WIND RIVER

System Architecture Exploration Using Wind River Simics

Table of Contents

Executive Summary ... 1
Investigating Hardware Accelerators ... 1
Setup Overview ... 2
  Software Program ... 3
  Hardware Device ... 4
  Implementation Effort ... 5
  Verifying Correctness of Hardware and Software ... 5
  Device Driver ... 5
Performance Measurements ... 6
  Model Abstraction Level ... 6
Automation of System Setup ... 6
Automation of Experiments ... 7
Experimental Results ... 8
  Initial Testing and Basic Optimization of Driver ... 8
  Mapping the Performance Landscape ... 8
  Hardware Accelerator Speed ... 10
  mmap Driver ... 11
  Final Evaluation ... 11
  Applying Simics Analyzer ... 13
Sanity Checks ... 14
  Software and Hardware ... 15
Conclusion ... 15

Executive Summary

Wind River Simics allows product teams to adopt a development methodology where physical system hardware is replaced by Simics virtual platforms running on a standard PC. This allows customers to efficiently define, develop, and deploy their products, usually much faster, with higher quality, and with fewer risks than when using physical hardware alone.

Simics virtual platforms run the same binary software as physical hardware and are fast enough to be used as an alternative for software development and testing. Simics virtual platforms are unique. They are fast and accurate enough to run a full software stack from hypervisor to application, and they guarantee repeatable software execution, full visibility/control of the virtual target hardware, and true reverse execution.

A key part of this development paradigm is to quickly get to running code and use real code for as much development as possible. For new hardware designs, the virtual platform is best created and used as an executable specification for the final system. These specifications should be used before the hardware design is finalized to allow them to be exercised by real software.

This paper provides an example of how a hardware accelerator can be explored, specified, and verified using a fast functional virtual platform, while knowing the main performance requirements on the accelerator and the characteristics of the software that will drive it.

You will see how to use Wind River Simics and fast functional simulation to do the following:

- Move a software function into a hardware accelerator.
- Define and refine the hardware-software interface.
- Analyze the performance requirements of the accelerator.
- Determine when a hardware accelerator is faster than keeping a pure software implementation.
- Provide an executable specification for the detailed hardware design.

Investigating Hardware Accelerators

In modern computer system design it is common practice to offload functions from software to hardware accelerators (also known as offload engines) to increase system performance and reduce the processor's computation load. The goal can be to increase throughput, decrease jitter, or reduce the power consumption of the system by using dedicated hardware and lower processor clock frequencies. A key part of the system design is to determine whether and when a hardware accelerator makes sense, compared to a pure software implementation on programmable processor cores. The preferred flow is to start with a software implementation of the function and quickly prototype how implementing the functionality in dedicated hardware affects the system performance and behavior.

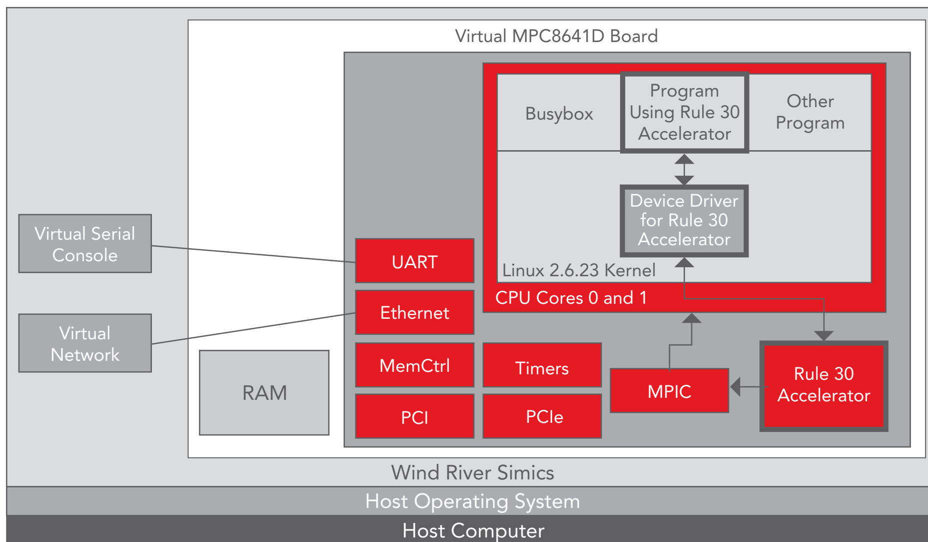

Figure 1: The virtual system setup

Such a prototype needs to include not just the computation in the hardware but how the software on a system drives the hardware and the specification of the hardware-software programming interface. With a fast transaction-level functional virtual platform, the interface can be defined, refined, and redesigned in a matter of hours because of the high level of abstraction used.

