakpoints are limited to instruction breakpoints and do not allow to monitor memory locations. This is fairly limited for an investigation.

An operating system can offer monitoring of more memory addresses through the paging mechanism. The *paging mechanism*, that will be discussed in detail later in this section, offers a facility to set permissions on a per-page level. The size of a usual page is $4KiB$. If a page is accessed by the process without sufficient privileges an exception is raised that is caught by the kernel.

To set a watchpoint the kernel reduces the permission of the page that contains the monitored address. Every access to this location will raise an exception and hand the control to the operating system. This exception enables the kernel to forward these events to an attached debugger.

2.1.3 Separated Debugger

The low latency information gathering techniques discussed before require privileged access to the process. A debugger that wants to rewrite the code of a debugged program has to have access to its address space. However, the operating system separates the address spaces of user spaces processes to prevent manipulation.

The operating system has to offer an interface for a debugger to debug programs without integrated debug code. In Unix kernels this is done via the `ptrace` system call that gives the debugger access to the programs address space. In addition it allows various manipulations like changing file descriptors and signal handlers.

If the debugger has access to the process data structures used by the context switch it can also inspect and manipulate the registers.

2.1.4 Scriptable Debugger

Analysis of binary targets will benefit from some form of automation.

When looking for specific write accesses some programs can generate a lot of events. If the memory value is written in a loop every iteration will trigger the watchpoint. In most investigations only a subset of these accesses are interesting, either accesses at a specific point in time or from a particular location. This requires a mechanism to activate and deactivate the monitoring.

Invariants as they are defined by Eiffel language are a valuable tool. When these are not integrated in the program the attached debugger has to offer an interface to define complex invariants.

These challenges require some interface for the programmer to define automated actions. One can either include this code in the debugger which then needs to be recompiled for each change or a script language is included. An integrated script language offers a fast interface to the programmer to define complex behavior

2.2 Complex Model

On system level there is no operating system underneath the program so the debugger cannot use OS supported debugging. Either the program itself is an operating system or it runs without an operating system. This renders the execution environment much more complex compared to simple user space applications. Architectural properties of the hardware are exposed to the program. In addition the environment is restricted and fragile.

2.2.1 Restricted Environment

At this point the programmer has to think about the environment in which the debugged code is executed in.

Imagine the programmer wants to add code to an operating system to output information about the system state. In a user space program this is everywhere possible. However in the OS there are some limitations.

For instance at boot time the operating system has to initialize the UART device before it can utilize the driver to output information. This also applies to devices like hard drives. Therefore for a small time frame the software has no output channel.

Within interrupt handlers timing is often critical. Interrupts that execute longer than expected can disturb the timing of device drivers. Also they can violate deadline constraints in real-time operating systems. Therefore, the programmer can not use a long-running function call that, for example, outputs data over the serial device. Similar constraints apply to many other areas of an OS kernel.

A additional restriction applies to environments like interrupts handlers. Functions like `printk` cannot be used in these routines. This creates a blind spot similar to the time before driver are initialized.

Within a kernel little or no protection from errors is provided. This make the environment fragile. When a user space application causes a segmentation fault or a similar

non-recoverable exception, the operating system can take over and terminate the application. How in the kernel, important structures like page tables and scheduler data are not protected. A kernel thread that writes to those locations will succeed and leave the operating system in an invalid state. Any further actions like debug output can then cause a kernel panic.

Virtual Machines

An execution environment that provides a low latency and in all situations usable solution to output values can avoid these situations. A virtual machine offers a good way to provide a low latency output facilities and debugging functions in a fragile environment. Virtual machines simulate a complete system for a guest kernel. It looks to the guest as if it is the only kernel on the system. Virtual machines either run unmodified or VM-adapted operating system (Paravirtualization). In many cases the host and the simulated architecture are identical but also the simulation of different architectures is common.

To provide a low latency output method the virtual machine can offer a facility similar to system calls. A signal can trigger the virtual machine monitor which stops the execution of the virtual machine. The virtual machine monitor can then output information to a screen or even slow devices like hard disks. Based on the application these are called *hypercalls* or *guest annotations*.

A virtual machine monitor can read data form the virtual machine when the state of the guest operating system is invalid.When a guest OS panics the VM machine can dump the memory without the risk of a file system corruption due to corrupted device drivers.

2.2.2 Aliasing

Aliasing is an architectural property that is exposed at system level. A memory value is accessible through multiple memory address. This makes the tracking of variables more complicated than observing a simple address.

Modern processors provide *segmentation* and/or *paging* capabilities. For debugging these introduce some problems. This section will mostly talk about paging because segmentation is rarely used nowadays.

The address translation enables an operating system to provide a continuous virtual address space for processes. Every process receives the full addressable space. Each of these address spaces is divided into *pages*. Pages that are not used by the program are not necessary present in physical memory. If a process tries to access a page that is not in memory the paging mechanism generates a *page-fault*. This gives the OS the opportunity to place the page in memory and establish a mapping of a virtual address to the physical address. The page itself can be placed everywhere in physical memory.

Multiple virtual addresses can be mapped to the same physical page. It is called aliasing because the data is accessible through multiple addresses. This is a problem for memory watchpoints, which are limited to monitoring a single virtual address.

An example are again the *x86* debug registers. The Intel System Programming Guide[Int13, Chapter 17] states that the comparison of debug register and the referenced address is done before translation to a physical address.

In addition the relations of physical addresses to virtual addresses are not time invariant. To free valuable system memory, currently unneeded pages can be swapped out and transfered to a hard drive. After a page fault the page can be put back everywhere in physical memory. Therefore the physical address changes while the virtual address remains constant.

When the debugger has to track accesses to a value in memory should watch both the virtual and the physical address. It has to monitor the physical location because multiple virtual addresses can map to the same physical address. Also it has to monitor the virtual address because the physical location in the memory can change over time.

Both the virtual and the physical address are available at the time of translation in the *memory management unit* (MMU). Before the translation only the virtual address is available. Afterwards, all accesses are done using the physical address. This makes the address translation mechanism the prevalent choice for tracking both.

2.2.3 Operating System Semantic Gap

On system level the debugger occasionally wants to retrieve information about the semantic state of an operating system in addition to source code line and file information.

For example, the programmer wants to analyse the behavior of multiple processes with respect to different schedulers. To get information about the runtime of processes the debugger has to recognize process switches and distinguish individual processes. Additional information is hidden in the operating system data structures, e.g. if a process is ready. To retrieve this information the debugger has to know how to interpret this data structures.

Guest Annotations

As with *printf* expressions the high level information can be passed to the debugger trough guest annotations. This option was discussed before in the context of restricted environments. Hypercalls or guest annotations from the guest OS can be used to pass high level information to the debugger.

Guest annotations provide a low latency and convenient way to output semantic information of the guest OS. For instance the scheduler could output ready processes. However similar to *printf debugging* this changes the program code.

Reimplementation of OS Functions

In the scheduler example an external hardware debugger or a virtual machine could easily read all relevant data structures since it has unlimited access to the memory. The meaning of the raw data is encoded in the data structures. To overcome the semantic gap the debugger can reimplement some of the operating system facilities in order to interpret this data structures. However the data structures that carry the process

information vary between operating systems and versions. This makes the adaption to new guest versions labor intensive.

2.3 Asynchronous Events

At system level a more complex hardware model is exposed. Asynchronous events of the hardware implementation become visible to the debugger. Asynchronous *exceptions* generated by external devices pre-empt the linear program execution and transfer control to handler code.

If no external events and data is used an execution would always follow the same path given the same initial state. When asynchronous events are added to the equation this is not true anymore. The program will only follow the same path when all asynchronous events in every execution are raised at the same time in relation to the program stream. This make it hard to reproduce executions. In particular in the iterative debugging pattern the programmer executes the program repeatedly to construct the chain of causes as discussed in the Introduction.

2.3.1 Asynchronous Events

Interrupts are similar to traps put require special attention. Both interrupt the program stream and trigger the execution of an exception handler. However, traps are generated by an instruction in the program. Therefore, the exception will always raise at the same position. But unlike traps, interrupts are not deterministic. They depend of the behavior of the external device and non-deterministic signals.

Record and Replay

Record and replay is a technique to avoid varying interrupt locations. In a first iteration all asynchronous events are recorded. In successive iterations this data is read and the events are raised at the same point in time.

A variety of methods allow to record and replay asynchronous events. The most common use virtual machines[KDC05] or operating system[MQB+08] support.

When a programmer tries to capture a rarely occurring behavior it can take days for the crash to happen. If the execution is recorded it requires a lot of memory space to save all important events. When the program is replayed it has to execute the same instruction stream. This again takes a long time. Idle times can be excluded during replay as an optimization. To avoid long replay durations the recording system can save checkpoints[KDC05]. Checkpoints are snapshots of the machine state. At checkpoints the replay system can start execution without knowledge of previous states.

2.3.2 External Information

Interrupts often signal the reception of data by an external device, for instance, a network interface controller that received a packet from the network. If the network is neither simulated nor under full control by the debugger network packets can arrive at

any time. An interrupt handler will then trigger the readout of the packet buffer. If the sending host is not under control by the debugger the information is non-deterministic.

Record and Replay

To retrieve the received information in a subsequent iteration the debugger records the data in the first iteration and replay it in all following executions. This is similar to the recording of asynchronous events.

In the record step the debugger has to identify all read accesses to the device. The received data has to be saved. In contrast to the record and replay of asynchronous events this information has to be tied to the reading instruction not to the point in time. Data written to the device is not saved because it is generated again in every new iteration.

2.4 Parallel Execution

Multi-processor systems, in particular shared memory systems, were present in servers for decades. Nowadays they find their way into handheld devices like mobile phones. Multi-processor systems consist of multiple processing units each executing their own instruction stream.

In single core systems non-deterministic behavior is caused by asynchronous events and external data as discussed in the previous section. The order of memory accesses is defined by the order of the instructions in the program stream. If multiple processes are executed the interleaving of these is defined by the scheduler. Hence, reproducing the position of external events also fixes the order of memory accesses.

