CHALMERS

Hardware Assisted Breakpoints in the Linux Kernel for LEON SPARC

Bachelor's thesis in Computer Science and Engineering

DAVID WILKINS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Bachelor's thesis 2020

Hardware Assisted Breakpoints in the Linux Kernel for LEON SPARC

DAVID WILKINS

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Hardware Assisted Breakpoints in the Linux Kernel for LEON SPARC
DAVID WILKINS

Supervisor: Daniel Hellström, Cobham Gaisler
Supervisor: Jan Jonsson, Department of Computer Science and Engineering
Examiner: Peter Lundin, Department of Computer Science and Engineering

Bachelor's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology / University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden 2020

Hardware Assisted Breakpoints in the Linux Kernel for LEON SPARC
David Wilkins
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Embedded systems can be found in a wide variety of applications ranging from performing simple tasks in a dish washer, to handling critical operations in satellites. There are many different operating systems to choose from when designing such a system. The Linux kernel is open source and a popular alternative for embedded systems. It has a large and active community and supports many different architectures.

LEON SPARC is a series of processor cores developed by Cobham Gaisler based on the SPARC v8 architecture. They are especially designed for *System on Chip* (SoC) solutions which are commonly used in embedded systems.A number of LEON processor cores are fault-tolerant allowing them to be used in space applications. This, and many other applications, creates a high demand for testing and debugging.

SPARC is one of the architectures supported by Linux. This support is maintained by Gaisler and other SPARC developers. Gaisler is interested in extending the debugging option on Linux for SPARC by implementing support for hardware assisted breakpointing.

This thesis report provides an implementation of hardware assisted breakpoints in the Linux Kernel for LEON SPARC. In addition, a GDB patch was created to provide a use case for the new kernel support.

Keywords: Linux kernel, SPARC, LEON, Gaisler, hardware assisted breakpoints, watchpoints, embedded systems, debugging.

Sammanfattning

Inbyggda system används brett i olika användningsområden från att utföra enkla uppgifter i en diskmaskin till att hantera kritiska uppgifter i satelliter. Det finns många olika operativ system att välja mellan när man utvecklar ett sådant system. Linux kärnan har öppen källkod och är ett populärt alternativ för inbyggda system. Den har en stor och aktiv användarbas av utvecklare och stödjer många olika arkitekturer.

LEON SPARC är en serie av processorer utvecklad av Cobham Gaisler och utvecklad av SPARC v8 arkitekturen. De är speciellt utvecklade för *System on Chip* (SoC) -lösningar vilket är vanligt inom inbyggda system. En rad av LEON-processorer är feltoleranta vilket gör dem användbara i rymden. Det och många andra användningsområden har höga krav på testning och felsökning.

SPARC är en av de arkitecturer som stöds av Linux. Det stödet underålls av Gaisler och andra SPARC utvecklare. Gaisler vill uttöka felsökningsstödet i Linux för SPARC genom att implementera hårdvarustödda brytpunkter.

Det här examensarbetet bidrar med en implementation av hårdvarustödda brytpunkter i Linux kärnan för LEON SPARC. Dessutom skapades en patch av GDB som bidrar med ett användningsfall för det nyutvecklade stödet i Linux kärnan.

Nyckelord: Linux-kärnan, SPARC, LEON, Gaisler, hardvaru-assisterade breakpoints, bevakningspunkter, inbyggda system, felsökning.

Acknowledgements

This project has been conducted in collaboration with Cobham Gaisler which has supervised the project and provided necessary resources. I want to thank the people at Cobham Gaisler for the opportunity to work on this project and for their guidance. Special thanks to my supervisor Daniel Hellström and Andreas Larsson for their support.

This bachelor's thesis marks the end of a three year computer engineering programme at Chalmers University of Technology. I want to thank my supervisor Jan Jonsson for helping write this report.

David Wilkins, Gothenburg, May 2020

Terminology

- **Breakpoint** - A debugging tool that stops execution of a program when a specified instruction is reached.
- **Inferior** - A process debugged by GDB.
- **Kernel** - The most fundamental part of an operating system.
- **Kernel space** - Where the kernel resides.
- **Signal** - Used for interprocess communication.
- **System call** - An interface to the kernel's services.
- **Target (general)** - The system that is being targeted for debugging or compilation.
- **Target (GDB)** - An interface to the inferior process which is in part defined by the target architecture.
- **Task** - Linux implementation of a process.
- **Trap** - An internal hardware event generated by the processor.
- **Trap frame** - The state of the processor stored on the event of a trap.
- **Process** - An instance of an executable program.
- **User space** - Where user applications reside.
- **Watchpoint** - A debugging tool that stops execution of a program when an address location is read from or write to. Sometimes referred to as a *data breakpoint*.

Contents

1

Introduction

1.1 Embedded Systems

An embedded systems is computer systems designed for a specific task, operating within a larger system. A common example of an embedded system is the modern automobile, providing many different features ranging from anti-blocking systems to multi-media services. These features are typically provided by embedded system that are programmed for that one task. Some embedded systems operate in real-time, some operate under harsh conditions. These factors put a higher demand on testing and debugging embedded systems compared to an average desktop application.

1.2 LEON SPARC

Cobham Gaisler is a company providing a library of digital hardware designs used in embedded applications. LEON SPARC is one of the product lines developed by Gaisler. The LEON processor cores are based on the SPARC v8 architecture. They are especially designed for System on Chip (SOC) solutions which are commonly used for embedded systems. They are appropriate for a wide range of commercial applications ranging from GPS recievers to multimedia units. Gaisler also develops fault-tolerant versions of the LEON processors which are used for aerospace applications.In actuality, the LEON processor has it's roots in fault-tolerant space applications since it started as an initiative by the European Space Agency (ESA).[1] LEON can run several different operating systems such as RTEMS and Linux or it can run programs without any operating system, commonly referred to as *bare metal*.

1.3 Purpose

Cobham Gaisler is interested in extending the debugging options on Linux for SPARC by implementing support for hardware assisted breakpoints and watchpoints. The LEON processor does provide hardware support for these features but the Linux kernel does not utilize it. The aim of this report is to investigate the missing link as well as providing an implemented solution. The solution intends to bring the hardware support for breakpoints to the user space. The project also aims

to provide a use case of the implemented design by providing a GDB patch that makes use of it. The implementation will be tested with handwritten manual tests.

The project aims to reach the following goals.

1. Describe the existing hardware support for breakpoints provided by the LEON SPARC processor.
2. Describe the systems in place, in the Linux kernel, that the implementation will be based upon.
3. Describe the internal structure of GDB that is responsible for placing hardware breakpoints.
4. Provide a patch to the Linux kernel that implements hardware breakpoint support for the LEON SPARC processor.
5. Provide a patch to GDB that can place hardware breakpoint on the LEON SPARC processor running Linux.
6. Test the new hardware breakpoint features of the Linux kernel.

1.3.1 Delimitations

The implemented kernel support will only support user space breakpoints, not kernel breakpoints.

The provided GDB-patch will only implement support for native targets, not remote targets. Although the kernel implementation may support it just as well, this will not be tested nor tried.

This project is based on the LEON 4 hardware specifically. Although most of the relevant specifications are the same across versions, some details may differ.

1.4 Disposition of Report

This report will first cover some theoretical knowledge necessary to understand the rest of the report. This includes some fundamental concepts regarding Linux, debugging and software development for embedded systems. Next the methods used for examination, implementation and testing will be outlined. In chapter 5 the results are presented. It begins by describing the hardware support provided by LEON SPARC and the incomplete software support provided by the kernel. It also explains the inner-workings of GDB. The chapter continues by presenting an implementation in the kernel that provides hardware breakpoint support as well as a patch to GDB that can use this implementation. The chapter ends by introducing a few tests and their result. Finally the utility of the hardware and the implementation will be discussed together with possible future improvements.

2

Theory

2.1 The Kernel

The kernel is the foundation of the operating system providing it's most essential system services. It acts as an interface to the hardware resources and provides an environment for user applications, called the *user space*. In order to protect the hardware resources, two execution modes are introduced, *kernel mode*, also known as *supervisor mode*, and *user mode*. Programs running in user mode has no access to hardware resources, this is reserved to the kernel running in kernel mode. Therefore, user applications rely on *system calls* for these services.