You will see how fast functional simulation is used to do such design exploration for a particular hardware accelerator candidate. The very fast simulation speed is leveraged to do a broad evaluation of implementation alternatives and cover a large number of different parameter values and software variations. Simics can run complete real software stacks to perform all tests using a complete Linux software stack on the target machine, making it possible to investigate exactly how the operating system and driver architecture affect overall performance.

Instead of actually designing a hardware accelerator for the algorithm in a hardware design language such as SystemC or Verilog, the software algorithm written in plain C is incorporated directly into a Simics device model at the functional level, written in Simics DML. The hardware design is not analyzed to determine how fast it would be in an actual circuit but rather a range of hardware computation latencies is assumed and the performance of the system is tested under each assumption. In this way, an executable specification is created for the hardware accelerator, and constraints for a later implementation step are provided.

Setup Overview

In this scenario, there is a software program implementing a particular algorithm (a cellular automaton known as rule 30), running on a dual-core Freescale MPC8641D Power Architecture–based system-on-chip (SoC). There is a benefit to offloading the execution of this algorithm from software to hardware, but we want to know how the intricacies of Linux affect the overall performance and how fast the hardware needs to be. Note that if a slower accelerator (higher computational latency) is possible, the hardware implementation can be smaller in terms of chip real-estate and use a slower clock frequency, reducing power.

Figure 1 shows the overall system setup. Since it is a virtual platform, a custom accelerator can be added into the MPC8641D SoC without much problem. The device is added to the virtual platform and mapped into the memory map of the processor cores. It is connected to the MPIC interrupt controller to enable it to interrupt the processor.

In addition to the device, a device driver is written so that the Linux kernel can access the device, and a test program is run in the user space to test the performance of the device from a user-level program.

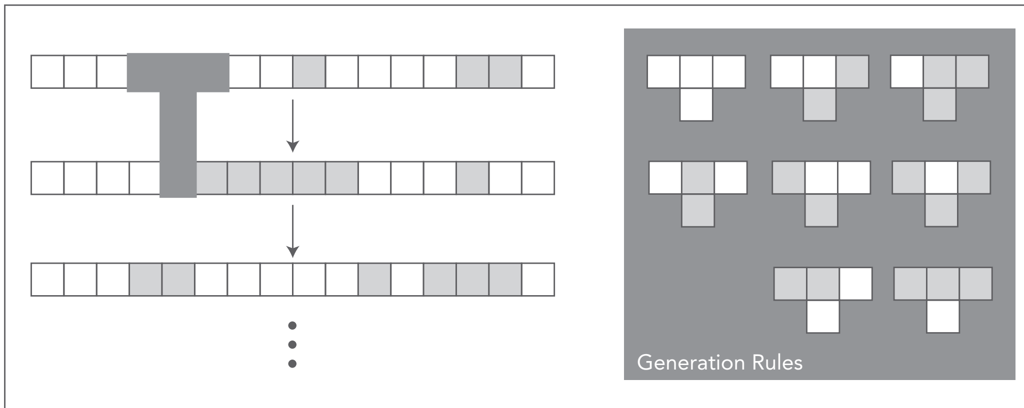

Figure 2: Rule 30 operation

Software Program

The program for which to investigate acceleration implements a cellular automaton known as rule 30 (a hardware implementation of rule 30 is available as open source from InformAsic). This is an interesting algorithm that computes one line of data at a time, generating very complex and unpredictable patterns from simple rules. As illustrated in Figure 2, each line consists of a number of binary elements, and the value of each element in a line is based on the value of the three elements above, to the left, and to the right of it.

The result of running this algorithm from an input, starting with a few bits set, is a characteristic "Christmas tree," as illustrated in Figure 3.