In the multi-processor or multi-core case this order of memory accesses is not defined by external events. Since the execution on multiple cores is not synchronized the interleaving is arbitrary. If one core reads a memory location and a second writes a value to the same location the outcome is different depending on their order. When the location is written first the read will yield the just written value. Otherwise, it will yield the previously written value. As the control flow may depend on this value it is obvious that this changes the program flow and results.

The solution is to have a constant order of memory accesses over all cores.

Deterministic Execution

One could serialize the execution of multiple cores. Similar to external interrupts this fixes the interleaving of memory access. While this introduces a great performance penalty - the speedup through parallel execution is completely ignored - it may be easy to implement in some situation as we will see in Section 3. This implicitly fixes the order of the memory accesses on all cores.

Page Ownership Protocol

The second - less strict - method is to fix the interleaving of memory accesses as proposed by Dunlap et al.[DLFC08]. To be more precise the order of writes in relation to reads

and the order of write in respect to writes is important. The read - read order can vary because, independent of the order, both instructions will yield the same value.

The write events that transfer information from one core to another can be detected and ordered using a page ownership protocol. This can be implemented in OS kernels and virtual machines using the same principle.

The page permissions are set that either all cores can read or only one core can write to a page. When there are only reads all cores can access the page. If one core writes to this page it is assigned exclusive access and the event is registered. These exclusive accesses define the memory order in time. After being saved and replayed, the same order is enforced using yet again the paging mechanism.

3 Deterministic QEMU

This section will walk through the design decisions that were made and techniques that were used.

As discussed in the Introduction the goal of this thesis is to develop a system-level debugging tool. It has to have the ability to run regular x86 operating systems in an SMP setup.

As shown in Section 2 virtual machines offer a starting point. Virtual machines allow to run operating systems under controlled conditions. They offer the advantages of an external debugger and can be adapted to the needs of the debugger. The debugger attached to the virtual machine can read general purpose and even internal status registers. Also it can suspend the execution to inspect and change the machines state.

The implementation aims for an *iterative debugging* process. In the *iterative debugging* paradigm, the programmer repeatedly executes the operating system to gather more information in each step. This requires the execution to be reproducible. Otherwise the OS control flow follows a different path and the programmer would collect conflicting information.

To reproduce executions one can choose from two strategies. *Deterministic execution* and record-and-replay. Both make sure that asynchronous events are raised at the same point in time in every execution, given the same initial state.Record and replay saves the points in time in the first iteration and re-raises the event at the same point in successive executions. In contrast, *deterministic execution* does not save the asynchronous events. Due to the structure of the virtual machine events, that are asynchronous on real hardware, are raised at the same point in time in each iteration.

The second goal is to offer a scriptable interface that enables the programmer to control the execution of the virtual machine. Because it will be integrated into the virtual machine it can read all internal state of the virtual CPUs.

3.1 QEMU

This implementation will be based on QEMU (Quick EMUlator)[1]. QEMU is an open-source virtual machine written in *C* that supports many architectures. It is licensed under GPL hence the code is easily available and free of charge. The community is large, active and provides support for many CPU-architectures and recent CPU features.

This implementation will be based on the TCG version of QEMU. QEMU's back end offers two modes of operation KVM and TCG. KVM (Kernel-based Virtual Machine) is a kernel module that provides a common interface to hardware virtualization extensions like Intel VT-x or AMD-V. TCG (Tiny Code Generator) on the other hand translates the

[1] http://wiki.qemu.org

guest binary code to the instruction set of the host using an intermediate code format. The KVM version offers significantly higher speed due to the hardware acceleration and multi-threaded execution. But for our application, TCG is favorable because it offers much more control over the execution than the hardware accelerated version.

While former versions of QEMU relied on specific versions of gcc it now uses TCG. TCG translates each guest basic block called a *translation block* and puts it into the *translation cache*. Multiple translation blocks are linked directly to increase execution speed. The host then executes the code in the translation cache. This execution of translated code permits to append additional debug code into the translation cache that is executed together with the guest.

In addition to QEMU, many other open-source virtual machines exist such as VirtualBox, Xen, and Bochs. QEMU was chosen because these have properties that make it impossible or hard to meet the requirements. First of all Xen is a paravirtualized virtual machine. In contrast to QEMU it requires changes to the operating system code. Operating systems for Xen will not run on real machines and vice versa. Because we can not debug all OS code, Xen is unfeasible for system-level debugging. VirtualBox does allow to execute unmodified operating systems. It offers a software driven simulator similar to QEMU, but the code is patched and not translated. Therefore, the changes are visible to the executed operating system. Bochs offers full system emulation, however, it is slow and not feasible for huge programs. This made QEMU the preferable choice for the implementation of the system-level debugger.

Based on QEMU previous work was done by Michael Peter and Dmitry Nedospasov. The first extensions date back to version 0.9.1 released in 2008. Michael Peter implemented TLB tracing, guest annotations and basic logging for the x86 version of QEMU. I will describe the implementation in more detail later. In 2010 Dmitry Nedospasov ported these changes to the then recent version 0.12. Additionally he cleaned up the original implementation. The development for QEMU went on since then. Base of the implementation covered in this thesis is version 1.0.

Basic Structure of QEMU

Before there is more details one has to understand the basic architecture of QEMU. The virtual machine is simulated by multiple threads (Figure 3.1). The two important ones are the main thread and the CPU thread. These threads are synchronized by a *central lock*. Only one of them is executing at a time.

The CPU thread executes all binary code a real CPU would execute. It is running a core loop that is executing the code. This loop can be interrupted by a flag so interrupts are handled. While the loop executes it holds the central lock.

If multiple CPUs are executed the TCG version executes them interleaved. The CPU switch either occurs on external signals or if a CPU gives up control voluntarily.

After the startup function started the CPU's execution it runs the `main_loop_wait` function. This function waits on an event file descriptor (*eventfd*). If an event is notified (e.g. by the host timer) on this file descriptor the function wakes up and interrupts the CPU thread in order to acquire the central lock. It then executes elapsed timer handlers and handlers for asynchronous events.

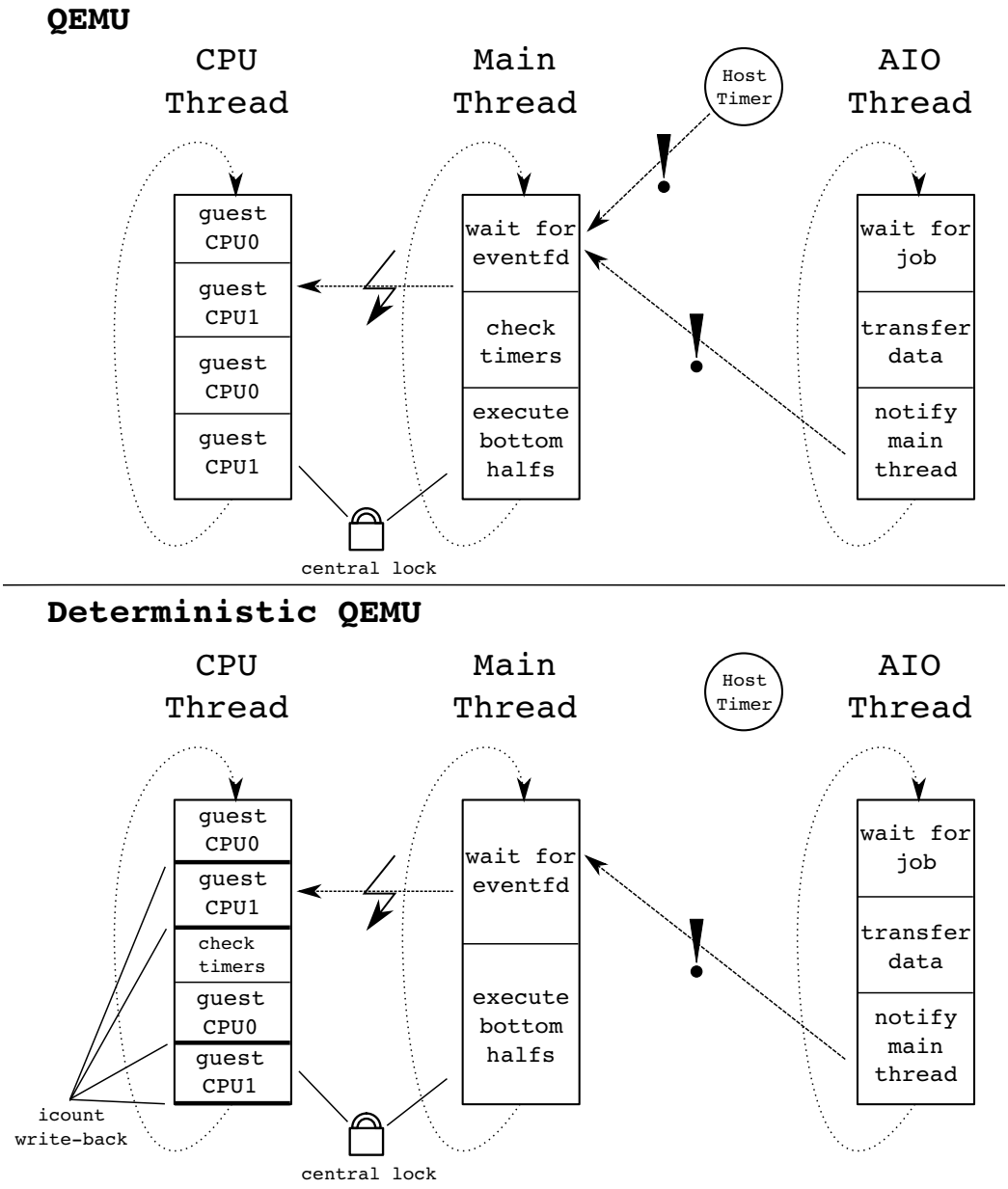

Figure 3.1: QEMU Thread Model

Most parts of the guest execution are deterministic in QEMU. In fact everything that is not implemented in multiple threads is deterministic. The execution of instructions including memory accesses are deterministic. Not deterministic are asynchronous events like timer interrupts and asynchronous I/O such as *direct memory access* (DMA) transfers.

