ABACUS Implementation

Softcore CPU Profiling with CVA5 and Linux

Author: Rajnesh Joshi

Fall 2024

Abstract

This report reviews the design and process behind creating an implementation of a CPU profiler which follows the ABACUS (hArdware Based Analyzer for the Characterization of User Software) framework [1]. The CPU profiler in this report profiles the five-stage Linux-capable RISC-V soft core CPU, CVA5 [2]. Interfacing and usability details are covered for individuals seeking to profile baremetal software, or userspace software running on an operating system with Linux kernel device drivers.

Contents

Chapter 1

Introduction

The ability to characterize and monitor the performance of software allows designers to gain feedback on the impact of their software or hardware design on software execution performance. For example, in software, the difference in the order of a nested for-loop might result in a series of either cache hits or misses depending on the programming language [3] (i.e., whether the array in cache is stored in row-major or column-major). In hardware, different branch prediction algorithms or cache replacement policies might result in completely different software performance depending on the pattern of software.

Software profiling provides information on the performance of software, giving designers the understanding of which hardware or software design parameters might need to be changed to increase the performance of their software. Different profiling methods exist, each with their own inherent limitations such as [4]:

- Software-based Simulation: Allows for a large visibility, providing a number of different metrics about the performance, but at a great slowdown.
- Software profiling: Associates certain software code segments to certain hardware events but also slows done application execution.
- Hardware-based performance counters: Hardware counters which collect details on certain interactions such as `hpmcounter` in RISC-V. Dependent on the microarchitecture, and ISA, limiting the scope of visibility and potential for growth in collecting new desired information.

Instead of relying on the previous aforementioned options, dedicated hardware units profile software running on a CPU core with lower overhead compared to software-based profiling solutions [4]. Hardware units with software configurability may be used for data collection on important system events such as cache misses, branch mispredictions, and memory read and writes.

ABACUS, hArdware Based Analyzer for the Characterization of Userspace Software, provides a framework that defines a generic profiler which collects data about the processor through debug nets, and provides an interface for software to access the profiler's collected data, and even control the profiler.

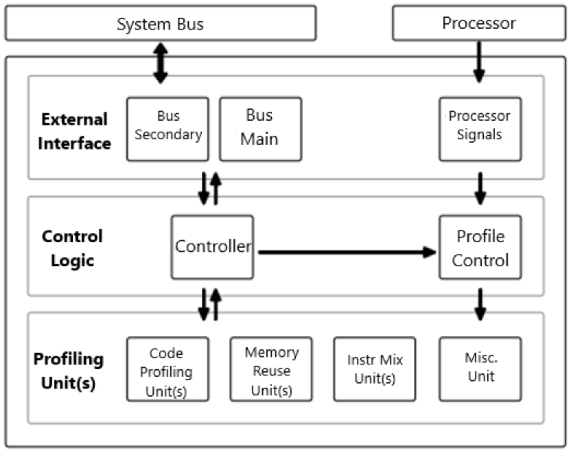

Figure 1.1: Example ABACUS CPU Profiler

The goal of this paper is to present the design of an ABACUS framework adherent CPU profiler which profiles the OpenHW CVA5 RISC-V softcore processor on an FPGA using LiteX. Usability information for profiling baremetal code, and userspace software in Linux with a character device driver that allows for a simple API to use ABACUS will be provided.

Chapter 2

Background

2.0.1 LiteX

LiteX is an open-source System-on-Chip (SoC) builder, and IP library that can be used to create FPGA designs [5]. One can liken it to be an open-source version of similar tools provided by FPGA vendors such as AMD Vivado, except with no GUI support.

The LiteX ecosystem and community has demonstrated support for seamless integration of several different existing RISC-V softcore processors, as well as, OpenSBI (an open-source Supervisor Binary Interface and bootloader) and Linux driver support for their IPs such as LiteUART.

2.0.2 RISC-V and CVA5

RISC-V is an open-source Instruction Set Architecture (ISA) based on Reduced Instruction Set Computing (RISC). An ISA defines "aspects such as the supported data types, which instructions the processor is able to execute, the registers and how the hardware manages main memory," and being RISC entails that the CPUs are "designed to execute really basic instructions, relying on simple and efficient hardware architecture" [6].

There are a number of open-source RISC-V processors such as BlackParrot, VexRISCV, and OpenHW CVA5. CVA5 is a 32-bit processor which supports RV32IMAD (32-bit RISC-V instruction set with multiplication/division, atomic instructions, and double-floating point operations) [7], and supervisor and user-level privilege for operating system support.