2.1.1 System Calls

A system call is a type of procedure that allows user applications access to kernel services. They work similarly to any other function call, the main difference being that they enter kernel code, thus the CPU switching to kernel mode. *Ptrace* and *fork* two examples of system calls mentioned in this report.

2.1.2 Processes and Threads in Linux

A process is the instance of a program. A new process is created with a `fork` system call which creates a *child* process of the calling *parent*. A defining feature of a process is that they are assigned different address spaces containing its stack and heap among other things. On creation a child will be assigned a copy of the parents address space. This means that any changes made to one of them will not affect the other. This is not the case for threads which share the address space with other threads under the same process.

There are two types of threads, *user threads* and *kernel threads*. User threads run entirely in user space with the kernel unaware of their existence. This means that scheduling user threads is handled by a user library such as *pthread*. Kernel threads are scheduled by the kernel allowing for greater concurrency and even parallelism on multi-core systems.

In Linux, support for kernel threads are provided in the shape of *lightweight processes*. Creating a new process using the `clone` system call allows specific resources

to be shared between parent and child. This blurs the lines between a process and a thread. This is especially true considering that lightweight processes and traditional processes use the same data structure to store their process descriptors, namely *tasks*. In Linux, a task is represented by the `task_struct` structure. Each `task_struct` contains a `thread_struct`, specifiying the processor execution state. On a task switch, the fields inside `thread_struct` are written to the CPU registers, restoring the thread state.

Lightweight processes can be organized in *thread groups*. This makes them behave more as single program, responding to some system calls in a more unified manner. [2]

2.1.3 Signals

Signals are a form of lightweight communication between processes provided in Unix systems. They are short messages containing only a number that defines the signal. Different signals communicate different events and cause the receiver to take different actions. Each signal has an associated default action that the receiver will take, however these default actions can be changed. A process can also decide to block signals it doesn't want to receive. below is a table of some available signals with their default actions.

signal	action	description
SIGTRAP	Core dump	Breakpoint hit
SIGILL	Core dump	Illegal instruction
SIGCHLD	Ignore	child process stopped or terminated
SIGCONT	Continue	Continue a stopped process
SIGKILL	Terminate	Force process termination
SIGSTOP	Stop	stop process

Note that a core dump action involves writing a core dump file to the current working directory and terminating the process.

In addition to these actions, a process can also decide to catch a signal. Catching a signal will cause a specified signal handler to be called.[2]

2.2 Exceptions and Traps

Exceptions are a way of communicating hardware events to software, most commonly to the kernel. They are generated on the execution of an instruction, making them synchronous. *Interrupts* are similar to exceptions but are asynchronous and usually comes from external hardware. Exceptions are commonly used to indicate a *fault* however this is not always the case. *Traps* are a subcategory of exceptions that does not indicate a fault, however the term *trap* and *exception* are sometimes

used interchangeably.There are usually many different types of traps and exceptions specified for an architecture. Each type has it's own entry in a *trap table*. These entries specify a *trap handler* for their respective trap type. Thus the trap handler is a function that is called on a trap event. In Linux a trap handler usually sends a signal to the process that caused it, notifying it about the event. [2]

2.3 Principles of Debugging

There are many different ways to debug a program. It can be done by special support from the hardware or entirely in software. Two important terms are the *host* and the *target*. The host is the system of which your debugger is running, while the target is the system executing your program. When the host and the target system are the same you are debugging natively. However it is often possible to debug a program running on a machine different from the one running the debugger. This is what is referred to as remote debugging. Remote debugging is a common practice for embedded systems since it requires less storage from the target.

2.3.1 Breakpoints and Watchpoints

A breakpoint can be set on a specified instruction telling the program to stop execution when it is reached. Breakpoints can be implemented in two main ways. They can be implemented in software, known as *software breakpoints*, or with special support from the hardware, referred to as *hardware breakpointing*.

A watchpoint is similar to a breakpoint, but it triggers when an address location is accessed rather than executed. They are used to detect a read or write operations to variables. The terms "breakpoint" and "watchpoint" are often used interchangeably. Sometimes a breakpoint is referred to as a type of watchpoint, this is how the Linux kernel tends to define it. However some times the opposite is considered to be true. In this report the term "breakpoint" will be used, for bot types, when not referring to watchpoints specifically.

2.3.1.1 Software Breakpoints

The most common way to implement software breakpoints is to have a special breakpoint instruction within the architectures instruction set. When a breakpoint is set the operation code for the specified instruction is replaced by the operation code for the breakpoint instruction. The operation code for the original instruction is then stored inside a breakpoint table. The program runs normally until the breakpoint instruction is reached. Executing the breakpoint instruction generates a trap, pausing the execution of the program. At this point the original instruction is once again placed on the address. Since the execution of the breakpoint instruction has already started the original instruction must be re-fetched in order to execute.

The benefit of using software breakpoints is that any number of breakpoints can be placed. However software breakpoints need to alter the memory addresses which

means they can only be placed on writable memory. Since some software are also installed on ROM, such as the kernel, software breakpoints are limited.

Software watchpoints are implemented by stepping through the entire program and checking whether one of the watchpoint addresses has changed their value. This has a significant impact on performance and is not very practical.

2.3.1.2 Hardware Breakpoints

Hardware breakpoints are implemented using special hardware registers. These registers store the addresses of the instructions that the breakpoints are placed upon. While fetching each new instruction, the CPU compares the address to the ones stored in the breakpoint registers. If there is match, a trap is generated.

Since hardware breakpoints are stored inside dedicated registers, the amount of breakpoints is limited to the amount of registers provided by the hardware. However they can be placed on any type of memory. It also has the added benefit of providing watchpoint capabilies without performance loss.[3]

2.3.1.3 Kernel Breakpoints and User Breakpoints

Hardware breakpoints can be placed on any type of memory, this allows them to be places inside kernel space as well as in user space. Breakpoints placed in user space are referred to as *user breakpoints*. They are only sensitive to the process that created them. This means that the breakpoints are only set in hardware while their process is running. Breakpoints placed inside the kernel are called *kernel breakpoints*. They can be triggered by any process that steps on their address. Since they are sensitive to the entire system, they must be placed on all CPU's in that system. When a system supports both kernel breakpoints and user breakpoints, the debug registers must be shared between these two types.[4]

2.4 Embedded Software Development

Software development for embedded systems require tools specialized for the task. Many of these tools are specific for a certain processor, architecture or operating system. Gaisler provides support for their hardware design in the shape of compilers, build tools, simulators and debugging solutions. They also contribute with patches to the Linux source code for the SPARC architecture.

2.4.1 Compilers

Development of embedded applications requires a cross-compiler. This is a compiler that runs on one system but compiles executables for another system. The host system is typically a common desktop computer such as Windows running on an Intel chip. The target is the embedded system you want to develop applications for, such as Linux running on a LEON processor. However the target can also be

a processor with no installed operating system. This requires a *bare metal cross-compiler*. GCC can often be used a cross compiler, being open source allows target developers to provide patches and configurations to it. The compiler is an essential part of a *toolchain*.

2.4.2 Toolchains

A toolchain is a set of development tools that are used for software development. This mostly involves a compiler but can also include device drivers and even debuggers. They are commonly used in development for embedded software to provide support for a target system.

Gaisler provides toolchains for LEON with several target operating systems, including Linux and RTEMS. There is also a toolchain for bare metal compilation called Bare-C Cross Compiler System (BCC), available for LEON. Most of these toolchains are based on GNU libraries specifically modified for different LEON targets.

2.4.3 Build Tools

The Linux kernel is used in a wide variety of applications using a broad range of underlying hardware. It can be configured in many different ways to fit your specific requirements. The underlying system for configuring the kernel is called *kbuild* which uses a mechanism called *kconfig*. It is structured as a hierarchy with a configuration file for each configurable source directory.

Buildroot is a common tool that simplifies the configuration and building process for Linux systems. it provides a user interface and it can be used to generate a kernel image, file system, boot loader and even a toolchain. The kernel image is then loaded on to hardware or a simulator and booted.

2.4.4 Simulators

During development of software for embedded processors it is preferable to use a simulator or emulator of the target hardware. This is often more flexible than actual hardware. Gaisler provides a simulator for the LEON processor called TSIM.