3.2 Basic Debugging

A useful debugger has to provide information gathering features in addition to deterministic execution and script support. These mechanisms will provide the base for the rest of the implementation. Upstream QEMU offers basic debugging features. However these function are fairly limited and not well maintained.

It provides a *gdb* stub that is used to attach an external debugger. Basic features like break- and watchpoints as well as single-stepping are supported, but it lacks advanced hardware control. For example the debugger can only read physical memory (no virtual user space memory), has no access to QEMU's internal clocks and cannot directly control QEMU. The debugger is not aware that it is controlling a virtual machine.

Therefore *Deterministic QEMU* will not use the gdb stub but use some of the already integrated debug functions. Also the existing implementation by Michael Peter already provides three features. Recording of QEMU system events, marks in the guest code that trigger additional events, and memory monitoring of virtual and physical address spaces. All events are written to a log file.

Logging

The logging offers a method to record QEMU system events like CPU switches, CR3 (control register of *x86* architecture) reloads and timer expiration. All events are identified by a major and minor debug identifier. The *major ID* is used to specify the general event whereas the *minor ID* is used for subevents that provide additional information.

The logging routine is called from the QEMU source code using the function `dbg_log()`. This function expects four arguments. A double-word (32bit integer) that contains both a 16bit major ID as well as a minor ID of the same size. The remaining three values are 64bit integers that are chosen by the programmer. They can be used to record additional information such as pointers.

The `dbg_log()` function does not log the event immediately but allows the user to filter the events. Logging can be en- and disabled at event granularity in the QEMU source code. This requires to recompile QEMU.

Guest Annotations

As the second basic feature guest annotations offer a versatile low-latency solution to transfer information from the guest to the debugger. These were also already implemented by Michael Peter and ported to QEMU 1.0.

A guest annotation is a special instruction that is inserted in the guest code by inline assembly, triggers debug routines, and exports data at run-time when it is executed.

To avoid false events this instruction should be rarely used in regular code. This leaves two options, an opcode that is unused or useless. Unused opcodes cause portability issues because they can be used in later processor models and cause exceptions on real machines. Therefore, an unused instruction is the most portable one. The instruction also has to have `NOP` semantics. If it does not, the compiler will ignore optimization opportunities. For instance, an instruction that overwrites registers causes the compiler to reload local variables from the stack.

The useless instruction `and %eax, %eax` was chosen. Like a `NOP`, the instruction does no useful calculation and is not used by compilers. However, it changes the status registers.

QEMU recognizes the instruction when it decodes the guest binary code and before it generates host code. When a guest annotation is found the TCG generates a call to a helper function into the translation block. Therefore QEMU only checks for the instruction at translation time.

Because this code is put into the translation cache it is executed every time the basic block is executed. This function calls the `dbg_log()` function, which generates an event in the log file. Hence no additional delay is introduced during regular executions.

The event gives control information about the guest code. However it is useful to add additional information in the log file. This is done by putting values in registers that are read by the debugger after the instruction was identified. A preprocessor macro (Listing 3.1) is provided to a user that writes arbitrary variables in registers and inserts the guest annotation.

```
#  define maglog(major, minor,d0,d1,d2,d3,d4)            \
     { unsigned d;                                       \
        asm volatile("and %%eax, %%eax"   \
    :"=a"(d),"=d"(d),"=c"(d),"=b"(d),"=D"(d),"=S"(d) /*output*/  \
    :"0"(((major) << 16) | (minor)),"1"(d0),"2"(d1), /*input*/   \
     "3"(d2),"4"(d3),"5"(d4)                                     \
    :"cc"); }) /*clobber*/
```

Listing 3.1: Macro for inserting guest annotation

A major and minor ID are put into register `EAX`. Please note that these are not equal to the IDs that are used in QEMU events. The arguments `d0-d4` are stored in other *x86* registers.

This store operations require additional operations, which are generated by the compiler, that move the values in the chosen registers. Also the macro thrashes register values that have to be restored afterwards.

It is obvious that guest annotations are a low latency operation for the guest OS. The instruction looks like one useless executed opcode. All executed debug code is hidden from the guest. As this is a plain atomic instruction, it can be used in all situations. Unlike printing functions like `printk()` it has no limitation on the execution environment and can be used in situations like interrupt handlers.

Memory Tracing

The third basic debugging feature - in addition to logging and guest annotations - is memory tracing. All memory loads are translated by the paging mechanism and served directly by the simulated RAM.

The paging mechanism is a good place to implement the tracing. Every memory access goes through the MMU whether it is a RAM or a memory mapped I/O access. At this point the virtual address is translated into the physical address. This means both addresses are available at this point. The ability to track both addresses is important because of aliasing as discussed in Section 2.

Also memory tracing based on the translation mechanism gives a solution for tracking memory locations. To understand why, some details about QEMU's TLB implementation are required. QEMU simulates a virtual *translation lookaside buffer* (TLB) that acts like a real TLB. The TLB serves as a cache for the translations of a virtual to a physical address that is done by the MMU. Therefore, it is not necessary for the MMU to walk the page table for each translation. When a translation is performed that is not cached it is inserted into the TLB.

This insertion is used to implement the memory monitoring feature. At every insertion the debugger checks if the page's virtual or physical address range contains a monitored address. If this test yields a positive result the entry in the TLB is marked with a `TLB_SPECIAL` flag. This flag is written to an unused bit in the entry.

At every lookup that reaches the TLB, QEMU checks for the `TLB_SPECIAL` flag and - if present - executes a debug handler. This handler has to verify if the access actually touches a monitored address because it is triggered at page size granularity. When it was no false positive an event is generated.

In conclusion, *TLB tracing's* overhead is low because it limits the address check to interesting pages. The performance penalty for pages that contain no monitored address is a single flag check. However accesses to pages that are monitored can generate many false positives. But still the latency is low because the event handler is left after a quick range check.

3.3 Deterministic Execution

For the aimed for *iterative debugging* process the execution of the guest OS has to be reproducible. To achieve this I will use deterministic execution of the guest code by the virtual machine. Deterministic execution is easier to implement in comparison to record and replay.

Record and replay captures all non-deterministic behavior and replays them, when the programmer wants to reproduce the execution. For record and replay the virtual machine has to implement a method to record and another to replay executions. The record functionality has to identify asynchronous events and save the point in time when they occurred. At replay these points have to be found again and the event played back. In particular at replay time the virtual machine has to find the play-back points without too much overhead. Also a significant amount of data has to be stored. This requires

efficient handling and a good selection of saved data. If the selection is not good more data will be saved then required for playback.

In contrast to record and replay the virtual machine does not need to save asynchronous events. Due to the implementation of the virtual machine they happen at the same time in every iteration.

Both techniques require changes to the virtual machine. Record and replay needs a precise event insertion mechanism at replay time. Deterministic Execution depends on changes to interlink the generation of asynchronous events with the execution of guest code. However, deterministic execution does not require the selection and saving of recorded events. This makes deterministic easier to implement.

The deterministic implementation covers various types of formerly non-deterministic events and implements them deterministically. These are timer interrupt generation, clock readings, and the interleaving of multiple cores in SMP. This section will go through each of them and explain the changes.

Before I go into more details on how QEMU was patched to execute deterministically want to describe the extension to the command line interface. A new argument was added to the QEMU *command line interface*.

The new `-dbg` argument sets up all system level analysis tools described in this chapter. A follow-up integer value enables or disables the deterministic mode. This sets a global variable `deterministic_mode` that triggers changes that make the execution deterministic.

3.3.1 Interrupt Generation & Handling

In the single processor case only the generation of interrupts has to be deterministic. This means the interrupts have to happen at the same position in the instruction stream in every iteration.

In general all exceptions that are synchronous and deterministic in hardware are both also in QEMU. Software interrupts and exceptions such as page faults are generated deterministically. As on real hardware these interrupts are generated in the executing guest code. When a page fault is raised the execution is suspended immediately and a page fault is raised. The CPU thread breaks out of its guest execution loop and the corresponding handler is executed. This is inherently deterministic because all execution is done by a single thread.

Only some interrupts are inserted asynchronous. Two important functions implemented asynchronous timers interrupts and DMA requests.

Timers

Timers are most likely implemented asynchronously for performance reasons. All time counters are mapped to two virtual clocks `rt_clock` and `vm_clock`. Both clocks and therefore the timers are mapped to host clocks.

When a hardware timer, like a new APIC timer, is set up, a new timer is created on the virtual clock. A host timer is then set to the earliest deadline of all VM timers.

The SIGALRM handler is executed when the host timer expires. This handler wakes up the main thread, which interrupts the CPU thread. The main thread now checks for expired timers and executes the registered timer callback functions. These callback functions simulate the actual hardware and raise the virtual interrupt line to the processor.

When the host timer handler interrupts the CPU thread depends on scheduling decisions of the host operating system. Thus it is non-deterministic and not under control of the virtual machine.

Because the clocks are mapped to the host the time source is basically the host. This raises new problems that will be discussed in Section 3.3.2.

Deterministic Timer Expiration

The expiration of timers is non-deterministic because the check is triggered by a host timer. Instead the check has to be triggered by a synchronous event.

The check itself is done in the qemu_run_all_timers function in the main thread. In replayable QEMU this function was moved into the main thread outside of the CPU execution.

Because the check is now done in the CPU thread, the host timers that wake up the main thread can be disabled. Instead another mechanism has to interrupt the CPU execution frequently and deterministically. This mechanism is covered later.

Whenever the CPU execution is left the timers are checked. Because this will be deterministic all interrupts are generated deterministically.

In the new structure the timers are checked by periodic checking instead of host timer signals. This means many executions of the function are useless because they find no expired timers which leads to more overhead.

DMA

The second important function that is implemented asynchronously is DMA.

DMA request are handled by a thread pool that is created in addition to the two threads discussed before (Figure 3.1. These threads handle asynchronous I/O(AIO).

The request is issued from a coroutine in the guest code. When the request is issued the coroutine suspends and the main thread can continue executing guest code.

The AIO thread executes the request independent of the main thread. When it is done it wakes up the main thread.

