**Diplomarbeit**

# Remote Debugging via Firewire

Julian Stecklina

30. April 2009

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

| | |
|---|---|
| Betreuender Hochschullehrer: | Prof. Dr. rer. nat. Hermann Härtig |
| Betreuender Mitarbeiter: | Dipl.-Inf. Bernhard Kauer |

Technische Universität Dresden
Fakultät Informatik

## AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

| | |
|---|---|
| *Name des Studenten:* | **Julian Stecklina** |
| *Studiengang:* | Informatik |
| *Immatrikulationsnummer:* | 3014126 |
| *Thema:* | **Remote Debugging via Firewire** |

*Zielstellung:*

IEEE 1394, auch Firewire genannt, definiert ein weitverbreitetes serielles Bussystem, mit dem man Geräte und mehrere Rechner verbinden kann. Die dafür eingesetzten OHCI Controller unterstützen dabei einen Betriebsmodus, der direkten Zugriff auf den Hauptspeicher sowie das Auslösen von Interrupts ohne Beteiligung des Betriebssystemes erlaubt.

Gute Debugfähigkeiten von Kern und Anwendungen sind obligatorisch für das Minimieren des Entwicklungsaufwandes und die Akzeptanz eines Betriebssystems. Bisherige, speziell auf das entsprechende Betriebssystem zugeschnittene Debugger sind mit erheblichen Eingriffen in den Kern verbunden.

Ziel dieser Arbeit ist es, aufbauend auf GDB, eine Debugarchitektur zu entwickeln, die Remote Debugging über Firewire erlaubt. Besonderes Augenmerk ist auf möglichst minimale Änderungen im untersuchten System zu legen. Weiterhin sollte diese Architektur auch einfach in virtuellen Maschinen wie Qemu oder KVM implementierbar sein.

Um die Anwendbarkeit dieser Architektur unter Beweis zu stellen, ist ein Prototyp eines Debuggers für Nova zu entwickeln.

| | |
|---|---|
| *verantwortlicher Hochschullehrer:* | Prof. Dr. Hermann Härtig |
| *Betreuer:* | Dipl.-Inf. Bernhard Kauer |
| *Institut:* | Systemarchitektur |
| *Professur:* | Betriebssysteme |
| *Beginn:* | 01. 11. 2008 |
| *Einzureichen:* | 30. 04. 2009 |

Dresden, 30. 10. 2008

[illegible]
Unterschrift

## Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 30. April 2009

Julian Stecklina

## Abstract

Traditionally, the legacy RS-232 serial port has been used to connect a remote debugger to its target system. The debugged operating system had to be enhanced with support code to handle this connection. This thesis explores a novel approach to debugging that uses the Firewire bus not only as fast communication medium but also as remote manipulation tool. This approach promises to minimize support code in the debugged system by injecting the code needed for remote debugging at runtime.

Based on the GNU Debugger, a remote debugger for the NOVA microhypervisor has been implemented that fulfils these promises. The result is a fully functional debugger fit for everyday use. Furthermore, the bulk of its implementation is not specific to NOVA and can be quickly adapted to other kernels in need of a debugger.

# Contents

# List of Tables

# List of Figures

# 1 Introduction

*"There are two ways to write error-free programs; only the third one works."*
Alan J. Perlis

## 1.1 The Need for a Kernel Debugger

The kernel is one of the most critical parts in a reliable and secure system. A single bug in the kernel can potentially undermine both of these qualities, reliability and security. Yet despite decades of research in programming languages and methodology, there is still no silver bullet that lives up to its promise of advancing software engineering to a point, at which bug-free software becomes reality [10]. Thus every sufficiently complex program contains errors.

It is commonly known that the amount of errors in a program correlates to lines of code. By minimizing lines of code, the amount of—potentially disastrous—bugs can be reduced. The microkernel approach [23] is one way of achieving code reduction in the kernel.

Due to high complexity of systems code, a potential for errors remains even in microkernels and different methods can be used to spot them. One approach is to use formal verification to prove correctness against a specification. Although initial success is reported in the literature [19, 30, 36], a complete verification would encompass not only the kernel, but also the compiler and the processor to be truly meaningful.[1] Given the complexity of modern general-purpose microprocessors, a complete verification may not be practical.

Another approach is using a programming language for systems code that aids in the discovery of bugs at compile time. One recent proposal is the *BitC* language [29], which supports a rich type system that enables the programmer to express even complicated operations without resorting to type-unsafe code. While BitC looks promising, it has yet to spread beyond its origin, the EROS group. The vast majority of systems code is still implemented in unverified C or C++.

## 1.2 Systems Programming and Reusable Tools

Writing systems programs still remains an art of engineering. According to popular lore, a good workman—and, by extension, engineer—is known by his tools. The proverb probably predates the information age by a fair number of years, but its essence survives unchanged.

[1] Even a verified compiler can fail to meet its specification. This situation is illustrated best by "transitive" trojan horses [26]. A malicious compiler can substitute arbitrary code when compiling the verified compiler, resulting in a compromised binary [35]. The same reasoning applies to *buggy* compilers. A buggy compiler may introduce hard-to-find bugs in every binary program it produces. Processors are, of course, susceptible to the same problem.

The quality of the systems programmer's work is greatly influenced by the availability and quality of his tools.

Tools available to systems programmers are mostly written and maintained by themselves, yet time spent on such tools is mostly frowned upon by chief executives, as there is no tangible progress of the project. One consequence is that tools are largely neglected or considered a burden instead of being improved when they fail to meet expectations [15].

A quickly retargetable system debugger is such a tool with the potential of boosting productivity in a working or scientific environment in which experimental systems code is written to evaluate new ideas. Such projects commonly do not have the resources to develop a suitable debugger. Being able to use a proper debugging environment from the start of a project, can be a great asset to productivity.

But before developing this idea further, I want to shift the focus to Fiasco's debugging facilities to make another point.

## 1.3 Avoiding Stale Code

For a decade the Fiasco kernel [20], a second-generation microkernel, has been one of the research vehicles of the Operating Systems Group of Technische Universität Dresden. It contains *JDB*, a sophisticated in–kernel debugger.

| **Component** | **Lines of code** | **Percentage** |
|---|---:|---:|
| *JDB* | 20,614 | 23 % |
| *Support code* | 9,914 | 11 % |
| *JDB + Support code* | 30,528 | 34 % |
| *Fiasco total* | 88,215 | 100 % |

Table 1.1: The size of JDB, the kernel debugger of the Fiasco microkernel, compared to the size of Fiasco itself. The measurement includes machine–specific code for all platforms Fiasco supports.

JDB alone accounts for about one fourth of Fiasco's source code.[2] Adding libraries that are only used by JDB, the part of Fiasco devoted to debugging grows to one third. These libraries are used to process compressed data, disassemble instructions, and evaluate regular expressions.

A quick review of the code reveals that the compression library, an ancient version of *zlib*,[3] is vulnerable to specially crafted compressed data streams. These vulnerabilities are not security relevant. An attacker with access to the kernel debugger does not need to bother with security vulnerabilities.

However, this stale library and its long-fixed bugs hint that accumulating code in the kernel that does not contribute to its main functionality might be a problem in the long run. It would have been disastrous, if a future developer chose to use zlib's features to implement critical functionality.

[2] Code size measurements were generated using David A. Wheeler's 'SLOCCount'.

[3] *zlib* is a compression library that is used in many open-source projects. Its current version can be obtained from `http://www.zlib.net/`.

| Medium | Duration |
|---|---|
| *RS-232 (9600 Bit/s)* | 18 min |
| *RS-232 (115200 Bit/s)* | 91 s |
| *Firewire* | < 0.1 s |

Table 1.2: The theoretical duration of a one megabyte data transfer using the serial port (RS-232) compared to Firewire.

It is unlikely that the Fiasco project is going to remove JDB, which is otherwise a great debugger, but other kernel projects might want to avoid having to maintain a large debugger in the kernel source.

## 1.4 Remote Debugging

An alternative to using an in–kernel debugger is remote debugging using the PC's serial port with the GNU Debugger (GDB). Because of its support for a wide range of platforms and its free availability, GDB is favored by many developers [16].

By using remote debugging, a large debugger inside the kernel becomes superfluous. Instead, only a comparably small stub is needed to handle commands transmitted using the serial connection from the debugging host on which GDB is running. These commands range from simple, like reading and writing of memory and registers, to more complicated ones, like setting breakpoints or—in a multithreaded target—manipulating threads.

Adding rudimentary support for remote debugging to a kernel is simple [17]. Supporting larger parts of the protocol, however, is a quite involved task, because its semantics is poorly documented. Anecdotal evidence suggests that debugging stubs are implemented in an ad-hoc fashion. Their maintenance is largely ignored once they seem to work.

Additionally, the serial connection used for communication between stub and GDB is painfully slow for today's standards (Table 1.2). Remote debugging with GDB using the serial port is clearly not an optimal solution. But how can it be improved?

## 1.5 Enter: Firewire

During the last years, several operating systems have been adding support for the IEEE 1394 serial bus, commonly known as Firewire, as communication medium for remote debugging. Section 3.2 deals with them in more detail.

Suffice it to say that these implementations use Firewire because of its speed, but ignore its more intriguing features allowing the implementation of a debugger with *less to no support code in the target kernel at all* compared to classical remote debugging with the GNU Debugger. Exploring these features was one of the main incentives for this thesis.

## 1.6 Goals

The goal of this thesis is to devise and implement a remote system debugger based on the GDB that is able to perform basic debugging operations on the kernel of an operating system. That is, it should be able to:

- control execution of the system;
- inspect and manipulate data structures;
- set break- and watchpoints.

This debugger should require *considerably less changes* to the operating system kernel than remote debugging over a serial connection, thereby reducing the amount of code that has to be maintained inside the kernel. It should improve upon the interactive speed of remote debugging and, at the same time, provide *at least* the same features as remote debugging.

Lacking any kind of proper debugger, the NOVA microhypervisor [34] has been chosen as test case for this architecture.

## 1.7 Terminology

So far, I have relied on the implicit understanding of certain terms, but at this point it is necessary to agree on the concrete meaning of these terms to avoid confusion in later chapters.

The **target system** or simply **target** is the computer running a operating system that is to be debugged. The **target kernel** is the kernel of the operating system running on the target system. The kernel contains a **debugging stub**, which is the part of the kernel that had to be added to support a debugging framework. The **host system** or simply **host** is the computer running the debugger. The debugger will almost exclusively be the GNU Debugger in our case.

## 1.8 Organization of This Thesis

The next chapter gives the required technical background information. Its sections are intended to be self-contained. The impatient or already informed reader may skip it and refer to it as the need arises. Chapter 3 gives an overview about system debugging in general and introduces several debugging frameworks that already utilize Firewire. The generic design of the debugging architecture is presented in Chapter 4. It is complemented with the description of the implementation of the monitor and kernel stub for the NOVA microhypervisor as well as the Linux kernel in Chapter 5. In Chapter 6, I answer the question, how this architecture compares to classical remote debugging and whether the developed architecture performs well enough to be used for other purposes, such as profiling or tracing. Chapter 7 sketches the solutions of several open problems that can form the basis of future projects. The closing chapter, Chapter 8 summarizes this thesis and gives some final remarks.

# 2 Technical Background

This chapter starts with overviews of several technologies that are touched by this thesis. I will address the serial port, PCI, USB, and Firewire. Following that, I continue describing interrupt handling on the PC platform, which is particularly important to the discussion in Chapter 5.

## 2.1 The Serial Port

The serial port has been a standard external interface of the PC since the original IBM PC design introduced in 1981, but the underlying technology, RS-232, exists far longer.

It was a primary method of connecting external peripheral devices to the PC, but is today classified as 'legacy port' by hardware vendors and thus slowly disappearing. A reason for its obsolescence is its low speed, typically not more than 115,200 bit/s, which makes the serial port unsuitable for data–intensive applications.

The advantage of the serial port is its simplicity and its availability, which still makes it a viable option as communication medium for many applications.

## 2.2 The PCI Bus

The Peripheral Component Interconnect (PCI) bus, introduced in 1993, is the standard expansion bus system in the PC. Typically, PCI devices come in the form of expansion cards that can be plugged into a free PCI slot on a PC's motherboard.

PCI is used to connect a variety of different internal devices, such as graphics, network, and sound cards. It is also used to connect external buses, such as Firewire and USB, to the PC using adapter cards. Firewire controllers usually come in the form of a PCI device.

The PCI bus provides three address spaces to connected devices. Accesses to any of these address spaces are performed by `read` and `write` transactions on the PCI bus.

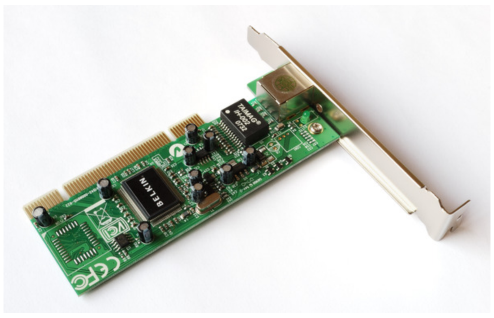


Figure 2.1: A typical PCI expansion card.[1]

[1] Image provided by Wikimedia Commons:
`http://commons.wikimedia.org/wiki/File:GB_Network_PCI_Card.jpg`

The *configuration space* is used by system software to automatically discover available PCI devices and configure them by assigning each PCI device windows of the *memory* and *input–output* address spaces. Communication with a device is then performed by accessing memory or input–output ports in these windows. Automatic configuration was a major contributor to the success of PCI.

Unless claimed by other devices, the PCI bus controller claims accesses to the memory address space and forwards them to the system's memory controller. This mechanism, which is called Direct Memory Access (DMA), allows PCI devices to access the system's main memory independently of the processor.

To learn more about the PCI bus, I recommend reading the PCI standard itself [25]. It includes an excellent, concise introductory chapter.

## 2.3 The Universal Serial Bus

The Universal Serial Bus (USB) is a serial bus standard meant to overcome the shortcomings of peripheral handling in early PC designs. Its major advantage is the replacement of a multitude of different connectors (PS/2, RS-232, . . . ) with one common connector used by a diverse set of peripherals, such as keyboards, mice, or printers.

Because of its single type of connector, automatic device configuration and hot plug support, USB has been very successful.