2.4.5 Debuggers

There are many different available software debuggers that can be used for embedded applications. The GNU Debugger (GDB) is a popular open source debugger that can be used natively or remotely (see section 2.3. The open source nature of GDB allows developers to freely provide support for their specific target system.

Software debuggers work well for debugging user space applications but have limited capabilities for debugging software working closer to the hardware. For this reason a hardware debugger is often desireable. They usually allow greater control of the hardware and are very useful for debugging the kernel.

3

Method

This project will be conducted using tools provided by Gaisler. This includes build tools, tool chains as well as TSIM. Their toolchain for Linux is used to build the Linux kernel which provides LEON configurations of GCC and GDB. The resulting kernel image is loaded on to TSIM and run. The kernel and hardware will be debugged using TSIM's internal features. The GDB patch will be debugged using GDB. Manual tests will be loaded onto the target file system and executed using a shell script.

This report is also heavily based on researching LEON, SPARC, the Linux kernel and GDB. This is primarily done using hardware manuals, documentation and source code.

3.1 Building and Compiling

For this project Linuxbuild is used as the front end build system. It is provided by Gaisler and is based on buildroot. It provides an easy way to get started with building the kernel for LEON by gathering all the necessary tools.

The source code can be downloaded directly in the Linuxbuild interface from Gaisler's website. The source code for Linux version 4.9 is used since it is the latest stable version for LEON.

Linuxbuild is configured to use the LEON Linux toolchain for building the kernel, which can also be downloaded on Gaisler's website.

3.1.1 Configuration

When configuring the kernel, most of the options are the left as the default provided by Linuxbuild. However some configurations are necessary specifically to this project.

The default configuration will build for LEON 3, while this project aims to work on LEON 4. These two versions has different base addresses. LEON 3 will start on address 0x40000000 on start up while LEON 4 will start on address 0x00000000. The start address need to be re-specified to 0x00000000 inside Linuxbuild or the

kernel will try to boot on the incorrect address.

In addition GDB needs to be built to the target if we are to use it natively on the target. There are configuration options for this in Buildroot. Since We will be providing our own patch, we want to set a custom source path that buildroot will should build from, leading to the source code we implemented.

3.2 Simulating

The simulator used for this project is TSIM, which is provided by Gaisler. It simulates the hardware of the LEON processor allowing commands to be issued through a command line interface. These commands are used to load executables, resetting and running the simulator, amongst other things. TSIM provides debugging built-in debugging commands which are useful for debugging the kernel. The simulator can also halt the simulated CPU at any time, allowing you to investigate the processor state.

3.3 Debugging

The debugging commands in TSIM will be used when debugging the kernel. These features include breakpointing, watchpointing, backtracing, disassembling and reading and writing to registers as well as memory locations.

GDB will be debugged by an other instance of GDB. This is possible but requires a dummy program for the debugged GDB session to debug.

3.4 Testing

The ptrace layer will be tested using manually contructed tests. These test will try the basic features of the implementation. They are pre-compiled and placed in the target file system. A shell script is also supplied to run all the tests and print the results.

4

Result

In this chapter the results of the defined goals will be recounted. First the existing support in hardware, kernel space and user space will be described, this part is purely investigatory. Then an implementation solution for hardware breakpoints in the kernel will be provided, which describes this project's contribution to the Linux kernel. This is followed by an implementation of a GDB patch using the new found kernel support, this recounts the contribution made to GDB. Finally the tests for the kernel implementation will be described.

4.1 Hardware support

The LEON processor is based on the SPARC architecture. This means that SPARC provides a certain standard and LEON is an implementation that adheres to this standard. Some specifications are detailed by the SPARC architecture while some are determined by the LEON implementation. SPARC does not mandate any support for hardware breakpoints, meaning that much of the support is specific to LEON. However SPARC does provide some important specifications and definitions that does influence LEON's implementation.

In this section, the existing hardware support for breakpoints will be recounted. This includes the relevant standards from the SPARC architecture, the LEON implementation of these standards and how hardware traps are resolved by the hardware.

4.1.1 SPARC

This section will establish the relevant specifications provided by SPARC.

4.1.1.1 Ancillary State Registers

The SPARC architecture provides two primary kinds of registers. These are *general purpose registers*, also known as *r-registers*, and *status/control registers*.

Amongst the status/control registers there is a subcategory called *ancillary state registers* (ASRs). Sparc provides a total of 31 ASRs. ASR1 - ASR15 are reserved for future use and may not be implemented. However ASR16 - ASR31 are *implementation-dependant* meaning that what they are used for and their privilege

levels are determined by the implementation, in our case LEON. They are read from and written to using the RDASR and WRASR instructions respectively.

4.1.1.2 Configuration Registers

The configuration registers is another group of status/control registers defined by SPARC.

The *Processor State Register* (PSR) holds status information and contains several fields that controls the processor. One of these is the *Enable Trap* (ET) bit which enables and disables traps for the processor. If ET is one, then a trap is handled as normal. If ET is zero, incoming interrupting or deferred traps will be ignored, however precise traps will cause the CPU to enter error mode. This will halt the CPU requiring a reset in order to restart it. There is also the *Supervisor* (S) bit which is set when the CPU is in kernel mode. In addition the *Previous Supervisor* (PS) bit stores the mode that the CPU was in before the event of a trap or a system call.

The *Trap Base Register* (*TBR*) defines where to look for a handler on the occurance of a trap. Bit 31 through 12 contains the *Trap Base Address* (TBA) which defines the 20 bit address space in memory where the trap handler table resides. Bit 11 through 4 defines the trap type (*tt*). It functions as an offset into the trap table. It is set by the hardware to point to the handler of the current trap and retains its value until the next trap. Finally the 4 least significant bits are filled with zeroes and are never used.

The *Program Counter* (PC) and the *Next Program Counter* (NPC) is also considered status/control registers. Each instruction takes 32 bits of memory, this means that NPC points 4 bytes after PC.[5]

4.1.1.3 Register Windows

An implementation of SPARC can have an IU containing between 40 and 512 general purpose registers.[5] However only 32 of these registers are accessable at a given time. A program can access 8 global r-registers and 24 mapped r-registers. The mapped registers provide a *register window* which consists of 8 local registers(L0-L7), 8 in-registers(I0-I7) and 8 out-registers(O0-O7). This register window changes its mapping on a context switch providing new registers to the callee. During a context switch, the out-registers of the caller is remapped as the in-registers of the callee, thereby passing arguments between adjacent contexts. Therefore the in-registers and the local registers are together referenced as the *register set*, consisting of 16 registers. Note that the amount of available register sets depend on the amount of available general purpose registers implemented in the IU.

The window of the current context is indicated by the *Current Window Pointer* (CWP), which is a field inside PSR. On a context switch CWP is decremented or incremented by 16. CWP is decremented using the SAVE instruction and incremented using the RESTORE instruction. Both of these instructions works as an

ADD insruction with the side affect of decrementing or incrementing CWP respectively. The values that are to be added resides in registers from the current context while the result is stored in registers in the new context. If there are too many nested SAVE instructions all register sets will be occupied. Since the decrement CWP is a modulo NWINDOWS operation, the output of the last context will be the input of the first context. This would cause problems so instead a trap is generated, storing the registers on a stack. Each window has a stack pointer which resides in register O6 as well as a frame pointer in I6. This means that the stack pointer of the current context is the frame pointer of the next context. The stack pointer points to the top of the stack and the frame pointer points towards the beginning of the context. Therefore [6] SAVE and RESTORE must be used when calling a subroutine if it is to use any of the registers within a register window, with SAVE residing in the beginning of the routine and RESTORE at the end.[5]

4.1.1.4 Traps

When a breakpoint is hit, a trap is generated, causing the CPU to execute a trap handling routine. This routine prepares for a call to the trap handler before entering it.