Chapter 3

Profiler Design

This section outlines how the presented CPU Profiler implementation was designed. *Figure 3.1* shows the high-level design.

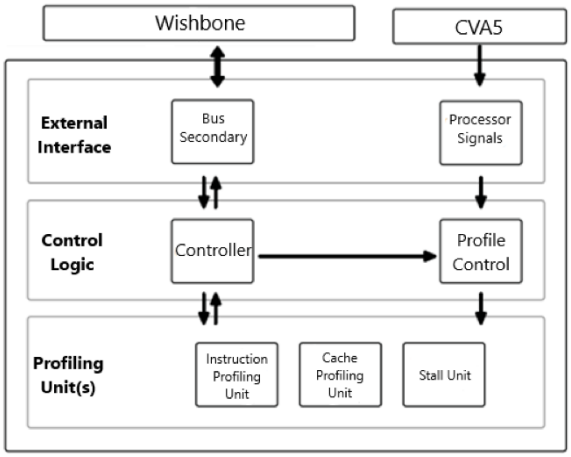

Figure 3.1: Implemented ABACUS CPU Profiler

In contrast to *Figure 1.1*, the **System Bus** and interfaced **Processor** is explicitly defined to be Wishbone and CVA5 respectively. The **Profiling Units** are also explicitly defined. The **Bus Main** interface is not implemented as Direct-Memory-Access (DMA) is not required since the amount of data collected in the tracing by the implemented profiling units is not large enough to warrant offloading it to memory.

To briefly break down *Figure 3.1*, beginning at the top, CVA5 interfaces with the profiler in two ways. First, the processor must provide its nets to the CPU profiler which is where the arrow that leads into the *Processor Signals* block comes from. These nets are used for profiling. The other interface is the interface that software will use to obtain data from the profiler by accessing its registers via bus read and writes with the CPU as the main, and the profiler as the secondary.

Beneath the top ("External Interface") layer is the middle layer which contains the "control" logic of the CPU profiler. In our implementation, this is effectively a few registers that software may access to enable or disable certain profiling units which each respectively profile different characteristics of the CPU.

At the bottom layer is the *Profiling Units*, which take certain signals from the CPU and use them to calculate values that are relevant for the information they are trying to profile. For example, the *Cache Profile Unit* takes signals that are derived from the instruction- and data cache, and calculate the number of hits, misses and requests, as well as the amount of clock cycles it takes to refill a cache line after a miss, and store this data to registers which software can read from the bus interface.

3.1 Top-level and Bus Logic

The top-level hosts the external interfaces, the registers that software will write to for controlling the profiling units, and the registers it will read from for obtaining the profiling data. The top-level instantiates all of the profiling units. The supported bus interface in this implementation is Wishbone.

3.2 Profiling Units

The profiling units do the analysis and computation based on the incoming nets from the CPU, and provide results to the registers in the top-level for software to read. The implementation of the ABACUS profiler in this paper includes three profiling units, *Instruction Profiling Unit*, *Cache Profiling Unit*, and *Stall Unit*. The choice for choosing these units was made by referencing example profiling units covered in ABACUS literature [4], and by surveying the current maintainers and developers of CVA5 at SFU's Reconfigurable Computing Lab to know what information about the CPU's performance that the profiler could provide at run-time would be useful for their work.

3.2.1 Instruction Profiling Unit

The instruction profiling unit samples issued instructions, and sorts them into bins based on the type of instruction that was issued. Registers store the frequency of each instruction type to provide information on the frequency of certain instructions that are executed in a program. It is important to emphasize that only issued instructions are sampled, since not every instruction that is fetched from cache or memory is executed if a branch condition, a flush, or anything that changes program flow occurs.

Instruction Type	**Covered Instructions**
Load	LB, LH, LW, LBU, LHU, LUI
Store	SB, SH, SW
Addition	ADD, ADDI, AUIPC
Subtraction	SUB
Branch	BEQ, BNE, BLT, BGE, BLTU, BGEU
Jump	JAL, JALR
System Privilege	ECALL, EBREAK
Atomic Instructions	LR.W, SC.W, ...

