Performance Measurements of VProc on Verilator

Simon Southwell

June 2024

Preface

This document is from an article written in June 2024, and uploaded to LinkedIn, on performance measurements of the VProc virtual processor with Verilator.

Simon Southwell
Cambridge, UK
June 2024

Contents

Introduction

I have written about the VProc virtual processor before, in terms of what it is, what it is used for, how it works and the extended possibilities for co-simulation and co-design using such a system. That article garnered some interest and a query about whether VProc is, or could be, supported on the open source Verilator cycle based Verilog simulator (Chili.CHIPS*ba). Although the latest code did not have Verilator support, it had been run with Verilator in its early days, and so support was easily added and I made an announcement, when this work was complete. The motivation behind the query was about speed of execution, with Verilator, as either C++ or SystemC based, being a fast simulator which could lend itself for use in near real-time and/or interactive simulations, and it was wondered just how fast VProc might run along with Verilator and what the overhead of VProc might be. I had made some measurements back in the day and found at that time that the percentage time spent in the PLI domain was around 0.1%. Now that was a long time ago, and I agreed to make some new measurements with an up-to-date Verilator and the latest VProc. This was completed recently and the results summarised and shared with the original inquirer.

When it was also suggested that I might write a blog on this work I didn't, at first, think there could be much interest in this niche subject, being rather specialised to the virtual processor. However, on reflection, I have used some of the features of Verilator that may not be familiar to all users, have migrated VProc to run on it using the DPI-C interface, and have used my rv32 RISC-V C++ instruction set simulator, running on VProc, to drive an HDL test environment. The RISC-V CPU model runs an example program that uses FreeRTOS as its operating system and spawns a couple of concurrent tasks that use the delay features and display progress messages to a UART based terminal. The whole ting was also profiled using Verilator features and gprof.

So this work brings together many different components, some of which I have written about before. The VProc article has already been mentioned, but I also wrote a set of articles on real-time operating system based on FreeRTOS and, in part 4, I introduced the rv32 ISS on which to port FreeRTOS and run an example program to demonstrate the basic features. These articles tend to focus on one key subject and often there is no equivalent information on how to take separate components, such as these, and bring them together to form a new environment that's useful in different ways.

In this article, then, I want to gather together these different components to construct a simulation with Verilator on which we can take some performance measurements. In addition we will also be using GTKwave to display what's going on. Along the way we will look at the integration processes followed and which Verilator features were used and how to use them, including generating waveform data, 'profiling' the simulation to get performance information, and the `--timing` switch to support inter-cycle timings (Verilator is purely cycle based by default). All these features will be looked at under the microscope of how this affects performance and when and where these are appropriate to trade performance over enabled simulator features.

Integrating VProc with Verilator

At its heart, the Verilator logic simulator is a cycle based simulator. What does this actually mean? Well, to contrast, an event-based simulator, such as one would find in a commercial product (though not exclusively), will simulate detailed timing for all events including those between the events generated by the edges of system clocks. This means that arbitrary delays can be modelled and, relevant to our discussion, delta-cycle calculations can be made. A simple model of how such a simulator works is to think about time slots down to the resolution of set for the simulation (`` `timescale 1ns/1ns `` for example). As the simulation runs, any change at a given time may cause other changes in logic values, and these are added to the back of a 'fifo queue' of changes that need to be evaluated for that time slot. Some might be for the current time slot and be placed in its own queue or might be for a future time slot and will be placed in its queue. When, for a given change, all the calculations have been evaluated, the simulator pops that event from the queue and moves on to the next one for the time slot. These calculations could even change the value of the signal just popped and will be placed again at the back of the queue. Indeed, if one is not careful, then an infinite loop is created (`forever a = ~a;`) and the simulation will hang. When all the events have been processed for a time slot, with the queue empty, then the simulation will move to the next time slot that has any events to evaluate—and so on. Thus, a simulation need not have any clocks and all be done with delays and waits etc., or any arbitrary mix.

With a cycle based simulation, a clock is required, and evaluation of state is done all at once for a whole clock cycle, with no events of delays in between. The simulator might allow a delay syntactically but will probably collapse this to a 0 delay equivalent, placing the event at the beginning of the cycle. This limits what can be simulated and not all possible HDL code can be run. Usually, this is described as

being limited to 'synthesisable' code, as logic synthesisers also can't handle arbitrary delays and events and can process only combinatorial and synchronous logic. But this isn't a truly accurate description of a cycle based simulator, as it can handle behavioural code, like `$display` for example, which a synthesiser can't. One might think that this restriction makes cycle-based simulators not very useful and too restrictive, so why use one? Well, the answer is that, because they limit the logic evaluation to clock edge events, substantially reducing the number to process, they can run very much faster than event based simulators, and if one can restrict the test environment to use only the allowed syntax, then many more test vectors can be put through the (possibly very complex) IP being developed, which is a clear advantage. How does this affect running VProc on Verilator?

Well, the VProc virtual processor has the concept of delta-cycle updates. That is it can read or write to state in a delta-cycle (within a time slot), not advancing simulation time, just as discussed above for an event simulator (covered in more detail in my article on VProc and in its manual). This is important for using VProc in scenarios where it is driving a bus, interface, or any set of ports, that are wider than its 32-bit generic memory mapped bus. In these cases, the pins/ports of the module in which it's instantiated can be memory mapped onto VProc's bus and updated and/or read in a set of delta cycle accesses before allowing the simulation to advance time. The functionality for driving the ports is then modelled in C/C++, providing an API to higher level software running on the virtual processor. Thus, any number of pin/ports (to a limit of 4 billion or so!) can be serviced. This is how the PCIe, USB and TCP/IP models work. So, VProc won't work with Verilator, right? Well, a couple of things help in this regard.

