Simulation-Based Full-Stack System Profiling

Motivation

What got me into this project is because of my experience working on the FireAxe [1] case studies. One case study was to measure the tail-latency effects of an Golang application. Interestingly, we were seeing different tail-latency characteristics in our SoC compared to x86 server machines as we increased GOMAXPROCS. It was frustrating that we weren't able to clearly describe why our system was behaving in a certain way, even though we had complete control over our SoC simulation environment.

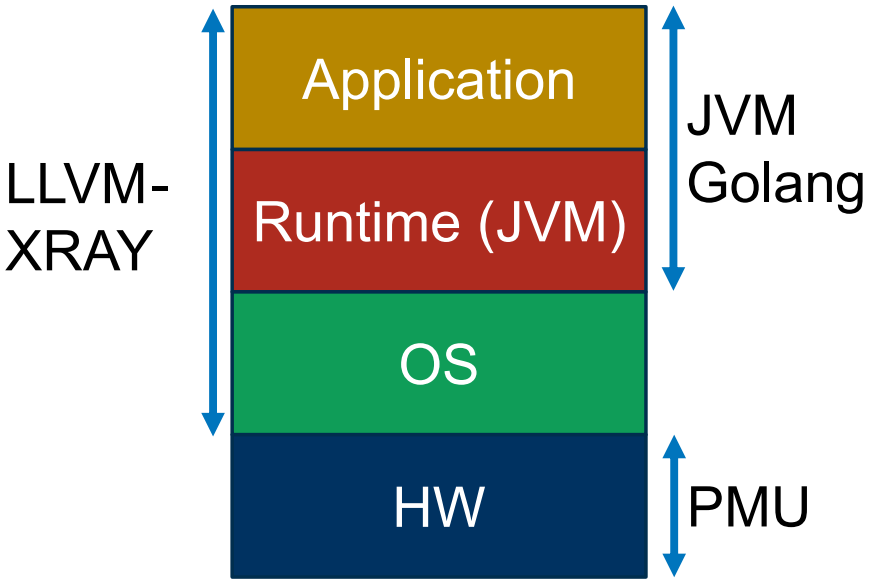

Figure 1: There is no single profiler that provides a complete view of the system—from hardware, OS, and runtime to application.

Although we could have used existing software profilers within simulation, this approach had its own shortcomings. First, the profiler ecosystem is fragmented. That is, there is no single profiler that provides a complete view of the system—from hardware, OS, and runtime to application. There are profilers for high-level language runtimes such as Java or Golang that tells you about the language runtime such as JIT, green thread scheduling, and garbage collection. Hardware counter based profilers (VTunes, TIP, TEA) generates CPI stacks and provides information about the microarchitectural events. LLVM based profilers such as X-Ray generates flame-graphs for user-space applications while the Linux perf tool generates flame graphs for the kernel. However, no single software profiler provides a unified view of the system. For instance, no profiler can tell you about why latency spikes happen in a go runtime system for certain applications because this is a combination of the golang runtime's scheduler, Linux's thread scheduler and microarchitecture. Specifically, there is no profiler that will show you which green thread is getting mapped to which OS thread by the Golang runtime, which OS thread is getting mapped

to which physical cores by Linux, what the coherency traffic looks like when the threads are run. If we had this magical profiler, we could 1) understand which green threads are getting mapped to which OS thread 2) which OS thread is mapped to which core 3) what happens in the memory subsystem during execution.

Furthermore, existing profilers show a fundamental tradeoff between profiling resolution and perturbation. Sampling profilers (e.g., perf, XRay) impose low overhead, but their millisecond-scale resolution prevents observing rare tail events. Also, in runtime-managed languages such as Java, sampling profilers add systematic skew, such as safepoint bias. On the other hand, tracing profilers (e.g., Go's execution tracer) provide nanosecond-scale observability but are limited in depth and bandwidth to reduce perturbation. In practice, trace-based profilers are run multiple times, each with a limited number of instrumentation hooks enabled at a time to prevent excessive perturbation. After enough runs, profilers use post-processing methods such as trace alignment to provide an overall view of what is going on in the system. Unfortunately, this method suffers from non-determinism and noise within the system.