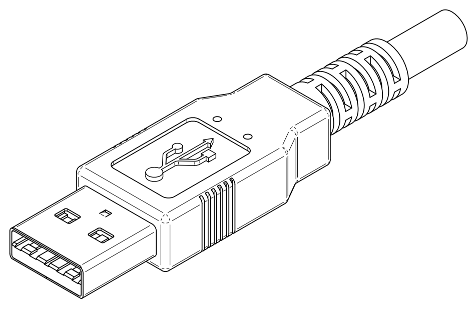


Figure 2.2: The USB Type A connector.[1]

USB is designed to reduce the cost of peripherals. It implements a star topology with the host computer at its center and follows a strict master-slave philosophy, in which *only the host can initiate transactions* on the bus and every connected device acts as a slave, which needs only a very simple (and cheap) controller. Not only does USB not support DMA, there is also no support for interrupts as well, requiring the host to periodically *poll* devices that are traditionally interrupt-driven, such as keyboards. A consequence of this approach is that direct peer–to–peer communication between peripherals is not possible.

Communicating with USB devices usually requires a complete USB software stack, which is a large and complicated piece of code. Communication via USB devices has therefore not been a prominent choice for debuggers. To alleviate this problem the USB Debug Port has been introduced.

The debug port is a special USB device that allows to connect two USB–capable PCs, between which it provides a bidirectional communication channel. Support for the USB Debug Port is an optional feature of USB host controllers [4]. If a host controller supports it, system software is able to communicate using the debug port without a complete USB

[1] Image provided by Wikimedia Commons:
`http://commons.wikimedia.org/wiki/File:USB_Type_A_Plug_BW.svg`

| | RS-232/Serial Port | USB | Firewire |
|---|---|---|---|
| Introduction | 1969 | 1996 | 1995 |
| Prevalent use | peripherals, terminal | peripherals | multimedia, industry |
| Speed | 50–115,200 Bit/s[1] | 1.5–480 MBit/s | 400–3200 MBit/s |
| Connectible devices | 2 | 128 | 63 |
| Topology | point–to–point | star | tree |

Table 2.1: Comparison of the serial port (RS-232), USB, and Firewire. The USB and Firewire figures are from current revisions of the respective standards, USB 2.0 and IEEE 1394-2008.

stack. The specification limits the data rate in this mode to eight bytes per time slot on the bus, which limits its overall bandwidth to a maximum of 0.5 MBit/s.

## 2.4 The Firewire Bus

The IEEE 1394 standard describes a serial bus capable of high transfer speeds (up to 3.2 GBit/s in later revisions of the standard [5]) as well as isochronous real-time data transfers common in audio and video processing. Several vendor-specific names for IEEE 1394 have been introduced since its inception, *Firewire* being the most popular one.

It is competing with USB, but has a slightly different scope. IEEE 1394 (Firewire) arose out of the need to replace SCSI for the connection of external devices, such as disks, scanners, and printers. Using SCSI for external devices is largely obsolete today; it has been superseded by IEEE 1394, USB, and eSATA in virtually all but high-end applications.

Owing to the SCSI heritage, the Firewire bus is at its core quite different from USB; it treats connected devices more like a network, in which every participant, called *node*, has equal abilities.

To consider a small example, a single Firewire bus can be used to connect two PCs, a disk, a camera, and a printer. Each device acts as a node in the network and can communicate with every other device. The two PCs can exchange IP packets over the bus, while one of them concurrently accesses the disk, while the camera sends a picture to the printer.

This admittedly synthetic example shows that the Firewire bus represents not only a means for the connection of peripheral devices, but a general networking mechanism that is open to completely unforeseen applications.

### 2.4.1 Bus Topology and Addressing

The Firewire bus consists of nodes, each with a unique 64-bit identifier, the Globally Unique Identifier (GUID). Nodes are connected in a peer-to-peer fashion to form a tree. Multiple buses may be connected by bus bridges to form larger networks. There is no cycle detection in IEEE 1394; networks have to be noncyclic.

Nodes in the network are addressed by NodeIDs that are constructed during the initial bus enumeration and resemble the position of the node on the bus. NodeIDs can change

[1] Higher speeds are possible, but uncommon on the PC.

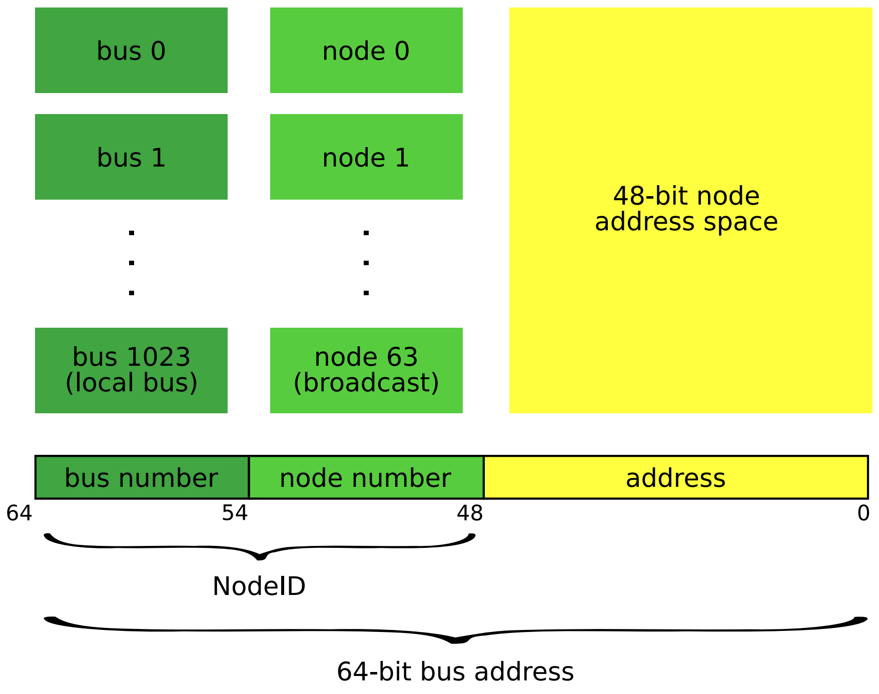


Figure 2.3: System software sees the Firewire bus as 64-bit address space, which is statically split into address spaces for every node on the bus. The bus and node numbers form the NodeID, which can change when the topology of the bus changes.

on each bus reset. Because bus resets are caused, for example, when the topology changes, applications should use GUIDs to differentiate nodes from each other.

Each node provides an 48-bit address space. Together with the 16-bit NodeID (formed by the combined bus number and node number), all nodes on the network form a 64 bit address space, which is depicted in Figure 2.3.

### 2.4.2 Transactions

IEEE 1394 defines a protocol stack with three layers:

- The *Physical Layer* arbitrates bus access and translates link layer requests into electrical signals,
- the *Link Layer* provides an acknowledged datagram service and handles isochronous data transfers, and
- the *Transaction Layer* that provides the three primitives to access the bus address space: `read`, `write`, and `lock`.

`read` and `write` transactions are used to read and write portions of the bus address space. The `lock` transaction performs an atomic *Compare–and–swap* operation that can

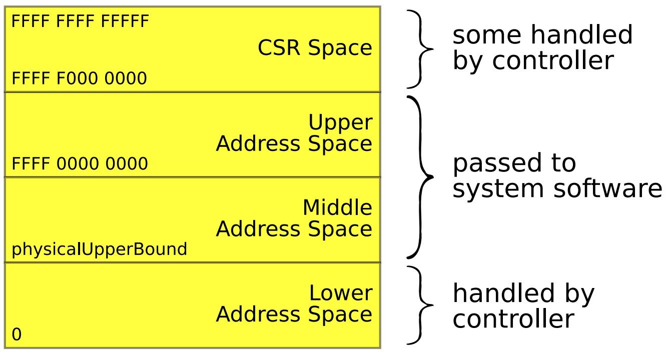


Figure 2.4: An Open Host Controller logically distinguishes four regions in a node's address space. Transactions touching the Lower Address Space are handled by the controller itself without informing system software. `physicalUpperBound` is a hardware register of the controller that can be programmed by system software. Transactions touching the Middle and Upper Address Space are passed to system software. Accesses to the CSR Space are special and are handled in part by the controller.

be used to safely update remote data structures in the presence of concurrent access from different nodes on the bus.

`read`, `write`, and `lock` are called *asynchronous* transactions to differentiate them from isochronous streams, which provide time guarantees. Because streams play no role in this work, I always refer to asynchronous transactions, when I use phrases, such as 'Firewire bus transaction' or 'transaction'.

## 2.5 The Open Host Controller Interface

The IEEE 1394 standard only defines the wire protocol necessary for interoperability between Firewire nodes. The Open Host Controller Interface (OHCI) complements IEEE 1394 with a standardized way of accessing the bus from software by specifying an interface of a IEEE 1394 controller (the *Open Host Controller*). Virtually all PCI Firewire controllers on the PC market try to be OHCI compliant, with only a few rare exceptions [31].

The OHCI implements the link layer of the IEEE 1394 bus with additional support for the transaction layer. Because PCI devices are the typical realization of OHCI, the specification also includes a chapter about implementing an Open Host Controller on the PCI bus [2].

### 2.5.1 Automatic Handling of Incoming Transactions

The OHCI is a complex interface and most details are not interesting in the scope of this work. One feature in particular, however, stands out.

Incoming asynchronous transactions are handled by an Open Host Controller according to the memory region they address (see Figure 2.4). Specifically, transactions touching the Lower Address Space are passed directly to the PCI bus avoiding the need for the system's

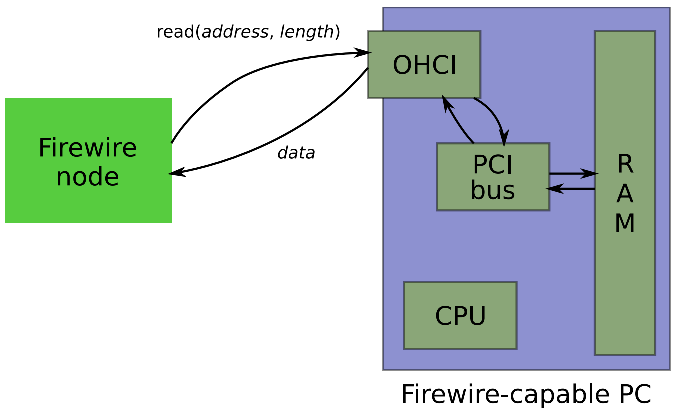


Figure 2.5: Remote memory access using the Firewire bus. A node on the bus can read and write arbitrary parts of the host's memory without its processor being involved, if the Open Host Controller allows physical requests from this node.

processor and operating system to be involved, effectively opening the PCI memory address space to all nodes on the Firewire bus. The intended use of this mode of operation is to implement fast and low-latency communication protocols.

The border between Low and Middle Address Space is configured by setting the `physicalUpperBound` hardware register of the Open Host Controller. By setting it to its maximum value, the Middle Address Space shrinks to zero and almost the complete bus address space of the node is mapped to physical memory. Accesses to the Lower Address Space are called *physical requests*.

### 2.5.2 Security Issues

Making a node's physical memory accessible from all other nodes on the bus presents a gaping security hole [7, 9] allowing arbitrary nodes on the bus to obtain and manipulate sensitive information (e.g., cryptographic keys or passwords). The OHCI includes several mechanisms to mitigate that.

The most effective solution is to disable the Lower Address Space by setting the `physicalUpperBound` register to 0. No transaction (except accesses to the CSR space) is then automatically handled in hardware and the operating system can inspect each transaction. This might incur a considerable performance overhead. Another way is to selectively enable access for trusted nodes via the `AsynchronousRequestFilter` and `PhysicalRequestFilter` hardware registers. In recent PCs, the Input–Output Memory Mapping Unit (IOMMU) can also be used to effectively constrain the Open Host Controller's capabilities, but IOMMUs are not widely available, yet.

From a security point, a Firewire port that is accessible to attackers should be considered as dangerous as an accessible PCI port. The difference is that custom "attack" PCI cards [11] are expensive, but the dangerous potential of Firewire ports can be exploited by comparably cheap off–the–shelf devices as shown by [14].

## 2.6 An Overview of Interrupt Handling on the PC

Several design decisions in the following chapter are based on the low-level characteristics of the PC platform. This short discourse can by no means even begin to cover the PC platform as a whole, so, on the following pages, I will concentrate on key aspects of the hardware platform that are crucial to the rest of this thesis. These aspects will mainly revolve around interrupt handling on the PC and debugging features of the Intel processor architecture.

For a complete in–depth description of the processor architecture, you are referred to Intel's set of processor manuals [22]. Specifically, the first manual "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture" is a good entry point for learning about the Intel architecture.

### 2.6.1 PCI Message–Signaled Interrupts

If a device on the PCI bus needs attention by the operating system, it signals an interrupt. The traditional way to do this, is for the device to assert one of the interrupt lines of the PCI bus. Because there are only a small number of interrupt lines (typically four), interrupt lines must be shared, if the number of devices exceeds the number of available interrupt lines.

Having multiple devices assigned a single interrupt line has several disadvantages [3]: If one of these devices signals an interrupt, the corresponding interrupt handler of the operating system has to check *all* devices sharing the interrupt, possibly creating a considerable interrupt latency.

PCI host controllers conforming to newer PCI specifications offer an alternative to interrupt lines, the message-signaled interrupt (MSI). Instead of asserting an interrupt line, a device performs a `write` transaction into a special part of the PCI memory address space to express its call for attention. PCI Express, the successor to the PCI bus, uses message-signaled interrupts exclusively.

The processor to which the interrupt is to be delivered (if there are more than one) and the type of interrupt to be generated are encoded in the address and data of the `write` transaction.

### 2.6.2 Interrupt Handling on the Processor

If an interrupt arrives at the processor, the corresponding handler procedure must be executed. For that purpose, the processor maintains an Interrupt Descriptor Table (IDT) with one descriptor for each possible interrupt. There are three different types of descriptors that each cause the processor to take a different action, when the corresponding interrupt is signaled.

- *Trap Gate* and *Interrupt Gate* descriptors cause the CPU to execute a handler procedure specified by a pointer in the descriptor. The handler procedure is executed similar to a normal function call by preserving the state necessary to return to the interrupted code on the current stack.[1] The difference between Trap and Interrupt Gates is that the latter disables interrupts upon entering the handler.