The SPARC architecture defines three different types of traps. Precise traps, deferred traps and interrupting traps. A precise trap is caused by an instruction and is generated before the instruction executes. A deferred trap is caused by an instruction but may be generated several instructions later. Interrupting traps are usually not related to a specific instruction and they tend to be caused by external signals, typically used for I/O requests.[5]

The trap caused by a hardware breakpoint is categorized as a precise trap.[7]It causes the CPU to perform the following steps:

$$ET \leftarrow 0$$

$$PS \leftarrow S$$

$$S \leftarrow 1$$

$$CWP \leftarrow ((CWP - 1) \mod NWINDOWS)$$

$$I[1] \leftarrow PC$$

$$I[2] \leftarrow nPC$$

$$TBR.tt \leftarrow \textit{trap type}$$

$$PC \leftarrow TBR$$

$$nPC \leftarrow TBR + 4$$

First traps are disabled inhibiting any interruptions to the CPU. Then the current supervisor mode is stored inside PS before changing the supervisor mode to 1. After this the register window is decremented to provide a new context. Then the values

of PC and NPC are stored providing a reference to the triggering address. The next step is to provide the trap handling routine, this is done by writing the trap type to the *tt* field inside TBR. When TBR points to the correct routine the value is written to PC. Also TBR + 4 is written to NPC pointing to the second instruction of the trap handling routine.[5]

4.1.2 LEON

This section will explain LEON's implementation of the SPARC architecture regarding the support for hardware breakpoints.

4.1.2.1 The Integer Unit

LEON's *Integer Unit* (IU) implements the integer instructions for SPARC. This IU has a seven stage pipeline, generating traps on the sixth stage. This is where the address inside the breakpoint register is compared to the address of the current instruction. This is done for every instruction. If there is a match, a breakpoint trap will be generated.

4.1.2.2 Ancillary State Register Implementation

The LEON processor provides breakpoint registers by implementing four pairs of ASRs, ASR24 to ASR31. The first register in a pair stores the address where the processor should break. The second register stores a mask. The IU will only compare the address bits corresponding to the ones set inside the mask. This is used by watchpoints to cover a range of addresses such as structures or arrays.

Besides the address field and the mask there are three control bits, *IF*, *DL* and *DS*. IF is stored inside the address registers least significant bit. It indicates that the trap should be generated on the instruction-fetch phase. This is the bit used to enable a **breakpoint** specifically. The other two control bits are specifically used for **watchpoints** and they are stored inside the mask register. DL indicates that the trap should be generated when data is loaded from the address. DS indicates that the trap should be generated when data is stored on the address. Having both of these bits set would create an *access watchpoint*, which triggers on both read and write. Having all three control bits unset would disable the breakpoint.

Besides the 8 breakpoint registers there is also a *LEON Configuration Register* which provides information about how the processor was configured. This task is assigned to ASR17. Among other things it informs the amount of breakpoints supported on the current configuration. This is stored in the *Number of Watchpoints* (NWP) bit field assigned to bit 5-7. LEON can support a maximum of 4 hardware breakpoints, however it can be configured to provide fewer.[7]

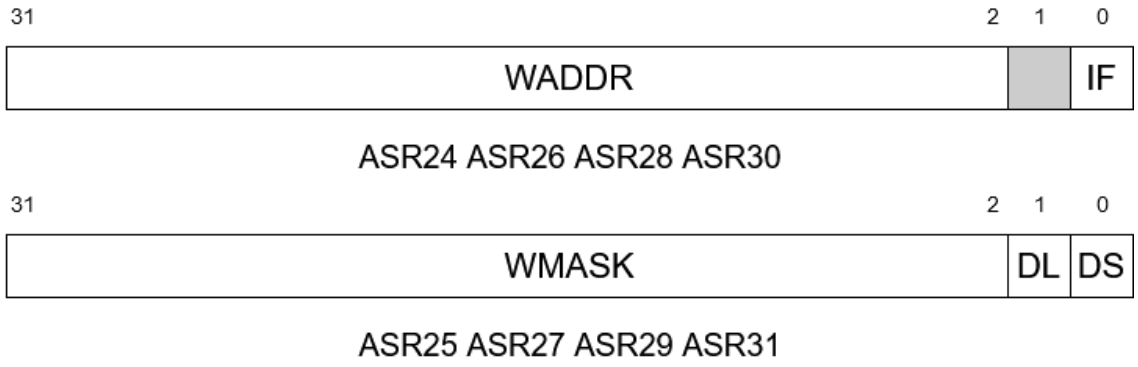

Figure 4.1: Breakpoint registers

4.2 Kernel Space Support

In this section the existing Linux kernel space support will be explained. This involves the ptrace layer and giving control to the correct trap handler.

4.2.1 Ptrace

Ptrace (Process Trace) is a system call part of the Unix system interface. It allows one process, the *tracer*, to observe and control the execution of another process, the *tracee*. It also allows the tracer to examine and update the tracee's memory. A tracee will usually be a child of the tracer, however it is possible to attach to an already running process. The tracee is in actuality not a process, but rather a thread inside the target process. This means that multi-threaded processes require one attachment per thread. The threads that are not attached are not debugged. The ptrace system call has the following signature.

```
1 ptrace(request, pid, addr, data);
```

The `request` argument specifies what ptrace should do. Some common requests are `GET_REGS` and `SET_REGS` which reads or writes to the tracees registers respectively. The `pid` argument is the thread ID of the tracee you want to call to. The `addr` and `data` arguments are pointers which are used differently by different requests.

When a process is traced it reacts differently to signals. All signals, except for `SIGKILL`, will cause the tracee to stop execution. The tracee will then notify the tracer about this handing over control to the tracer.[8]

4.2.2 Trap Handling

When examining the trap table on the LEON Linux target, it is clear that a trap handler is already implemented. However all it does is inducing a kernel panic. Most importantly the subroutines that issues control to the trap handler is already in place. These subroutines saves the current user state of the processor, known as the *trap frame*.

4.2.2.1 Trap Frame

The trap frame is a set of registers that defines a processor state. On the event of a trap, the current trap frame is saved on the stack. This will allow the CPU to recover this state after the trap has been handled. This is not to be mistaken for the thread state described in section 2.1.2. The trap frame only stores registers that are available to the user while the thread state contains the registers used by the kernel. The trap frame for a specific architecture is defined by the `pt_regs` struct. The trap frame for the SPARC 32 architecture consists of `psr`, `pc`, `npc`, `y` and `u_regs`. The first three fields are self-explanatory. The *Y register* has not been cover in this report, but is a register used for multiplication and division. `u_regs` stands for *user registers* and consists of the global registers and the in-registers.

4.2.2.2 Trap Handling Procedure

As described in section 4.1.1.4, a new context is created for the trap and the CPU jumps to the trap's table entry. The table entry for a hardware breakpoint contains the following lines of assembler code.

```
1 t_wpt:          move    %psr, %l0
2                 ba      do_watchpoint
3                 mov     %wim, %l3
```

First PSR is stored as the first local variable inside this context. Then the routine will unconditionally branch to the `do_watchpoint` label. This causes the execution of the following instructions.

```
1 do_watchpoint:  SAVE_ALL
2                 wr      %l0, PSR_ET, %psr
3                 WRITE_PAUSE
4                 add     %sp, STACKFRAME_SZ, %o0
5                 mov     %l1, %o1
6                 mov     %l2, %o2
7                 call    handle_watchpoint
8                 mov     %l0, %o3
9                 RESTORE_ALL
```

This subroutine begins with the `SAVE_ALL` macro. It is used for setting up the register window and the trap frame for the trap handler and must be included in every trap entry point that intend to call a handler written in C-code. Next the ET-bit in PSR is set, re-enabling traps. The next 3 lines passes the arguments to the trap handler. The first argument is a pointer to the stack location of the trap frame. The next two arguments are PC and NPC that were stored in local registers by the hardware. Finally PSR is stored as the last argument. When the arguments are set, the trap handler is ready to be called using the call instruction.

4.3 GDB Internals

GDB makes use of the ptrace layer to take control of another process. This process is referred to as the *inferior*.

The GDB program is divided into three primary subsystems. The user interface, symbol handling, and target system handling. The target system handling, also called the *target side*, is responsible for execution control and target manipulation. The symbol side/target side division is informal with a some overlap between the two. The scope of this report never extends beyond the target side.[9]

The source code of the target side is divided into different families. The *tdep* family (files ending with *-tdep.c*) implements debug support for specific target architecture. The *nat* family (files ending with *-nat.c*) provides support for native debugging on specific systems. The following source files are provided in both families.

- sparc-*family*.c: Implements Sparc-specific debugging
- linux-*family*.c: Implements Linux-specific debugging
- sparc-linux-*family*.c : Implement Sparc-Linux specific debugging

other files of interest :

- breakpoint.c : Implements top-level breakpoint handling
- target.h : Defines the interface of the target side to the rest of the debugger.
- target-delegate.c : Generated file, providing delegate functions for the target.