As described before, the main thread interrupts the CPU thread and acquires the central lock. After this was completed, it executes the rest of the coroutine which are called bottom halves in QEMU. These raise the virtual interrupts.

This behaviour is inherently non-deterministic. It depends on the contention and the scheduling decisions of the host when the AIO thread is executed. Therefore, the CPU loop can be interrupted at an arbitrary time.

The handling of the non-deterministic behavior introduced by DMA is not covered by the *Deterministic QEMU* implementation. Guests that use DMA will experience non-determinism. In a later version DMA can also be reimplemented synchronous.

External Devices

A third source of asynchronous behavior are peripheral devices that accept external input. The most common device of this type is a network interface controller (NIC).

When a NIC receives a network packet it raises an interrupt to give the virtual CPU the opportunity to receive the packet. This changes the behavior of the virtual CPU depending on the time the packet arrives.

It is hard to replay external input when the external source like a host in a network is not controlled by the virtual machine. The only option is to record and replay the information.

While in record and replay setups can be easily extended to perform this recording, it is laborious in a deterministic setup. This setup will not cover external data sources. It is often sufficient to generate load, if required, in a separate process on the same system.

3.3.2 Time Source

In both deterministic execution and record and replay the time source plays an important role. For deterministic execution the VM needs a time source that is independent of the real time. Otherwise, if the VM is stopped, the clock will still advance and the insertion point of timer interrupts will change.

As discussed before QEMU uses the host clock to emulate the virtual clocks. This is not feasible for deterministic execution.

Instead it is possible to use the instruction count to determine the current time. When the clock is tied to the instruction counter, the time is always synchronous with the executed instructions. It is obvious, that timers based on the instruction counter expire at the same instruction in every iteration.

When no instructions are executed the time is halted for the virtual machine. This allows the VM debugger to execute code of arbitrary length without changing the execution of the guest. After the debugger code is executed the guest is resumed from the same state. To the guest it appears as if no time expired.

Tying the time to the instruction count causes problems in some modes of operation. Most processors offer idle and power saving modes that cause a shutdown of the processor's execution units. This means it does not execute any instructions and the instruction based clock is not advancing. When a OS kernel has no processes to schedule, it is going into idle mode and waits for a timer interrupt to wake up the CPU. When the clocks are not advancing the OS will be never woken up. In this case one either patches the operating to use busy waiting instead or the clock is deterministically advanced to the next deadline.

QEMU already offers an option to use instruction based timers called *icount* mode that can be used after some changes. If enabled the mode counts executed instructions and uses this number to determine the current time.

The instructions are counted by inserting additional code into the translated code. For every translation block prequels and sequels are generated. These contain the number of instructions that where translated into the block. At execution time these are collected and added to a global *icount* variable after the loop that executes guest code was left.

In icount the CPU loops exits the guest code execution more frequently to advance the CPU Clocks. If the loop is not left clocks would show odd readings because the collected *icount* value is not folded into the global counter. Two reads from the TSC-register would yield the same value because icount is only updated between CPU loop iterations.

To exit regularly each guest code iteration is granted a budget of instructions. The budget is decremented with each executed translation block. When the budget is consumed or the CPU is interrupted by an external signal the CPU's iteration is left. The budget is the smaller of either 65535 instructions or the delta to the next timer deadline.

For *Deterministic QEMU* the guest execution has to exit more frequently to service the timer expiration check which was moved from the main thread to the CPU thread. As a reminder, this is necessary because the CPU thread is not interrupted by the main thread in *Deterministic QEMU*. Thus each CPU iteration is granted a smaller maximum budget of 600 instruction.

When all virtual CPUs are in idle mode QEMU's *icount* mode advances the clocks to the next deadline. This behavior is called warping. While one would expect this technique to be deterministic it showed non-deterministic effects and was disabled for later investigation. Therefore the guest kernel has to be patched to use busy waiting instead of idle modes.

The rate of retired instructions is lower than the clock frequency. In order to achieve a realistic clock speed the global counter is multiplied with a scaling factor. If this scaling factor is not given QEMU's *icount* mode will adjust the scaling factor to fit to the host clock. This is non-deterministic. Instead for deterministic execution the scaling factor has to be set to a fixed value together with the enabled `-dbg` argument.

3.3.3 Deterministic SMP

The multiprocessor case puts more requirements on the implementation. Again the goal is to support *deterministic debugging*. For deterministic SMP the interleaving, in particular of memory access, has to be in the same order in every execution.

However, the already existing changes make also the SMP execution deterministic. In the TCG version of QEMU all virtual CPUs are executed by a single thread. The guest code of multiple CPUs is executed interleaved. When a CPU's guest code execution loop is exit the thread switches to a different CPU. In non-deterministic QEMU the loop is interrupted by the main thread trying to acquire the central lock.

The main thread does not interrupt the execution in *Deterministic QEMU*. As a reminder, this excludes the usage of DMA and external data sources as these two still use the main thread.

Instead the interleaving is controlled by the *icount* mechanism as discussed in Section 3.3.2. After a fixed number of instructions the guest code execution is left and it switches to a different CPU. So the interleaving of the guest code of all CPUs and the timer expiration is always fixed. This makes also the SMP execution deterministic.

3.4 Scriptable Refinement

The base implementation already offers basic automation through guest annotations. The generated code checks the payload of guest annotations for magic values to toggle debug functions. For example based on register values it turns on and off memory tracing.

While this introduces a basic form of automation it integrates debug logic together with the guest. If the guest crashes it can not control the debugger anymore. A better alternative, to control the debugger, is a script engine independent of the guest.

A script engine provides an interface for the programmer to wield control over VM runs, to inspect, and control the machine state. For a system level analysis tool based on QEMU it gives access to the VM's internals. This can ether be an interactive prompt (which can also be a GUI), a stub for a common debugger, a scriptable interface to the debugger, or a combination thereof. A scriptable interface is most feasible for the iterative debugging process, which is the main case of this thesis.

An interactive prompt is the best option for short investigations. The programmer can stop the program and examine the machine's state. But in an longer iterative debugging sessions one wants to use automation to save repeating work.

A script language gives the programmer the capabilities to automate the investigation. In each step of the iterative session the programmer can add additional debug code until the scripts can determine the origin of the bug.

3.4.1 Lua Script Engine in QEMU

Several stand-alone script languages offer the capability to be embedded into an application. Examples are popular languages like Perl, Python and Lua[dFIF94]. While all of these languages offer an embedding API, Lua is interesting in particular. Lua was designed from the start as an embedded language.

Replayable QEMU's command line interface offers, in addition to the `-dbg` argument, a `-lua` argument. This initializes the Lua runtime and executes a Lua script that is passed as a file name. The script is executed after Lua was initialized. This gives the programmer the chance to set up the debug environment before the VM's execution starts.

The interface offered to the programmer is event based and built around the low-level logging events introduced in Section 3.2. Every time the `dbg_log()` function is called and a Lua handler is registered for this event control is handed over to the script engine. Handlers can be registered at startup or any time the control is passed to Lua.

Whenever an event is generated *deterministic QEMU* has to look up if a handler is registered. To reduce the latency for the negative case the lookup is split in two steps as illustrated in Figure 3.2. QEMU implements a bit mask with one bit for every event in C. When a registered event is found the control is handed to a Lua dispatch function. Otherwise, it does not call the Lua runtime. Lua stores the function registered for each event in a hash table. The dispatch function looks up the handler function in a hash map and calls it. CPU state, event IDs, and additional payload are passed to the handler as arguments.

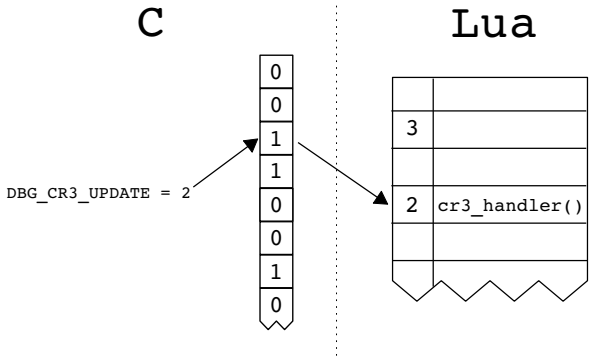

Figure 3.2: Lua Event Callback Lookup

3.4.2 Debugging API

Embedded Lua is designed around the principle of a shared stack. The stack is used to transfer arguments as well as result for calls from Lua to C and vice versa. If a C function calls a Lua function it has to look up the function in the Lua state and push it together with the arguments on the stack. For a reverse call the C function has to grab the arguments from the shared stack. From the Lua script perspective it makes no difference if the called function is implemented in C or Lua.

Wrapper Generator

While the initialization of the Lua runtime is simple, providing access to C function and structures is labor-intensive. When exported functions are not used exclusively by Lua, it is necessary to write a wrapper function which grabs the arguments from the common stack and does a regular C function call. In addition, all functions have to be registered in the global Lua namespace. Complex data structures have to be wrapped in setter and getter functions in order to access the fields. Because this is a common problem, automated tools exist to avoid repetitive programming.

Wrapper generators generate these interface functions from a description file. A generator that supports many interpreted languages is SWIG[2]. The glue code is generated from plain C/C++. The description code can contain C function declarations and implementations, structures and C++ objects, as well as Lua code. In the best case one can use a regular header file to generate the wrapper. However in many cases the description needs hand tuning because the SWIG complier is easily confused by complex or compiler-dependent C features. From this description C code is generated that offers a function to insert all exported functions into Lua's state.

[2] www.swig.org

API

The offered API gives detailed control over *Deterministic QEMU*. All of the basic features are exposed to Lua. In addition it allows to control and inspect the state of CPUs as well as to use the debug features. For example the script can read clocks and start timers.