The significance of a full-stack profiler may not seem so intuitive. However, we can only optimize parts of the system that we can see and understand, and as mentioned above, not one profiler provides visibility into the entire stack, preventing us from understanding cross-stack optimization opportunities. This leads to whether there exists any cross-stack optimization opportunities in the first place. Although I don't have a clear answer to the above question, we can see empirical evidence in [2]. This article shows that even "Hello, World!" exhibits end-to-end latency shaped by scheduler waits, page faults, interrupts, power-state resumes, and cross-core interactions which are events invisible to existing profilers. This argues for a whole-system, nothing-missing view that captures cause and effect across user space, kernel, and hardware.

In this work, by using simulation, we enable a complete side-channel approach to profiling so that we would have cycle-level resolution and effectively unlimited bandwidth. This eliminates the need for trace alignment while tracing out all the system events that we are interested about with very high resolution. Furthermore, it serves as a baseline for evaluating in-band profilers. Rather than speculating the cause of perturbation, we can accurately attribute perturbation to different sources such as stack-unwinding overhead, cache contention, or systematic biases by comparing in-band profilers with our out-of-band profiler.

Co-Simulation Methodology

Naive approaches for extracting state from an FPGA include using scan chains or inserting numerous probes into the system. Generating scan dumps frequently destroys simulation throughput, while doing so infrequently reduces the profiling resolution. In addition, scan chains can lengthen the critical path of the design, lowering the achievable FPGA clock

frequency. Probes, on the other hand, significantly increase engineering complexity. For example, suppose we want to obtain the value stored at a particular memory address. We would need to scan both the DRAM state and the SoC cache to determine whether that address has been written to.

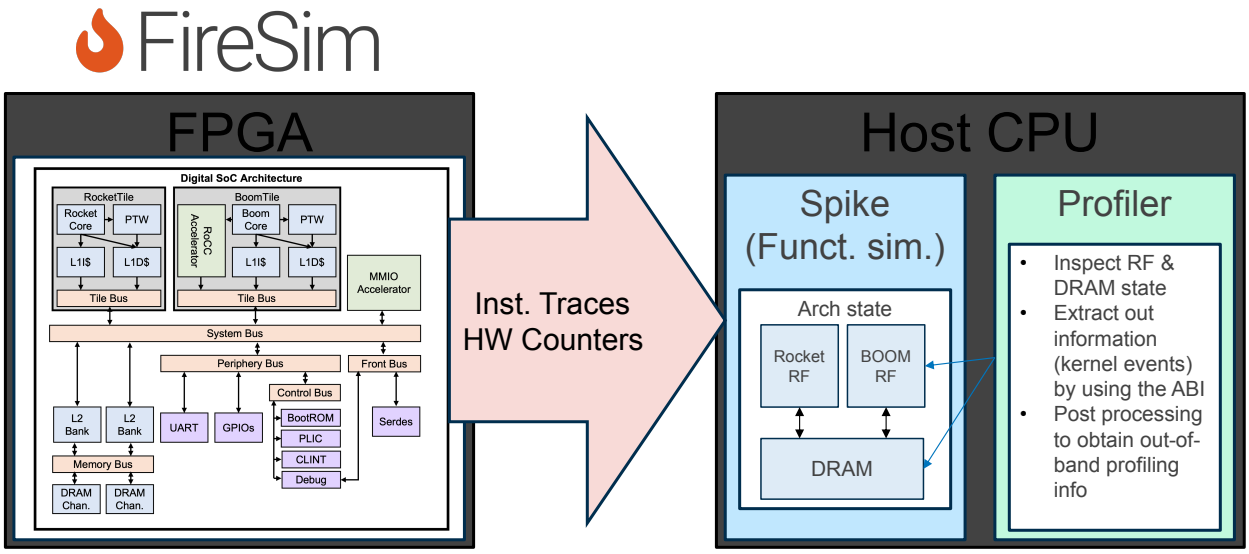

Figure 2: Architecture of our profiler implementation. It uses co-simulation to maintain a portrait of the SoC state on the host machine.

To address these challenges, our profiler is built using co-simulation between the target RTL running on the FPGA and a functional model running on the host CPU. Figure 2 shows the overall architecture. The target device runs on the FPGA as usual, but we exfiltrate committed instruction traces along with additional signals such as interrupt lines. We also extract hardware counter information, which serves as profiling data. The committed instruction traces and hardware signals are then fed into a functional model (Spike) to reconstruct the architectural state of the SoC. Once the full architectural state of the FPGA has been recreated in memory on the host, we can probe any register or memory location directly for profiling.