The core algorithm is very easy to describe in software, if a single C char is used to represent each bit. In this case, the

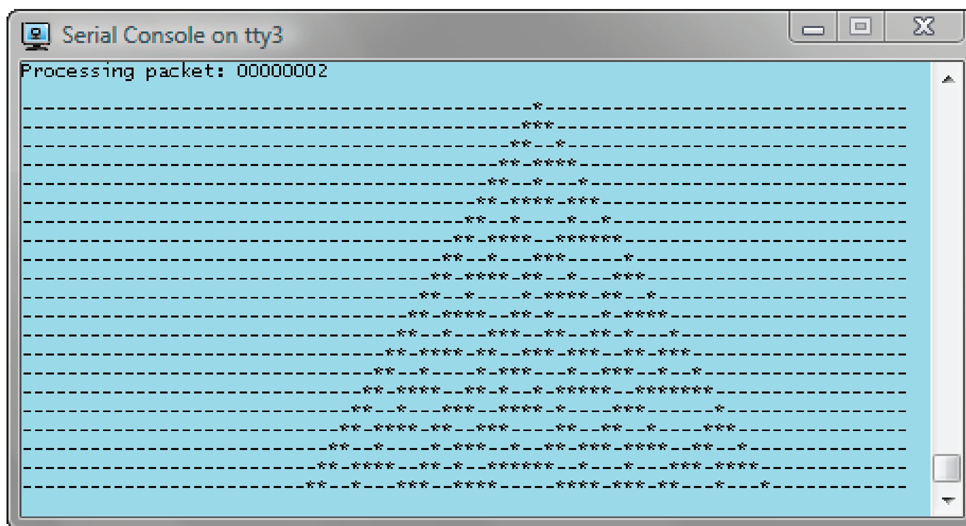

Figure 3: Rule 30 results

```
const uint8_t g_rules_array[8] = {0,                /* 000 */
                       1,              /* 001 */
                       1,              /* 010 */
                       1,              /* 011 */
                       1,              /* 100 */
                       0,              /* 101 */
                       0,              /* 110 */
                       0               /* 111 */
};

for(i=0;i<line_length;i++) {
      int  bit_left_index = (i==0) ? (line_length-1) : (i-1);
      char bit_left = source[bit_left_index];
      char bit_mid = source[i];
      char bit_right = source[(i+1)%line_length];
      int  index = bit_left * 4 + bit_mid * 2 + bit_right;
      dest[i]=g_rules_array[index];
}
```

Figure 4: Rule 30 char-based implementation

very straightforward C code can be used, as shown in Figure 4. The rules array corresponds to the rules illustrated in Figure 2.

An implementation has been written using a bit-based representation of the data. This requires some more code but is about as fast as the byte-based implementation. It also corresponds to the reasonable hardware implementation because it does not waste 8 bits to represent the value of a single bit of information.

The software program runs the rule 30 algorithm using a variety of implementation options, as instructed on the Linux command line when the program is started. To provide input data to the algorithm, a set of test initial lines (called "packets") were compiled into the software program.

The program tests a particular implementation in this manner:

- Loop over a certain number of packets (initial lines).
- Pick up a packet from the test data set and truncate it to the selected line length for this test run.
- For each packet, run the rule 30 algorithm for a certain number of iterations (when the successive lines that are generated are plotted, it results in the characteristic Christmas tree display).

Typically each implementation variant is tested by running it on 1,000 packets for 99 iterations per packet (generating a pattern of 100 lines), which means that for each test case, the core loop of the algorithm is gone through 99,000 times.

Hardware Device

The hardware device is written using Wind River Simics DML, a device modeling system provided with Simics. DML is used to describe the programming register layout of the device, the interrupt and I/O handling toward the processor and the management of computation delays. The core computation of the device is the bit-based implementation taken directly from the test software. What changes is how the input data arrives in the algorithm and how results are reported. Rather than as a function call from a test software driver, it is invoked over a memory-mapped device programming interface.

Figure 5 shows the registers in the programming interface, as expressed in the DML file. The programming interface works like this:

- Set operation parameters into the control, rule_set, and line_length registers.

```
bank regs {
    parameter register_size = 4;
    register version       @ 0x00 "Device version register";
    register control       @ 0x04 "Device control register";
    register status        @ 0x08 "Device status register";
    register reset         @ 0x0c is (write_only) "Reset register (write only)";
    register irq_num       @ 0x10 is (read_only) "IRQ assigned to device";
    register rule_set      @ 0x14 "Rule set (bit encoded)";
    register line_length   @ 0x18 "Line length (in bits)";
    register start_compute @ 0x1c "Start computation";
    register input[32]     @ 0x20 + 4*$i is (write_only) "Input buffer";
    register output[32]    @ 0xa0 + 4*$i is (read_only)  "Output buffer";
}
```

Figure 5: DML device programming interface

```
attribute time_to_result {
    parameter documentation = "Delay in from start of
                              operation to results are available";
    parameter type = "f";
    parameter allocate_type = "double";
    parameter configuration = "optional";
}
```

Figure 6: DML device latency coding

```
simics> ra0->time_to_result = 10.0e-9
```

Figure 7: Scripting latency

- Write input data to the input array, up to 1024 bits in units of 32 bits. This representation is bit-based, with one bit per cell in a line.
- Start the computation by writing to start_compute.
- Wait for an interrupt to signal completion, or spin on the value of the status register to set the operation complete flag.
- Read results from the output array.