4.3.1 Breakpoint handling

The breakpoint handling in GDB is broken down to two stages. First the user issues a command setting the breakpoint. This causes GDB to initialize the breakpoint. This involves creating a breakpoint based on the user input and checking if that type of breakpoint is supported. The second stage occurs when the user decides to run the inferior. Right before the program starts, all the initialized breakpoint are inserted. When the inferior is stopped by a breakpoint hit all of the inserted breakpoints are removed. This is the method for handling hardware breakpoints as well as software breakpoint. When it comes to hardware breakpoints, it is important that the breakpoint registers are cleared when switching to a new task. GDB does nothing to ensure this, entirely relying upon the kernel to handle it. Since hardware breakpoints are architecture specific, how they are manipulated is defined by the *target*.

4.3.2 The Target Stack

In GDB a target is an interface between the debugger and the inferior. There are many types of targets including executable files, processes, threads, core dumps or architectures. GDB can have several loaded targets during a single session. All of these targets are stored in a stack of *strata*. A stratum is a boundry within the stack containing targets of a certain type. In other words, thread targets are stored in the thread stratum and process targets are stored in the process stratum. The strata are organized in decreasing importance in the stack meaning that targets in a more important stratum will always be accessed before a target in a stratum of lesser importance, regardless of which of them were placed ins stack first. At the bottom of the stack is always a dummy target, making sure the stack is never empty.

4.3.3 Target Operations

Each target has associated *target operations*. They provide support for the target which may be architecture specific. For instance, a native process target running on a SPARC Linux host will be provided a SPARC Linux target operation set. If GDB tries to call a target operation that is not implemented by the top target, GDB will continue to the next target in the stack until an implementation is found.

A target operation is called in the following manner. The caller accesses a target operation of the current target via a macro definition with the `target_` prefix. The current target points to a set of target operations that consists of auto generated delegate functions. These delegate functions calls the same target operation for the target beneath itself.The target beneath belongs to the same stratum as the current target, however it is this target that actually implements the operation. If the operation is not implemented the same delegate function will take its place. This will cause GDB to continue down the target stack until a target implementing the operation is found. If no other target provides an implementation, the dummy target will be reached. The dummy target uses a default implementation returning an error value indicating the lack of support. This return value gets returned to the caller which acts by raising an error or simply ignoring it. The call sequence with no implementation in the stack is illustrated in figure 4.2 while figure 4.3 illustrates the procedure when there is an implementation.

Inside *target.h* the `target_ops` structure is defined which contains a set of function pointers for a target. Among these operations there are a few related to hardware breakpoints.

- `to_insert_hw_breakpoint` - insert a hardware breakpoint in target
- `to_remove_hw_breakpoint` - remove a hardware breakpoint in target

These are the operations that need to be implemented in the SPARC Linux architecture specific code.

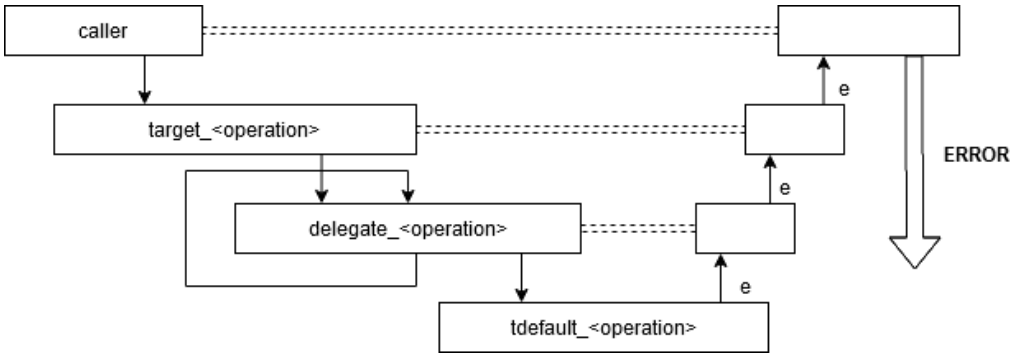

Figure 4.2: Sequence diagram for a target operation with no implementation in the target stack

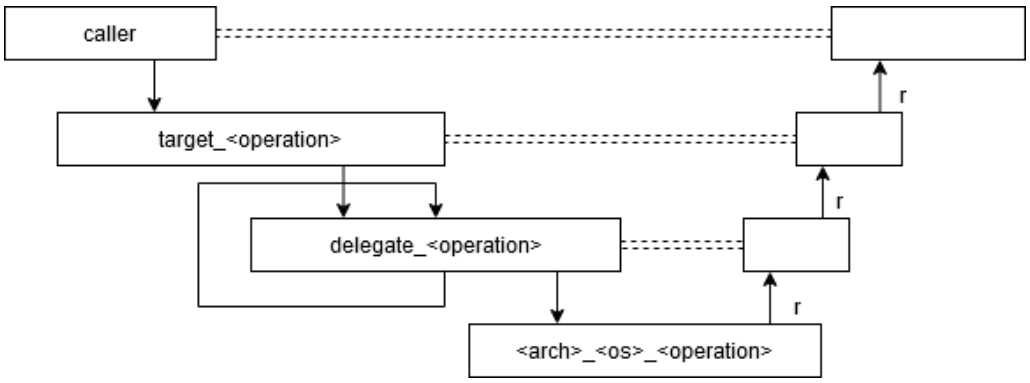

Figure 4.3: Sequence diagram for a target operation with an implementation in the the target stack

4.4 Patching the Kernel

This section describes the patch created to provide hardware breakpoint support in the kernel. This includes an implementation of the trap handler and an extension of the ptrace layer. This extension allows user space applications to insert, remove, enable and disable breakpoints as well as changing their masks. It also provides information about how many breakpoints are set, how many are supported, and their types. In addition it specifies an protocol to access all these new features. This extension is also responsible for handling the logistics of the breakpoints.

4.4.1 Implementing the Trap Handler

The role of the trap handler is to send a signal to notify a process about the trap. Signal transmissions are divided into two phases, signal *generation* and signal *delivery*. Signal generation involves editing a data structure inside the receiving process's descriptor, providing information about the upcoming signal. Delivery means notifying the receiving process of the new signal. This usually happens at the end of the signal generation. However if the receiving process is not currently running on the CPU, the kernel needs to force the process to react to the signal.[2]

As described in section 4.2.2 most of the hard work of the trap handler is already in place. The provided trap handler resides in `arch/sparc/kernel/traps32.c` and has the following signature.

```
1 void handle_watchpoint(struct pt_regs *reg, unsigned long pc,
    unsigned long npc, unsigned long psr)
```

In other words, the values of the PC, NPC and PSR registers are available through the arguments. In addition the `pt_regs` struct is the provided trap frame.

The purpose of the trap handler is to send a signal to the trap inducing process. This is done by first constructing a `siginfo_t` structure, which holds the signal information. The relevant information provided by the structure are the *signal number*, *signal code* and *address*. The signal number is set to `SIGTRAP` to indicate a breakpoint and the signal code is set to `TRAP_HWBKPT` to indicate a hardware breakpoint or watchpoint. The address is stored as a pointer to the location where the breakpoint hit. This is achieved by providing the `pc` argument as a void pointer. By calling `send_sig_info` the signal can be generated and delivered, passing the signal number and the info structure as arguments.

4.4.2 Extending the Ptrace Layer

Ptrace system calls are implemented by the architecture specific function `arch_ptrace`. For 32 bit SPARC systems this function is defined inside `arch/sparc/kernel/ptrace_32.c`. Here most of the common ptrace requests are identified using a switch statement which calls the approriat request function.

4.4.2.1 Logistics in Ptrace

The kernel handles breakpoint manipulation in two steps. First a kernel software representation is created or updated by one of the available ptrace requests. This software representation is tied to a specific task inside linux, namely the tracee. The second step happens when this task is scheduled to run on the CPU. This is when all the breakpoints tied to the task is written to the breakpoint registers on the CPU. This concept is illustrated in figure 4.4

A data structure `arch_hw_breakpoint` is defined to represent a breakpoint in the kernel. A list of `arch_hw_breakpoint` pointers is stored for each `task_struct` inside it's `thread_struct`. This makes the breakpoint list part of the thread state regardless if the owning task is a lightweight process or a traditional process. Each list can fit four pointers meaning that each task can insert four hardware breakpoints each. Throughout the coming sections there will be a distinction between a **set** breakpoint and an **inserted** one. An inserted breakpoint is created and stored inside a `thread_struct` but is not necessarily represented in one of the hardware registers. A set breakpoint is represented in a register allowing it to cause a trap. Since this solution implements user breakpoints, an inserted breakpoint will only be set if it's task is currently running. The breakpoint structure looks as follows.