There are several challenges that must be addressed to make co-simulation work. The most significant issue is that components outside the core also update the architectural state of the SoC. Representative examples include timer interrupts from the CLINT and I/O interrupts from the PLIC. MMIO device responses present another source of nondeterminism. Finally, atomic operations can cause divergence between the RTL and functional simulations. For instance, when two cores contend for a lock, differences in interleaving may result in one core acquiring the lock in the functional model while a different core acquires it in the RTL simulation.

We address these issues by reasoning about system state from each processor's perspective. As long as the functional model of the processor observes (i.e., loads) the same values produced in the RTL simulation (FireSim), the two will exhibit consistent behavior. Using

the SoC's device tree, we can identify loads from MMIO devices (including the PLIC and CLINT) and inject the corresponding values into the functional simulator. For atomic operations, we detect the instruction opcode and apply the same approach. Finally, for I/O device interrupts, we delay asserting the interrupt in functional simulation until the same interrupt signal is observed in RTL simulation.

Case Study: Instrumenting the Linux Thread Scheduler

Profiling on the host is performed as follows. Because we control the entire simulation environment, including the application binaries of interest, we can obtain object dumps of both the applications and the kernel, and then use program counter (PC) values as triggers for profiling actions. For example, we know the PC corresponding to the entry point of the `fork` system call. When the functional simulation reaches this PC, we use the ABI to extract the function call arguments. If an argument is a pointer, we query the corresponding memory location to dereference it. Since the functional simulation maintains a copy of the page table, we can directly query memory values using the virtual addresses stored in registers. Finally, we use struct offsets to determine the values of individual fields.

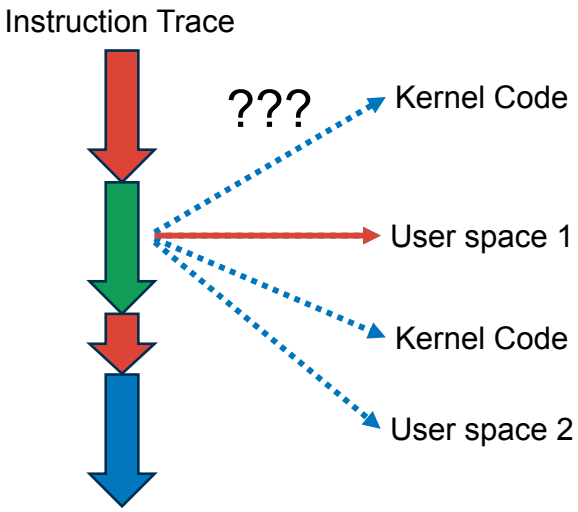

Figure 3: Given a stream of instructions, we do not know which process each instruction maps to.

The above technique assumes that we know which application binary the current process corresponds to. Figure 3 illustrates a limitation of this approach: a stream of PCs could map to any program, including the kernel, making it difficult to identify which binary the stream belongs to. Kernel code is relatively easy to identify, since its PCs occupy the top half of the virtual memory space (starting with 0xfffff). Differentiating between arbitrary user-space programs, however, is more challenging. Fortunately, by intercepting the kernel's `fork`, `exec`, and `CFS scheduler` code, we can determine which process each instruction stream maps to.