- *Task Gate* descriptors cause the CPU to perform a *task switch* to a task that is supposed to handle the interrupt.

Task switching is a seldom used feature of the Intel architecture. When the processor performs a task switch, it stores its current state in the Task State Segment (TSS) of the current task and loads the contents of the new task's TSS. I will refer to this as *hardware* task switching to distinguish it from manual context switching done by the operating system.

Most systems programmers frown upon using hardware task switching, because it is perceived to be slow compared to manual context switching.

### 2.6.3 Debugging Features

The Intel processor architecture includes support for single-stepping and breakpoints. The core of this functionality is provided by the `TF` (trap) flag in the `eflags` register, the breakpoint instruction `int 3`, and the debug registers `dr0`-`dr7`.

Debug events are reported as either debug exception (#DB) or breakpoint exception (#BP), depending on the type of event, and can be handled by the operating system exactly as other exceptions and interrupts.

The TF flag is used to implement single-stepping. When this flag is set, a debug exception (#DB) is generated after each completed instruction. The processor clears the flag before entering the debug exception handler.

To set instruction breakpoints, the `int 3` instruction can be inserted by a debugger (or compiler) at the designated place in the application's code. Because this involves changing the code of the application by the debugger, these breakpoints are referred to as *software breakpoints*. When the CPU encounters this instruction, a breakpoint exception (#BP) is triggered and control is transferred to the corresponding handler procedure.

Another way to set breakpoints involves the debug registers. These breakpoints are versatile, but only four can be active at any given time. These breakpoints are referred to as *hardware breakpoints*. To set a hardware breakpoint, the address must be written into one of the lower four debug registers. Flags in `dr7` are then used to enable these breakpoints and to specify their type. There are several different types of hardware breakpoints. The most important are:

- *Instruction breakpoints* cause the processor to generate a debug event when the instruction at the specified address is to be executed.

- *Read breakpoints* cause a debug event when the address is about to be read as data.

- *Read/Write breakpoints* cause a debug event when the address is used in either a read or write operation.

[1] Actually, the stack is switched when the interrupt handler is to execute in another privilege level, but that opens a whole new can of worms that adds little to the point I am trying to make, so I hope the informed reader forgives me this slight inaccuracy.

These debug events are delivered to the operating system as #DB exceptions. What type of event triggered the exception, can be identified by interpreting `dr6`.

To handle code that itself intents to modify the debug registers, the processor can be instructed to generate a debug event on every access of the debug registers. A debugger may then choose to emulate these accesses.

## 2.7 Summary

This chapter gave a brief overviews of the serial port, USB, PCI and continued with a more in–depth discussion of the Firewire bus. Following that, I described message-signaled interrupts and interrupt handling on the PC. The last part of this chapter focused on hardware support for debugging on the 32-bit Intel Architecture.

The next chapter concentrates on the approaches to remote system debugging and already existing debuggers using Firewire.

# 3 Related Work

*"If you're not part of the solution, you're part of the precipitate."*

Henry J. Tillman

The previous chapter discussed details of the Firewire bus and gave an overview about handling hardware interrupts on the PC platform.

This chapter discusses existing system debugging approaches. I introduce debuggers that have support for Firewire and argue how this relates to recent security threats that involve the Firewire bus. The ideas presented in this part give rise to the design of a remote kernel debugger that is presented in the next chapter.

## 3.1 Approaches to System Debugging

Traditionally, there have been several approaches to kernel and system debugging. This section presents the most popular ones with the intention to show the gap this work tries to fill.

### 3.1.1 In–Circuit Emulator

One possibility to perform system debugging is to use an in–circuit emulator (ICE). ICEs come in many different types, such as bond–out processors, which expose additional pins to control execution and to allow access to internal signals and registers.

In any case, the ICE provides a controlling interface to another computer, the debugging host, running the ICE control software. In essence, it is a debugger built into the hardware.

ICEs are used, for example, in debugging embedded systems that lack resources or output devices to support a software debugger. ICEs are also available for commodity hardware, where they provide features that are out–of–reach for software debuggers, such as breakpoints on certain bus events [12].

Intel has a long history of in–circuit emulators that goes back to at least the Intel 80286 [13]. The 80286's ICE is actually quite interesting in the scope of this work. To generate a breakpoint on a specific event, the ICE control software on the host would monitor the 80286's bus, until a matching event occured. At this point, it signals the processor to stop execution and dump its internal state into a predefined memory location that the control software can inspect. The rough idea is similar to the design I propose in the next chapter.

ICEs are in many aspects superior to other choices, but their price is prohibitive for many applications.

### 3.1.2 In–Kernel Debugger

A less expensive choice is to implement debugging functionality in the operating system kernel itself. An example of this, the Fiasco microkernel's debugger JDB, has already been discussed in the introduction. Another example is FreeBSD's kernel debugger.

The great advantage of this approach is that the debugger has direct access to all kernel data structures. Debugging commands, such as "send a signal to process *n*", are easily implemented.

One drawback in my eyes is that the line between kernel functionality and debugging functionality can become fuzzy. The debugger becomes an integral part of the kernel and most parts of it cannot be reused. It also adds code to the kernel, such as instruction decoding and parsing of debug information, that does not contribute to the main functionality of the kernel and contributes a maintenance burden.

Lastly, writing an in–kernel debugger is no small accomplishment and costs valuable development time not every project can spare.

### 3.1.3 Remote Debugging

In addition to debugging applications that run on the same computer, the GNU Debugger can talk across a serial line or TCP connection to a debugging stub on another computer. This mode of operation is most useful when debugging operating system kernels and embedded systems where the stub is integrated in the kernel itself. Its key problems have already been outlined in the introduction:

While remote debugging is a workable alternative to an in–kernel debugger, it still adds considerable code to the kernel, because implementing GDB's remote protocol is not an easy task. Additionally, it relies on RS-232, a slow legacy communication channel that is not available in many recent PCs.

Using a debugger outside the kernel incurs loss of functionality. Certain operations, such as the "send a signal to process *n*" example above, become difficult, because they have to be expressed in the debugger's command language. If the command language is insufficient to express the desired action, the user is out of luck.

### 3.1.4 System Emulator with Debugging Extensions

Virtual machines or emulated systems can be used for debugging as well. For example, the system emulator QEMU [8] implements a GDB stub in its emulation layer, which can be used to attach the GNU Debugger to an unmodified operating system. Thus, QEMU acts as a kind of primitive ICE in its emulated system.

This approach adds to the advantages of remote debugging that no kernel stub has to be developed at all, the slow serial connection to the target is replaced with a fast TCP connection from GDB to QEMU, and test machines can be comfortably started as QEMU processes.

Unfortunately, this solution is not applicable, when hardware devices are needed that the system emulator cannot emulate. Additionally, having an operating system work in an emulated system does not imply that it necessarily works on a real system.

The purpose of this work is to mitigate these last two disadvantages by providing a comparable debugging experience on real hardware.

## 3.2 Debuggers Using Firewire

The idea of using Firewire as a kernel debugging aid is not new. There are currently at least three major operating systems that provide support for Firewire as a communication medium for remote system debugging. Except for the FreeBSD implementation, documentation is either sparse or outdated.

### 3.2.1 Windows' Kernel Debugger

One of the earliest implementations is in the kernel debugger of Microsoft Windows XP. Available documentation about the actual mode of operation is scarce, but indicates that the debugger on the host writes debug commands directly into the target's memory to be processed by the stub [18].

### 3.2.2 FreeBSD's dcons

The `dcons` driver [27] is a console driver for the FreeBSD operating system that provides two in–memory communication ports. One is used for normal console in- and output, the other is a communication medium for the GNU Debugger.

These ports are implemented as memory buffers that can be read by either FreeBSD's kernel memory interface on the same machine or from a remote machine using Firewire. This interface is abstracted by the `dconschat` utility providing a uniform interface two both access modes. `dconschat` is able to open a TCP/IP server to allow connections from GDB or allow the user to access the remote system's console. The latter option also makes it possible to use FreeBSD's in–kernel debugger over Firewire.

`dcons` is fully integrated into FreeBSD since FreeBSD 6 and is actively supported.

### 3.2.3 Linux' firescope and fireproxy

A slightly different approach [1] has been implemented for the Linux kernel. It is split into two tools:

- *Firescope* is a tool that reads arbitrary memory of a target system that runs Linux and is connected via Firewire. It is specially geared towards reading the remote system's kernel `printk` buffer, which contains the latest part of the kernel log.

- *Fireproxy* implements a GNU Debugger backend, but is very limited in the functionality it provides. It offers only remote memory access to the target, which allows to inspect kernel data structures. Controlling execution, inspecting registers, inserting breakpoints, and all other features of the GNU Debugger are not supported.

Development seems to have ceased in 2006 with both tools being left in a functional, yet experimental development stage.

### 3.2.4 Limitations of These Approaches

The FreeBSD implementation uses Firewire as a fast communication medium for its GDB stub and its in–kernel debugger. From a pragmatic standpoint, `dcons` simply removes the need for a serial cable. It does not remove the need for either the in–kernel debugger nor the GDB stub in the FreeBSD kernel.

Firescope and Fireproxy exploit Firewire's remote memory access feature, but do not go beyond that. Firescope provides introspection into a running kernel but no means to manipulate its execution. Although Fireproxy uses the GNU Debugger as a frontend, it does not offer functionality that is commonly associated with debugging.

### 3.2.5 Going One Step Further

The feature of directly reading and writing a computer's memory via Firewire has sparked a series of creative hacks with the aim of circumventing security features of the operating system [14].

An example is to use remote memory access via Firewire to change the user ID of a running process. After all, it is just a field in a kernel data structure. A normal process started by an unprivileged user can thus be elevated to superuser privileges.

Going one step further, it is possible to inject arbitrary code by overwriting code that is surely executed, such as the operating system's interrupt handlers. Although this offers great possibility for mischief, it is also a intriguing way to monitor and control execution of a computer from a remote system.

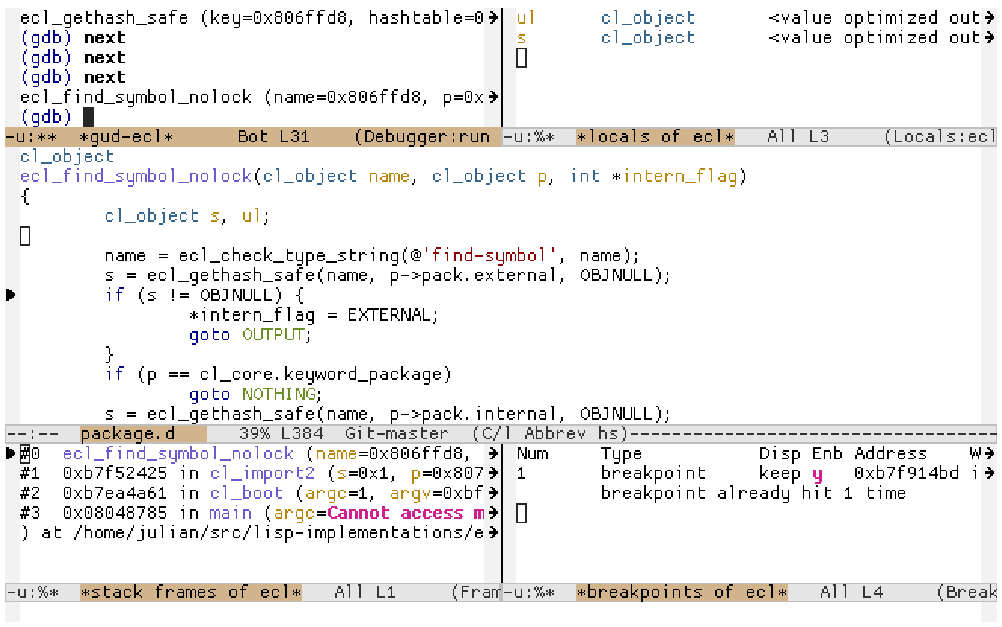
## 3.3 The GNU Debugger

GDB, the GNU Debugger, is the de-facto standard application debugger on most Linux-based and BSD-derived operating systems on various hardware platforms. It is a source-level debugger with excellent support for C, C++, Objective C, and to different degrees Fortran, Java, Pascal, Modula-2, and Ada [32]. GDB is free software protected by the GNU Public License.

GDB itself features only a command-line interface, which can be unwieldy for complex debugging tasks. For a more pleasant debugging experience the programmer can choose between a multitude of frontends to GDB.

Personally, I appreciate GUD, the Grand Unified Debugger, which is a frontend integrated into Emacs, as well as DDD, the Data Display Debugger. The latter is a highly useful tool to graphically explore the structure of data in a running program [38]. Screenshots of example sessions can be seen in figures 3.1 and 3.2.