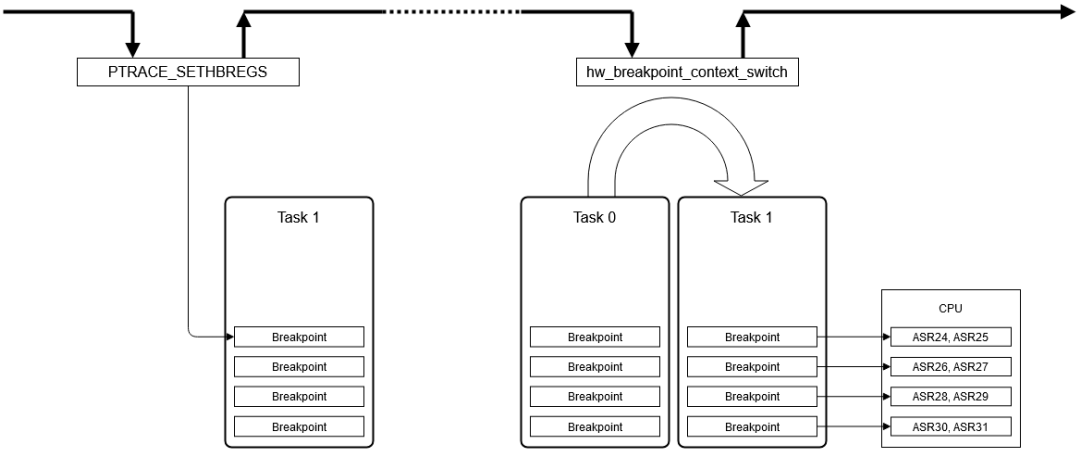

Figure 4.4: Placement of a user breakpoint

```
struct arch_hw_breakpoint{
    unsigned int address;
    enum hwbp_type type;
    int reg;
    int slot;
    int enabled;
    unsigned int mask;
}
```

The `address` and the `mask` fields contain the values that is to be written to the address and mask registers. The `reg` field defines which ASR is the address register for this breakpoint. This field is strongly connected to the `slot` field which identifies the placement of this breakpoint in the list. The slot determines which register is used, for example slot 0 uses ASR24 and ASR25 while slot 1 uses ASR26 and ASR27. The `type` enumeration can be on of four values defining what type of breakpoint it is. The type can be defined as a read watchpoint, write watchpoint, read-write watchpoint or a breakpoint. The `enabled` field simply states if the breakpoint is enabled or not.

4.4.2.2 Defining an Extended Interface

Currently the ptrace layer does not provide any way of setting hardware breakpoints for SPARC. Therefore a custom ptrace request need to be implemented. This is done by defining a new macro `PTRACE_SETHBREGS` in the header file and implementing a function `set_hbregs` in the source file. This request will be used for all of the hardware breakpoint manipulation. `PTRACE_SETHBREGS` could be defined to any unused request value, in this case it is defined to be 27. On a `PTRACE_SETHBREGS` request the data argument is used as a code to determine what should be done. The `addr` argument has different meanings depending on the code. Either it is used to indicate the address the breakpoint should be triggered by, a mask value or the slot number of the breakpoint. See table 4.1 for the protocol of the `PTRACE_SETHBREGS` request.

Code	request	meaning of `addr`
0	insert write watchpoint	watchpoint address
1	insert read watchpoint	watchpoint address
2	insert read-write watchpoint	watchpoint address
3	insert breakpoint	watchpoint address
4	change mask in slot 0	mask value
5	change mask in slot 1	mask value
6	change mask in slot 2	mask value
7	change mask in slot 3	mask value
8	remove breakpoint	slot
9	enable breakpoint	slot
10	disable breakpoint	slot

Table 4.1: Protocol for hardware manipulation requests

In addition to the actual breakpoint manipulation, ptrace need to be able to provide some information about the breakpoints. For starters it needs to be able to provide how many hardware breakpoints are supported on the machine. Ptrace also needs to be able to tell the user space how many breakpoints are currently available. Finally the user space need to be able to provide the type of any of the inserted breakpoints. These features are made available through the `PTRACE_GETHBREGS` ptrace request. Similar to `PTRACE_SETHBREGS` it uses the `data` argument to distinguish between different requests. See table 4.2 for the protocol of the `PTRACE_GETHBREGS` request. Note that retrieving the type requires a slot number to identify the correct

Code	request	meaning of `addr`
0	get support	unused
1	get available amount	unused
2	get type	slot

Table 4.2: Protocol for get-requests

breakpoint. The other two requests require no information from the caller.

4.4.2.3 Implementing the Requests

`PTRACE_SETHBREGS` and `PTRACE_GETHBREGS` are implemented with the `set_hbregs` and `get_hbregs` functions, which are called by `arch_ptrace`. To minimize cluttering inside the ptrace source file, these functions are kept short and only identifies the request based on its code. The actual work is delegated to a series of of functions from a new source file `arch/sparc/kernel/hw_breakpoint.c`. This source file is accompanied by the new header file `arch/sparc/include/asm/hw_breakpoint.h`. Each type of request calls its respective function as explained in the following table.

request type	function	arguments
insert	`hw_breakpoint_insert`	address, type
change mask	`hw_breakpoint_change_mask`	mask, slot
remove	`hw_breakpoint_remove`	slot
enable	`hw_breakpoint_enable`	slot
disable	`hw_breakpoint_disable`	slot
get type	`hw_breakpoint_get_type`	slot
get amount	`hw_breakpoint_get_available_amount`	none
get support	`hw_breakpoint_get_supported_amount`	none

In addition to the arguments specified in the table, all but `hw_breakpoint_get_supported_amount` also takes a `task_struct` specifying the associated task.

4.4.2.4 Inserting and Removing Breakpoints

Inserting a breakpoint is done with the `hw_breakpoint_insert` function. All it does is creating a new breakpoint and placing it in the thread's list. A `arch_hw_breakpoint` pointer is allocated using `kmalloc` (memory allocation for kernel space) and values are assigned to its fields. The `type` and `address` values are provided as arguments to the function. The `slot` value is gathered by iterating through the list looking for a `NULL` pointer. If one is found it's index is used as the slot number for the new breakpoint. If no available slot was was found, the ptrace request will return an error. The `reg` value is induced by simply multiplying `slot` by 2 and adding 24, since ASR24 is the first breakpoint register and each breakpoint takes 2 registers. Finally the mask is given a default value corresponding a word-size breakpoint. If an other mask is desired, the change-mask-request should be used. On success the this request returns the slot value of the new breakpoint. This value is assumed to be stored by the caller, so the breakpoint can be referrenced by other requests.

A breakpoint is removed by the `hw_breakpoint_remove` function. Here the breakpoint is gathered from the list using the provided `slot` as index. That position in the list is assigned a `NULL` pointer, then the breakpoint is deallocated using `kfree`.

`hw_breakpoint_change_mask` simply writes the value of `mask` to the `mask` field of the breakpoint at index `slot` in the list. Similarly `hw_breakpoint_enable` and `hw_breakpoint_disable` sets the `enabled` field to 1 and 0 respectively.

4.4.2.5 Context Switching

None of the previously mentioned request functions change the actual registers, they merely alter the breakpoints software representation in the kernel. The registers are only assigned values on a context switch to a task with breakpoints in it's list.