Some parts of the API are exposed through functions, other parts are directly exported as data structures. Data structures are not limited to variables. Event IDs are used as preprocessor defines in QEMU. These are exported to Lua. This means the same constant name can be used for the handler registration as well as for generating the event. An example for an event ID is shown in Figure 3.2

CPU State The most important exported data structure is the `CPUState`. For every virtual CPU QEMU uses a `CPUState` structure. This structure contains the whole CPU state as well as all registers, hidden or not. Because some fields are common for multiple architectures the structure definition is composed from many files in the QEMU build process. Some fields are also aligned, which is not supported by SWIG. Therefore, it is almost impossible to let SWIG generate wrapper functions. This leaves the only option to copy the structure type definition that is the result of this generation process. While this works great for the user it has to be adapted to every new QEMU version and architecture.

Numerous functions are exposed to the Lua script engine. The reminder of this section will go through some of them. A complete list can be found in Appendix A.

Timers The Lua API offers instruction accurate timers. These timers use QEMU's clock facilities. As discussed these clocks were changed to use the instruction count as the time source. There are two types of clocks CPU-local clocks and a global clock. The global clock counts instructions of all CPUs. The local clocks are attached to individual CPUs. All timers in QEMU are associated to a clock. When the timer expires the guest execution is left and a callback handler is executed. The leaving of the guest execution is instruction accurate.

The clocks as well as functions to register timers are exposed to Lua. Therefore, the Lua engine can register instruction accurate timers on the clocks. Every timer has an associated Lua callback function that is registered in a hash table. The lookup is similar to log events. A generic C callback function is called by QEMU. This calls a Lua dispatch function that looks up and calls the Lua handler.

Instruction accurate timers enable the debug scripts to exactly time changes to the guest execution. One could delay the execution of one CPU for a fixed amount of instructions executed on another CPU. However, the QEMU implementation only allows to stop all CPUs at once. For this purpose two new functions `cpu_stop()` and `cpu_resume()` have been implemented and exported to Lua.

DWARF Information The relation of machine instructions to a source code is set by the compiler. The scheduling decisions of the compiler determine where instructions are

placed. Fortunately most compilers such as gcc allow to output relational information. This information is included it the ELF binary using the open DWARF[3] standard.

The DWARF data contain different information about the binary such as symbol names. But the most important information is the section that relates code addresses to lines in the source code. A script is provided together with *deterministic QEMU* that exports this information to a Lua data structure. The Lua data structure enables the debug script to output the corresponding source code position. This shows that *deterministic QEMU* is easy to extend due to the integrated script engine.

The Lua interface of *deterministic QEMU* provides the function `qemu.dwarf_find_line()` to look up the line information corresponding to a given code address.

[3] http://www.dwarfstd.org/

4 Evaluation

This section will evaluate the feasibility of the implemented features in *deterministic QEMU* as well as the *iterative debugging* approach. The evaluation of the *iterative debugging* approach is important because many VM-based system level debugger favor record and replay over deterministic execution.

4.1 Synthetic Use Case

The application runs on a Linux platform executed by a *x86* SMP system. Buildroot[1] was used to generate a operating system image using version 3.2.6 of the Linux kernel.

The application consist of two individual processes. Listings of the source code can be found in Appendix B. It implements a server application that transforms strings. Process 1 - the receiver - receives data from the network and sends them through a buffer to process 2 - the filter. After the filter is triggered by a flag within the memory it moves the string to a new buffer. During this operation it exchanges every 'e' by 'a'. After the translation is finished the new buffer's offset is written to a shared variable. Then the filter process goes back to waiting state. The receiver then sends the information to the requester.

The application works as expected. But for some inputs the receiver is terminated by the operating system due to a segmentation fault.

4.1.1 Deterministic QEMU

A segmentation fault means the process is trying to access a virtual address that was not assigned to the process. The operating system outputs information about the current position in the program (instruction pointer) and the offending address. In a one process application the programmer can lock for this location and trace back the source of the offending address. Yet this application is special.

Each process is running in its own address space but they share data through shared memory. The processes map a POSIX shared memory object into their address space. Each process may map the same shared memory object to different locations in their address space. This means in each address space a virtual address maps to the same physical location.

To monitor both addresses one could run both applications in their own OS supported debugger. But it is unclear how both applications behave in their own debugger. Therefore, the setup is executed in *Deterministic QEMU*. In *Deterministic QEMU* it is simple to monitor both the physical and the virtual address.

[1] http://buildroot.uclibc.org/

4.1.2 Segmentation Fault Location

The operating system outputs information about the segmentation fault to the console. However this information is not directly available to the debugger. From the hardware's perspective it look like any page-fault.

When a process accesses a virtual address that is not mapped to a physical page in RAM a page-fault is raised and the control is transfered to the kernel. The Linux kernel than decides if the memory area belongs to the virtual address areas that are assigned to the process. If this is not the case the process is terminated using the `SIGSEGV` signal.

Segmentation Fault Address

So only Linux knows if it is actually a segmentation fault. This shows the semantic gap discussed in Section 2.2.3.

A guest annotation will export the information to the debugger. Kernel version 3.2.6 implements the range check in the file `arch/x86/mm/fault`. This file (Listing 4.1) is annotated using the `maglog` macro. It uses the major ID 3 and outputs the offending address and instruction address in the process. The message that is output by the kernel is generated in line 3.

```
765     maglog(3,0,(int)address,(int)regs->ip,0,0,0);
766     if (unlikely(show_unhandled_signals))
767         show_signal_msg(regs, error_code, address, tsk);
768
769     /* Kernel addresses are always protection faults: */
770     tsk->thread.cr2     = address;
771     tsk->thread.error_code  = error_code | (address >= TASK_SIZE);
772     tsk->thread.trap_no = 14;
773
774     force_sig_info_fault(SIGSEGV, si_code, address, tsk, 0);
```

Listing 4.1: Exerpt from arch/x86/mm/fault.c

The guest annotation causes *deterministic QEMU* to generate an event for the integrated Lua engine. Therefore in addition to the annotation a handler has to be implemented and registered. Listing 4.2 (for complete script, see Appendix C) shows the implementation of this handler. The implementation reads the major and minor ID from the CPU's state. If the major ID is 3 and therefore a segmentation fault event the information transfered by the annotation is output. In line 10 the function `maglog_handler` is registered as the handler for all events generated by guest annotations.

```
1  function maglog_handler(cpustate, major_idx, minor_idx, payload)
2      major = qemu.get_maglog_major(cpustate)
3      minor = qemu.get_maglog_minor(cpustate)
4
5      if major == 3 then -- segmentation fault
6          io.stderr:write(string.format("segmentation fault at %x eip=%x \n",
               cpustate.edx, cpustate.ecx))
7      end
```

```
8  end
9
10 qemu.register_event_handler(qemu.DBG_USER, maglog_handler)
```

Listing 4.2: maglog Event Handler

The debugger now outputs the address of the instruction that dereferenced the offending address. But this has no obvious relation to a line in the source code. Therefore the information has to be added.

Source Code Information

As described in Section 3 *deterministic QEMU* offers a script to transform DWARF debug information to a Lua data structure. Listing 4.3 shows an excerpt of the data structure that is imported in the debug script.

```
dwarf = {}
dwarf.receiver = {}
dwarf.receiver[0x80487ac] = "receiver.c:21"
dwarf.receiver[0x80487bb] = "receiver.c:22"
dwarf.receiver[0x80487c7] = "receiver.c:23"
[...]
dwarf.filter = {}
dwarf.filter[0x804860c] = "filter.c:18"
dwarf.filter[0x8048618] = "filter.c:20"
dwarf.filter[0x8048624] = "filter.c:21"
[...]
```

Listing 4.3: "Lua DWARF Line Information"

Using `qemu.dwarf_find_line()` it is now possible to display line information in addition to the instruction pointer. Listing 4.4 shows the resulting output.

```
segmentation fault: access to bfca0fff at eip=8048cc4(receiver.c:116)
    esp=bf9e3890
```

Listing 4.4: Annotated Segmentation Fault with Line Information

A quick look into the source code shows that at this line the pointer to the return buffer of the filter process is dereferenced. This address is calculated by adding the offset returned by the filter to the base address of the shared memory. Apparently this address in corrupted. Once both values are output by a guest annotation one can see that the offset value is unusual.

4.1.3 Location of Writing Instruction

Now that we know the source of the offending address it is time to track down its source. The corrupted `ret_ptr` variable is located in the shared memory. As described both processes write to and read from the same shared memory. This means multiple virtual addresses point to the same physical address. Hence we have to track access to the physical address. Fortunately *Deterministic QEMU* offers tracking of physical addresses.

Detecting Stores

To enable tracking of writes to the variable the Lua script has to identify the location of the variable. One could either reconstruct the information from DWARF symbol information or inform the script through a guest annotation. I will use a guest annotation. The receiver executes a guest annotation with the major ID 1. The guest event handler (Listing 4.5) is extended and now activates the memory tracing.

```
if major == 1 then --start tracing
    io.stderr:write(string.format("tracking addr=%x\n", cpustate.edx))
    qemu.dbg_start_tlb_tracing_phys(cpustate, cpustate.edx, cpustate.edx + 1)
end
```

Listing 4.5: Memory Tracing Activation

This enables the generation of a `DBG_STORE_ACCESS` event. To receive this event the Lua script has to register a new event handler. The handler receives all important information, the virtual address, the physical address, and the current instruction pointer.

In the first part of the investigation it was obvious that all position information was within the `receive` process. But now both processes - the receiver and the filter - can write to the shared memory. This means Lua receives an instruction pointer from *Deterministic QEMU* but it is not certain to which virtual address space it belongs. So it is important to distinguish different processes.

Process Tracking

The Linux kernel switches between the currently running tasks. To identify the current process one could annotate or interpret the task structures. However, this requires more patching of the Linux kernel and is version dependent.

A context switch is easy to identify for *Deterministic QEMU*. The `CR3` *x86* control register contains the base address of the current page table. Hence, it changes during a context switch. When the `CR3` register is written QEMU issues a `DBG_CR_UPDATE` event.

Yet not every kernel task has its own page table. Multiple threads in one process use the same address space. Therefore they use the same page table base address. Processes do use different address spaces and have their own distinct `CR3` value. This can be used to identify the individual processes since there are no threads in this setup.