Although being written for application development, GDB has made its way into kernel debugging as well. Together with being free software, this makes GDB an obvious choice for this project.



```
ecl_gethash_safe (key=0x806ffd8, hashtable=0→
(gdb) next
(gdb) next
(gdb) next
ecl_find_symbol_nolock (name=0x806ffd8, p=0x→
(gdb) 
-u:**  *gud-ecl*       Bot L31     (Debugger:run
```

```
ul       cl_object       <value optimized out→
s        cl_object       <value optimized out→

-u:%*  *locals of ecl*    All L3       (Locals:ecl
```

```
cl_object
ecl_find_symbol_nolock(cl_object name, cl_object p, int *intern_flag)
{
        cl_object s, ul;

        name = ecl_check_type_string(@'find-symbol', name);
        s = ecl_gethash_safe(name, p->pack.external, OBJNULL);
        if (s != OBJNULL) {
                *intern_flag = EXTERNAL;
                goto OUTPUT;
        }
        if (p == cl_core.keyword_package)
                goto NOTHING;
        s = ecl_gethash_safe(name, p->pack.internal, OBJNULL);
--:--  package.d      39% L384  Git-master  (C/l Abbrev hs)-----------------------------------
```

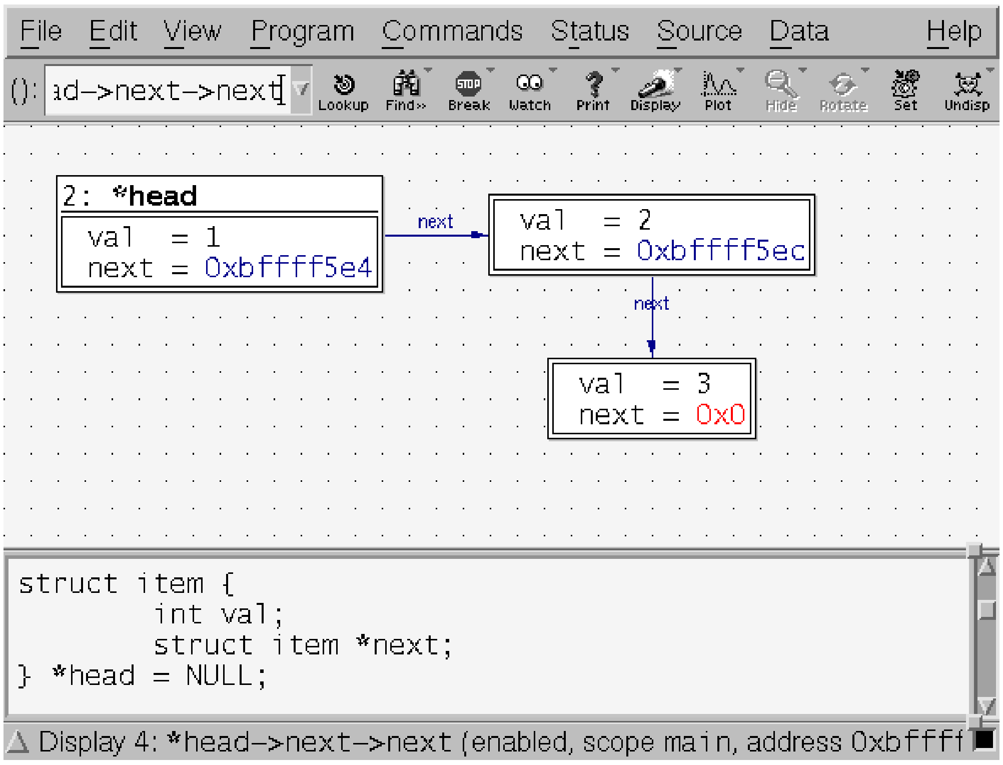
```
#0  ecl_find_symbol_nolock (name=0x806ffd8, →
#1  0xb7f52425 in cl_import2 (s=0x1, p=0x807→
#2  0xb7ea4a61 in cl_boot (argc=1, argv=0xbf→
#3  0x08048785 in main (argc=Cannot access m→
) at /home/julian/src/lisp-implementations/e→

-u:%*  *stack frames of ecl*    All L1      (Fram
```

```
Num     Type           Disp Enb Address    W→
1       breakpoint     keep y   0xb7f914bd i→
        breakpoint already hit 1 time

-u:%*  *breakpoints of ecl*    All L4      (Break
```

Figure 3.1: A screenshot of a source-level debugging session using GUD, an Emacs-based frontend for GDB.



```
struct item {
        int val;
        struct item *next;
} *head = NULL;
```

Display 4: *head->next->next (enabled, scope main, address 0xbffff

Figure 3.2: A linked list in DDD. The screenshot demonstrates the advanced data exploration tools of DDD.

## 3.4 Summary

This chapter discussed the common approaches to system debugging. It continued describing uses of Firewire for remote debugging: FreeBSD's `dcons` driver uses Firewire as a fast communication medium to an existing in–kernel debugger. The Linux tools `firescope` and `fireproxy` use Firewire's remote memory access feature to inspect a running Linux kernel, but do not implement a full debugger. Other, more creative approaches that use Firewire to inject code and manipulate control flow are reported from the security community.

The next chapter will outline the design of a system debugger that combines both approaches to yield an uninstrusive remote system debugger that can be quickly retargeted to new systems.

# 4 A Framework in Three Parts

*"If we complicate things, they get less simple."*

Professor at Cambridge University

So far I have only described technical details and the few existing related projects. I will now turn to the actual design of an unintrusive remote system debugger.

## 4.1 Initial Considerations

There are certain ideas raised in the introduction that are worth repeating. I strongly believe that it is detrimental to the health of an operating system kernel to keep code in it that is maintained as a second–class citizen. A in–kernel debugger or debugging stub is usually such code. System programmers usually have better things to do than maintaining their tools.

A way to avoid this situation, at least regarding debuggers, is to move as much of the debugging infrastructure out of the kernel to be maintained separately. In effect, this makes the debugger and the kernel or system it is intended to debug separate projects. Bad quality on the part of the debugger might then frustrate developers, but has no direct implications on the quality of the kernel itself.

The question that immediately follows is: What do we really need inside a kernel to debug it remotely? The answer largely depends on what features the hardware provides. Since the olden days, we have had RS-232, a simple communication mechanism that was present on nearly every PC and is only now slowly disappearing.

RS-232 provides a bidirectional communication medium with minimal effort on the kernel-side. Operations that need to be performed on the target system have to be encoded as data packets and sent across the wire. A debugging stub on the target then decodes them, performs the desired action and returns a result. Basically, this is GDB remote debugging in a nutshell. Even if one replaces RS-232 with another communication medium, such as the USB Debug Port[6], the concept remains the same. Something inside the target kernel has to interpret these packets.

Taking a step back, let us consider what the essentials for remote debugging are. A remote debugger has to:

- *inspect* and *modify* the state of the target, which is divided into its memory and processor–internal state, such as registers, and
- *control* its execution.

Table 4.1 lists which of these essential operations each of RS-232, the USB Debug Port, Firewire, and PCI can perform *without cooperation* from the target system. RS-232 and the

| | **RS-232** | **USB Debug Port** | **Firewire** | **PCI** |
|---|---|---|---|---|
| *CPU register access* | no | no | no | no |
| *Memory access* | no | no | **yes** | **yes** |
| *Execution control* | no | no | **interrupts** | **interrupts** |

Table 4.1: Support,for remote debugging operations without support from the target. Interrupt injection via Firewire is not an obvious feature and is discussed in Section 5.2.1.

USB Debug Port do not provide any of the necessary features and have to completely rely on support code in the target kernel.

An interesting category is formed by Firewire and PCI. Both can:

- cache-coherently access the target's memory and
- signal interrupts.

However, a debugger cannot be built without support from the target, based solely on these capabilities. The target kernel has to be enhanced with a tiny debugging stub. But, given these capabilities, what really needs to be in the target kernel to facilitate remote debugging? Surprisingly, only two things are needed. The first one is a way to safely stop and resume execution of the system upon reception of an interrupt. The second one is a way to expose processor–internal state in memory, where it can be easily read remotely.

It is important to mention that such a kernel stub is completely agnostic about the way the debugging host performs remote memory access and interrupt execution. The stub does not need to drive any hardware devices. Given the fact that the debugging host can remotely access memory, the stub can even be injected into the target system at runtime.

On the following pages, I present a design of a remote system debugger based on these premises.

The implementation presented in the next chapter focuses—for practical purposes—on Firewire, but a custom–built PCI card or enhanced virtual machine monitor would work just as well, as would any device that can inject interrupts and remotely access the target's memory in a cache-coherent way.

## 4.2 The Bird's Perspective

Let me give you a quick overview of the design I propose, before I discuss the individual parts. At the top of the debugging architecture is the GNU Debugger. Reusing the GNU Debugger frees us from the daunting task of disassembling object code, parsing debug information, writing an user interface and generally from reinventing the wheel.

The GNU Debugger can be used unchanged in this framework. Any recent version is sufficient when it proves stable enough[1].

[1] The GNU Debugger has been found to be quite instable in certain situations. Most notably when debugging functions written in Assembler and assembled using gas with debugging symbols enabled.

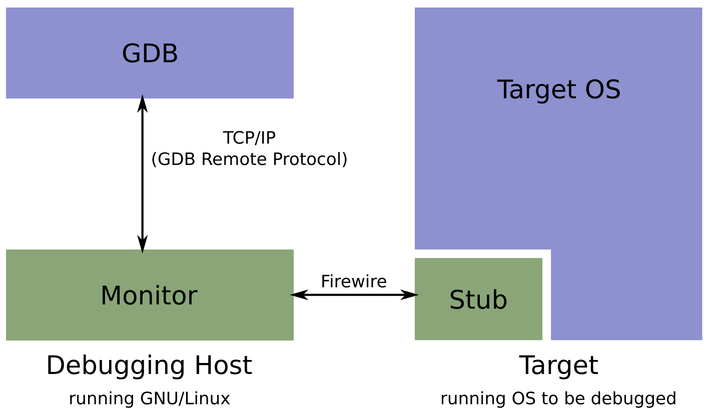


Figure 4.1: The three parts of the debugger framework: The GNU Debugger, the monitor, and a kernel stub in the target kernel. Firewire is used for communication.

Normally, the GNU Debugger uses a serial connection directly to the target machine. In our case, it connects to the *monitor*, another application running on the host, via a TCP/IP connection. The monitor's purpose is to interpret requests coming from the GNU Debugger and execute them using one of its backends, such as the Firewire backend, which I describe in more detail below. On the target system, a tiny kernel stub is needed to expose processor–internal state.

## 4.3 The Monitor

The monitor acts as a server for the GDB remote protocol. It translates requests coming from the GNU Debugger via the TCP/IP connection into operations on its target.

The design of the monitor focuses on reusability. It is implemented in three parts, which are shown in Figure 4.2.

### 4.3.1 The GDB Server

The first major part of the monitor is the GDB server, which implements a large part of the GNU Debugger's remote protocol. Details of the protocol implementation are addressed in Section 5.3.1 on page 34. Each request from GDB is executed by operations on the controller.

### 4.3.2 The Controller

The controller is the heart of the monitor. It abstracts the services provided by the target kernel stub into an easily usable interface. The major tasks that the controller abstracts are:

- Starting and stopping of the target using interrupts,
- translating virtual into physical addresses,

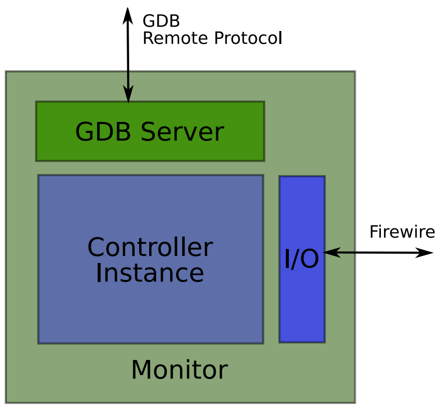


Figure 4.2: The monitor consists of three parts: The GDB server implements the GDB remote protocol and manages communication with the GNU Debugger. The controller abstracts the capabilities of the target kernel stub. Lastly, an input–output layer hides the intricacies of the underlying communication medium, such as Firewire.

- accessing the target's register set.

The controller is intended to serve as a middleware between GDB requests and the kernel stub, but its interface is completely agnostic about the GNU Debugger and can be used not only as backend for other debuggers, but also for other types of applications as well. In Section 6.2, I describe a primitive statistical profiler, which is implemented using the controller interface, to assess the cost of monitor interrupts.

### 4.3.3 The Input–Output Abstraction

The low-level communication with the target is encapsulated to allow for different kinds of communication.

While the intended medium is a Firewire bus connected to a physical target system, it is also possible to implement this interface for a virtualized target running in a process emulator or a virtual machine. As most virtualization systems already facilitate debugging, the latter is mostly meant as a debugging vehicle for the debugging architecture itself.

By abstracting from the concrete type of communication to the target system, the controller and the GDB server can be agnostic about the communication medium and are presented with a clear interface that mirrors closely what is possible using Firewire, as it is the most restrictive medium.

An implementation of an input–output backend for the controller has to support only three methods:

- The `read` and `write` methods are used to read arbitrary chunks of physical memory on the target;
- the `nmi` method is used to interrupt the target with an non-maskable interrupt (NMI).

The Firewire implementation of these input–output methods is obviously the most straight-forward one, as the `read` and `write` operations map directly to Firewire primitives and `nmi` is implemented as a special write operation, as explained in Section 5.2.1.

Support for targets running in virtual machines is a bit more complicated, because at the time of writing no widely–used virtual machine emulates a Firewire device.[2] Fortunately, the same operations that are possible via Firewire can be directly implemented in the virtual machine monitor. An implementation for QEMU exists and is described in Section 5.3.3.

## 4.4 The Target Kernel Stub

Although much of the debugging functionality is moved outside the target kernel, a small kernel stub is still needed, but its complexity is extremely low. This kernel stub consists of an interrupt handler to handle interrupts from the monitor and is responsible for exposing processor-internal state and safely stopping and resuming the target kernel.

For the sake of simplicity, only uniprocessor systems are supported. Section 7.2 shows how this design can be extended towards multiprocessor systems.

### 4.4.1 Contexts in the Target Kernel

The functionality provided by the kernel stub is implemented using transitions between contexts. Contexts are execution states that are switched upon receiving an interrupt from the monitor or upon debug events. These contexts are completely orthogonal to what the underlying operating system may call context, process, or thread. The implementation of contexts is machine–specific. An implementation for the 32-bit Intel architecture is described in the next chapter.

The target kernel executes at any one time in one of three contexts.

- The `run` context is the one in which the kernel and user-mode applications are executed. The target system spends most of its time in it.
- The `interrupt` context is entered whenever the target receives an interrupt from the monitor.
- The `halt` context is entered either when a debug event occurs on the target system or directly when the `interrupt` context is told to stop execution.

The `run` context is special, because, whenever it is left, the current state of the processor must be stored in the command space, which I will discuss to below. When control returns to the `run` context, its state is restored from the command space.

### 4.4.2 Communicating with the Stub

As the monitor can only communicate using interrupts and memory accesses, a certain area of the target's memory, the *command space*, is used for communication. The command space serves two purposes.

[2] Incidentally, shortly before finishing this work, an effort to implement a virtual Open Host Controller for QEMU has been started. At the time you are reading this, QEMU might be worth a second look.

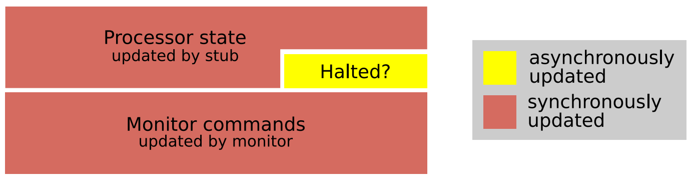


Figure 4.3: The command space is logically divided into two parts: Processor state and status reported by the target and commands sent by the host.