Firstly, in recent times (since I last use Verilator in earnest), a `--timing` flag has been added to Verilator which does, in fact, allow it to process events between cycles to some degree and, importantly for VProc, allows delta-cycles updates. This will necessarily slow down simulation, and we shall be looking at this later. But it does make VProc compatible with Verilator without modification.

Secondly, if one doesn't need to drive wider buses with VProc, then delta-cycle updates are not required. However, Verilator won't compile the VProc Verilog code associated with delta-cycles as it stood, and so VProc was modified to be able to disable this with a parameter (`DISABLE_DELTA` set to non-zero) and to remove any delays if compiling for Verilator (`VERILATOR` defined). With these changes VProc can be compiled for Verilator without using the `--timing` switch.

In our environment, VProc will be configured for non-delta cycle operations as we only need the 32-bit bus interface, and so the HDL compiles for Verilator without issue.

Programming Interface

Verilator was originally a Verilog simulator, but now has support for SystemVerilog (which didn't exist when Verilator first started), including the DPI-C programming interface. Since VProc has support for DPI-C (used with Vivado), and this is the simplest of the programming interfaces to use, advantage was taken to use it with Verilator. The VProc Verilog remains unchanged but is compiled as SystemVerilog, though included via a new file, `f_VProc.sv` which simply defines `SV_VPROC` so that VProc includes `vprocdpi.vh` to gain access to DPI features and change calls to PLI type `$` tasks, to the equivalent DPI function calls (this was already in place for Vivado support).

SystemC or C++

Verilator works by converting the Verilog code to C++ software, which is then compiled with a C++ compiler. However, that code can be selected to be SystemC compatible (itself a C++ library), using the SystemC simulation engine, or as pure C++, where the simulation engine is part of the generated code.

VProc is written in C and C++ and introducing a new system into the environment would only complicate things. Of course, if you have a pre-existing SystemC environment you want to use, it makes perfect sense to use this. For our purposes, we will stick to using the C++ output.

VProc C/C++ Compilation

VProc software and accompanying user code normally compiles to a shared object (`VProc.so`)—known as a dynamically linked library (DLL) on Windows. This shared object is then loaded either at elaboration (e.g., Vivado uses `xelab -sv_lib VProc.so ...`) or at run time (e.g., Questa uses `vsim -pli VProc.so ...`). With Verilator, however, the Verilog is being converted to C++ and then compiled. Thus, the VProc code needs to be included in this compilation. Fortunately, VProc compilation (via the provide make files) already compiles everything to a static library first (`libvproc.a`), before compiling the shared object. We can use this library to include the VProc software for linking with Verilator code. Verilator will take `g++` command lines on its own command line, basically without modification, and so adding `-L../ -lvproc` will link this VProc software. Note that the link path is a

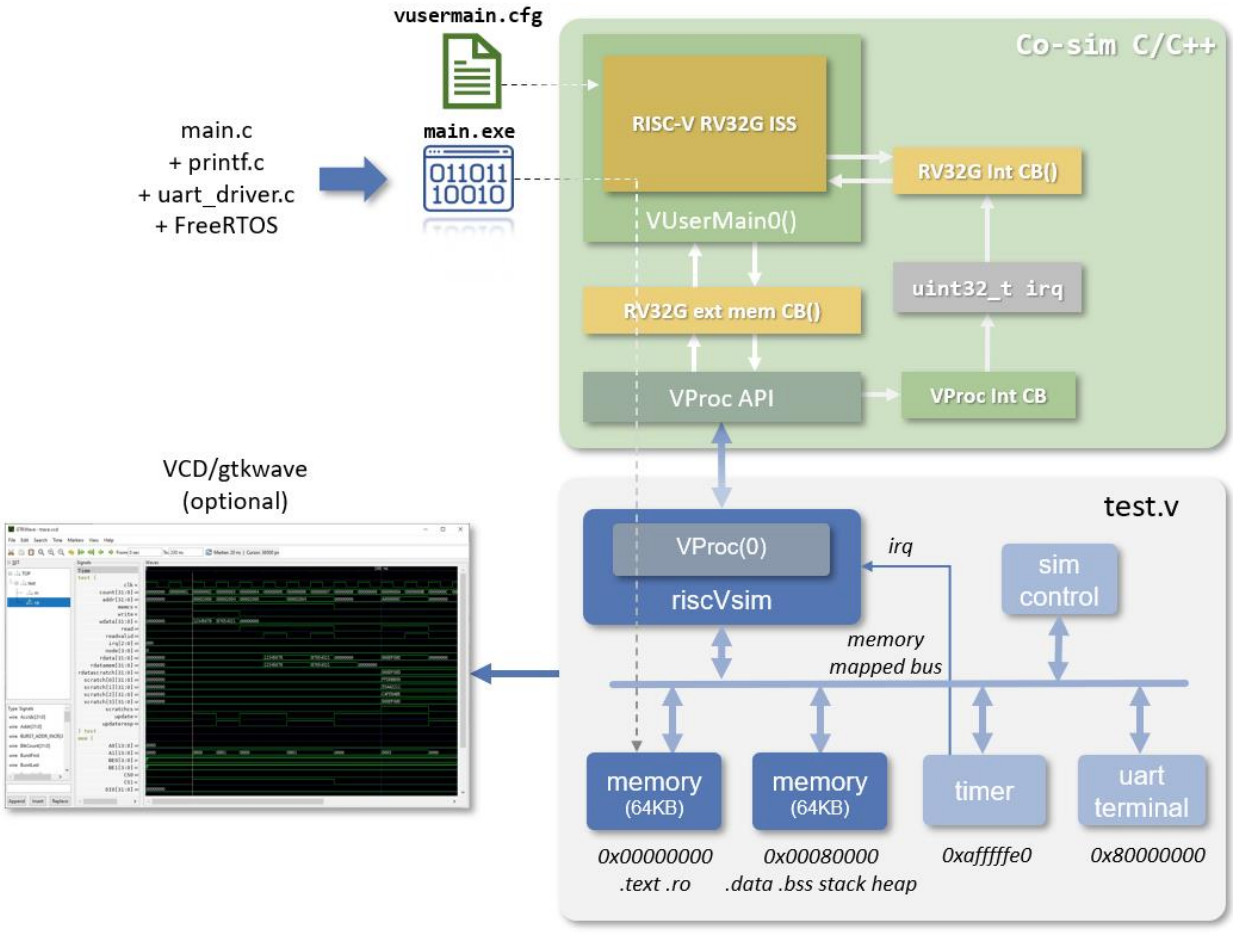
directory above where `verilator` was executed, since Verilator compiles in the work directory it created, which is a directory below. All of this detail will be wrapped up in make files, so is abstracted away and need not be remembered when using the system.