Table 3.1: Instruction Categories covered by the Instruction Profiling Unit

3.2.2 Cache Profiling Unit

The cache profiling unit keeps track of the number of cache requests, hits and misses, and measures the amount of clock cycles it takes to replace a line in the cache after a miss. This allows for a measure of memory reuse, and direct comparison of the performance provided by different cache replacement policies. Profiling support is available for both data- and instruction caches.

3.2.3 Stall Unit

The stall unit tracks branch and RAS (return address stack) mis-predictions, as well as causes of stalls in the issue stage of the CPU pipeline. This allows for better data collection on which branch prediction algorithms perform better on which software patterns, and understanding what causes the most stalls to the issue stage. The causes of stalls to the issue stage that are recorded by this profiler include:

- `No Instruction`: No instruction was ready to be issued.
- `No ID`: CVA5's issue block assigns an ID to any instruction that enters the issue stage, and if it runs out of IDs to assign, then this will cause a stall.
- `Flush`: The CPU pipeline was flushed for a certain reason.
- `The issue unit was busy`: The issue unit was occupied and could not issue a new instruction.
- `Operands were not ready`: An operand of an instruction still needed to be computated, and thus was not ready (i.e., data dependency).
- `Hold`: Generic causes: Instruction page faults, invalid fetch addresses, invalid instructions, etcetera.
- `Multi source`: Stall occurrences due to multiple aforementioned stall causes.

3.3 Memory Map

Top-level registers beginning at `ABACUS_BASE_ADDRESS`.

Register	Offset	Access
Instruction Profile Unit Enable	0x04	R/W
Cache Profile Unit Enable	0x08	R/W
Stall Unit Enable	0x0c	R/W

Table 3.2: Profiling unit enable registers

Instruction Profile Unit registers beginning at `ABACUS_BASE_ADDRESS` + 0x100.

Register	Offset	Access
Load Counter	0x000	R
Store Counter	0x004	R
Addition Counter	0x008	R
Subtraction Counter	0x00c	R
Branch Counter	0x010	R
Jump Counter	0x014	R
System Privilege Counter	0x014	R
Atomic Instruction Counter	0x01c	R

Table 3.3: Instruction Profile Unit Registers

Cache Profile Unit registers beginning at `ABACUS_BASE_ADDRESS` + 0x200.

Register	Offset	Access
ICache Request Counter	0x000	R
ICache Hit Counter	0x004	R
ICache Miss Counter	0x008	R
ICache Line Fill Latency Counter	0x00c	R
DCache Request Counter	0x010	R
DCache Hit Counter	0x014	R
DCache Miss Counter	0x018	R
DCache Line Fill Latency Counter	0x01c	R

Table 3.4: Cache Profile Unit Registers

Stall Unit registers beginning at `ABACUS_BASE_ADDRESS` + 0x300.

Register	**Offset**	**Access**
Branch Misprediction Counter	0x000	R
RAS Misprediction Counter	0x004	R
Issue stage, no instruction Counter	0x008	R
Issue stage, no ID Counter	0x00c	R
Issue stage, flush	0x010	R
Issue stage, unit was busy	0x014	R
Issue stage, operands not ready	0x018	R
Issue stage, hold	0x01c	R
Issue stage, multi-source	0x020	R

Table 3.5: Cache Profile Unit Registers

Chapter 4

System-on-Chip Design

4.0.1 LiteX

LiteX is an open-source SoC tool that is free to use. CVA5 is already integrated into the LiteX ecosystem making it fairly easy to use in SoC designs already. LiteX uses MiGen, a Python-based HDL, to connect HDL modules like CVA5, ABACUS, PLIC or CLINT which are written in System Verilog, with their own IP cores like LiteUART, LiteDDR, and so on. Software support via the BIOS, and drivers for several different IP cores are provided by LiteX. LiteX invokes Vivado when doing synthesis, placement and routing when building bitstreams for Xilinx boards.

Designing with CVA5 on LiteX

To begin designing with LiteX, follow the README tutorial on Github for downloading and setting up LiteX locally. Furthermore, to bring up Linux on CVA5 using LiteX, please refer to the README at this Github repository.

Adding an IP to the CVA5 LiteX SoC

For SoC designs in LiteX which use CVA5, the portion of the top-level design which instantiates CVA5, and user-made IP cores is defined in: `/litex/litex/soc/cores/cpu/cva5/core.py`. This file instantiates CVA5 (which is technically in a wrapper which you can find in `pythondata-cpu-cva5/.../system_verilog/examples/litex/litex_wrapper.sv`)

To explain how to add a new custom IP to the SoC, connect a signal between it and CVA5, and add it to CVA5's Wishbone peripheral bus, we provide a dummy example (Dummy IP).