When switching to a new task, the Linux scheduler calls `swith_to` which is a macro defined by architecture specific code. SPARC has defined this as series of function calls followed by a long inline assembly function. By placing a call to

`hw_breakpoint_context_switch` it can be called every time the scheduler switches task. One of the macro's arguments is a pointer to the new task. Passing this pointer to `hw_breakpoint_context_switch` gives the function access to the right task. This function iterates through all the breakpoints in the task. If a breakpoint is a `NULL` pointer the registers associated with that slot will be cleared. if it is not it will set the breakpoint. A breakpoint is set by writing to it's address register and mask register. Using bit operations the enable-bit, associated with the breakpoints type, is set. The resulting values are written to the registers through a call to `set_asr`

4.4.2.6 Writing to Registers

`set_asr(reg, value)` writes `value` to the ASR register supplied by the `reg` value. Trying to write to a ASR register that is not ASR17 or ASR24-ASR31 will result in failure returning an error. However providing a valid register number will call one of the `set_asr` helper functions. There is one helper function for each valid register. They contain inline assembly code that writes `value` to the register. Below is the helper function for ASR26.

```
noinline void set_asr26(int value){
    __asm__ __volatile__("wr %0, %%asr26"
                         :
                         : "r" (value)
                         );
}
```

4.4.2.7 Getting Hardware Support

`hw_breakpoint_get_supported_amount` simply returns NWP bit field from ASR17. The value of the register is fetched and shifted 5 bits to the right before it's masked to only include the 3 least significant bits. The result is the number of watchpoints supported by the current processor configuration. The value of an ASR is gathered with inline assembly, similar to when writing to one.

```
noinline void set_asr17(void){
    int value
    __asm__ __volatile__("rd %%asr17"
                         : "=r" (value)
                         :
                         );
    return value;
}
```

4.4.2.8 Getting a Breakpoint Type and Number of Available Slots

`hw_breakpoint_get_type` and `hw_breakpoint_get_available_amount` are very simple functions. The former simply returns the type of the given breakpoint slot. The latter loops through the task's list counting the amount of `NULL` pointers.

4.5 Extending GDB

The GDB patch described here provides the ability to use hardware breakpoints in GDB by implementing specific target operations. It provides a way for GDB to represent and handle hardware breakpoints. However it does not fully support watchpoints.

4.5.1 Hardware Breakpoint Handling in GDB

Hardware breakpoints are represented in GDB similar to how they are represented in the kernel. However in GDB, the struct is named `sparc_linux_hw_breakpoint`. The fields represent the same things as their kernel counter parts. Some fields have been omitted since GDB does not need to know about them. The enumeration `target_hw_bp_type` is an exact match to the one used in the kernel indicating a breakpoint or any of the watchpoint types.

```
1 struct sparc_linux_hw_breakpoint{
2   unsigned int address;
3   unsigned int mask;
4   enum target_hw_bp_type type;
5   int enabled;
6   int hw_slot;
7 };
```

Just like in the kernel each inferior has a list of breakpoints. This time a list is a struct named `sparc_linux_inferior_bps` containing an array of `sparc_linux_hw_breakpoint` pointers and an identifier. This identifier specifies which inferior the list belongs to. These lists are stored globally on a vector and they are created on demand.

4.5.2 Inserting Hardware Breakpoints in GDB

The target operation `to_insert_hw_breakpoint` is used for inserting hardware breakpoints (note that it is not used for watchpoints). This operation is implemented by `sparc_linux_insert_hw_breakpoint`. Amongst it's arguments is a `bp_target_info` struct pointer named `info`. This struct stores information about the breakpoint the user wants to insert. This address where the user wants to place the breakpoint, `reqstd_address`, is found here. This address is made to align by clearing the two least significant bits, making the result a multiple of four. The corrected address then stored in `info`. Then the breakpoint list for the current inferior is retrieved. If the inferior does not have a list, one is created. This is followed by a new breakpoint struct being allocated. It is given the corrected address, the type is set to indicate a breakpoint and it is set to be enabled. Before inserting the new breakpoint, it is placed in the list. If this fails, due to the list being full, the breakpoint is deallocated and an error message is returned. If it succeedes the ptrace request for breakpoint insertion is called with the type is sent as the code.

test name	test case	result
`test_break`	place a single breakpoint	PASSED
`test_break_disable`	disable a breakpoint	PASSED
`test_break_enable`	re-enable a breakpoint	PASSED
`test_break_multi`	place two breakpoints	PASSED
`test_break_multi_same`	place two breakpoints on the same instruction	PASSED
`test_break_multi_consec`	place two breakpoints on to consecutive instructions	PASSED
`test_watch_global`	place a watchpoint on a global variable	PASSED
`test_break_local`	place a watchpoint on a local variable	PASSED

Table 4.3: Table of ptrace tests

The return value of the ptrace call is stored in the breakpoint's `hw_slot` field.

4.5.3 Removing Hardware Breakpoints in GDB

Breakpoints are removed by the `to_remove_hw_breakpoint` operation which is implemented by `sparc_linux_remove_hw_breakpoint`. It is provided the same set of arguments as `sparc_linux_insert_hw_breakpoint`. A breakpoint is removed by first creating a dummy breakpoint based on the values in the `info` field, then looking for a matching breakpoint in the inferior's list. If none is found it returns an error. However if a matching breakpoint is found, ptrace is called to remove it. This time the code is set to 8 to indicate a removal and the `address` argument is given the slot value of the matching breakpoint. Two breakpoints are said to match if both their address and their type are the same. Finally it is also removed from the list.

4.6 Testing Ptrace

The implementation of the ptrace layer is tested using manual testing. A test in this case is a C-program that uses one of the implemented features of the extended ptrace layer and validates the behavior. If ptrace behaved as expected, the test will exit with a `SUCCESS` status, if it did not it exits with a `FAILURE` status. This is a form of end-to-end testing which does not test the internal functions of the ptrace layer. This means that the tests can still pass when unintended behavior occurs inside ptrace as long as the breakpoints are hit and the inferior behaves as intended. All the test programs can be run using the `test.sh` shell script. It simply runs each test and prints their success status to the console. Table 4.3 lists all of the tests.

The following example shows the code of `test_break` and illustrates how a test works.

```
/*places a breakpoint on a function and enters the function*/
int main(void)
{
  pid_t child;
  int status;
  int stopped = 0;         //if child stopped by trap
  int finish = 0;          //if child could finish
  int slot;
  siginfo_t siginfo;

    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        raise(SIGSTOP);
        func();
        return EXIT_SUCCESS;
    }
    else {
        wait(NULL);

    slot = ptrace(PTRACE_SETHBREGS, child, func,
   INSERT_WATCHPOINT_EX);

    if(slot < 0)
      return EXIT_FAILURE;

    ptrace(PTRACE_CONT, child, NULL, NULL);
    wait(&status);
    //if stopped by signal
    if(WIFSTOPPED(status)){
      //if stopped by trap
      if(WSTOPSIG(status) == SIGTRAP){
        //remove breakpoint
        ptrace(PTRACE_SETHBREGS, child, slot, REMOVE_WATCHPOINT);
        ptrace(PTRACE_GETSIGINFO, child, 0,&siginfo);
        //if trap raised on correct address
        if(siginfo.si_addr == func)
          stopped = 1;
        ptrace(PTRACE_CONT, child, NULL, NULL);
        wait(&status);
      }
    }
    //if process finish correctly
    if(WIFEXITED(status)){
      if(WEXITSTATUS(status) == EXIT_SUCCESS){
        finish = 1;
      }
    }
    if(stopped && finish)
      return EXIT_SUCCESS;

    return EXIT_FAILURE;
  }
}
```

Listing 4.1: example of test

In order to pass the test, the tracee must first signal that it was stopped, then it must signal that it finished correctly. After the `fork` the child will call ptrace allowing it to be traced, making it the tracee. Then `raise` is used to send a `SIGSTOP` signal to itself. Since the process is traced the signal will stop the child and notify the parent. The parent then places a breakpoint, specifying `func` as the address. At this point the test checks if ptrace sent back an error, in that case the test failed. Otherwise it tells the child to continue while waiting for the next signal, this time the signal status is stored in `status`. The child continues by calling `func`, this should cause a breakpoint hit, sending a `SIGTRAP` signal to the child. This signal would cause a core dump but since the process is traced the signal will notify the parent and stop the child. The parent proceeds by checking if the child was stopped by a breakpoint trap, passing the first part of the test. If so the child is allowed to continue and the parent once again waits for the next signal. The child has nothing left to do so it simply finishes with a `EXIT_SUCCESS` status. This notifies the parent again. The parent checks the exit status of the child, if the child exited successfully as well as stopped correctly the test has passed. This is one of the most basic test though all the other tests are quite similar and follows the same formula.