So now we have the means to integrate, with Verilator, VProc's own software, the user program to be run, and any libraries that this might use, without changing the basic way the VProc software is compiled.

The Test Environment

In general, we've seen how to integrate VProc HDL, its software and the program to run on the virtual processor, with Verilator. For the performance testing we want to do, we now need some specifics in terms of the test environment and what we are running on the virtual processor.

For this work we are going to run the rv32 RISC-V instructions set simulator (ISS) on the processor (with some lightweight integrating wrapper code), where memory mapped accesses will be directed at the logic simulation, and interrupts will be derived from the logic. The diagram below shows the intended setup.



The HDL wraps the VProc component in a simple module (`riscvSim`) to basically separate out the data and instructions buses (if configured this way, which is the default), presenting two master bus ports, and also has a 3-bit interrupt input port (for software, timer and external interrupt requests, though we will only use the timer interrupt). The VProc component is configured as 'node 0'. VProc may be instantiated in a simulation multiple times, but each must have a unique node number (a handle, if you will) to associate it with its own program. The entry point 'main' function for a node is `VUserMain<n>`, where *<n>* is the node number, so `VUserMain0` in the example.

On the memory mapped bus are instantiated two simple dual port RAM models, one for read-only data and the other for the dynamic data, and have single cycle access. A simple timer module implements the 64-bit `mtime` and `mtimecmp` timer registers, ticking at 1MHz, and generates an interrupt request if the `mtimecmp` value is greater than the `mtime` value. Both these registers can be accessed by the VProc code. A 'terminal' module, with a UART compatible register interface is used for printing to the console. Some simple simulation control is also implemented to allow the running VProc program to stop/finish the simulation by writing to a specified location. The test bench HDL is thus very lightweight.

VProc is running a program that instantiates the rv32 ISS C++RISC-V model, interfacing its memory accesses to VProc API calls and passing back interrupt state. On the ISS itself is running a demonstration program, that's loaded to the test bench memory models, and which has FreeRTOS running with an example program with two tasks running concurrently and displaying to the terminal. FreeRTOS is configured to run with the timer running at a 1µs period and uses the IRQ to schedule context switches, delays etc. By default, the ISS model's program is running out of the test bench's HDL memories, and thus makes an access across the VProc DPI-C interface every cycle, except when accessing the timer/terminal peripherals, where accesses are two cycles—but this is comparatively infrequent compared to memory. Thus, VProc is almost saturated in the number of accesses, which is (at first) what is trying to be achieved to measure performance by maximizing VProc activity.

All the code for this can be found in the VProc repository under `examples/riscv`.

Integrating the ISS

The `rv32` ISS can't simply be run as a VProc program, as it makes no reference to the API to drive the simulation interface. However, its integration is straightforward. I

won't go into great detail about the ISS, and more information can be found in the manual, but it's worth summarising the features relevant to our needs.

The ISS models a 32-bit RISC-V processor to the RV32GC + Zicsr standard. That is, it has all the basic operations of a general processor, along with compressed instructions and implements the (minimum) control and status registers. It also has an internal memory model of 1Mbytes and an internal `mtime/mtimecmp` timer model which is configurable to use either real-time or use the ISS's own internal clock cycle count. Both of these features, the timer and the memory, can be bypassed to use external models, which will come in handy for us. Lastly, the ISS has some callback function features to allow expansion and interfacing with other models (another handy feature). For our purposes, the model provides methods to register a callback to be invoked on any memory access (load or store instructions), where the callback can handle to access and return an elapsed cycle time for that access or say it did not process that access and hand it back to the ISS for processing, or a callback invoked when the ISS wants to read the current interrupt request state.