Note that the details of the register implementation are described later in the DML file. The overview declaration just provides the names and offsets of the registers for an easy overview of the programming memory map.

The operation parameters that can be set are interrupt notification and whether the device should copy the output result to the input array on operation completion. This would remove the need for the software to copy output data to the input array for chained computations. This is the mode known as "hwo" in the following experiments.

The latency to compute a result is set as a parameter in the device, using a Simics attribute (device parameters that can be set and read at any point during a simulation). Figure 6 shows the DML code to create the attribute. To change the latency, simply issue the Simics CLI command shown in Figure 7. This can be done at any point during a simulation run.

Implementation Effort

Implementing the hardware accelerator model at this level of abstraction is fairly quick and took only a few working days. Most of the time was spent iterating the device programming interface and the device driver implementation, to create a convenient programming interface for the software and to test that the complete software stack worked. Such evolution of the hardware-software interface is key to designing a programming interface that make sense from the perspective of the driver and the BSP.

The complete model source code is about 560 lines, including comments and debug printouts as well as the C-based computation kernel of around 100 lines. The code also contains version registers for the hardware, interrupt handling, unit testing support, and error checking.

Verifying Correctness of Hardware and Software

To verify that all software variants and hardware acceleration variants work correctly, the test software has some special modes that run two different implementation variants and compare the outputs. The byte-based implementation is the golden reference model and validates the software bit-based implementation as well as the hardware implementation (in all its operation modes) against it. This methodology found some bugs in the initial bit-based implementation regarding certain corner cases as well as an embarrassing bug indicating the output data of the computation in the hardware wasn't read.

Device Driver

The Linux device driver created for the device is a "char" driver and uses the Linux standard write() and read() calls to drive data into and read results from the device. ioctl() is used to set parameters and query the device state. As a result of initial performance exploration, a mmap() function was implemented where the user-level software can directly access the registers of the hardware.

The driver is compiled as a Linux kernel loadable module, to make it easy to change it on the target machine. If it was compiled into the kernel, each device driver change would have required a complete rebuild of the kernel as well as a reboot of the target system. Instead a checkpoint of a booted machine is used as the starting point for each iteration and the latest revision of the hardware module and device driver is added on the fly. In this way, iterating hardware and software changes takes only seconds.

Performance Measurements

Once the basic infrastructure is in place, performance data is collected. The simulation was set up to detect the start and end of the computation kernel as well as the start and end of each packet. Simics OS awareness was used to distinguish between user-mode and kernel-mode time.

The data collected was the minimum, maximum, and average of the following times:

- Compute time per line, start to end (the compute kernel)
- Compute time per line, time spent in user mode
- Compute time per packet, start to end
- Compute time per packet, time spent in user mode

Simics magic instructions (no operations in the code that Simics identifies that do not affect the execution semantics) delineate the start and end of each processing unit. In this way there are very precise measurements that do not suffer the variability inherent in using a serial console to print out the start and end times. In physical hardware, this would be similar to driving an output pin high and low and looking at the timing in an oscilloscope.

Model Abstraction Level

These simulations were performed using the standard Simics software timing (ST) level of abstraction. This is less detailed than SystemC TLM-2.0 LT in that memory access latencies are not accounted for, models are not allowed to do any kind of waiting before returning from a function call, and mandatory temporal decoupling is used to gain about an order of magnitude in simulation speed. However, we do model system time, clock interrupts, and the hardware latencies that matter. The processor is a fast in-order model with a fixed execution time per instruction, and there is no cache model or model of memory latency. The hardware model of the accelerator accounts for time by sending an interrupt to the processors after each computation is complete, and the delay from the point of starting a computation to the signaling of completion is varied to test different hardware speeds.

Automation of System Setup

Since we wanted to run thousands of test cases with various parameters, the loading and execution of the test software was optimized and automated using Simics check-pointing and scripting.