First, it holds a status field and the processor state of the `run` context, which the monitor can modify via remote memory accesses. This state is updated synchronously on every interruption by the stub. Its second purpose is to communicate the desired action that the monitor wants to perform to the target. There are only two different types of actions required.

- The `stop` command is used to force execution from the `interrupt` context into the `halt` context.
- The `resume` command is used to resume normal execution in the `run` context.

On each interrupt, the stub updates the status field in the command space to indicate whether:

- the target was interrupted executing normally, that is in the `run` context,
- the target was sleeping in the `halt` context, or
- the target is halted due to an unrecoverable exception. This state is not recoverable and the target can most likely not be resumed.

The stub, as presented so far, suffices to implement a fully functional debugger, but the monitor would have to periodically send interrupts to recognize when the target has entered the `halt` context due to a debug event. Polling with interrupts is undesirable, because it wastes processor time on the target even when no interaction with the debugger is needed.

By adding a field to the command space that is asynchronously updated by the stub whenever the target system enters the `halt` state, the monitor can learn this information by a simple memory access and defer interrupting the target, until it is certain that it is stopped.

### 4.4.3 Not Using a Ring Buffer

A major goal for the target kernel stub has been its utter simplicity. The current design uses an approach that reminds of a state machine with clear transitional events and states. The core logic can easily be expressed in one page of code and is robust even in the face of a crashing monitor. A new monitor can be started and can reattach to the running target without problems.

Using a ring buffer for communication was consciously avoided, because it would have required a producer–consumer approach, which is deemed unnecessary. A ring buffer requires careful synchronization and would have increased the complexity of the stub considerably.

The current design rests on the assumption that communication with the stub via interrupts is done infrequently. Operations, such as repeatedly stepping one instruction, violate this assumption and as consequence perform badly compared to other operations that require less interruptions (see Section 6.1.2).

For applications of the framework that suffer from this design decision it might be worthwhile to reconsider the decision not to use ring buffers.

### 4.4.4 Halting and resuming the target

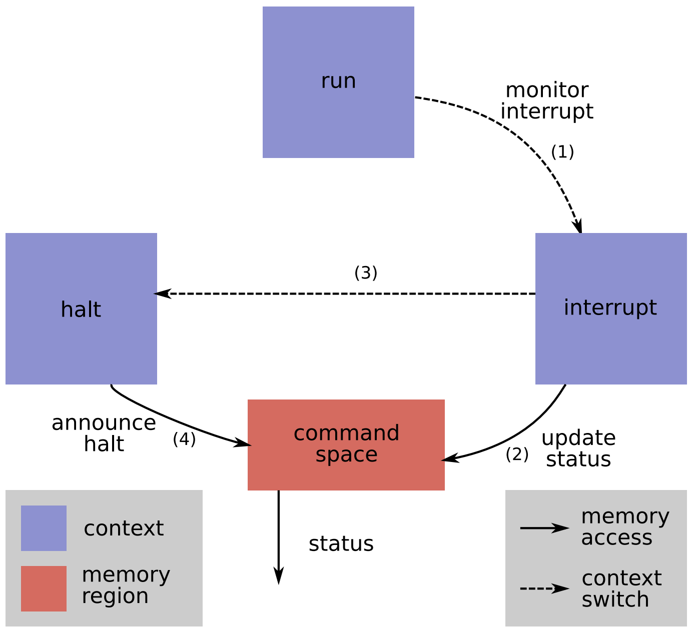


Figure 4.4: Halting the target is accomplished by forcing execution into the `halt` context.

To safely inspect the state of the target, it must be stopped. Figure 4.4 depicts the sequence of events that are initiated by the monitor to stop the target and obtain its state.

The monitor first issues the command to force the target into the `halt` context by writing it into the command space and then triggering an interrupt in the target. This causes the `interrupt` context to become active. The handler then updates the processor state in the command space and switches to the `halt` context where execution is caught in an idle loop.

When the monitor is instructed to resume execution, a similar chain of events is started that is illustrated in Figure 4.5:

The monitor tells the stub to return to the `run` context in an analogous way: It includes the command in the command space and signals an interrupt. The stub then updates the command space and carries out the command, which is returning control to the `run` context.

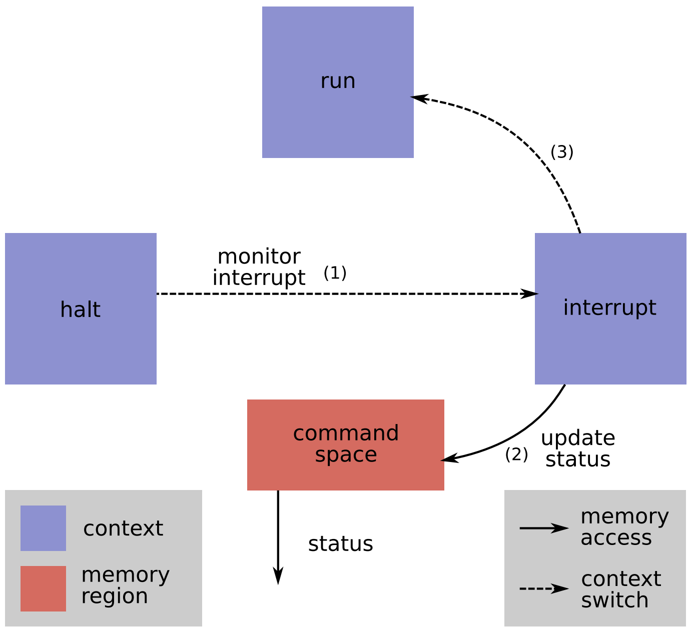


Figure 4.5: Resuming execution in the target after being halted is done by forcing execution into the `run` context.

### 4.4.5 Handling Debug Events

So far we have seen how to manually halt and resume execution of the target system. More interesting and useful are debug events such as breakpoints, watchpoints or processor exceptions.

If the processor architecture supports hardware-assisted debugging, the necessary interface is exposed via the command space. If the processor architecture lacks support for it, the monitor can inject code to achieve the same without involvement of the stub. How breakpoint injection is handled on the Intel architecture is explained in Section 5.2.4.

Debug events are handled quite different from manually halting the target. Upon encountering a debug event, the target switches to the `halt` context. Inside that context, it announces the occurrence of a debug event in the command space.

While the target system is running in the `run` context, the monitor polls the status field in the command space. When the command space indicates that the target has entered `halt`, the monitor signals an interrupt.

The interrupt causes a switch to the `interrupt` context, where the complete status information in the command space is updated. When the update is done, the target switches back to `halt` and sleeps again. The monitor can now pass control to the GNU Debugger, which figures out what caused the debug event on its own. Resuming the execution works exactly as explained in the previous section.

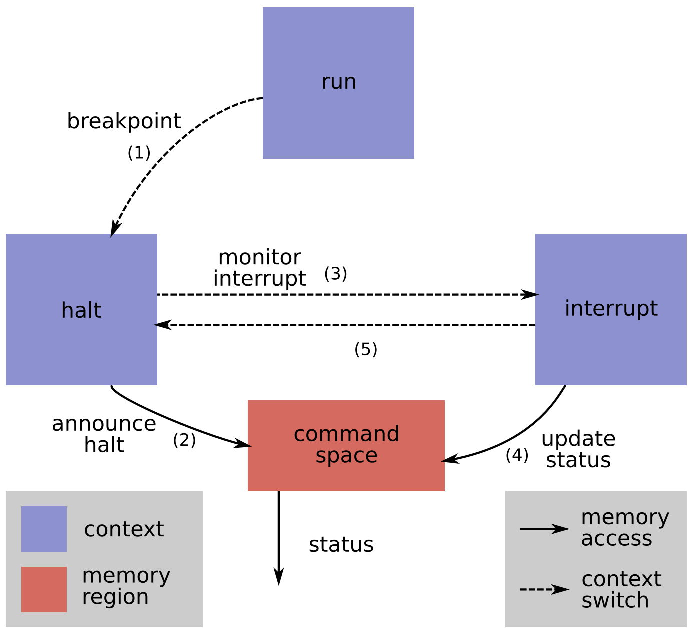


Figure 4.6: When a debug event occurs, the target switches to the `halt` context. The monitor detects this and extracts information via a roundtrip to the `interrupt` context.

## 4.5 Summary

This chapter presented an argument for a different approach to remote system debugging that relies on features provided by the Firewire bus. Building upon the GNU Debugger, a design was outlined that separates the debugger and the corresponding kernel stub from the system that is to be debugged. The next chapter will deal with implementation issues on the 32-bit Intel architecture.

# 5 Implementation

*“What I cannot create, I do not understand. ”*

Richard Feynman

In the preceding chapter, the generic design of a minimally intrusive system debugger was outlined. This chapter starts with a discussion of how the design maps onto real hardware and then go on to deal with the concrete implementation for two different kernels on the 32-bit Intel architecture.

## 5.1 The NOVA Microhypervisor

The main implementation and experimentation was done with the NOVA microhypervisor [33]. Its development has been started as part of the ROBIN project, whose goal was to develop a small, robust, and *open* operating system architecture for security–sensitive applications. The original project homepage can be found at `http://robin.tudos.org/`.

NOVA is an experimental hypervisor that allows running legacy guest operating systems alongside a multi–server user environment. A major design goal is to provide isolation for legacy systems and applications in order to limit the impact of a compromised component.

For the purposes of this thesis, NOVA makes an excellent testbed, because it is *small* and therefore easy to understand and modify. Furthermore, it currently implements neither a GDB stub nor an in–kernel debugger.

## 5.2 Mapping the Design onto the 32-bit Intel Architecture

There are several challenges with mapping the design, outlined in the previous chapter, onto commodity PC hardware, most of which deal with the kernel stub. The monitor itself arrives mostly as a consequence of design choices in the kernel stub and the perceived weirdness of the GNU Debugger’s remote protocol. I will return to the protocol in Section 5.3 on page 34.

### 5.2.1 Remotely Injecting Interrupts Using Firewire

I have been vague so far on how it is possible to remotely signal interrupts via Firewire. The key is creative use of features of the PCI bus and the OHCI.[1]

Let me recapitulate some facts from Chapter 2. Section 2.5.1 mentioned that it is possible to program the OHCI in a way that allows it to pass Firewire bus requests directly to the PCI

[1] I give full credit for this discovery to my thesis mentor, Bernhard Kauer. Without it, this work would not have been possible in this form.

bus. Section 2.6.1 explained that modern PCI devices can use special `write` transactions on the PCI bus to signal interrupts, so called message-signaled interrupts. The type of interrupt is encoded in this transaction.

Even if it seems almost ludicrous on first sight, it is possible to send a `write` request to a particular Firewire node, which is handed to the PCI bus by the node's OHCI, where it is interpreted as a message-signaled interrupt. All that is needed on the target, is a chipset that supports message-signaled interrupts and an enabled Advanced Programmable Interrupt Controller (APIC).

The format of message-signaled interrupts is quite elaborate [21], but for our limited purposes it suffices to interrupt the first physical processor with an NMI, which is done by simply writing the value 0x40 at the address 0xFEEE0000 in a node's Firewire bus address space. There is nothing more to it.

### 5.2.2 Handling Interrupts in the Kernel Stub

The design rests on the assumptions that the debugging host can signal interrupts and read arbitrary physical memory of the target. The implementation uses NMIs, because they cannot (easily) be masked by the operating system.

NMIs have traditionally been used for unrecoverable error conditions, such as failing hardware components, that have to be detected regardless in which state the operating system is. Nonetheless, safely handling NMIs on the 32-bit Intel architecture proves to be a challenge. There are basically two choices:

By using an *interrupt gate*—see Section 2.6.2 on page 11 for a brief discussion of interrupt handling—the operating system's current kernel stack is used to handle the interrupt. If the kernel has code paths without a valid stack, a delicate recovery in the handler is needed to avoid memory corruption.

The Linux kernel uses interrupt gates for NMI handling and has to recover, for example, when being interrupted directly after a system call. The recovery process employed by Linux to fix the smashed stack is intimately tied to code that might have been executing, when the interrupt was triggered, and is likely to frighten even experienced system programmers.

The second choice is to use a *task gate* to handle the NMI handler. On first sight, this solves all the aforementioned problems, because the CPU saves its state prior to the interrupt to memory and loads a new state including a new stack pointer for the handler procedure. Curiously, the task switch does not save two critical system registers (`cr3`, `ldtr`) whose contents get destroyed during the process.[2]

To safely resume execution, the interrupt handler must recover both registers prior to returning control. Fortunately, there are usually few places in a kernel that modify these registers. It is straight-forward either to deduce their contents from kernel data structures, when they are lost, or to keep shadow copies in memory.

The latter choice, using a task gate, is the one currently taken by the NOVA microhypervisor. However, NOVA goes only half the way and makes no attempt to restore the destroyed state, because it interprets NMIs as unrecoverable failure and deliberately panics.

[2] These registers are termed 'static fields' in the TSS by Intel's system programming guide [21]. Static fields are restored when the processor switches to the corresponding task, but never written by the processor.

I favored the task gate approach to NMI handling for the debugging stub, because it has two pleasant properties. Using hardware task switching reduces kernel–specific fix-up code to restoring the content of the two destroyed registers, instead of a complicated restoration procedure, as employed by Linux. The second property is that the processor frees the kernel stub from the job of storing the processor state, because it is already written to the TSS.

### 5.2.3 Representing Contexts

Because of the decision to handle NMIs with a separate hardware task, I decided to map contexts to hardware tasks. Each context—`run`, `interrupt`, and `halt`—is represented by its own hardware task.

Architectures that do not provide hardware-assisted task switching obviously need a different approach. The AMD64 architecture in its native 64-bit mode is one example of such an architecture. A possible implementation for AMD64 is outlined in Section 7.1.

### 5.2.4 Processor-Specific Parts of The Command Space

The command space is divided into processor state reported by the target and commands sent by the debugging host.

#### 5.2.4.1 Commands

Three commands have been implemented. These are `stop` and `resume`, which were already explained in Section 4.4.2.

To implement hardware breakpoints, the processor debug register's need to be written by the monitor. These registers are currently not contained in the generic processor–state in the command space, because reading and writing them is slow compared to accessing the processor's general–purpose registers. A special command `write_dr` was implemented for the purpose of updating the debug registers, which is only used when the user sets hardware breakpoints in the GNU Debugger.

During development, the stub had support for other commands, such as reading the debug registers. In theory, the contents of the debug registers indicate the type of debug event that took place. However, there seems to be no way to communicate that information to GDB, so support for this functionality has been dropped.