Deterministic QEMU offers facilities to identify processes. The function `task_track()` ties a `CR3` value to a process name. The reverse function `task_query()` looks up the corresponding name associated to the current `CR3` value. Therefore in both processes a new guest annotation was introduced to mark the current thread.

The store event handler can now be extended to print the line and source file name of each write operation (Listing 4.6) to the offset value. The last store to this location will give more information about the source of the wrong value.

```
function st_handler(cpustate, major_idx, minor_idx, payload)
```

```
  io.stderr:write(string.format("store from %s at %s eip=%x retaddr=%x
      vaddr=%x paddr=%x\n",
    qemu.task_query(cpustate),
    qemu.dwarf_find_line(dwarf, qemu.task_query(cpustate), payload.t0),
    cpustate.eip,
    payload.t0, payload.t1, payload.t2))
end
```

Listing 4.6: "Store Handler"

```
1  tracking filter task
2  tracking receiver task
3  tracking addr=bfbf1043
4  store from filter at filter.c:61 eip=80487e7 retaddr=80487e7 vaddr=b774b043
       paddr=1734043 ebp=bff0dfe8
5  store from receiver at receiver.c:122 eip=8048d1d retaddr=8048d1d
       vaddr=bfbf1043 paddr=1734043 ebp=bf9e3b38
6  store from filter at filter.c:61 eip=80487e7 retaddr=80487e7 vaddr=b774b043
       paddr=1734043 ebp=bff0dfe8
7  store from receiver at receiver.c:122 eip=8048d1d retaddr=8048d1d
       vaddr=bfbf1043 paddr=1734043 ebp=bf9e3b38
8  store from filter at filter.c:61 eip=80487e7 retaddr=80487e7 vaddr=b774b043
       paddr=1734043 ebp=bff0dfe8
9  store from receiver at receiver.c:122 eip=8048d1d retaddr=8048d1d
       vaddr=bfbf1043 paddr=1734043 ebp=bf9e3b38
10 store from receiver at receiver.c:105 eip=8048c04 retaddr=8048c04
       vaddr=bfbf1043 paddr=1734043 ebp=bf9e3b38
11 store from receiver at receiver.c:105 eip=8048c04 retaddr=8048c04
       vaddr=bfbf1044 paddr=1734044 ebp=bf9e3b38
12 segmentation fault: access to bfca0fff at eip=8048cc4(receiver.c:116)
       esp=bf9e3890
```

Listing 4.7: "Final Output"

Listing 4.7 shows the final output of the debug script. In line 11 one can see that the receiver writes to the offset variable. During normal operation only the filter process writes to this variable. A look in the corresponding line in the source code of the receiver reveals at this location it is copying the input to the buffer that is read by the filter. The buffer overflows and the values directly behind the buffer are overwritten. One of those is the offset value.

The successful investigations show the applicability of the *iterative debugging* process together with *deterministic QEMU*. During the investigation the output showed no conflicting information as one would expect from interleaving dependent executions. Together with guest annotations *deterministic QEMU* is feasible for real world application debugging.

5 Related Work

Early system level analysis tools were external devices used to verify timing constraints in real-time operating systems[Fry73, Pla84]. To gather in information about accessed memory locations and executed instructions the device is attached to the system bus of the CPU. This analysis is performed during a normal run. To perform predicate checks the analysis device has to be reasonable faster than the monitored system. Plattner estimated the required monitor to be $10 - 15$ times faster.

Real-time and hardware assisted analysis approaches became more rare as the processors became faster and more complex. As this puts more pressure on the requirements on suitable tools. This led to the development of VM-based solutions like the debugger developed in this thesis.

5.1 CHESS

CHESS[MQB+08] is a tool that supports the search for Heisenbugs. Heisenbugs are concurrency bugs that hide for a long time and only occasionally pop up under fragile configurations.

The tool's goal is to explore all interleaving non-determinism in the program. To explore this state space they implement a scheduler that is invoked at every call to a concurrency primitive. The program is only suspended at this points.

In a first iteration all scheduling choices are recorded. During the following iterations all permutations of scheduling decisions are tested. The search can be time consuming because the state space grows with the number of pre-emption points.

Because the program is only pre-empted at concurrency API calls and executed single-threaded the ability to find data-races is limited. The authors address this problem by using a data-race detector to introduce more pre-emption points.

The technique is implemented by intercepting API calls to concurrency primitives. No changes to the library and program are required. Further pre-emption points for data-race detection are introduced by binary instrumentation. The authors implementation supports Windows, Singularity, and .Net. This limits the application to user space programs.

To address input non-determinism CHESS records system call return values and error values. However it does not guarantee full determinism. For example data that is read from a file is not recorded. Therefore the underlying test infrastructure has to offer precautions to clean the environment before execution.

After a crashing or non-terminating execution has been found, the recorded interleaving can be replayed and attached to a debugger. The debugger can now single-step through the program while CHESS controls the interleaving.

Compared to *deterministic QEMU*, CHESS is focused on user-space applications. It covers the interleaving of multiple instruction streams and bugs that are caused by it. However, it lacks debugging features like breakpoints and state inspection.

CHESS is targeted towards user level applications that use OS APIs. For low level system analysis various tools have been proposed using virtualization.

5.2 TTVM

King et al. present the Time Traveling Virtual Machine(TTVM)[KDC05]. Instead of iterative debugging they propose reverse debugging. A virtual machine is used to record execution history and checkpoints. This allows the programmer to travel and single-step trough long running executions.

Instead of re-executing, reverse break- and watchpoints make it easy to find the last offending event. The programmer plays back the recorded iteration until the crash and then goes back in time to investigate the cause. Reverse debugging allows to restore and investigate corrupted or dropped stack frames.

As the implementation is based on ReVirt[DKC+02] it also uses User Mode Linux. User Mode Linux is a paravirtualized implementation of the Linux kernel. The authors claim it shares 98% of the code base with a vanilla kernel and can, therefore, be used for kernel debugging.

To debug bugs in device drivers, which are notoriously hard to find, they modified User Mode Linux. The proposed implementation can use unmodified device drivers using a trap-and-forward architecture.

TTVM features a more advanced debugging pattern compared to the *iterative debugging* approach. Time travel allows faster debugging since it does not require the re-execution of the virtual machine. However, unlike *deterministic QEMU*, the implementation is limited to paravirtualized kernels and, therefore, requires changes to the source code.

5.3 SMP-ReVirt

SMP-ReVirt[DLFC08] was presented as the successor to Re-Virt, which is also used as the VM in TTVM. They add SMP recording features to record modern multi-core CPUs.

The focus of this work is on recording and replaying complete system, in particular for intrusion analysis. SMP-ReVirt provides no build-in debug capabilities.

ReVirt was based on User Mode Linux where the SMP version is based on Xen. Xen was optimized towards speed and therefore performs well. However it uses a simplified hardware device models to reduce the driver overhead for the guest. This and many other guest modifications make analysing low level system code impossible or unreliable.

External events are recoded and tagged with position information. SMP-ReVirt uses hardware branch counters to quickly find reinsertion positions.

The authors identify the ordering of shared memory accesses as the main source of non-determinism in SMP. Replaying an SMP-system means restoring the order of memory

accesses. Dunlap et al. use a CREW(concurrent read, exclusive write) protocol to detect and play-back the order.

To implement the CREW protocol they use the paging mechanism on page level granularity. Therefore false-sharing is not limited to cache-lines but is also present on page level. This lets applications that contain a high level of sharing perform poorly while independent applications suffer little performance penalty.

In comparison to *deterministic QEMU*, the Xen approach makes the solution unfeasible for low level system code debugging. However, as described before the record and replay is superior to deterministic execution.

5.4 SimOS

SimOS[Her98, RHWG95] was proposed as a solution to collect execution characteristics of modern computer systems.

The analyser implements a deterministic virtual machine, that simulates a complete computer system. The virtual machine is implemented using the Embra Hardware Simulator[WR96] which uses binary translation to emulate the target systems CPU. The system is capable of running an unmodified version of SGI/IRIX.

To reduce the overhead of full system simulation SimOS implements different versions of the same architecture. This is useful because different types of analysis require varying amount of information. A cache analysis or the boot-up process do not need cycle accurate information. The limited detail is traded in for higher speed.

The main target of SimOS is performance analysis. However it is also equipped for debugging tasks. A gdb stub is provided for interactive control like single stepping. Symbol information of the executable can be used as additional context information.

Similar to *deterministic QEMU* SimOS offers an integrated interpreted language. The tcl interpreter allows to implement flexible information processing rules. It supports fast and simple event filtering and event counting for basic information collection.

Unlike *deterministic QEMU* which simulates *x86*, it only simulates the MIPS and Alpha architecture. SimOS is not under active development, at the time of writing of this thesis.

5.5 Simics

The currently most advanced commercial system level analysis tool is Simics[AM00, MCE+02] distributed by the Intel subsidiary Wind River Systems Inc.

Simics implements a full system simulator that is mostly used in processor development and debugging. The execution itself is fully deterministic for multi-core systems.

To overcome input non-determinism Simics supports the recording and replay of external events. In addition multiple simulations can be run clock-synchronized.

A number of features, some of them are also present in *deterministic QEMU*, allow to use Simics for debugging. Guest annotations and machine state changes can trigger handlers that are implemented in Python. The script engine has access to all hardware devices including bus systems and system memory.

The simulator offers "OS Awareness". It reproduces the facilities of operating systems like Linux to interpreted the data structures to overcome the semantic gap. This enables programmers to, for example, track processes[Win10].

Simics provides a similar set of features like *deterministic QEMU*. It is also used for investigations like conducted in this thesis' evaluation. However it is proprietary software and has to be licenced from Wind River.

5.6 Other Applications

Besides debugging various other applications for system level analysis tools have been proposed. Proposals include running production systems on top of a system level analyser in order to allow intrusion detection and secure logging at runtime. [CN01]

6 Conclusion

In this thesis a system-level analysis tool based on the virtual machine QEMU was designed and developed. The tool called *Deterministic QEMU* executes *x86* SMP production operating system kernels such as Linux and Fiasco.