1. Create signals that will connect Dummy IP and CVA5. Let's suppose the only signal between the two is a 32-bit *dummy_signal*

```
1         dummy_signal = Signal(32)
```

Listing 4.1: Declaration of signal between CVA5 and Dummy IP

2. Update the CPU parameters dictionary to assign the newly created signal to one of its ports. The port name will be on the left-hand of the assignment, and must be prefixed with either **o_** or **i_** based on directionality. Note: We are not creating a new port, this

port is assumed to already be defined in the top-level entity of the CVA5 wrapper, we are only connecting it to a signal.

```
        self.cpu_params.update (
            o_dummy_input = dummy_signal )
```

Listing 4.2: Connect signal to top-level port on CVA5 wrapper

3. Define a new interface to the Wishbone bus.

```
self.testbus= testbus = wishbone.Interface(data_width=32,
   address_width=32, addressing="byte")
```

Listing 4.3: Create new Wishbone interface with LiteX python handle

3. Instantiate IP core, and assign dummy signal to one of its 32-bit ports (we assume it has only one). LiteX will search for files in its compile order for one with an entity titled "top_level_entity_name" for reference of this IP, be sure it exists.

```
self.specials += Instance("top_level_entity_name",
i_dummy_signal = dummy_signal)
```

Listing 4.4: Instantiate Dummy IP and wire dummy_signal to its single input

3. Add the Dummy IP core as a bus secondary with the address of *test_base*.

```
soc.bus.add_slave("test", testbus, region=SoCRegion(origin=self
   .test_base, size=0x1_0000, cached=False))
```

Listing 4.5: Add Dummy IP core as bus secondary to the CPU

One thing to note is that we inform LiteX where the file of the IP Core we are adding is located so it knows where to find it.

```
    @staticmethod
    def add_sources(platform):
        cva5_path = get_data_mod("cpu", "cva5").data_location
        with open(os.path.join(cva5_path, "tools/compile_order"),
            "r") as f:
            for line in f:
                if line.strip() != '':
                    platform.add_source(os.path.join(cva5_path,
                       line.strip()))
        platform.add_source(os.path.join(cva5_path, "examples/
           litex/litex_wrapper.sv"))
        platform.add_source(os.path.join(cva5_path, "/localhome/
           dummy_folder/DummyIP.sv"))
```

Listing 4.6: Add Dummy IP Core source file for LiteX to reference (line 9)

4.0.2 Vivado

A functionally equivalent designed may be completed in the Vivado Block Diagram design flow provided the extension of the Profiler with a functional AXI4 or AXI4-Lite bus interface as interfacing with Xilinx IP Cores like AXI Interconnect relies on this interface. From a high-level, a block diagram that implements this profiler with CVA5 may look like Figure 4.1.