The VProc top level program (`VUserMain0`) creates an ISS object (pointed to by `pCpu`) and registers an external memory callback function (`ext_mem_access`) and an interrupt callback function (`iss_int_callback`) using, respectively, the methods `register_ext_mem_callback` and `register_int_callback`.

The memory callback is invoked for each load or store instruction, with an access type (indicating whether a read or write and size as byte, half-word, or word), with a data reference containing write data or for passing back read data, and the address. The current ISS time is also passed in, but we won't be using this. It is in this callback function that the type is decoded and the relevant VProc API functions called (via some simple memory wrapper code—`mem_vproc_api.cpp`). The wrapper code simply invokes `VWriteBE` (write with byte enables) or `VRead` (read word) VProc functions as appropriate, doing the necessary shifting of data, but also has the ability to call `VTick` to advance time for additional cycles, beyond the actual memory access, to model processing time of other instructions if desired. The setup we're using does not do this. A simplified version of the callback is shown below.

```
int ext_mem_access(uint32_t addr, uint32_t& data, int type, rv32i_time_t time) {
    int processed = 1;
    switch (type) {
    case MEM_RD_ACCESS_BYTE:  data = read_byte(addr);  break;
    case MEM_RD_ACCESS_HWORD: data = read_hword(addr); break;
    case MEM_RD_ACCESS_INSTR: data = read_instr(addr); break;
    case MEM_RD_ACCESS_WORD:  data = read_word(addr);  break;
    case MEM_WR_ACCESS_BYTE:  write_byte(addr, data);  break;
    case MEM_WR_ACCESS_HWORD: write_hword(addr, data); break;
    case MEM_WR_ACCESS_INSTR:
    case MEM_WR_ACCESS_WORD:  write_word(addr, data);  break;
    default: processed = RV32I_EXT_MEM_NOT_PROCESSED;  break;
    }
    return processed;
}
```

The ISS's interrupt callback simply returns the value of an interrupt request state variable (`irq`), a 32-bit value, with bit 1 the timer interrupt request bit. A pointer to a wakeup time is also passed in. This contains the current ISS time on calling the function and is updated to the time that it should be called next. This allows minimising of the calling of this function if the code knows that no new interrupts will occur before a minimum amount of time will elapse. In this example, we will call every cycle to avoid having to synchronise to the HDL timer and respond immediately to any change of state. To update the irq state, the `VUserMain0` program registers another callback function (`vproc_irq_callback`), but this time with the VProc code. The VProc API provides a function, `VRegIrq`, to register a callback which gets called whenever the VProc `Interrupt` port changes state, passing in the current state of the vector. This callback, then, simply updates the `irq` variable with the new state each time it's called, ready for the ISS to inspect it. Note that, although the user code is running in a separate thread from the simulator, which calls the `vproc_irq_callback` directly, this is thread safe, as explained in my last article on VProc. The two callback functions are shown below:

```
static uint32_t irq = 0;
uint32_t iss_int_callback(rv32i_time_t time, rv32i_time_t *wakeup_time) {
    *wakeup_time = time + 1;
    return irq;
}

int vproc_irq_callback(int val) {
    irq = val;
    return 0;
}
```

With this integration done, the `VUserMain0` program can load the RISC-V program to be run on the ISS model. The loads to memory of the program (via the `read_elf` method) will go through the external memory callback and will thus be loaded to the HDL memory models via VProc API calls. The ISS model is then ready to be run, using the `run` method, but this call is bracketed with the functions `pre_run_setup` and `post_run_actions`. These get the current real-time, and calculate the time taken by run so that the instructions-per-second rate can be calculated, which will be one of our major performance comparison points. When calling the run method, a configuration is passed in to alter the behaviour of the ISS model. One of these configurations is the number of instructions to execute before returning (otherwise the model will run forever). The ISS has lots of configuration points, which can be controlled from the command line (for the stand alone ISS) or via `vusermain.cfg` for this model. I won't detail them all, but relevant to this discussion, I will list those used:

- `-t` : specify test executable (default `test.exe`)
- `-T` : Use external memory mapped timer model (default internal)
- `-n` : specify number of instructions to run (default 0, i.e. run until `unimp`)
- `-h` : display help message

A `vusermain.cfg` file, located where the simulation is run, can be used to add these command line arguments to the program. An example contents of this file might be:

```
vusermain0 -T -n 10000000 -t ./main.exe
```

Note that the '`vusermain0`' associates the command line options with the specific `VUserMain`*<n>* program, and multiple node configurations could be specified in this same file. Other options are available, and if `-h` is added to the `vusermain.cfg` options, these will be displayed.

We're now good to go with the simulation—once we have a program to run on the ISS.

The RISC-V FreeRTOS Program

FreeRTOS configuration for Co-simulation

FreeRTOS has already been ported to the rv32 ISS, and this is documented in part 4 of my series of articles on real time operating systems, including configuring FreeRTOS in terms of the clock rate (really the timer rate) and the tick rate. For the standalone ISS, the clock/timer rate is set for 1MHz for a period of 1µs, and a tick rate of 100Hz for a period of 10ms. The location in the memory map of the `mtime` and `mtimecmp` registers is also defined.