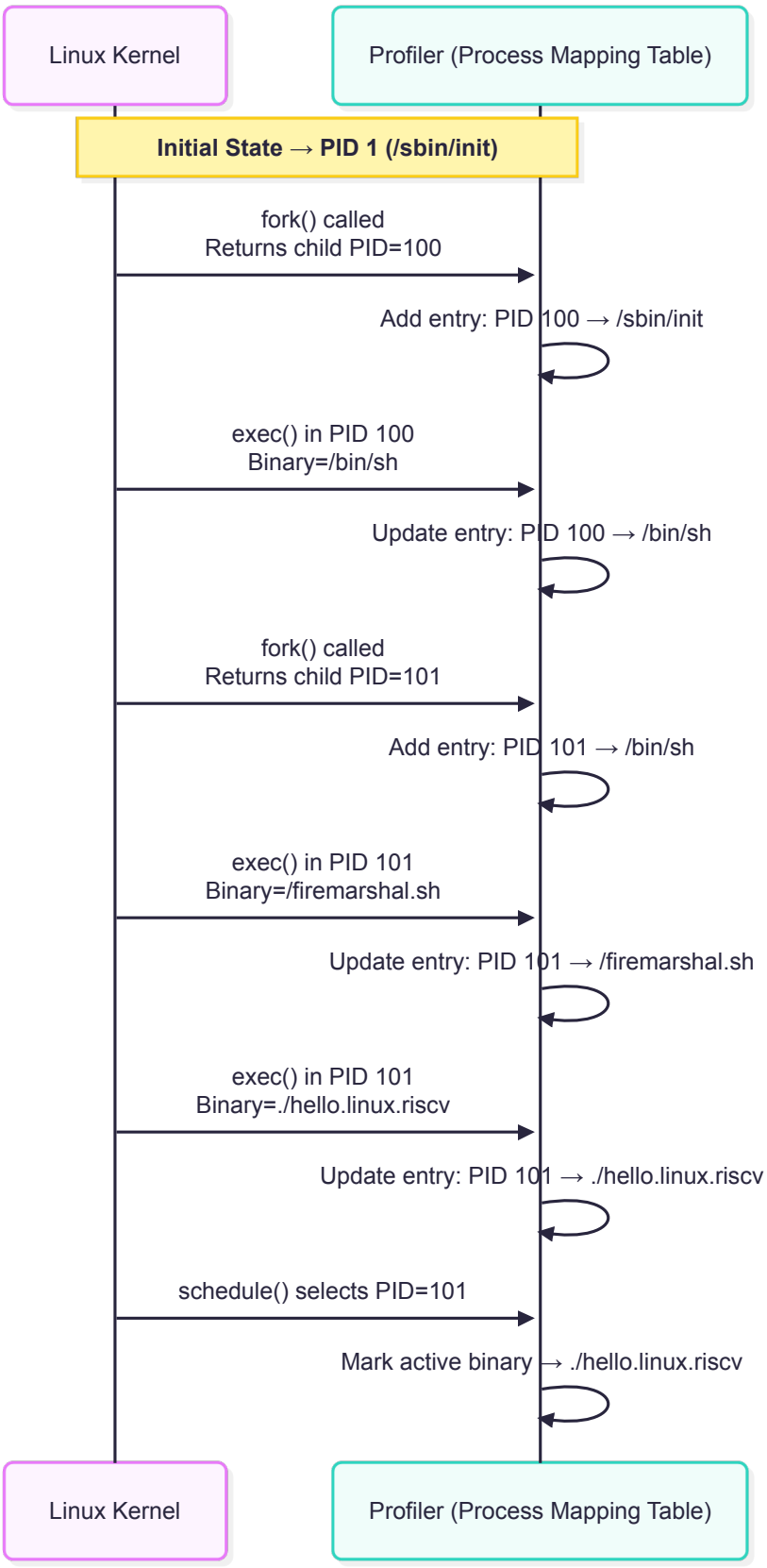

Figure 4: By intercepting the kernel's `fork`, `exec`, and `CFS scheduler` code, we can determine which process each instruction stream maps to.

The diagram in Figure 4 provides a high-level overview of this mechanism. In the following paragraphs, we describe each interception point in detail.

First, we intercept the `fork` system call. When `fork` is invoked, the kernel creates a new child process by allocating a new `task_struct`. The child inherits the parent's code, data, heap, and stack, with memory typically managed through copy-on-write. The function signature of the `fork` system call is:

```
pid_t kernel_clone(struct kernel_clone_args *args)
```

For the parent process, this function returns the `PID` of the newly created child. Therefore, on every `fork` system call, we can insert an entry into a profiler table that maps each `PID` to its corresponding binary. For convenience, we refer to this structure as the *process mapping table*.

Next, we intercept the `exec` system call. When `exec` is invoked, the kernel replaces the calling process's memory image with a new program. The function signature of the `exec` system call is:

```
static int do_execveat_common(int fd, struct filename *filename,
                              struct user_arg_ptr argv,
                              struct user_arg_ptr envp,
                              int flags)
```

We obtain the pointer value `*filename` by intercepting the second argument to this function using the ABI. Next, we need to inspect the layout of `struct filename` to obtain the path string to the executed binary.

The `struct filename *filename` argument points to a structure that contains the path string of the binary being executed. The `struct filename` is defined as:

```
struct filename {
  const char *name;
  const __user char *uptr;
  struct audit_names *aname;
  bool separate;
}
```

From the `struct filename` pointer, we then extract the `*name` field. By reading memory byte by byte starting from name until a null terminator is encountered, we reconstruct the path to the binary that the process is set to execute. Finally, we update the *process mapping table* in the profiler, replacing the entry for the current `PID` (the process that invoked `exec`) with the new binary.