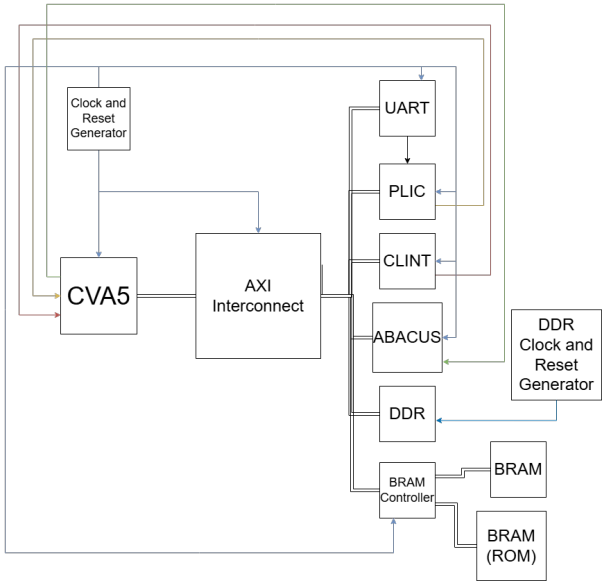

Figure 4.1: Vivado Block Diagram of ABACUS-CVA5 setup

- `Clock and Reset Generators`: Adds the free-running clock on the evaluation board into the FPGA design, and sources for different reset signals.
- `AXI Interconnect`: Routes transactions from the main (CVA5) to its secondary peripherals (PLIC, CLINT, ABACUS, UART, etc.), enabling the CPU to control them via software.
- `CVA5`: Modified-CVA5 core with profiling nets exposed to the top-level and wrapped in an IP.
- `ABACUS`: ABACUS CPU profiler that takes the nets from CVA5, and acts as an AXI secondary interface to the CPU via the AXI Interconnect, allowing the CPU to use software to control it.
- `CLINT`: Core-Local Interrupt Controller. A RISC-V specification-defined interrupt controller for **internal** interrupts, these are:
  - Timer interrupts (example use-case: operating system process time slicing)
  - Software interrupts (interrupts from software running on separate cores for multicore setups, not relevant for this design since it is single-core)
- `PLIC`: Platform-Level Interrupt Controller. A RISC-V specification-defined interrupt controller for **external** interrupts, such as keyboard presses. This is why it has a connection with the UART.
- `UART`: Allows the CPU to send data out and receive data from the PC. Special drivers (LiteUART.c) are written so the CPU knows how to use this.
- `DDR Controller`: Enables DDR access. Be sure to check if the evaluation board available has DDR.
- `DDR Clock and Reset Generator`: Separate clock and reset for DDR since it runs in a different clock domain.

- `BRAM (ROM)`: Holds the initial code that will be run (i.e., the PC at base address 0x0000 0000, which is the `bios.bin`).
- `BRAM`: Holds memory space for the initial BIOS code to read and write to during execution prior to initializing the DDR).

Chapter 5

Resource Utilization

On the Xilinx VCU118, with a clock frequency for the LiteX generated design of 125MHz, the resource utilization is outlined in the *Figure 5.2*. It may be possible for the design to be clocked higher.

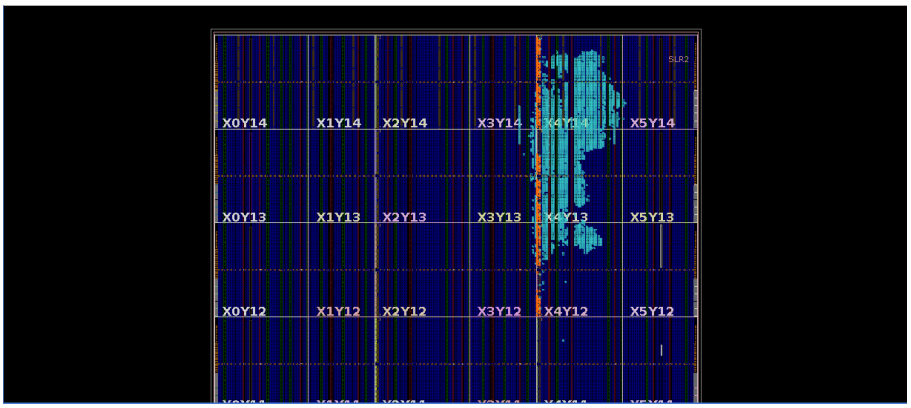

Figure 5.1: Placement of design on PL fabric

Resource	Utilization	Available	Utilization %
LUT	13497	1182240	1.14
LUTRAM	2422	591840	0.41
FF	9839	2364480	0.42
BRAM	22.50	2160	1.04
URAM	2	960	0.21
DSP	4	6840	0.06
IO	128	832	15.38
BUFG	5	1800	0.28
MMCM	1	30	3.33

Figure 5.2: Resource Utilization Summary

A sizeable portion of the LUT and FF utilization reflects the arithmetic operations (incrementing of counter registers), and register instantiations. The relatively high IO usage is due to the top-level SoC wrapper LiteX generates having several ports including some to the DDR, some to the LEDs, UART (rx/tx/rts/cts), clock, and reset.

Chapter 6

User Manual

ABACUS is meant to be used for profiling software. Software can be ran in baremetal or above the abstraction layer of an operating system, and ABACUS supports profiling both.