In order to use these settings unmodified on our new co-simulation system, the HDL timer model needs to increment the `mtime` register value every 1µs, which it has been set up to do. It will even scale with the system clock frequency, which is set by a test bench generic `CLK_FREQ_MHZ` set at 100 (MHz) by default, which is forwarded into the timer. The 10ms tick period can remain unchanged as this just determines when the OS will schedule the next time slice. The effect in HDL will be that the timer compare register, `mtimecmp`, at each time slice, will be updated with a value equivalent to 10ms into the future—`mtime` plus 10000. When `mtime` becomes equal or greater than `mtimecmp`, the timer module raises the timer interrupt request signal, and the OS performs the necessary time slice processing and reschedules for the next time slice, updating `mtimecmp` once more.

The Test Program

The demonstration test program for running on FreeRTOS spawns two concurrent tasks, which count down and display the count values to a UART terminal and then delays for 1 second, when running in the standalone ISS. The model is configured to use its own timer model with a real time clock, so the tasks do actually delay for 1s. The program can be configured to run for a given number of instructions executed and then terminate. In this mode it can measure the time taken whilst running the program and calculate and display the instructions-per-second rate at the end.

For the co-simulation test, with the timer being an HDL model not running in real time and the execution rate not likely to be as fast as the standalone ISS (which is what we're trying to find out), then a delay of 1s may be too slow to reasonably see the tasks have multiple loop iterations. So, the code can be scaled to reduce this delay by some factor. If the test program is compiled with `COSIM` defined, then a scale factor of 40 is used so that the delay is 25ms. The code can be found in the rv32 github repository's `freertos` directory, where there is a make file (`make`

USRFLAGS="-DCOSIM"). This, of course, requires the gcc RISC-V tool chain to be installed. However, a copy of the executable, main.exe, is checked into the VProc repository, in examples/riscV, which is where the simulation picks it up for execution, as specified in vusermain.cfg.

Drivers

In addition to FreeRTOS, the user code also needs some additional code to print the messages to the UART terminal. This comes in the form of a lightweight printf implementation for embedded systems called Tiny Printf by Marco Paland (printf.c) and a UART driver, uart_driver.c. The UART driver code is expecting a 16550 UART like register set that's memory mapped to an offset of 0x80000000, and this is what's implemented for the HDL UART console module in the test bench.

Running the Simulation

We now have all the pieces to run the simulation: the HDL test bench, VProc, the ISS and the integration software, FreeRTOS, and the application. To run the program on Verilator, in the examples/riscV directory, there is a makefile.verilator make file. This can be configured in different ways to alter the compiling of the C++ code and the Verilator switches. The original environment was setup to run a test program written in assembler, and the default compilation is for this. To run the FreeRTOS program, some settings, common to all tests, are needed:

```
make -f makefile.verilator USRSIMFLAGS="-GTIMEOUTCOUNT=0 -GRISCVTEST=1" run
```

This would be the default compile and run command, where the program runs forever and won't timeout within the HDL. The simulation can be terminated, though, using the -n option in the vusermain.cfg file, as discussed earlier, which is what we'll use for our tests. An example output for a simulation run of 10 million instructions is shown below:

```
VInit(0): initialising DPI-C interface
  VProc version 1.7.6. Copyright (c) 2004-2024 Simon Southwell.

****************************
*   Wyvern Semiconductors   *
*  rv32_cpu ISS (on VProc)  *
*     Copyright (c) 2024    *
****************************

Entered main()
task1: count = 20000
task0: count = 10000
task1: count = 19999
task0: count = 9999
task1: count = 19998
task0: count = 9998
task1: count = 19997
```

```
task0: count = 9997
task1: count = 19996
task0: count = 9996

Number of executed instructions = 10.0 million (120826 IPS)

- test.v:156: Verilog $finish
```

The IPS number show the instructions per second value for this run of the simulator.

Using GTKWave

By default, Verilator is compiled with the `--trace` command line option specified by the make file, which is needed to allow the generation of data suitable for displaying in a waveform viewer. To actually generate this data, the test bench has code at the top level, enabled on a generic called `VCD_DUMP`.

```
initial
begin
if (VCD_DUMP != 0)
    begin
      $dumpfile("waves.vcd");
      $dumpvars(0, test);
    end
end
```

The generic can be set on the make command line as well:

```
make -f makefile.verilator USRSIMFLAGS="-GTIMEOUTCOUNT=0 -GRISCVTEST=1 \
                                        -GVCD_DUMP=1" run
```

As `--trace` is the default setting, it is useful to disable this if not generating waves, and setting the make file's `SIMTRACE` variable to be empty will do this:

```
make -f makefile.verilator USRSIMFLAGS="-GTIMEOUTCOUNT=0 -GRISCVTEST=1 \
                                        -GVCD_DUMP=1" SIMTRACE="" run
```