#### 5.2.4.2 Synchronization and Robustness

When the monitor sends an interrupt to the target, it has to wait until the interrupt handler on the target system has finished updating the command space.

The current method uses a counter in the command space that is incremented by the handler, whenever it completes handling a monitor interrupt. The monitor polls this counter. When it increases, the monitor knows that it can safely access the command space again.

Additional fields have been introduced to the command space to increase robustness. At the start of the command space is a magic value. If the command space is aligned on a page boundary (the default), the monitor can quickly scan the target's memory to find it and does not have to rely on the user or the kernel binary to find it.

A simple check sum is used in the command space to make sure that neither the monitor nor the kernel stub operate on a corrupt command space. While this should never happen, during development it *did* uncover several bugs that would otherwise been hard to spot.

## 5.3 Implementing the Monitor

The gateway between GDB and the target kernel (stub) is the monitor that, as explained earlier in Section 4.3, consists of

- A server for the GDB Remote Protocol,
- the controller, which is an abstraction for the services provided by the target kernel stub, and
- an input–output abstraction.

Of these, the first two are dependent on the target processor architecture and deserve special attention. In the current state of this debugger, neither is dependent on the specific target kernel.

### 5.3.1 The GDB Server

The monitor implements a server for the GDB Remote Protocol. A GDB session can be attached to the monitor by using the `target remote` command from GDB's command line. This command establishes a TCP/IP connection to the monitor. Using this connection GDB issues commands, which the monitor answers. There is only limited support for asynchronous communication.

The protocol is packet-oriented and uses in its original form only the 7-bit ASCII character set. It is to large extents human-readable. As the protocol was originally only used for serial connections, it includes features to detect transmission errors and to retransmit broken packets.

Actions taken by the user are translated by GDB to one or several commands. Each command is sent as a separate packet. After a response packet is received, the next packet is sent. Except for a small subset of required commands, a server can choose which commands it answers.

I started with implementing the required set of commands that include reading and writing processor registers and memory as well as basic control of the target's execution. Afterwards, I added support for the extended breakpoint interface to allow the user to set hardware breakpoints from within GDB.

Each command from GDB is executed by operations on the controller. The interface provided by the controller is explained in the next section.

Certain points of the protocol are ambiguously specified or implemented by GDB in a bogus way. A lot of frustration in the early stages of the server was caused by spurious packet acknowledgements sent by GDB. These spurious acknowledgements are now silently ignored, apparently without ill effects.

Another example is the format of the commands to read and write processor registers. The protocol specification only includes the details for the MIPS architecture. To learn the exact layout for other architectures one has to consult GDB's source code. While digging through source code is not especially difficult, it is certainly distracting.

Additionally, GDB proves to be unstable in many situations. In my personal opinion, GDB and its documentation are in questionable shape for an application that had over two decades to mature.

### 5.3.2 The Interface Provided by the Controller

The controller presents the capabilities of the target kernel stub as a set of methods. Each of these methods is implemented as operations on an input–output backend.

Its main function is to hide low-level synchronization and data organization from the GDB server code. For example, the `interrupt` method, which interrupts the target, takes as parameter the actions to be taken, writes any altered processor state back to the target, does the actual interrupt, and reads the state of the target. The controller is also able to resolve virtual to physical addresses by using information from the command space and traversing the target's page tables.

The separation between GDB server and controller has been chosen to allow reusing much of the code, when this debugging approach is ported to another debugger. In this case, the GDB server has to be replaced, but the controller can be used unchanged.

### 5.3.3 The QEMU Input–Output Backend

QEMU is an open source processor emulator that can be seen as a hosted virtual machine monitor. It runs as a normal application on any GNU/Linux system and thus makes a convenient testbed for development.

Writing an interface from scratch that would expose the necessary operations that are also available via Firewire seemed excessive after inspecting the QEMU source code. It already includes a complete GDB stub that can be used to debug virtual machines. Problematic is that the stub handles only accesses to virtual memory and internally resolves them to physical addresses. Handling different address spaces in the target operating system is thus complicated. In addition the set of operations offered by QEMU's GDB stub differs from the set my monitor offers, creating a different and potentially confusing debugging experience for the developer.

I concluded that the best way to proceed was to use the existing GDB stub as interface to QEMU. Several changes had to be made to support the monitor's input–output interface:

- Memory access via the GDB stub has been changed to accept physical instead of virtual addresses.
- QEMU lacked a way to emit NMIs from the GDB stub. A generic change has been made that allows to issue arbitrary commands (including NMI requests) to be sent to QEMU via the GDB stub.

The whole patch to QEMU consists of 18 added and 15 removed lines of code. One downside of this approach is that the monitor has to implement a way to pretend to be GDB. Fortunately, the additional code is also quite small, because a major part of the GDB stub code, used by the monitor to answer queries from the real GDB, can be reused.

The major stumbling block has been the crude implementation of the protocol by QEMU, which, for example, sends certain response multiple times. A lot of time was spent working around these idiosyncrasies.

Another downside is that every communication with QEMU stops the virtual machine. It has to be explicitly restarted by the monitor afterwards. The stopping and restarting is the major reason why the QEMU backend performs significantly worse than a real target connected via Firewire. One approach at alleviating this slowdown is discussed in Section 5.6.

## 5.4 Integration into the NOVA Microhypervisor

Integration into NOVA went particularly smooth, because it has been the primary development vehicle. NOVA lends itself well to experimentation, due to its small size and clean code [34].

There are currently two different kernel stubs for the NOVA hypervisor. The initial implementation was developed as part of NOVA. A second implementation can be injected via Firewire and requires almost no changes to NOVA itself.

### 5.4.1 The Stub as Kernel Extension

The first C++ implementation of the kernel stub adds about 300 source lines of code (SLOC) to the NOVA hypervisor. Half of this code is the NMI handler and the other half is contributed by code that sets up task segments and interrupt descriptors.

The added code neither changes NOVA's architecture nor adds any code in the normal control paths after the initial bootstrap phase. The maintenance cost of this code is expected to be negligible.

### 5.4.2 Injecting the Payload

Prompted by the author of NOVA, I investigated, if the kernel stub has to be maintained in NOVA's source code at all. This gave rise to a second implementation of the kernel stub that is implemented as a separate binary that can be injected into a running NOVA system.

This implementation consists of about 300 SLOC as well, but written in assembly instead of C++. This "payload" is complemented by a small amount of code in the monitor to do the actual injection.

The injection is accomplished by writing the payload into free kernel memory and changing the handler of, for example, the timer interrupt to point to the payload's initialization code. When the next timer interrupt is delivered the payload's initialization code is executed, which installs its own interrupt handlers for handling NMIs and debug events. Finally, the original timer interrupt handler is restored and executed. Instead of the timer interrupt handler, any kernel function that is regularly executed can be used as an entry point.

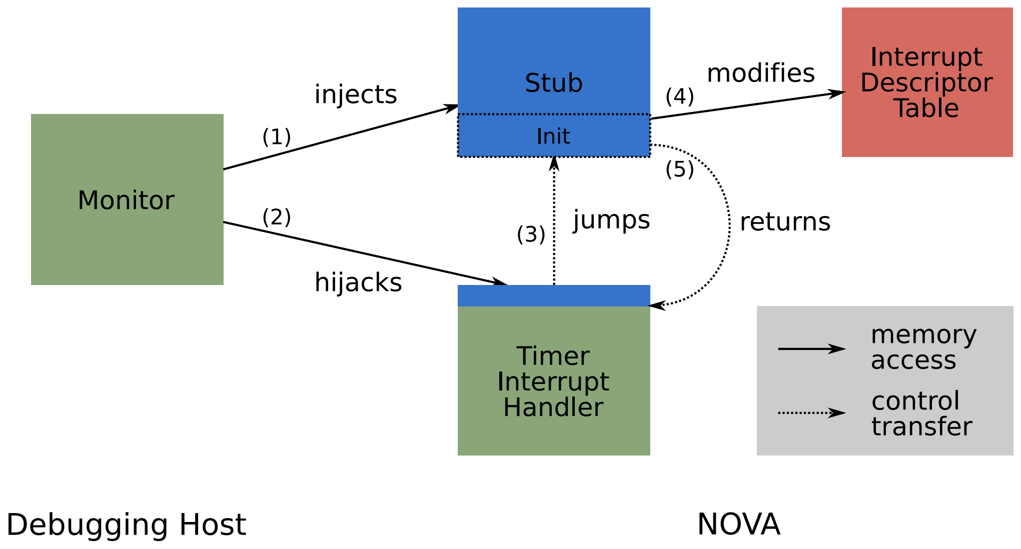


Figure 5.1: Injecting the kernel stub into NOVA. The monitor injects the stub into NOVA's kernel memory and changes the timer interrupt handler to execute the stub's initialization code. On the next timer interrupt, the initialization code is run. It adds the required hooks in the IDT. Then it restores the original timer interrupt handler and jumps back to it. At this point the stub is operational.

With an arbitrarily complex injection procedure, no changes to the kernel would be necessary. The current implementation cheats in two ways to simplify the injection procedure:

- By modifying the linker script of NOVA, it is assured that there actually is free space in kernel memory.
- The Global Descriptor Table (GDT) NOVA sets up is extended to include two unused entries, which can be used by the injected kernel stub.

With these simplifications, which required six lines of code to implement, the injection procedure is unproblematic.

A problem with the current implementation is that the injected NMI handler has to restore `cr3` after the task switch. The code in NOVA that would allow the stub to do this, is implemented in the form of C++ inline functions, which cannot be called from the stub.

The current solution is to manually use the *objdump* tool, a part of the *GNU binutils*,[3] to discover the in–memory layout of the involved kernel data structures. With this knowledge the recovery can be implemented in assembly language. This procedure is cumbersome and has to be repeated whenever the layout of these data structures changes.

It is possible to automate this procedure and generate the appropriate code by interpreting debug information contained in the kernel binary, but this is not implemented yet.

[3] `http://www.gnu.org/software/binutils/`

## 5.5 Integration into the Linux Kernel

As a proof-of-concept, the kernel stub was ported to the Linux kernel. Because `cr3` and `ldtr`—registers that are destroyed on a task switch—are not easily recoverable from Linux kernel data, the Linux kernel has been changed to keep shadow copies of these registers in memory. These shadow copies are then be used by the kernel stub to recover these registers. Because of this change, it has not been possible to implement the kernel stub as loadable kernel module.

Linux kernel preemption [28] has been disabled, because the debugger in its current form does not support multiple threads of execution well.

Support for the Linux kernel has been primarily written to prove that it is trivial to support new kernels. Without any prior exposure to Linux kernel programming, I have been able to port the kernel stub in about three days: Two days were spent analyzing the parts of the Linux kernel that are touched by the debugger. Another day was needed to do the actual implementation. Someone more familiar with the inner workings of the Linux kernel would most likely complete the port in a fraction of this time.

## 5.6 Performance Improvements

The initial interactive experience, especially with the QEMU backend, was perceived as poor. As experimentation with QEMU has been an important feature, improvement was necessary.

The major problem is that every communication with QEMU is costly. When printing data structures, GDB makes a lot of small read requests that each translate to a single request to QEMU, retrieving memory in larger chunks and caching it seemed like a good way to significantly reduce communication with QEMU. While the user is interacting with GDB, the target system resides in the `halt` context and no memory operations take place. Memory chunks that are retrieved from the target can therefore be cached until the target resumes execution.

The memory cache that has been implemented operates as write–through cache. It widens every memory request to multiples of its block size (currently 64 byte) in anticipation of accesses to nearby data. The cache is flushed when the target is instructed to continue execution.

This optimization had a huge positive impact on the interactive performance when dealing with a target running inside QEMU.

## 5.7 Programming the OHCI

The work of programming the OHCI has been integrated in a small program named *Morbo* that runs before the target kernel.

In its current form Morbo is a Multiboot-compliant kernel that locates an OHCI-compatible controller on the PCI bus and initializes it to allow requests to the complete memory address space of the target (see Section 2.5.1). To allow the monitor to issue interrupts via MSIs, it also makes sure the APIC is enabled, which is required for MSIs to function. After hardware initialization is completed it loads a Multiboot-compliant target kernel and executes it.

Using an external program to initialize the Firewire controller, is unproblematic as long as the booted kernel does not interfere by

- disabling the controller,
- disabling physical requests, or
- limiting the controllers capabilities in any other way, for example by using an IOMMU.

Currently, NOVA does not implement support for OHCI and leaves the controller alone. If someone implements an OHCI driver for NOVA, it must not perform any of these activities to avoid interfering with the debugger.

Morbo is not needed to boot Linux, because the default Firewire stack in Linux initializes the OHCI in the same way.

## 5.8 Summary

This chapter explained how the abstract design presented in the previous chapter maps to real hardware. It continued with details on the implementation for the NOVA microhypervisor and the proof-of-concept implementation for the Linux kernel. Certain interesting aspects of the implementation were presented, such as memory caching and programming of the OHCI.

The next chapter will evaluate performance and functional aspects of the implementation.

# 6 Evaluation

*“The wonder of systems codes is that they work at all[. . . ]”*

Jonathan Shapiro

The preceding chapter focused on implementation details of the debugging architecture. This chapter compares the implementation to classical remote debugging using RS-232 serial connections in terms of speed. Afterwards, to assess the underlying architecture’s performance characteristics and applicability to other domains, I discuss a simple statistical profiler, which utilizes the lower-levels of the implementation. But before I present numbers, let me revisit the goals stated in the introduction.

## 6.1 Revisiting the Goals

At the beginning of this thesis, I stated several goals for my debugger. The first part of these goals are to develop a system debugger based on the GNU Debugger that is able to:

- control execution of the system;
- inspect and manipulate data structures;
- set break- and watchpoints.

These goals have been achieved with the implementation of a working GDB backend. The debugger offers an equal set of features compared to traditional remote debugging over a RS-232 serial connection, as described in Section 3.1.3.

Doing just as well as the “old” solution would not be an achievement in itself. The second part of the goals is essential. I stated that this new remote debugging architecture should require *considerably less changes* to the operating system kernel that is to be debugged, while improving significantly on the interactive speed compared to remote debugging via serial connections.