6.0.1 Baremetal

Read and writes to the ABACUS internal registers were done by referencing and dereferencing their addresses with pointers. In baremetal, we have access to physical memory, and since we are aware of the addresses of the registers as per the memory map, we know which addresses to point to in order to access whichever register is of interest within ABACUS. For example:

```
volatile unsigned int* INSTRUCTION_PROFILE_UNIT_ENABLE = (
   volatile unsigned int*)(ABACUS_BASE_ADDR + 0x04);

int enable_instruction_profiling(void) {
    *(INSTRUCTION_PROFILE_UNIT_ENABLE) = (unsigned int) 0x1;
    return (*(INSTRUCTION_PROFILE_UNIT_ENABLE) == 0x1);
}
```

Listing 6.1: Enabling instruction profile unit in baremetal

It would be of interest to note how the CPU understands that accesses to certain portion of its address space translate into bus transactions instead of requests to memory. In CVA5 at least, a portion of the data cache is **non-cacheable** and this is reserved for bus peripherals. Any access to this region of memory translates to a bus transaction to a peripheral device on the bus that may have that address.

```
        # CPU Instance.
        self.cpu_params = dict(
            # Configuration.
            p_RESET_VEC      = 0,
            p_NON_CACHABLE_L = 0x80000000, # FIXME: Use
               io_regions.
            p_NON_CACHABLE_H = 0xFFFFFFFF, # FIXME: Use
               io_regions.
            # Clk/Rst.
```

```
            i_clk = ClockSignal("sys"),
            i_rst = ResetSignal("sys"),
        )
```

Listing 6.2: Declaration of non-cacheable regions in CVA5 (core.py)

Volatile in the above code is used to inform the compiler that the value of this register may change by external means (in other words, the profiling unit) so it isn't optimized out by the compiler [8].

6.0.2 Linux

Userspace software in Linux does not have access to physical memory, and instead uses virtualized memory. To interact with bus peripherals with software running on an OS, kernel drivers are needed since userspace software doesn't have the privilege to translate physical address regions into virtual address regions. Do note that the kernel space is virtualized but has more privileges than user space software.

Two options are available: the first is static kernel drivers, which are compiled with the kernel itself, and activated using device-tree bindings. This would be a more grueling process since it is time-intensive to re-compile the entire Linux kernel. There is merit behind designing static kernel drivers. For example, some drivers need to be initialized for the kernel to even boot, such as drivers for the UART device so the print output can be seen on the terminal. This is what is done for the LiteUART drivers as they are defined in the device-tree bindings, which informs the kernel of the devices in the platform.

```
            liteuart0: serial@f0001000 {
                compatible = "litex,liteuart";
                reg = <0xf0001000 0x100>;
                interrupts = <1>;
                status = "okay"; };
```

Listing 6.3: Liteuart DT binding

Instead of static kernel drivers, dynamic kernel drivers that extend the running base Linux kernel at run-time are more appropriate. The first part of writing this kernel driver is to map the physical address region of ABACUS to the virtualized kernel space to allow the kernel to interact with it. This is done using `ioremap` [9]. For example:

```
static void __iomem *abacus_base;

static int device_open(struct inode *inode, struct file *file) {
        abacus_base = ioremap(ABACUS_BASE_ADDR, 0x1000);
```

Listing 6.4: Mapping of physical address region to virtual address space

The above code maps 4KB of physical address space to virtual address space, and returns the pointer to `abacus_base`. Register read and writes may now continue as before like in baremetal with kernel function handles like `ioread32`, and `iowrite32`. Output of an example loadable kernel module when loaded into the kernel via `insmod abacus.ko` may look like the following in *Figure 6.1*.