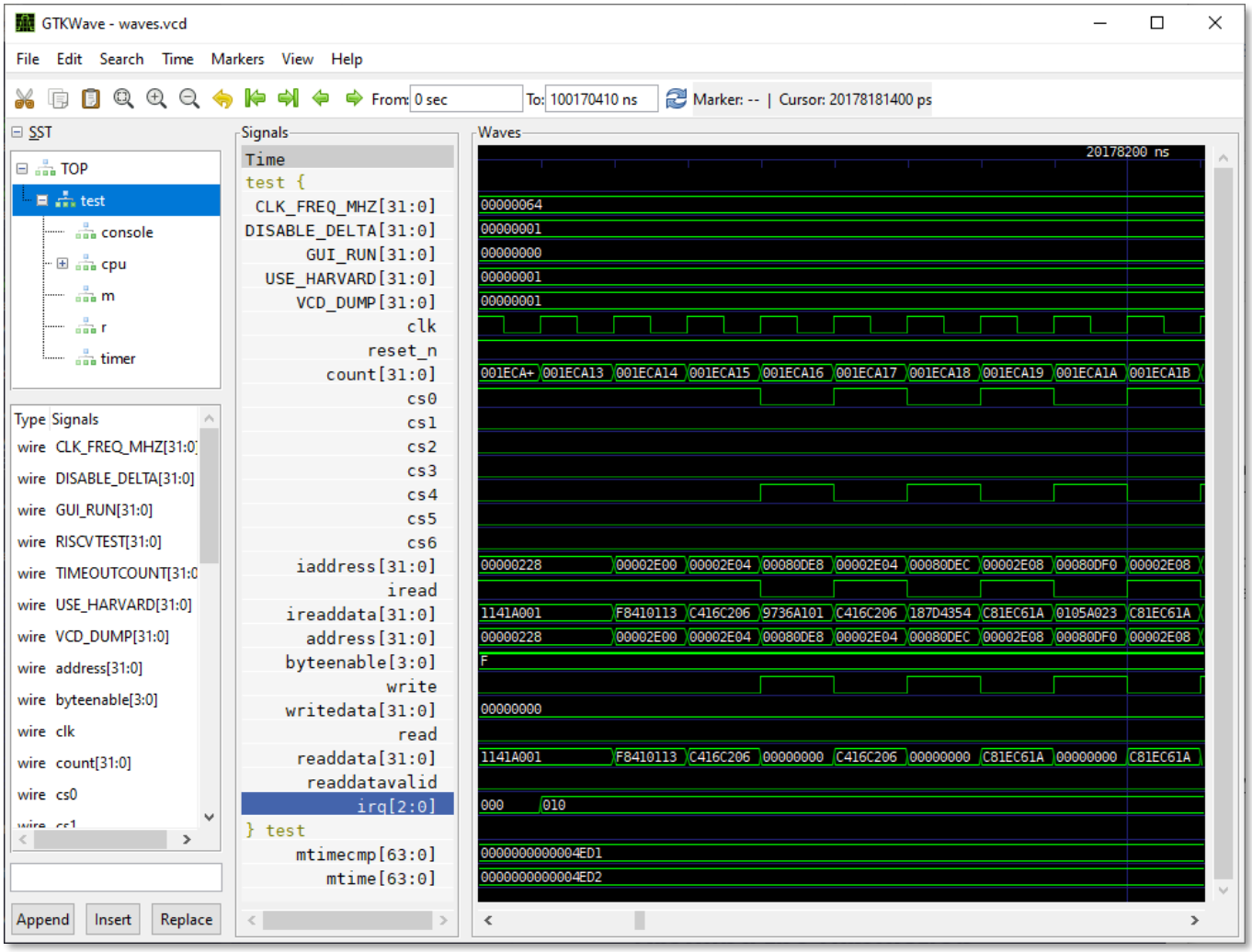
If `VCD_DUMP` is set (and `SIMTRACE` left at its default value), then the simulation will generate value change dump (VCD) data in a file `waves.vcd`, which can be viewed by GTKwave. This is simply a matter of running `gtkwave waves.vcd`. This will display a waveform window with the hierarchy on the left and an empty waveform pane. Clicking on any of the hierarchy blocks will list its signals in a pane underneath and these can be inserted/appended to the waveform window (even recursively, to get the signal of any sub-blocks). The setup of the waves displayed can be saved with `<ctrl>+S` and reloaded when `gtkwave` is run once more using:

```
gtkwave -a <file>.gtkw. waves.vcd
```

In fact , if the same file name (minus the file extension) is used as for the VCD file, e.g. `waves.gtkw`, then just a `-A` option is all that's needed and it will automatically load the `.gtkw` file. I.e.:

```
gtkwave -A waves.vcd
```

An example GTKWave window is shown below showing the raising of the timer IRQ and the code responding:



Profiling the Simulation

Verilator provides command line switches to enable 'profiling' of the C++ generated code. It actually just passes this on to the gcc/g++ compiler and linker but abstracts the details always using a single command line option, `--prof-cfuncs`, which will compile the code suitable for measuring the time spent in the individual functions of the code. The compiled code, when run, will now generate profiling data into a file, `gmon.out`, which can be processed by the `gprof` tool. But what is profiling and how does it work?

Profiling is the measurement of the proportion of execution time that's spent by individual parts of the code, either functions/methods, or individual lines of code, and providing a breakdown of these values to indicate where most of the time is spent so that bottlenecks in the code can be identified and possibly optimised to increase performance. There are two main ways to extract this kind of information—instrumenting the code or sampling. For instrumenting, the compiler will add extra functionality to the compiled code. For example, it may add incrementing a count associated with a function each time it is called. At the end of the execution, the counts for all the functions can be gathered and an output generated with this information. This method effectively modifies the code and will slow down the execution. Since it's likely that the *proportion* of time spent in particular section code of code is what's important, this may not be an issue. The sampling method sets up the executing code to be interrupted at regular intervals and the value of the program counter sampled and stored. At the end of execution, this sampled data is output. Using the debug information that was compiled into the code, the PC values can be associated with particular functions and even particular lines of code. This method is simpler and doesn't modify the code, but its measurements are statistical in nature due to the sampling, and sufficiently long execution runs need to be done to get relevant and accurate data. The output from multiple runs can be amalgamated and so, for example, the results from a suite of regression tests can be combined to get a better overall picture. The `gprof` tool uses both of these methods.