Deterministic QEMU implements several features that are required for system level operating system debugging. First these are basic information gathering facilities to handle the huge state space. TLB tracing, guest annotations, and logging of systems events that were developed by Michael Peter have been ported to a new QEMU version.

To reproduce executions that use asynchronous features of architectures like interrupts and multiple processors the VM was changed. The execution of these events is now deterministic and will happen at the same point in time in every execution given the same initial state. This allows to reproduce infamous Heisenbugs.

At last the Lua script engine was embedded into *deterministic QEMU* to provide a scriptable interface to the debugger. The exported functions allow to control all debug features and the VM's interleaving of cores.

The evaluation showed that the implemented functions provide a valuable tool for OS debugging. The investigations showed that *deterministic QEMU* is a helpful tool in an *iterative debugging* process. During investigations the programmer can re-execute the guest OS and retrieve non-conflicting information. Scripted control of the debugger saves a lot of repetitive work and allows to activate and deactivate tracing of events at runtime. However, the implementation of *Deterministic QEMU* is still limited and leaves much room for future improvements and new features.

6.1 Limitations

The presented implementation of *Deterministic QEMU* is still proof of concept. Mostly due to time constraint the implementation focuses on the core features and ignores many areas like portability, speed and even the reproduction of some bugs.

QEMU has support for many target architectures like *x86*, *ARM*, and SPARC. However, this work focuses on the *x86* architecture. Many features like guest annotations and the script engine are implemented in the *x86*-architecture specific code of QEMU. Hence, to support other guest architectures much of the work has to be repeated.

While many sources of non-determinism in the *x86* architecture have been replaced by deterministic counterparts two important sources have been left out. Guests that use DMA are still experiencing non-deterministic timing because asynchronous I/O is implemented in a separate thread. Like DMA external data sources can change execution path. The only possibility to replay those executions is to record and replay the incoming data which is not supported by *Deterministic QEMU*. During all investigations external data sources like network devices were disabled.

Even when operating systems do not use those features small patches are required to enable full determinism. The instruction based clocks are not advancing if no instructions are executed. The QEMU enhancements support to advance the clock during idle periods (warping) but the implementation is still non-deterministic. Hence, the idle mode was disabled in debugged operating systems.

During the evaluation the performance penalty was not measured. But one can easily see that *Deterministic QEMU* is slower due to removed optimizations. For instance in the original implementation QEMU uses the host timer to run guest timers. The execution of guest code is only exited if a timer expired. *Deterministic QEMU* has to periodically check for the expiration of timers and, therefore, introduces a significant overhead.

Finally the implementation is bounded in the ability to reproduce bugs. The evaluation showed that it can reproduces many bugs but its capabilities are limited in certain areas.

The ability to reproduce interleaving dependent bugs or Heisenbugs is limited. Interleaving dependent bugs depend on the interlacing of processes or memory accesses on multiple processors. However both are constant in *Deterministic QEMU*. First is constant due to deterministic timer interrupts the second due to fixed interleaved execution of multiple cores on one thread. Bugs that reproduce in this setup will always reproduce. But this hides all bugs that do not reproduce in this fixed interleaving.

As described before input data is not recorded and replayed. Thus bugs that are triggered by external data through hardware devices cannot be reproduced deterministically.

6.2 Future Work

As the last section showed there are many possibilities for improvements. However besides obvious improvements like enhanced speed, portability, and more exposed functionality in the script engine there are several new features that would greatly improve *Deterministic QEMU*.

6.2.1 Iterative Debugging Improvements

Iterative debugging with *Deterministic QEMU* is useful but time consuming. Every time the programmer wants to gather more information he has to re-execute the VM. In particular for long running applications like operating systems this is time consuming.

Checkpoints Checkpoint as implemented in various proposed system level analysis tools[KDC05, LBP+07] can greatly reduce the time to re-execute the VM. A checkpoint contains all information to resume the virtual machine from these points. In the first execution one or more checkpoints are saved. Every successive iteration can be restarted form this point. Upstream QEMU already supports checkpoints of running virtual machines. These facilities can be wrapped for the embedded Lua engine.

Record and Replay While checkpoints and deterministic execution allow to reproduce many behaviors they do not reproduce external data and arbitrary interleaving. Only recording and replay can reproduce these executions. The existing implementation already offers many facilities that can aid the implementation of record and replay. To implement the recording one could utilize the event generation of the current implementation. The event handlers can then be used to write the events with attached time stamps to a log file. The deterministic clock of QEMU can serve as a reliable time source required for replay.

Reverse Execution As discussed in the Introduction in an investigation the programmer searches the chain of causes backwards in time. Therefore the implementation of reverse execution as proposed by King et al.[KDC05] would aid the debugger. Instead of finding the last store before a corruption the programmer would define a reverse breakpoint that would find the offending store first. Also the last non-corrupted state could be restored easily.

Guest Annotations

Guest annotations proved useful during the evaluation. However they leave room for improvement.

When guest extensions are inserted in guest code they add additional instructions. Like instrumentation through *printf* this changes the timing of the code. In a future version of *Deterministic QEMU* this penalties could be avoided. As the clock is under control by the VM it could subtract the time required to execute the guest annotation.

The information exported from guest kernels is similar in between investigations. For example the detection of a segmentation fault is a common exercise. Thus patches that introduce guest annotations into common operating systems at distinctive points can reduce the debug time.

Automated Debugging

Based on *Deterministic QEMU* automated debugging techniques could be implemented. This section discussed that many non-deterministic bugs are interleaving dependent and QEMU's interleaving is fixed. However the interleaving of multiple processor and the expiration of timers can be manipulated by Lua. This puts us in position for automatic interleaving exploration.

One could implement naive fuzzing and try all possible interleaving. But the space of all interleaving is infinite. Researchers have proposed more goal-oriented methods. First Musuvathi et al.[MQB+08] proposed to detect data races first. During successive executions all possible outcomes of these data races are explored. This reduces the search space radically. A similar approach was proposed by Xiong et al.[XPZ+10]. They used static analysis to find ad-hoc synchronisation constructs. The code was then annotated and analysed with the same tool.

Glossary

API Application Programming Interface - Collection of functions that are offered to the programmer.

APIC Advanced Programmable Interrupt Controller - Interrupt controller for the *x86* architecture. Also provides a timer.

DMA Direct Memory Access - Hardware device that transfers data independent of the CPU between memories.

DWARF ELF Debug information format.

gdb GNU Debugger - A debugger used by many Unix operating systems.

Interrupt External signal that interrupts normal CPU execution and triggers the jump to an exception handler.

MMU Memory Management Unit - Device that assists the CPU with memory management. It translates a virtual address issued by the CPU to a physical address.

POSIX Standard for OS interface.

SMP Symmetric Multiprocessor - Multi-core architecture with multiple identical processing elements that share a single memory.

VMM Virtual Machine Monitor - Program that supervises the execution of a virtual machine.

x86 Processor architecture family by Intel.

Bibliography

[AM00] L. Albertsson and P. S. Magnusson. Using complete system simulation for temporal debugging of general purpose operating systems and workloads. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on*, pages 191 –198, 2000. 39

[CN01] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HOTOS '01, pages 133–, Washington, DC, USA, 2001. IEEE Computer Society. 40

[dFIF94] Luiz Henrique de Figueiredo, Roberto Ierusalimschy, and Waldemar Celes Filho. The design and implementation of a language for extending applications. In *IN XXI SEMISH*, pages 273–284, 1994. 27

[DKC+02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, December 2002. 38

[DLFC08] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 121–130, New York, NY, USA, 2008. ACM. 5, 15, 38

[Fry73] Richard E. Fryer. The memory bus monitor: a new device for developing real-time systems. In *Proceedings of the June 4-8, 1973, national computer conference and exposition*, AFIPS '73, pages 75–79, New York, NY, USA, 1973. ACM. 37

[Her98] Stephen A Herrod. Using complete machine simulation to understand computer system behavior. Technical report, Stanford, CA, USA, 1998. 39

[Int13] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manuals*, 2013. Version 045. 9, 13

[KDC05] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. University of Michigan, 2005. 5, 14, 38, 42, 43

[LBP+07] Oren Laadan, Ricardo A. Baratto, Dan B. Phung, Shaya Potter, and Jason Nieh. Dejaview: a personal virtual computer recorder. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 279–292, New York, NY, USA, 2007. ACM. 42

[LT93] N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993. 3

[MCE+02] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50 –58, feb 2002. 5, 39

[MQB+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association. 5, 14, 37, 43

[Pla84] Bernhard Plattner. Real-time execution monitoring. *Software Engineering, IEEE Transactions on*, SE-10(6):756 –764, nov. 1984. 37

[RHWG95] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the simos approach. *Parallel Distributed Technology: Systems Applications, IEEE*, 3(4):34 –43, winter 1995. 39

[Win10] Wind River Systems Inc. *Fixing an Intermittent Multi-core Bug with Wind River Simics*, 2010. 40

[WR96] Emmett Witchel and Mendel Rosenblum. Embra: fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '96, pages 68–79, New York, NY, USA, 1996. ACM. 39

[XPZ+10] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association. 43

[Zel09] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier, Amsterdam, 2. ed edition, 2009. 3

Appendix A

Lua API

```
--[ Callback Registration ]--
qemu.register_event_handler(event_idx, handler)
qemu.unregister_event_handler(event_idx)

--[ Retrieval of maglog ID ]--
qemu.get_maglog_major(cpu_state)
qemu.get_maglog_minor(cpu_state)

--[ CPU Index and State ]--
qemu.get_cpu_index()
    -- returns Integer
qemu.qemu_get_cpu()
    -- returns SWIGCPUState
qemu.cpu_stop(cpu_state)
qemu.cpu_resume(cpu_state)

--[ Clocks and Timers ]--
qemu.get_local_clock_ns(cpu_state)
    -- returns QEMUClock
qemu.get_global_clock_ns()
    -- retruns clock value
qemu.get_clock_ns(clock)
    -- returns clock value

qemu.start_global_timer(delta, callback, arg)
qemu.start_local_timer(cpuID, delta, callback, arg)

--[ Memory Tracking ]--
qemu.dbg_start_tlb_tracing(cpu_state, start_addr, end_addr)
qemu.dbg_stop_tlb_tracing(cpu_state)
qemu.dbg_start_tlb_tracing_phys(cpu_state, start_addr, end_addr)
qemu.dbg_stop_tlb_tracing_phys(cpu_state)

--[ Process Tracking ]--
qemu.task_track(cpu_state, name)
qemu.task_not_track(cpu_state)
qemu.task_query(cpu_state)
    -- returns name of current process

--[ Debug Information ]--
qemu.dwarf_find_line(dwarf_db, name, instr_ptr)
    -- returns file and line as string
```