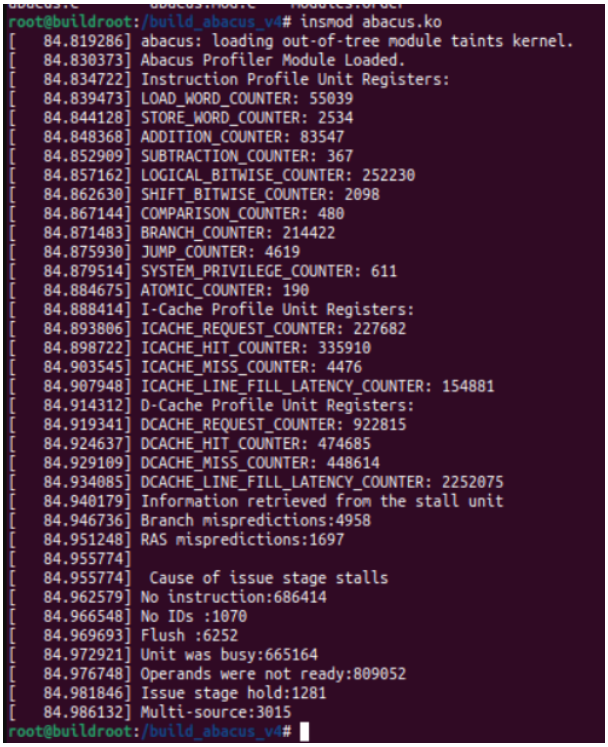

Figure 6.1: Loadable Kernel Module Output

The loadable kernel module in *Figure 6.1* runs completely in kernel space (hence why all the outputs are written to the DMESG). To provide userspace software with a method of accessing ABACUS to profile itself, kernel drivers must be written as a character device driver. This way the userspace software can open the device driver, and use its file descriptor to issue read and write operations which the driver handles. Without diving into the details of writing such drivers as literature is available on the topic online, two important functions are important to mention that are useful for designing such drivers: `copy_from_user` and `copy_to_user`. These functions copy buffers between user-space and kernel-space and vice versa, providing the necessary interface for user-space software to interact with the driver.

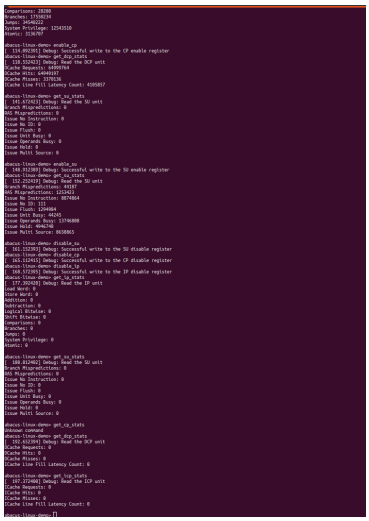

Figure 6.2: Userspace program leveraging ABACUS kernel drivers

6.0.3 Using the Linux Drivers

Insert the loadable kernel module to extend the base kernel at runtime with `insmod abacus_kernel_driver.ko`

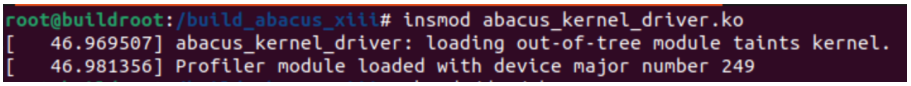

Figure 6.3: Abacus kernel driver loaded

The kernel module driver is written to have a dynamically assigned major number by the kernel. Populate **/dev** with the character device driver, **/dev/abacus**, with `mknod /dev/abacus c 249 0` (c for character device, 249 is major number, 0 is minor number).

The example userspace code in *Listing 6.5* uses the kernel driver to enable the instruction profiling unit, and get instruction profiling data.

```
#define DEVICE "/dev/abacus"

void enable_ip(int fd) {
    char cmd[16] = "enable_ip";
    write(fd, cmd, strlen(cmd) + 1); //string literal has
       automatic null character at end, but strlen returns the
       size of char[] not including the null character
}

void get_ip_stats(int fd) {
    char buffer[512] = "get_ip_stats";
    read(fd, buffer, sizeof(buffer));
    printf("%s\n", buffer);
}

int main() {
    int fd = open(DEVICE, O_RDWR);
    if (fd < 0)
        print("Could␣not␣open␣device\n");

    enable_ip(fd);
    get_ip_stats(fd);
}
```

Listing 6.5: Userspace code example

The remaining commands to read or write to the device in the API are shown in *6.1*.

enable_ip	write
disable_ip	write
enable_su	write
disable_su	write
enable_cp	write
disable_cp	write
get_ip_stats	read
get_su_stats	read
get_icp_stats	read
get_dcp_stats	read

Table 6.1: Provided commands, and file descriptor operation to use

Final Note for Readers Implementing Kernel Drivers for Bus Peripherals (such as ABACUS) on CVA5

The Linux image used by CVA5 was generated using Buildroot, a free and open-source tool for creating custom Linux distributions. The configuration of this image is quite similar to that of the Linux-On-Litex-VexRiscv project, with one important distinction: in the `linux.config` file, `CONFIG_MODULES=y` is set, which enables support for loadable kernel modules in the kernel.

It's important to note that Buildroot does not support adding a compiler to the target image. As a result, cross-compilation is necessary to place software onto the Linux image, typically using a root filesystem (rootfs) overlay. Similarly, kernel drivers must also be cross-compiled against the **exact kernel headers** of the built Linux image. Additionally, it is crucial to use the same toolchain that was used to build the image (which Buildroot creates unless told otherwise). Failure to do so will result in kernel modules that do not work. Finally, this may be unique to the version of Linux kernel that was compiled for this project, but a special C-flag was necessary in the Makefile for the kernel module [10], otherwise one may see the following linkage error: `Unknown relocation type 57`.