### 6.1.1 Code Reduction in the Kernel

Table 6.1 compares the size of Fiasco’s GDB stub code to the amount of code that I needed to add to Nova and Linux to make them compatible with my architecture. Roughly speaking, the stubs needed for this debugging architecture are about an order of magnitude smaller than Fiasco’s GDB stub. Basically, the stub in my approach revolves around an NMI handler whose source code fits on a single page.

With extra effort, as described earlier in Section 5.4.2, it is possible to decouple the stub from the target kernel source almost completely. This allows the stub to be maintained

| Kernel | Lines of code |
|---|---|
| *Fiasco's GDB stub* | 1219 |
| *Linux' GDB stub* | 1389 |
| *Nova monitor stub* | 144 |
| *Linux monitor stub* | 196 |
| *Nova injection support* | 6 |

Table 6.1: Comparison of kernel code devoted to the debugger stub. The lines of code for Fiasco do only include the parts of the stub used on IA-32. The Linux GDB stub code also only includes IA-32 support. The GDB numbers should be considered conservative, because they only include source files that are as a whole dedicated to the GDB stub. The real numbers are likely higher.

*outside* of the kernel source code, effectively removing almost all support code for this architecture from the kernel besides trivial changes that simplify the injection procedure.

### 6.1.2 Speed Improvements

A motivation for this work was the perceived slowness of remote debugging via RS-232. Using IEEE 1394 instead of RS-232 increases the available bandwidth of the communication medium by several orders of magnitude. Naïvely, one would expect a speedup of several orders of magnitude as well, but the reality is more complicated.

Operations, such as reading a single word of the target's memory, are more elaborate in this architecture than using a classical RS-232 remote stub in the target kernel. In the latter case, the GNU Debugger sends the request to read a word over the connection, the stub reads the provided address, and returns the data.

In our case, the debugger sends the same request to the monitor, which in turn has to resolve the given virtual address into a physical address. To do that, it needs to walk the target's page table. Walking the page table requires in the worst case, one additional memory access per page table level. This overhead is in most cases avoided by caching (see Section 5.6 on page 38), but each memory read due to a cache miss still costs about 76 µs to 150 µs (see Figure 6.1).

I intentionally decided against measuring the cost of debugging operations from inside the GNU Debugger. Instead, I measured the cost directly at the level of the remote protocol that it uses to communicate with debugging stubs. To achieve that, I implemented a benchmarking application that directly speaks the remote protocol and can be connected via either a RS-232 serial connection to a debugging stub or via TCP/IP to the our monitor. The results for basic operations are shown in Figure 6.2 on page 44.

The *stop reason* and *registers* operations require no communication with the target in our case. The monitor collects the needed information when the target stops, caches it, and hands it to the debugger over a local TCP connection. From the viewpoint of interactive performance, these operations can be performed basically for free compared to their RS-232 counterparts.

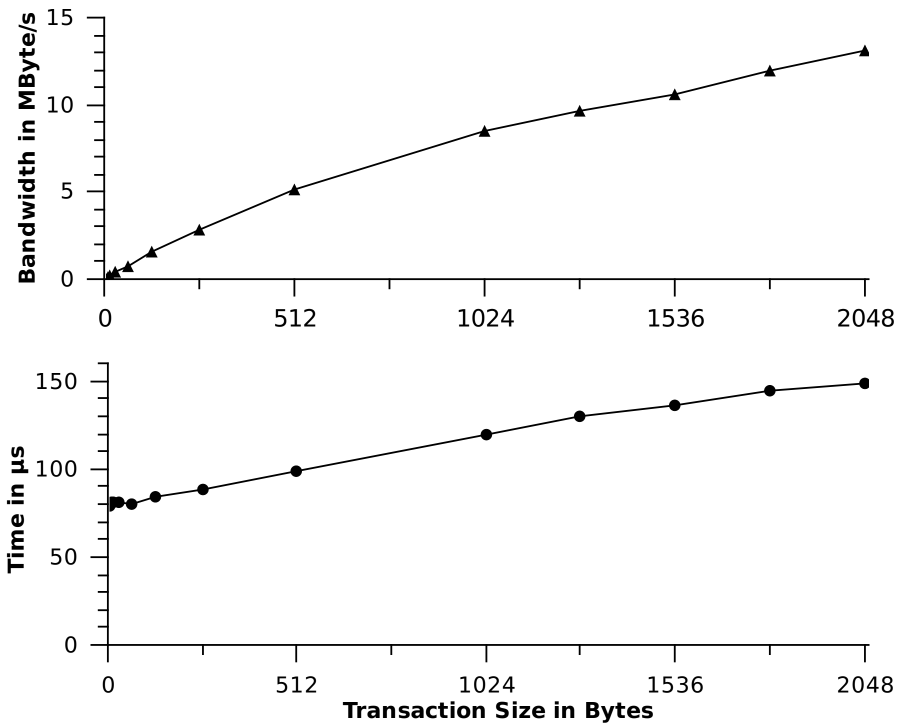


Figure 6.1: A low-level benchmark of `read` transactions on the Firewire bus. For this benchmark 400 MBit/s Firewire was used, due to lack of faster Firewire hardware. The first plot shows sustained bandwidth with transactions of different sizes. For this benchmark, transactions are issued sequentially. The second plot shows the latency of a single transaction of the specified size. IEEE 1394 limits asynchronous transactions to a maximum of 2048 bytes.

*stepi* is more expensive. Although it is still faster when performed over Firewire than over RS-232, the remaining margin is small. The main reason is that a single *stepi* requires two monitor interrupts—one to start execution and another one to figure out where the target has stopped—that each involve several memory transactions. If this cost is prohibitive for a future application of this debugging architecture, a redesign of the communication between monitor and stub may be required.

Reading arbitrary memory on the target should benefit the most from a wider communication channel. Figure 6.3 strikingly supports this intuition. It depicts the time needed to read 32 KByte of memory from the target using different request sizes. The GNU Debuggers default request size of 64 Byte incurs a large overhead, as it needs 512 separate requests to read the whole 32 KByte. However, even in this suboptimal case, reading memory is about a hundred times faster than reading memory over 115200 Bit/s RS-232. For interactive debugging, there is no need to optimize further.

It can safely be concluded that this debugging architecture outperforms remote debugging via RS-232.

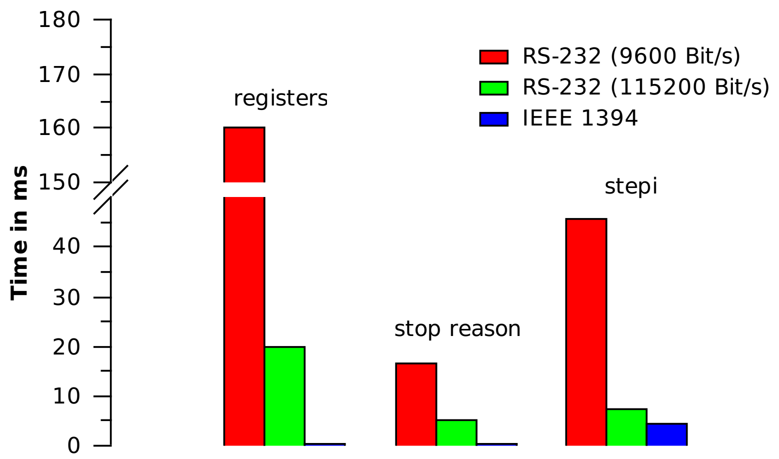


Figure 6.2: The amount of time needed for common debugging operations at the level of the GDB remote protocol. *registers* retrieves the contents of general-purpose registers. *stop reason* is used by the debugger to figure out why the target stopped. *stepi* steps a single instruction on the target system.

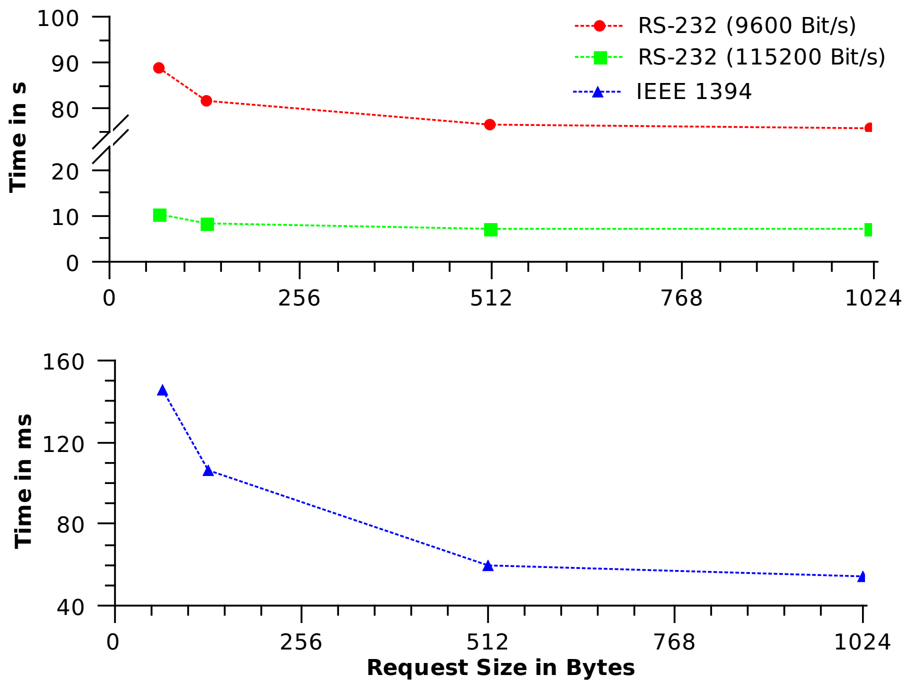


Figure 6.3: The time needed to read 32 KByte of memory from GDB over RS-232 and IEEE 1394 using different request sizes. GDB typically requests memory in 64 Byte chunks, which is obviously not optimal for our approach using IEEE 1394.

| Step | Cycles | Percentage |
|---|---|---|
| *NMI processing* | 600 | 11.1% |
| *task switching* | 4200 | 77.5% |
| *handler* | 620 | 11.4% |
| *total* | 5420 | 100% |

Table 6.2: The cost of handling a single monitor interrupt on the target system broken down by individual steps.

## 6.2 Measuring Monitor Performance: A Statistical Profiler

There are applications, such as statistical profiling, that can be implemented using the lower levels of this debugging architecture. A statistical profiler *samples* the execution state of the target at regular intervals, that is, it collects the instruction pointer and, optionally, parts of the current stack. The collected data is then analyzed to approximate the places where the target spends most of its time. These places are typically called "hot spots".

The results obtained by statistical profiling are only accurate, if the sampling process is cheap and does not disturb the target.[1] I implemented a primitive statistical profiler using the lower levels of the monitor to put the cost of memory accesses over the Firewire bus and monitor interrupts into perspective. As an additional bonus, I think this is an excellent example of what is possible with this framework.

The target system is equipped with a Pentium 4 processor with 1.6 GHz clock frequency, which is known for its high context switch cost and interrupt latency[24].[2] Because this debugging architecture relies on context switching at its core, the gathered results can be considered worst-case results. The Firewire hardware of the test machine is limited to 400 MBit/s transfers.

### 6.2.1 Cost on the Target System

The implemented profiler periodically issues monitor interrupts to obtain the current instruction pointer of the target. The overhead for the target system is processing these monitor interrupts, which are delivered as NMIs. Figure 6.4 shows the transitions between contexts in the target system, when it receives such an interrupt. The direct cost for the target system to receive a single profiling interrupt is thus the time needed to process the NMI request internally in the processor, two context switches, one to the NMI handler and one back, and finally the cost of processing the NMI handler itself.

Table 6.2 shows the time needed to handle a single monitor interrupt on the target, which is 5420 cycles or less than 4 µs, is largely dominated by the cost of task switching.

[1] I am deliberately vague on what constitutes disturbance of the target. These are considerations that are relevant, if the goal is to make performance measurements of the target system itself, which is not the scope of this work.

[2] Apparently, the test machine I have been using is the Pentium 4 test machine in the cited 7–year–old paper. This says *something* about the equipment available to students.

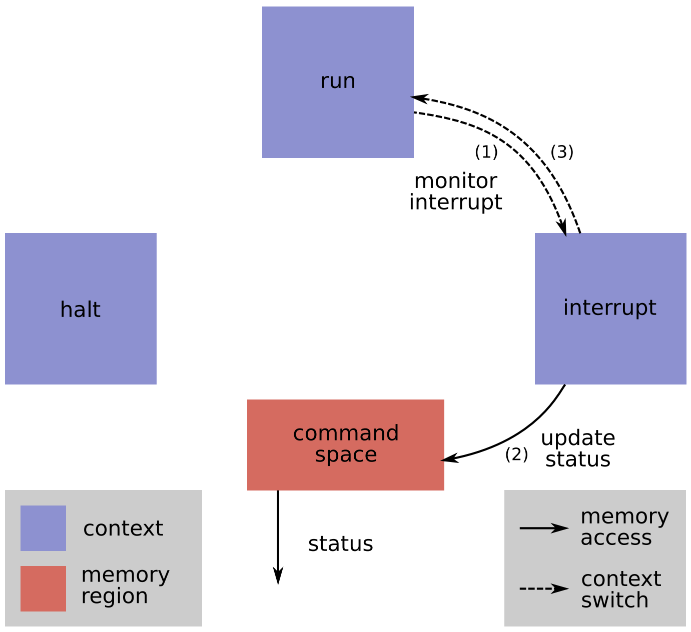


Figure 6.4: Sampling the execution state of the target requires only a single monitor interrupt, which causes the stub to update the command space. The target then resumes execution.

### 6.2.2 Monitor Interrupt Cost

From the perspective of the monitor running on the host, interrupting the target is more expensive, because it involves two transactions on the Firewire bus, one to generate the NMI and (at least) another one to check, whether the target has finished handling the interrupt. To actually do profiling, we need to collect the instruction pointer from the command space as well, which requires one additional transaction on the bus.

Judging from the results depicted in Figure 6.1, a word–sized transactions takes 80 µs to complete. Ignoring execution time, the monitor therefore theoretically needs at least 240 µs to take a sample. The measured cost is with about 300 µs not far off.

To put these numbers into perspective, I instructed the profiler to take samples as fast as it can. The current implementation sustains close to 3000 samples per second, which expected given the cost of 300 µs for a single sample.