5

Discussion and Conclusion

5.1 Method

5.1.1 TSIM vs Hardware

The implementation provided here has been developed and tested on TSIM. Due to the COVID-19 pandemic work had to be conducted remotely, removing the possibility for testing on real hardware. At the time of writing, TSIM has limited support for hardware breakpoints. The breakpoint mask registers only allows word-sized masks. This meant that only word-sized breakpoints and watchpoints could be tested and the change mask-request remains untested. In addition the DL bit was not functioning correctly, disallowing any writes to it. For this reason read watchpoint could not be tested. Therefore the test case that checks watchpoints only covers write watchpoints.

5.1.2 Testing and Validation

Testing the kernel is not as straight forward as testing a user space application. There are a few test frameworks available but none seemed to fit the needs of this project. KUnit is a unit test framework for the Linux kernel. However it was introduced in Linux version 5.5 and is not available in version 4.9 used in this project. KMemleak is a tool that checks for memory leakage in the kernel. This is available to Linux 4.9 but was not used due to lack of time. Additional tests were conceived but never completed. These includes ptrace requests with invalid arguments and a few edge cases.

5.2 Implementation Discussion

5.2.1 User Breakpoints Implementation

The purpose of the project was to bring hardware breakpoints to the user space. Therefore only user breakpoints were implemented and kernel breakpoints were deemed to be outside of the project's scope. A user breakpoint is tied to a specific task this makes sure that only the process you expect will trigger a breakpoint. However an added benefit is that more breakpoints becomes available. Instead of

sharing the breakpoint registers between all the processes, each process gets exclusive access to the breakpoint registers when they are running. In this project It also resulted in great simplifications in the code minimizing redundancy by differentiating breakpoint insertion from setting the registers.

5.2.2 Performance

However there is one drawback from tying breakpoints to their task. On each context switch all the breakpoint registers are updated. Since context switches occur very frequently this could have an impact on performance. This is ineffective and undesired since one of the proclaimed benefits of hardware breakpoints is that it does not impact performance. In order to minimize this effect the patch tracks how many breakpoints are inserted in each task and how many are set in the hardware in total. This allows the kernel to omit breakpoint updates for task switches that do not require it. However this introduces a shared variable in the kernel that are used by several processes.

5.2.3 Slot addressing

In the extended ptrace interface the breakpoints are addressed via their index in the task's list, which is directly tied to breakpoints hardware register. This requires the user space to store the index of each inserted breakpoint. It could be argued that this does not provide enough of an abstraction to the hardware. however it is a much simpler solution than addressing breakpoints by their addresses since they are not necessarily unique.

5.2.4 Other Architectures

The Linux-kernel has an internal performance event infrastructure that provides some levels of abstractions to hardware performance events. This is used mostly for performance profiling but it also supports hardware events such as hardware breakpoints.[10] This infrastructure is commonly used in hardware breakpoint implementations in architectures such as x86 and ARM. However the performance event infrastructure is dependant on support for atomic64 types. This support is not currently provided by by the SPARC 32 bit linux build. Therefore this could not be used and a more specific solution was implemented.

5.3 Future Work

5.3.1 Shared Resource and Race Conditions

The shared resource described in section 5.2.2 need to be protected either by using locks or atomic instructions. Alternatively another way of minimizing the amount of times the breakpoint regisers are updated need to be introduced.

5.3.2 Memory leakage

The provided patch allocates breakpoint structures dynamically. Inserting and removing will allocate and deallocate a breakpoint respectively. However breakpoints are never deallocated after the process has exited. This most likely leads to memory leakage within the kernel. KMemleak could be used to detect if this is an issue, if so a fix should be implemented.

5.3.3 Differentiating Watchpoints

On the event of a breakpoint trap, the LEON processor does not provide any indication of what type of breakpoint or watchpoint was hit. This information is sometimes necessary to provide proper watchpoint support inside debuggers. The trap handler could perhaps be made to differentiate between breakpoints and watchpoints by looking at what instruction caused the trap. For instance if the trap was generated by a read or write instruction the trap is assumed to be a watchpoint. One problem with this is that a breakpoint can might aswell be placed on a read or write instruction, which would cause the handler to believe it to be a watchpoint. A simpler solution would be for the hardware to indicate the type of the latest hit breakpoint in one of the status registers or to have separate trap types for breakpoints and watchpoints.The x86 architecture does something similar to this where *debug register 6* indicate which specific breakpoint was hit.[11] This allows the x86 trap handler to send this information to the trap inducing process.

5.3.4 Future GDB improvements

The current GDB patch does not provide hardware watchpoint support. Watchpoint handling requires several more target operation than hardware breakpoints. These were not successfully implemented in time. Part of the reason for this is that GDB need to distinguish watchpoints from breakpoints or they will not be detected.

5.3.5 Future Testing

The current tests cases covers the basic features of the hardware breakpoint support in the ptrace layer. Additional testing will have to be made before submitting this patch to mainline Linux. This involves more test cases, testing with multiple processor cores and multiple processes and kernel threads and more.

5.4 Sustainability and Ethics

5.4.1 Hardware Breakpoints for Embedded Systems Applications

Embedded systems are becoming more prevalent in society, especially considering the rise of Internet of Things. The LEON processor can be used in a wide range

of embedded applications but is most recognized for it's use in aerospace applications. These applications often provides services that are critical to infrastructure and defense as well as environmental and geological research, which often requires a high level of software correctness. Therefore it is crucial that there are proper tools for debugging these systems. Hardware breakpoints can be useful while debugging embedded software, since they can be placed on ROM. In addition the performance cost of software breakpoints could be problematic in applications with though timing constraints, necessitating hardware breakpoints. Furthermore many embedded systems interacts with it's environment via I/O ports. In these applications watchpoints can be highly valuable for debugging. Since software implementations of watchpoints has significant performance costs they are rarely used, especially not when timing constraints are involved. Previously a hardware debugger would be required to use hardware breakpoints or watchpoints on the LEON processor running Linux. The provided patches allows hardware breakpoints to be set by GDB, making the feature available for a wider range of developers and hopefully contributing to more reliable software.

5.4.2 Ethics of Open Source

The Linux kernel and GDB are both open source projects. There is a constant debate regarding the ethics of open source, especially in critical software. The most common critique being the difficulty to maintain quality and consistency. However some argue that bugs can be fixed more quickly if there is a large open source community involved. Nevertheless the criticism is important to bare in mind when patching an open source project. One important principle is to change as little code as possible so you don't break something without realizing it. This means only ever touch architecture specific code unless you really know what you are doing. This is especially true for the Linux kernel which is used extensively in various applications. It also means that patches need to be thoroughly tested before it is submitted to the mainline repository.

5.5 Conclusion

The goals established in this report where partly met. The Linux kernel patch provides support for hardware breakpoints and watchpoints but lack a way of distinguishing between the two on trap generation. The GDB patch illustrates that the kernel patch is useful for placing breakpoints but has also exposed the need for differentiation between watchpoints and breakpoints in the kernel. Placing hardware breakpoints in GDB works as intended but watchpoints do not. The extended ptrace layer passes a few simple tests but further testing is needed.

Bibliography

[1] J. Gaisler, J. Andersson, and R. Weigand, "Next generation multipurpose microprocessor," in *Data Systems in Aerospace (DASIA)*, 2010.

[2] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[3] C. D. Udma, "Chapter 16 - software development tools for embedded systems," in *Software Engineering for Embedded Systems* (R. Oshana and M. Kraeling, eds.), pp. 511 – 562, Oxford: Newnes, 2013.

[4] P. Krishnan, "Hardware breakpoint (or watchpoint) usage in linux kernel," in *Ottawa Linux Symposium*, 2009.

[5] SPARC International Inc., *The SPARC Architecture Manual Version 8*.

[6] R. P. Paul, *SPARC Architecture, Assembly Language Programming, & C*. Englewood Cliffs, USA :Prentice Hall, 1994.

[7] Cobham Gaisler, Gothenburg, Sweden, *CGRLIB IP Core User's Manual*, 2019.

[8] M. Kerrisk, *Linux Programmer's Manual*. Linux, 2019.

[9] GNU, *GDB Internals Manual*, 2018.

[10] J. Kukunas, *Power and Performance*. Elsevier Science & Technology, 2015.

[11] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3*, 2016.