Chapter 7

Future Work

As with most projects, there is capacity to improve the implementation of ABACUS, and there exists different avenues to explore with ABACUS for research purposes. An obvious improvement that may come to mind would be to increase the number of profiling units to allow profiling and tracing of different performance characteristics that are not covered in the presented implementation. Extending the implementation to cover any additional feature as shown in *Figure 1.1* that was not implemented such as the Bus Main interface to memory providing a DMA access would be a good opportunity for improvement. Adding a AXI interface support would be strongly beneficial to introduce the design into a Vivado Block Diagram work flow.

It may prove interesting to implement a run-time reconfigurable branch prediction algorithm or cache replacement policy (which doesn't corrupt the state of the CPU while running Linux), and use ABACUS to seamlessly see which types of software work best with what types of branch prediction algorithms or cache replacement policies. Finally, via Buildroot, the Linux image may be modified to include *debugging, profiling, and benchmark* software, which would allow users to directly compare the data provided by a software profiling tool like *perftest* with ABACUS.

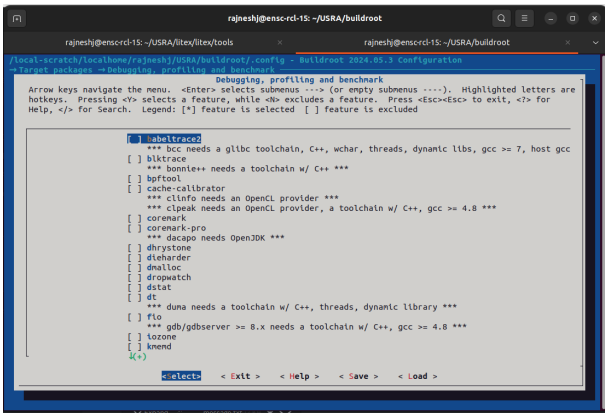

Figure 7.1: Debugging, profiling and benchmark in the Buildroot menuconfig

Chapter 8

Conclusion

In conclusion, this paper covered the motivation and design of a profiling unit which has the ability to characterize baremetal, and user software running on an operating system. With an easy-to-use API, users can leverage ABACUS to inform them about the quantitative effects that their hardware or software design has on software performance by referencing values provided by the profiling units.

Chapter 9

References

[1] L. Shannon, E. Matthews, N. Doyle, and A. Fedorova, "Performance Monitoring for Multicore Embedded Computing Systems on fpgas," arXiv.org, https://arxiv.org/abs/1508.07126.

[2] E. Matthews and L. Shannon, "Taiga: A new RISC-V soft-processor framework enabling high performance CPU architectural features," TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features, https://ieeexplore.ieee.org/document/8056766/

[3] S. Prasanna, "Modular: Row-major vs. Column-major Matrices: A Performance Analysis in Mojo and NumPy," Modular.com, Apr. 10, 2024. https://www.modular.com/blog/row-major-vs-column-major-matrices-a-performance-analysis-in-mojo-and-numpy

[4] E. Matthews, L. Shannon, and A. Fedorova, "A Configurable Framework for Investigating Workload Execution," Jan. 2011. Available: https://people.ece.ubc.ca/sasha/papers/fpt-2010.pdf.

[5] F. Kermarrec, S. Bourdeauducq, J.-C. L. Lann, and H. Badier, "LiteX: an open-source SoC builder and library based on Migen Python DSL," arXiv.org, 2020. https://arxiv.org/abs/2005.0250

[6] G. Masera, S. Brennsteiner, and F. Babbaro, "A RISC-V based accelerator for NFC Signal Processing," Dec. 2022.

[7] Openhwgroup, "Openhwgroup/CVA5: The core-V CVA5 is an application class 5-stage RISC-V CPU specifically targetting FPGA implementations.," GitHub, https://github.com/openhwgroup/cva5

[8] D. Gisselquist , "Accessing the registers of a SoC+FPGA," Accessing the registers of a SOC+FPGA, https://zipcpu.com/blog/2018/11/03/soc-fpga.html

[9] J. Corbet, "Ioremap() and memremap()," ioremap() and memremap(), https://lwn.net/Articles/653585/

[10] "Unknown relocation type 57," Reddit, https://www.reddit.com/r/RISCV/comments/1elwjj4/unknown$_r elocation_t ype_5$7/.