First, the standard Linux 2.6.23 kernel for the MPC8641D was booted on the virtual MPC8641D machine, and we took a checkpoint after the boot completed and the target software arrived at shell prompt. The checkpoint contains the complete software and hardware state and can be brought back into Simics almost instantaneously. Note that the checkpoint is completely portable and can be opened on any Simics installation on any host machine, which makes it possible to run several different simulation runs in parallel on multiple hosts.

Starting from the checkpoint, a series of scripts do the following to put the target system into a state where measurements could be performed:

- Load checkpoint.
- Add the rule 30 accelerator device.
- Load the device driver onto the target and initialize it.
- Load the test software onto the target.
- Initialize a Linux process tracker.

Adding the device was done using Simics' ability to add objects to a simulation at arbitrary points in time. Figure 8 shows the Simics script operations for creating a new accelerator and connecting it to the target memory map and MPIC interrupt handler. The line starting with @ is inline Python scripting, which is used to access certain rare Simics API functions from the Simics command line. The ccsr_space.add command updates the memory map for the on-chip devices section of the MPC8641D.

```
@SIM_create_object("rule30_accelerator","ra0",[["queue",conf.cpu0],["irq_dev",[conf.
pic,'internal_interrupts']],["irq_level",23], ["time_to_result", 1e-3]])
ccsr_space.add-map ra0:regs 0xf0000 0x200
```

Figure 8: Adding the rule 30 hardware accelerator to the setup

Serial Console on argo_0.soc.uart[0]

```
~ # mount /host
[simicsfs] mounted
~ # cp /host/sample_rule30_driver_linux_2.6.23.ko .
~ # umount /host
~ # insmod sample_rule30_driver_linux_2.6.23.ko phys_addr=0xf80f0000
Initializing rule30 device driver
Rule30 device at 0xf80f0000
Mapping rule30 device from p:f80f0000 to v:0xf1024000
Device hardware recognized, version 1.0
Allocated device numbers: (240, 100)
Registered Linux IRQ 39.
Initialization of rule30 device successful
~ # mknod /dev/rac c 240 100
~ # ./rule30.elf hardware_bit 1 10 78 2 /dev/rac
hardware_bit mode selected
[hardware_bit] ioctl set rules to 0x1e
[hardware_bit] Processing packet, seq:     0 length   78
-----------------------------------*-------*-----------*----------------------
[hardware_bit] - Done
-----------------------------------*-------*-----------*----------------------
----------------------------------***-----***---------***---------------------
---------------------------------**--*---**--*-------**--*--------------------
--------------------------------**-****-**-****-----**-****-------------------
-------------------------------**--*----*--*---*---**--*---*------------------
```

Figure 9: Automated loading of software on the target

To load the software in an efficient way, the simicsfs virtual file system on the target was used. Simicsfs allows for mounting any directory on the host disk as a file system and accessing the host files as if they are local files on the target machine. The driver and test program were copied from the host to the target every time there was a run (they are cross-compiled on a Linux x86 host), which means the relevant parts of the target software stack can be updated without rebuilding a target disk image and rebooting the target.

The target system operations are automated using Simics serial console scripting, where Simics scripts wait for prompts (or other output) and enter interactive Linux command-line commands. Figure 9 shows a screenshot of the state of the target serial console as the target software application is loaded after the device driver is loaded and initialized. Note that the simicsfs file system (/host) is mounted and unmounted for each file. Do not necessarily run a simulation with simicsfs mounted because it might impact repeatability and determinism.

All the previous steps were wrapped into a single Simics script, which when invoked brings the turnaround time to less than a minute for a recompile and test of the device model, device driver, or the test program. All the user needs to do is make changes and start Simics with the script to set up the hardware and load the software. Compared to using a remote physical hardware board, this is much more convenient and much faster.

Automation of Experiments

Based on the automated setup of the target machine for experiments, another script was built to perform the actual performance experiments. This script does the following:

- Configures experimental parameters as a set of Simics command-line variables
- Sets up the target machine
- Loops over a range of packet sizes, and for each packet size runs the test program in a variety of modes with a variety of hardware latency parameters
- Collects timing data for each packet size and mode and writes it to a file for later processing in Excel

The script controls the mode of execution and behavior of the test program by giving it different Linux command-line parameters. The hardware is controlled by changing the latency setting using a Simics attribute (note that this means the latency of the accelerator is not constant through a single simulation run but changes during the run before each test program execution). The target system is not restarted between each run. The variation caused by the execution history of the target Linux is insignificant. If the target Linux state mattered, each run would be started from the same checkpoint.