Lastly, we intercept scheduling decisions in the Completely Fair Scheduler (CFS) to determine which binary the current instruction stream corresponds to. Linux scheduling is performed in the `schedule` function, which appears throughout the kernel wherever a context switch may be appropriate—for example, when issuing an I/O request and waiting

for its response, or before exiting a system call. By intercepting `schedule` before a system call exits, we can identify which `PID` will be scheduled next. Since we maintain the *process mapping table*, this allows us to determine the binary that will execute next.

Within `schedule`, the function `pick_next_task_fair` selects the next task to run. Its signature is:

```
struct task_struct *
pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags
*rf)
```

The return value is a pointer to the `task_struct` for the task that will be scheduled next. By reading the `PID` field from this structure, we can determine the next process to run and map it to the corresponding binary.

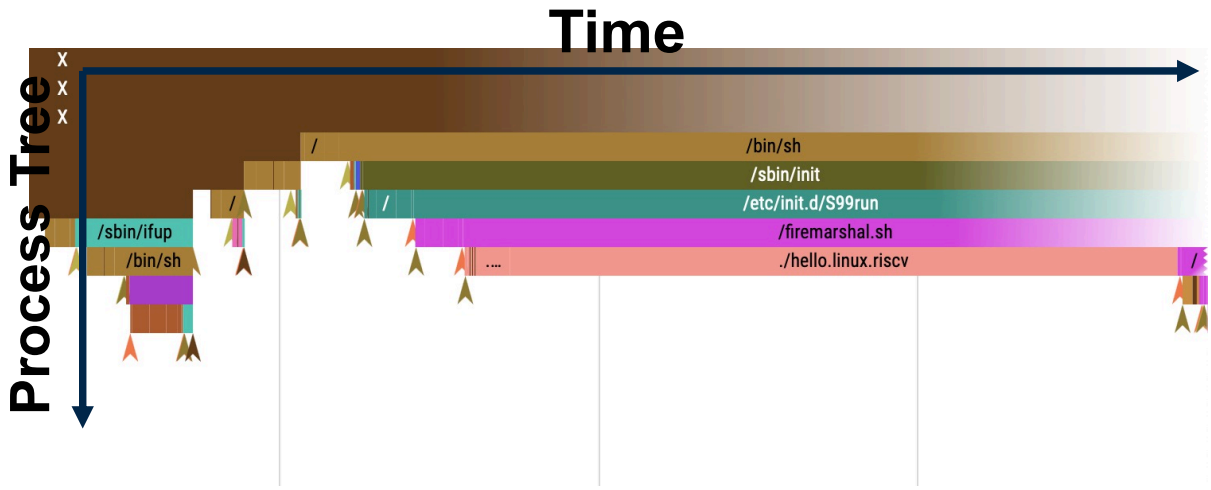

Figure 5: By intercepting the `fork`, `exec`, and `schedule`, we are able to create a process tree out of band.

Figure 5 illustrates the process tree captured by intercepting `fork`, `exec`, and `CFS scheduling` decisions. Each horizontal bar corresponds to a process, with its label indicating the binary currently mapped to the `PID` in our *process mapping table*. Starting from `/sbin/init`, the kernel launches initialization scripts such as `/etc/init.d/S99run`, which in turn invoke `/firemarshal.sh` and eventually the target binary `./hello.linux.riscv`. Alongside these, auxiliary processes like `/sbin/ifup` and `/bin/sh` are spawned as part of system setup. Because our profiler records every context switch and updates the mapping table on `fork` and `exec`, we can reconstruct this full execution tree and associate each instruction stream with its originating binary. The diagram therefore provides a faithful timeline of process creation and binary execution on the FPGA, derived entirely from the interception mechanism described above.