Lacking a suitable benchmark running on the Nova hypervisor, I can only estimate the overhead this profiling generates on the target from the time needed to process a single monitor interrupt. Disregarding additional indirect costs due to cache effects, the target spends about 1 % of its execution time processing monitor interrupts.

Given the low overhead on the target machine, it may be desirable to increase the rate at which monitor interrupts can be generated. I believe that this can be accomplished in the current framework by starting certain Firewire bus transactions concurrently, but this has not yet been pursued due to lack of time.

## 6.3 Summary

This chapter showed that the debugging architecture presented in the preceding two chapters outperforms classical remote debugging using RS-232 serial connections both in software engineering effort and speed. It requires considerably less effort to adapt an operating system kernel to this architecture, while offering the same feature set.

Additionally, this debugging architecture can be the foundation for other applications as well. I presented a primitive statistical profiler that utilizes the facilities already provided in the monitor and argued that it, even in its infantile form, imposes little overhead on the target system.

The next chapter will outline worthwhile future directions for this architecture.

# 7 Future Work

There are several dimensions on which the design and implementation of this debugging architecture can be extended. This chapter shall explore some of the most obvious in more detail.

## 7.1 Porting to 64-Bit

The 32-bit Intel architecture (IA-32) is slowly fading out in favor of its successors. The AMD64 architecture and its twin Intel 64 are an evolutionary step of the IA-32 architecture into the world of 64-bit computing. Processors based on this architecture are able to run unmodified operating systems and applications written for the IA-32 architecture, but require changes to the operating system, if any of the new features are to be used. Adapting the debugging architecture to support the 64-bit extensions is unavoidable in the near-term future.

Fortunately, most of the AMD64 architecture is compatible to its predecessor IA-32. However, the hardware-assisted task switching used in the implementation of the kernel stub for IA-32 is not present in AMD64 in its native 64-bit execution mode.

To handle interrupts on a known–good stack, AMD64 introduces the Interrupt Stack Table (IST). It describes up to seven different stacks that can be used for interrupt handlers. To handle an interrupt on another stack, the interrupt handler's entry in the IDT points to one of the stacks in the IST.

Saving the processor state has to be done manually, because the stack switching of AMD64 only saves the old stack and instruction pointer. Fortunately, it does not destroy processor state, as the IA-32 task switching does. Because context switches are initiated by the reception of an interrupt, the stack switching support of AMD64 together with manual saving and restoring of processor state suffices to implement them.

To determine which context was interrupted it is necessary to inspect the instruction pointer. As this is only required in the `interrupt` context, which handles NMIs, the code only has to distinguish the `halt` from the `run` context. Because the `halt` context executes only a handful of machine instructions, the distinction is trivial.

Changes to the monitor would include adapting it to the different format of processor state in memory. Apart from reading and writing the processor state, the monitor could be used unmodified.

The adaptions required for AMD64 are not implemented yet. Based on the previous observations, however, it should be straight-forward to implement them.

## 7.2 Supporting Multiprocessor Systems

Multiprocessor systems are already permeating the market of desktop computers and are beginning to appear in embedded designs as well [37]. Adapting this debugging architecture to multiprocessor systems, therefore, seems like a viable goal. Surprisingly, adding multiprocessor support would change little of the design. Most code could be reused as well.

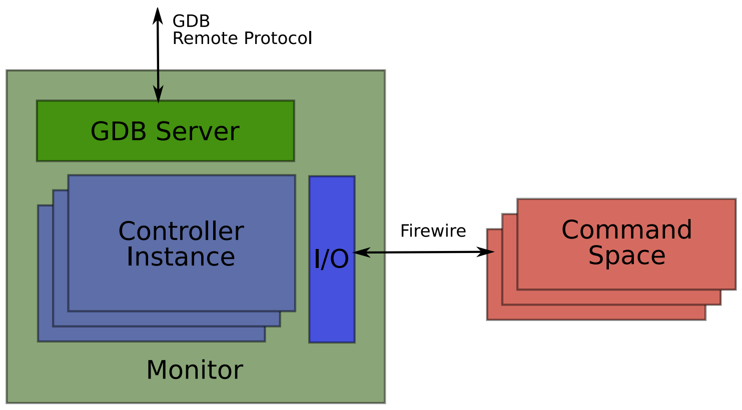


Figure 7.1: Design changes required for multiprocessor systems are the introduction of per–processor controller instances and per–processor command spaces.

The key idea is to use one controller instance and one command space per processor. This solution would require a method to individually interrupt specific processors. The current technique of injecting MSIs via Firewire (see Section 5.2.2 on page 32) provides this functionality.

Most of the work to support multiprocessor systems would revolve around the GDB server in the monitor. As GDB already has support for threads in the target, exposing the different CPUs of the target as threads to GDB should be straight-forward. Nonetheless, the usefulness of this approach is small, because GDB does not support different address spaces, limiting its use to cases in which all CPUs share a common address space.

## 7.3 Build One to Throw Away

If there is one thing that I definitely have learned, then it is that the GNU Debugger is—in my opinion at least—at its core insufficient for system debugging. This conclusion may sound like a harsh criticism, if you consider the pervasiveness of GDB in kernel debugging, but I believe it to be justified.

The problems I see stem from the fact that the GNU Debugger was intended as an application debugger for a UNIX-like system. From an application debugger's point of view it makes sense to assume that there is only *one* address space with *one* application. A system debugger needs a broader scope. What that scope exactly is, may be the question of a future thesis, but a system debugger at least has to have a concept of address spaces, each containing different programs.

Consider the following example. The system being debugged runs multiple instances of the same program. Ignoring “features” such as address space randomization, the executable code of each process will probably occupy the same virtual address range, albeit in different address spaces. If a user wants to set a breakpoint in exactly one of these processes, he is out of luck with GDB. To the GNU Debugger address *n* in one process is indistinguishable from the same address *n* in another process.

The missing concept of address spaces in GDB is just one aspect that makes it undesirable as a system debugger. There are many more. For example, the GNU Debugger lacks support for:

- physical addresses and virtual memory in general,
- multiple processors,
- specific processor registers, such as the control registers and model–specific registers (MSRs).

If a debugger is to be truly useful as system debugger, it would have to support all of these features. I do not believe they can be integrated into the GNU Debugger in a sane way without rewriting large parts of it. In the end, it may make more sense to start developing a system debugger from scratch.

## 7.4 Summary

This chapter outlined possible ports of this debugging architecture to 64-bit systems and discussed design changes for multiprocessor machines. Finally, an argument for writing a system debugger from scratch was made.

The next and final chapter will wrap up this thesis by summarizing its achievements.

# 8 Conclusion And Outlook

In this thesis, I implemented, based on the GNU Debugger, a fully functional remote system debugger for the NOVA microhypervisor, which in its final version requires almost no changes to NOVA itself.

Using Firewire as communication medium and remote manipulation tool, the size of the required kernel stub could be reduced by an order of magnitude compared to traditional remote debugging using RS-232. In contrast to traditional remote debugging, the required kernel stub does not have to be included in the target kernel, but can be independently developed and injected at runtime. Porting the kernel stub to new target systems, is believed to be simple. A proof–of–concept port to the Linux kernel has been done in three days.

A major accomplishment of this work was the separation of the debugger from its target system. Separating both has several advantages. On the one hand, it removes the burden of maintaining the debugging infrastructure from system programmers. On the other hand, it allows one remote debugging infrastructure to be used for several different target systems.

Other minor achievements are the considerable speed boost of using Firewire instead of RS-232 and the possibility of using the low–level interface of the monitor for other applications, such as profiling, which was cursorily explored in Section 6.2.

Although immediate goals for the developed debugger would be to port it to 64-bit and multiprocessor systems, I regard the major future goal to be the replacement of the GNU Debugger. Section 7.3 argued that the GNU Debugger is at its core insufficient for system debugging, even though it is currently pervasively used for that purpose. During the course of this thesis, it has become clear that system debugging needs a fresh start.

# Acronyms

| | |
|---|---|
| APIC | Advanced Programmable Interrupt Controller. 32, 38 |
| DMA | Direct Memory Access. 6 |
| GDT | Global Descriptor Table. 37 |
| GUID | Globally Unique Identifier. 7 |
| ICE | in–circuit emulator. 15 |
| IDT | Interrupt Descriptor Table. 11, 37, 49 |
| IOMMU | Input–Output Memory Mapping Unit. 10, 39 |
| IST | Interrupt Stack Table. 49 |
| MSI | message-signaled interrupt. 11, 38, 50 |
| NMI | non-maskable interrupt. 24, 32, 33, 36, 37, 41, 45, 46, 49 |
| OHCI | Open Host Controller Interface. 9, 31, 32, 39 |
| PCI | Peripheral Component Interconnect. 5, 6 |
| SLOC | source lines of code. 36 |
| TSS | Task State Segment. 12, 32, 33 |
| USB | Universal Serial Bus. 6, 7 |

# Bibliography

[1] Using physical DMA provided by OHCI-1394 FireWire controllers for debugging. http://www.suse.de/~bk/firewire/debugging-via-ohci1394.txt. Online, accessed 16-12-2008. 17

[2] 1394 Open Host Controller Interface Specification, January 2000. Release 1.1. 9

[3] Key Benefits of the I/O APIC. http://www.microsoft.com/whdc/archive/io-apic.mspx, December 2001. Online, accessed 17-12-2008. 11

[4] Enhanced Host Controller Interface Specification for Universal Serial Bus, March 2002. Revision 1.0. 6

[5] *IEEE 1394b-2002*, December 2002. Amendment to IEEE Std 1394-1995. 7

[6] USB2 Debug Device: A Functional Device Specification, March 2003. Revision 0.9. 21

[7] Michael Becher and Maximillian Dornseif. Feuriges Hacken - Spaß mit Firewire. In *21C3: Proceedings of the 21st Chaos Communication Congress*, December 2004. 10

[8] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005. 16

[9] Adam Boileau. Hit by a Bus: Physical Access Attacks with Firewire. In *RUXCON*, 2006. 10

[10] Frederick P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. 1

[11] Christian Böhme. PCI-to-PCI-Bridge mit sicherheitsrelevanten Eigenschaften. http://os.inf.tu-dresden.de/papers_ps/boehme-beleg.pdf, July 2005. 10

[12] Robert R. Collins. In-Circuit Emulation: A powerful hardware tool for software debugging. *Dr. Dobbs Journal*, July 1997. 15

[13] Robert R. Collins. In-Circuit Emulation: How the Microprocessor Evolved Over Time. *Dr. Dobbs Journal*, September 1997. 15

[14] Maximillian Dornseif. 0wn3d by an iPod: Firewire/1394 Issues. In *PacSec Applied Security Conference*, 2004. 10, 18

[15] Norman Feske. A case study on the cost and benefit of dynamic RPC marshalling for low-level system components. *SIGOPS Operating Systems Review*, 41(4):40–48, 2007. 2

[16] Bill Gatliff. Embedding with GNU: GNU Debugger. *Embedded Systems Programming*, September 1999. 3

[17] Bill Gatliff. Embedding with GNU: the gdb Remote Serial Protocol. *Embedded Systems Programming*, November 1999. 3

[18] Tom Green. 1394 Kernel Debugging Tips and Tricks. Slide presentation at the WinHEC 2004, `http://download.microsoft.com/download/1/8/f/18f8cee2-0b64-41f2-893d-a6f2295b40c8/DW04001_WINHEC2004.ppt`, 2004. Online, accessed 06-03-2009. 17

[19] Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. Applying source-code verification to a microkernel: the VFiasco project. In *SIGOPS European Workshop*, pages 165–169, New York, NY, USA, 2002. ACM. 1

[20] Hermann Härtig, Michael Roitzsch, Adam Lackorzynski, Björn Döbel, and Alexander Böttcher. L4-Virtualization and Beyond. *Korean Information Science Society Review*, December 2008. 2

[21] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, September 2008. Order Number: 253668-028US. 32

[22] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, March 2009. Order Number: 253665-030US. 11

[23] Jochen Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, 1995. 1

[24] Frank Mehnert, Michael Hohmuth, and Hermann Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002*, pages 124–133, 2002. 45

[25] PCI Special Interest Group. *PCI Local Bus Specification*, December 1998. Revision 2.2. 6

[26] Andreas Pfitzmann. *Diensteintegrierende Kommunikationsnetze mit teilnehmerüberprüfbarem Datenschutz*. Springer, 1989. 1

[27] The FreeBSD Project. dcons(4) manual page. `http://www.freebsd.org/cgi/man.cgi?query=dcons&sektion=4&manpath=FreeBSD+7.0-RELEASE`. Online, accessed 16-12-2008. 17

[28] Steven Rostedt and Darren V. Hart. Internals of the RT patch. In *Proceedings of the Linux Symposium*, 2007. 38

[29] Jonathan Shapiro. Programming Language Challenges in Systems Codes: Why Systems Programmers Still Use C, and What to Do About It. In *PLOS '06: Proceedings of the 3rd workshop on Programming languages and operating systems*, page 9, New York, NY, USA, 2006. ACM. 1

[30] Jonathan Shapiro, Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark Miller. Towards a verified, general-purpose operating system kernel. In *NICTA Formal Methods Program Workshop on Operating Systems Verification*, page 1, 2004. 1

[31] (PCILynx-2) IEEE 1394 Link Layer Controller. `http://focus.ti.com/lit/ds/symlink/tsb12lv21b.pdf`, June 2006. Online, accessed 17-12-2008. 9

[32] Richard M. Stallman, Roland H. Pesch, and Stan Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, 2002. 18

[33] Udo Steinberg. NOVA Microhypervisor Interface Specification. Technical report, Technische Universität Dresden, 2009. 31

[34] Udo Steinberg and Bernhard Kauer. Microhypervisor & Virtual–Machine Monitor Implementation Overview. Technical report, Technische Universität Dresden, 2008. 4, 36

[35] Ken Thompson. Reflections on Trusting Trust, 1983-ACM Turing Award Lecture. *the Communications of the ACM*, 27(8):761, 1984. 1

[36] Harvey Tuch, Gerwin Klein, and Gernot Heiser. Os verification: now! In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association. 1

[37] Philipp Zech. Embedded Multi-Core Processors. Seminar Embedded System Design, Institute of Computer Science, University of Innsbruck, January 2008. 50

[38] Andreas Zeller and Dorothea Lütkehaus. DDD—a free graphical front-end for UNIX debuggers. *SIGPLAN Not.*, 31(1):22–27, 1996. 18