Appendix B

Use Case Programs

```
1  #include <stdlib.h>
2  #include <signal.h>
3  #include <stdio.h>
4  #include <sys/mman.h>
5  #include <sys/stat.h>
6  #include <sys/types.h>
7  #include <sys/socket.h>
8  #include <arpa/inet.h>
9  #include <fcntl.h>
10 #include <unistd.h>
11 #include <string.h>
12
13 #include "maglog.h"
14
15 #define SHM_SIZE 4096
16
17 #define BUFFER_OFFSET 0
18 #define FLAG_OFFSET   64
19 #define ADDR_OFFSET   67
20
21 int main(int argc, char** argv){
22     printf("Receiver Process\n");
23     maglog(2,0,"receiver",0,0,0,0); //trace process
24
25     /* create shared memory object */
26     int shared_mem_fd = shm_open("/awesome_filter_mem", O_RDWR | O_CREAT,
           S_IRUSR | S_IWUSR);
27     if(shared_mem_fd == -1){
28         printf("can't create shared memory\n");
29         return 1;
30     }
31
32     /* set size of shared memory object */
33     if(ftruncate(shared_mem_fd, SHM_SIZE) == -1){
34         printf("can not set size of shared memory\n");
35         return 1;
36     }
37
38     /* map shared memory object into address space */
39     void* shared_mem = mmap((void*)0xbfbf1000,
40                             SHM_SIZE,
41                             PROT_READ | PROT_WRITE,
42                             MAP_SHARED,
```

```
43                              shared_mem_fd,
44                              0);
45      if(shared_mem != (void*)-1){
46          /* do work */
47          char*  buffer = (char*)shared_mem + BUFFER_OFFSET;
48          volatile char*  flag = (char*)shared_mem + FLAG_OFFSET;
49          volatile int* ret_ptr = (volatile int*)((char*)shared_mem +
                ADDR_OFFSET);
50
51          /* open socket */
52          struct sockaddr_in sockAddr;
53          int socket_fd = socket(PF_INET, SOCK_STREAM, 0);
54
55          if(socket_fd == -1){
56              printf("Cannot create socket\n");
57              exit(1);
58          }
59
60          memset(&sockAddr, 0, sizeof(sockAddr));
61
62          sockAddr.sin_family = AF_INET;
63          sockAddr.sin_port = htons(1100);
64          sockAddr.sin_addr.s_addr = INADDR_ANY;
65
66          if(bind(socket_fd, (struct sockaddr*)&sockAddr, sizeof(sockAddr)) ==
                -1){
67              printf("Cannot bind socket\n");
68              close(socket_fd);
69              exit(1);
70          }
71
72          if(listen(socket_fd, 10) == -1){
73              printf("Cannot listen on socket\n");
74              close(socket_fd);
75              exit(1);
76          }
77
78          *flag = 0;
79          *ret_ptr = 0;
80
81          maglog(1,0,ret_ptr,0,0,0,0); //trace memory addr
82
83          int connect_fd;
84          char input[255];
85          while(1){
86
87              connect_fd = accept(socket_fd, NULL, NULL);
88              if(!(connect_fd > 0)){
89                  printf("Error accepting connection\n");
90                  close(socket_fd);
91                  break;
92              }
93
94              int len = read(connect_fd,(void*)input,254);
95              if(0 == len){
```

```
96                  close(connect_fd);
97                  continue;
98              }
99              input[len] = 0;
100
101             if(*flag == 0 && *ret_ptr == 0){
102                 int offset = 0;
103                 char* src = (char*)input;
104                 while(*(src+offset) != 0){
105                     *(buffer + offset) = *(src + offset);
106                     offset++;
107                 }
108                 *(buffer + offset) = *(src + offset);
109                 *flag = 1;
110             }
111             while(1){
112                 if(*flag == 1 && *ret_ptr != 0){
113                     char* new_buffer = shared_mem + *ret_ptr;
114                     int len = 0;
115                     char output[255];
116                     for(;*new_buffer != 0; len++, new_buffer++){
117                         output[len] = *new_buffer;
118                     }
119                     output[len] = 0;
120                     write(connect_fd, shared_mem + *ret_ptr, len+1);
121                     *flag = 0;
122                     *ret_ptr = 0;
123                     break;
124                 }
125             }
126             close(connect_fd);
127         }
128
129         /* close socket */
130         close(socket_fd);
131     }else{
132         printf("can not map shared memory\n");
133     }
134
135     /* close file pointer */
136     close(shared_mem_fd);
137
138     /* unlink shared memory object */
139     if(shm_unlink("/awesome_filter_mem") == -1)
140         printf("can not unlink shared memory\n");
141
142     /* unmap shared memory */
143     if(munmap(shared_mem,SHM_SIZE) == -1)
144         printf("can not unmap shared memory\n");
145     return 0;
146 }
```

Listing B.1: receiver.c

```
1 #include <stdio.h>
```

```
2  #include <sys/mman.h>
3  #include <sys/stat.h>
4  #include <sys/types.h>
5  #include <fcntl.h>
6  #include <unistd.h>
7  #include <string.h>
8
9  #include "maglog.h"
10
11 #define SHM_SIZE 4096
12
13 #define BUFFER_OFFSET 0
14 #define FLAG_OFFSET   64
15 #define ADDR_OFFSET   67
16 #define NEW_BUFFER_OFFSET    100
17
18 int main(int argc, char** argv){
19
20     printf("Filter Process\n");
21     maglog(2,1,"filter",0,0,0,0); //trace process
22
23     /* create shared memory object */
24     int shared_mem_fd = shm_open("/awesome_filter_mem", O_RDWR | O_CREAT,
           S_IRUSR | S_IWUSR);
25     if(shared_mem_fd == -1){
26         printf("can't create shared memory\n");
27         return 1;
28     }
29
30     /* set size of shared memory object */
31     if(ftruncate(shared_mem_fd, SHM_SIZE) == -1){
32         printf("can not set size of shared memory\n");
33         return 1;
34     }
35
36     /* map shared memory object into address space */
37     void* shared_mem = mmap(NULL,
38                             SHM_SIZE,
39                             PROT_READ | PROT_WRITE,
40                             MAP_SHARED,
41                             shared_mem_fd,
42                             0);
43     if(shared_mem != (void*)-1){
44         /* do work */
45         char*  buffer = (char*)shared_mem + BUFFER_OFFSET;
46         volatile char*  flag = (char*)shared_mem + FLAG_OFFSET;
47         volatile int* ret_ptr = (volatile int*)((char*)shared_mem +
               ADDR_OFFSET);
48         char*  new_buffer = (char*)shared_mem + NEW_BUFFER_OFFSET;
49
50
51         while(1){
52             if(*flag == 1 && *ret_ptr == 0){
53                 int offset;
54                 for(offset = 0; *(buffer + offset) != 0; offset++){
```

```
55                      if(*(buffer+offset) == 'e')
56                          *(new_buffer+offset) = 'a';
57                      else
58                          *(new_buffer+offset) = *(buffer+offset);
59                  }
60                  *(new_buffer+offset) = 0;
61                  *ret_ptr = (int)((void*)new_buffer - (void*)shared_mem);
62              }
63              sleep(1);
64          }
65      }else{
66          printf("can not map shared memory\n");
67      }
68
69      /* close file pointer */
70      close(shared_mem_fd);
71
72      /* unlink shared memory object */
73      if(shm_unlink("/awesome_filter_mem") == -1)
74          printf("can not unlink shared memory\n");
75
76      /* unmap shared memory */
77      if(munmap(shared_mem,SHM_SIZE) == -1)
78          printf("can not unmap shared memory\n");
79      return 0;
80  }
```

Listing B.2: filter.c

Appendix C

Lua Segmentation Fault Investigation

```
1  dofile("dwarf.lua")
2
3  function maglog_handler(cpustate, major_idx, minor_idx, payload)
4      major = qemu.get_maglog_major(cpustate)
5      minor = qemu.get_maglog_minor(cpustate)
6
7      if major == 1 then --start tracing
8          io.stderr:write(string.format("tracking addr=%x\n", cpustate.edx))
9          qemu.dbg_start_tlb_tracing_phys(cpustate, cpustate.edx, cpustate.edx
               + 1)
10     elseif major == 2 then -- process tracking
11         if minor == 0 then
12             qemu.task_track(cpustate, "receiver")
13             io.stderr:write(string.format("tracking receiver task\n"))
14         elseif minor == 1 then
15             qemu.task_track(cpustate, "filter")
16             io.stderr:write(string.format("tracking filter task\n"))
17         end
18     elseif major == 3 then -- segmentation fault
19         io.stderr:write(string.format("segmentation fault: access to %x at
               eip=%x(%s) esp=%x\n",
20             cpustate.edx,
21             cpustate.ecx,
22             qemu.dwarf_find_line(dwarf, "receiver", cpustate.ecx),
23             cpustate.ebx))
24     end
25 end
26
27 function st_handler(cpustate, major_idx, minor_idx, payload)
28     io.stderr:write(string.format("store from %s at %s eip=%x retaddr=%x
           vaddr=%x paddr=%x ebp=%x\n",
29         qemu.task_query(cpustate),
30         qemu.dwarf_find_line(dwarf,qemu.task_query(cpustate),payload.t0),
31         cpustate.eip,
32         payload.t0,payload.t1,payload.t2,
33         cpustate.ebp))
34 end
35
36 qemu.register_event_handler(qemu.DBG_USER, maglog_handler)
37 qemu.register_event_handler(qemu.DBG_STORE_ACCESS, st_handler)
```