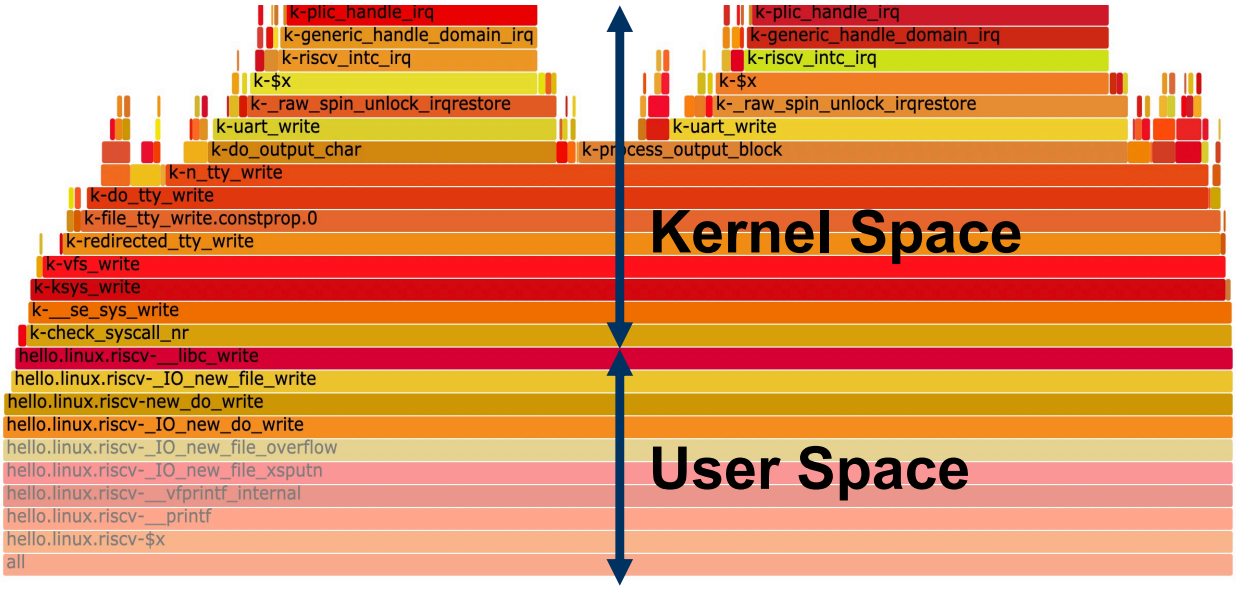

Figure 6: As we have information about which binary corresponds to the current instruction stream, we can use it for more profiling operations such as stack reconstruction.

With the mapping from instruction traces to binaries in place, we can reconstruct full stack traces by performing stack unwinding during simulation. Each committed instruction trace is tagged with its corresponding binary using the *process mapping table*, which allows us to distinguish between user-space and kernel-space execution. When the instruction pointer lies in user space, we unwind through the application's stack frames using the debug information from the application binary. When the pointer crosses into kernel space (e.g., during a system call or interrupt), we continue unwinding through the kernel's stack frames. By stitching these together, we obtain unified flamegraphs that capture the complete control flow across user space and kernel space. This enables profiling views where application functions (e.g., printf) are directly connected to the kernel routines they invoke (e.g., ksys_write), giving us a holistic view of performance bottlenecks spanning the entire software stack.

To summarize, by intercepting three key system calls in the kernel related to scheduling, we were able to create an out-of-band `PID` to binary mapping. Using this, not only can we visualize what the kernel scheduler is doing, but also use it as a stepping stone for adding additional instrumentation in the profiler for user space applications.

Case Study: Golang Application

We next extended profiling support to runtime-managed languages (Go in our example). Then, we decided to overlay our kernel-level profiler on top of the profiling information obtained from Go's built-in execution tracer. Although this approach introduces some perturbation as the Go runtime is instrumented, it significantly reduces the engineering

complexity by avoiding the need to reverse engineer the runtime's scheduling and memory managament details. Furthermore, as our profiler operates out of band, we can attribute exact cycles to each kernel event, thereby translating into the timestamp domain used by Go's profiler. This enables us to seamlessly overlay the two profiling information into a unified view that provides us with kernel events as well as the application behavior.