Using gprof to Process the Output

The `gmon.out` data is not human readable and has to be processed with `gprof`. This is simple enough with the following command example, as used for the profiling of this system, where the top level of the Verilog is `test` and the resulting executable is generated as `work/Vtest` (or `work/Vtest.exe` on Windows).

```
gprof work/Vtest gmon.out > gprof.log
```

The resultant log produces a lot of information in text format, which is more human readable than `gmon.out`, but needs some interpretation. It is divided into two parts—a flat profile and a call graph. The flat profile provides a table of functions with cumulative percentage of time, as well as information in seconds, both cumulative and self, and the number of calls. It is ordered on the seconds spent in just that function (self seconds). The call graph gives information of the call structure, showing the function calls and all the child calls from that function to build up a picture of execution, and contributions to the cumulative time for that function. It is quite difficult to interpret as it stands, but there are visualisers available, such as `gprof2dot`, which can turn this information into an actual annotated graph.

For our purposes, we will concentrate on the flat profile as the main measurement we want is the time spent in VProc and its running program versus the time spent in Verilator code and the functions modelling the Verilog. An example (edited) fragment of a flat profile from `gprof` is show below:

```
Flat profile:


Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 11.90      1.04     1.04 159324067     0.00     0.00  Vtest___024root___eval(Vtest___024root*)
  5.38      3.45     0.47 79662032     0.00     0.00  Vtest___024root___eval_nba(Vtest___024root*)
  4.12      4.24     0.36 79662032     0.00     0.00  VlDelayScheduler::resume()
  3.78      4.57     0.33 238986098     0.00     0.00  VlCoroutineHandle::~VlCoroutineHandle()
  2.52      5.35     0.22 79662034     0.00     0.00  VlDeleter::deleteAll()
  2.35      5.55     0.20 358479147     0.00     0.00  VlTriggerVec<3ul>::clear()
```

The example shows a function on the right, with the percentage time spent in that function in the first column. The next column is a cumulative addition of the time spent in each function, with the time just for that function after this. The number of calls to the function is shown in the fourth column, followed by the average time spent in a call (in milliseconds) with the average for the total time, including calls to child functions. For our performance measurements we can limit ourselves to just the percentage scores as only the relative times are important.

Some additional information can be extracted from the `gprof.log` information using the Verilator tool `verilator_profcfunc`, which can cross reference the data with actual lines of Verilog. E.g.,

```
verilator_profcfunc gprof.log > profcfunc.log
```

This produces some useful summaries as well as a table similar to the flat profile but annotated with the 'type' of the function, such as whether part of a Verilog block (`VBlock`), part of the Verilator library (`VLib`) or whether C++ code. An example set of summaries is shown below:

```
Overall summary by type:
  % time  type
   44.68  C++
   40.71  Common code under Vtest
    6.75  VLib
   10.75  Verilog Blocks under Vtest
   -2.89  Unaccounted for/rounding error

Overall summary by design:
  % time  design
   44.68  C++
    6.75  VLib
   51.46  Vtest
   -2.89  Unaccounted for/rounding error

Overall summary by module:
  % time  module
   44.68  C++
    6.75  VLib
   40.71  Vtest common code
    1.89  f_VProc
    1.08  mtimer
    1.26  riscVsim
    5.60  test
    0.92  uart
   -2.89  Unaccounted for/rounding error
```

Using all this information, there's still some extraction to be done to add up all the time spent in VProc functions (mercifully few), and in the code running on the virtual processor, which is mostly the ISS and the callback functions. Extracting this data with a few `grep` commands and some regular expressions allows the total percentage times to be calculated for the VProc code and the program running on it, with the rest of the C++ percentage, as calculated by `verilator_profcfunc`, being Verilator generated code. So what results were seen?

Performance Measurements

ISS Baseline Performance

Before we take measurements for the co-simulation environment it will be useful to take a baseline performance measurement of the standalone ISS. This was run for 200 million instructions and took 16 seconds to give a rate of 12.5MIPS. This varies

from run to run, so let's say **12.5MIPS ± 1MIPS**. This is fairly fast and is why we can use actual real-time timings to run an application program. This, then, is the stake in the ground with which to compare the co-simulation tests.

Co-simulation Performance

With the setup as has been described, with the ISS executing out of the HDL memory models, crossing the VProc boundary nearly very cycle, an ISS instruction execution rate was measured of just under **200KIPS**. Comparing this to the model running standalone as pure C++ baseline measurement discussed above, at >10MIPS, we have dropped two orders of magnitude in performance, though can still get hundreds of thousands of instructions per second, which is still useful, and faster than an HDL model. What was the VProc contribution?