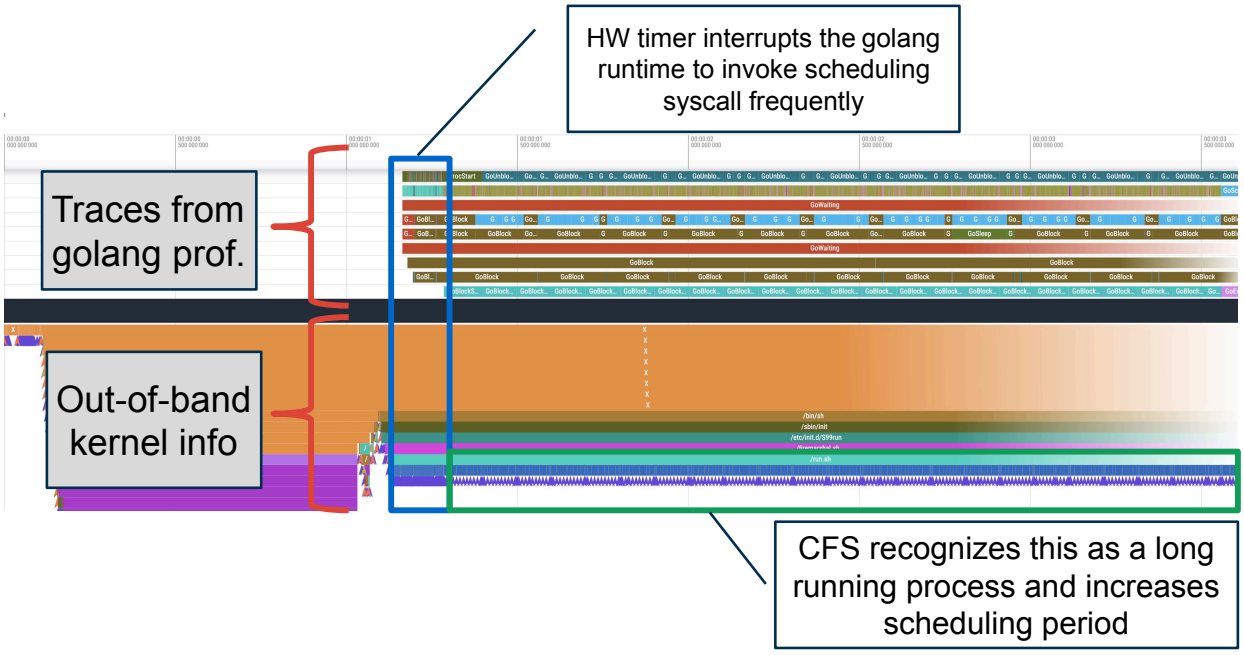

Figure 7: Result of overlaying the Go execution tracer with our outofband kernel profiler. We can see the interaction between the golang internal scheduler and Linux's CFS.

Figure 7 shows the result of overlaying the Go execution tracer with our out-of-band kernel profiler. The top portion corresponds to traces generated by the Go runtime, which expose goroutine scheduling events such as `GoBlock`, `GoWaiting`, and `GoSleep`. The bottom portion contains the kernel-level process and context-switch information captured independently by our profiler. Early in execution, the Go runtime relies on frequent hardware timer interrupts to drive its internal scheduler, causing a burst of scheduling-related syscalls that appear densely in the trace. Over time, however, the Linux Completely Fair Scheduler (CFS) recognizes the application as a long-running process and gradually increases the scheduling period, reducing the rate of these invocations. By aligning the Go runtime events with kernel-level information in a common timeline using cycle-accurate timestamps, we can directly observe this transition and correlate high-level goroutine scheduling with the underlying kernel behavior. This produces a unified view of execution that spans from language runtime decisions down to the OS level.

Future Work

There are many potential directions this work can extend into.

First, for this method to work well, we need a functional simulator that is modular and capable of executing in various modes. That is, it must be able to run in free-standing mode where there is no input from the outside and the simulation runs as a master. It must also support a slave mode where it can be used to develop co-simulation frameworks. Spike, despite it having relatively high performance, could be improved on these axese and building a new functional simulator from scratch would be a worthwhile endeavor.

Second, we could translate our approach into real hardware by implementing a dedicated SoC event-tracking unit that can be integrated directly into silicon. Such a unit would enable low-overhead profiling of production applications, including those written in interpreted languages, without requiring intrusive instrumentation. Users should be able to add hint instructions within their application to log events that they are interested in. We think that having higher visibility into the system behaviors will open up new possibilities for software performance optimization.

Finally, as an interesting case study, we could use this to evaluate scheduler efficiency in heterogeneous big.LITTLE-style systems where the interaction between application behavior and core selection policies remains poorly understood.

Bibliography

[1] J. Whangbo *et al.*, "FireAxe: Partitioned FPGA-Accelerated Simulation of Large-Scale RTL Designs," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 501–515. doi: 10.1109/ISCA59077.2024.00044.

[2] R. L. Sites, "Benchmarking "Hello, World!": Six Different Views of the Execution of "Hello, World!" Show What Is Often Missing in Today's Tools," *ACM Queue*, Oct. 2018, doi: 10.1145/3291276.3291278.