This simulation was profiled (with no `--trace` option), for a run of 10 million instructions and got the following percentage times:

- 58.6% of the time was in C++ code
    - 12.9% of the time was in VProc code
    - 1.4% of the code was in the ISS code
    - The rest of the time (44.3%) was in Verilator C++ functions

This, of course, is the extreme case of interaction every cycle, but the model can be configured to run using its program from an internal memory model. Simply switching off using the external HDL memory (leaving just peripheral accesses as external), brought the instruction rate back to >**10MIPS**. This is somewhat artificial, as simulation time isn't advancing whilst running instructions (i.e. CPU infinitely fast and its timings not synchronised with the simulation). So code was added to the external memory callback function to give the ability to inspect the ISS model's own internal clock cycle count, and if 1000 cycles or more have elapsed since the last inspection, the simulation is allowed to advance for that period (via a `VTick` API call), synchronising simulation time with the ISS model's concept of time and reducing the number of VProc transactions by orders of magnitude. If a non-memory access is needed, the times are sync'd at that point as well, and the time difference reset, so that these accesses are not delayed and are in the expected places in the simulation. With this setup, the simulation ran at around **1.2MIPS**. Not the maximum rate seen in the standalone ISS, but a much better result. This new setup was profiled as well and produced the following results:

- 44.7% of the time was in C++ code
    - < 0.1% of the time was in VProc code

- 3.3% of the code was in the ISS code
- The rest of the time (41.3%) was in Verilator C++ functions

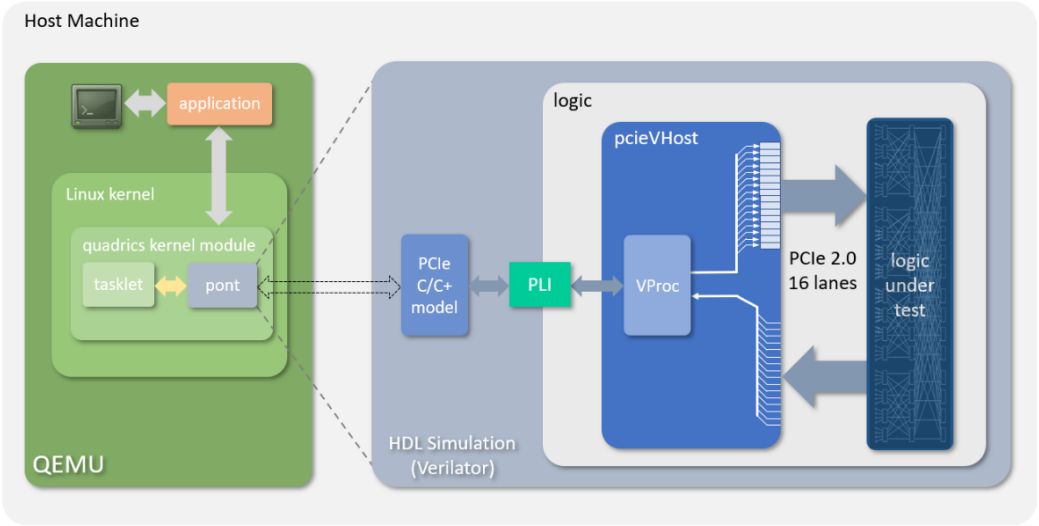
In addition to these two basic runs, the effect of Verilator switches was measured on the performance. In particular using the code optimisation switch (`-O3`) to compile the C++ code for speed, the `--trace` switch for wave data generation (though with no VCD data generated), and the `--timing` switch for support for delays and delta cycles (though none used). None of these appeared to make a significant difference.

The `-O3` switch being present or not made no discernible difference (maybe this is the default?). With `--timing` added, less than 1% reduction in execution time was seen, and with `--trace` (without generating VCD) just under 2% degradation in speed was observed.

Conclusions

The conclusions from this work, then, are that, with the full-on VProc access setup, VProc *and* the ISS model are still taking less than 15% of the compute time when fully saturating the VProc interface. The setup as it stands has all the real functionality in the ISS model and VProc interface, and it still makes only a smallish contribution to compute effort. The observed instruction rate (~120KIPs) is still limited mostly by the Verilator simulation. If the rest of the test environment was more realistic, with more HDL IP, such as complex units or sub-modules under test, the percentage contributions from VProc will quickly go even further down. Taking advantage of the internal memory model of the ISS raised the execution rate by an order of magnitude, by reducing the number of accesses across the PLI—also by an order of magnitude. The ISS model still used only around 4% of the execution time, but the VProc interface code did not even register on the profiling statistics (and would need many more cycles of simulation to do so, I suspect), making no real contribution to the overall execution time.

This ties in with the original results as measured way back when I first put VProc together. Here it was driving a PCIe endpoint of a complex HPC network chip using my PCIe C++ model (driven by kernel code running on QEMU), and updating every cycle, where it measured less than 0.1% time spent in the PLI. The diagram below shows this real-world verification environment:



So, my conclusion is that, in a practical real-world setup, the simulation rate would be still be limited by the logic simulation compute time, with minimal contributions from VProc code and the programs that it's running—even if accessing the simulation every cycle. Scenarios in which the VProc software is, say, a TCP/IP socket connected to another application could add significantly to the software side contribution to the run time, but this is why VProc does not use these kinds of method to communicate between the VProc API software and the simulation, and why it is a low overhead solution.

What VProc does allow is the utilisation of complex software models for very fast generation of test vectors over an interface that adds little to no measurable overhead, accelerating simulation to a rate where real-time or interactive execution could be possible.