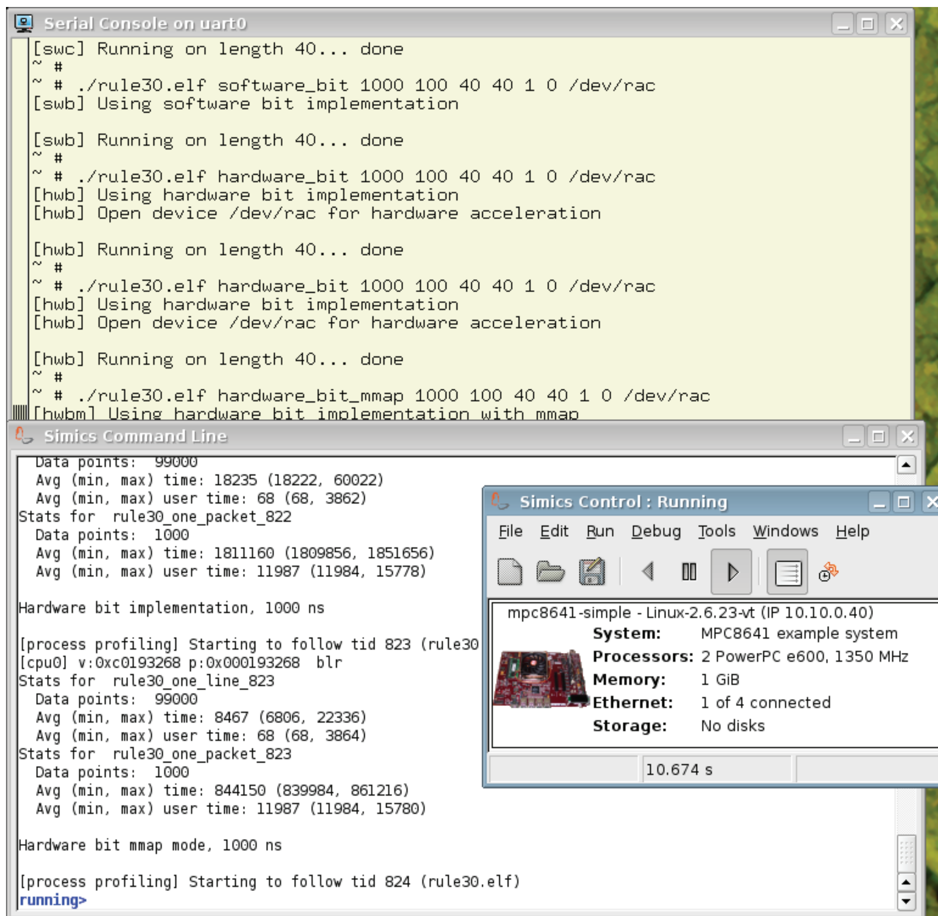

Figure 10: Automated testing and performance measurement

Each run can cover many minutes of target time and take a few hours to run on the host. The automation lets it run completely unattended, including on multiple host machines, running many experiment variants in parallel.

Experimental Results

Over the course of the software-hardware integration, a series of experiments was performed. The following describes the most important steps in the process.

Initial Testing and Basic Optimization of Driver

In very early testing of the device driver, it was noted that the only sensible way to push data into the accelerator was to use a single write() call for the entire set of data and a single read() call to retrieve all results. Writing input a word at a time is many times slower because the overhead of a Linux kernel call is fairly significant, as described later. This simplified the driver because it did not have to maintain a file position abstraction. Another early optimization was to use static memory allocation in the driver, which sped things up by about 25%. The initial version of the driver used kmalloc() and kfree() to create and delete a kernel buffer for the data from the user space each time read() or write() was called.

Next, there were some odd variations in the measured end-to-end execution times because the Linux kernel was scheduling the test process on different cores at different times. This is part of the normal functionality of the kernel, but since a clean comparison of implementation alternatives was needed, with as many variables under control as possible, processor affinity was used to tie the compute process to core 0 for all tests.

Quite a bit of information about how to do Linux device drivers was clear, and it is possible to evaluate their performance in a fast functional simulator. The volume of test cases executed brought attention to the scheduler. Only about one run in 30 was affected, and it required quite a few program runs to make the issue stand out.

Mapping the Performance Landscape

To get an overview of the performance landscape with the hardware accelerator and the basic device driver, a set of experiments was run where the length of the packet was varied from 10 to 1020 bits, in increments of 30 bits. The hardware latency was varied from 1 ns to 10000 ns, logarithmically. The purpose was to determine the order of magnitude of speed where the hardware accelerator would become relevant.

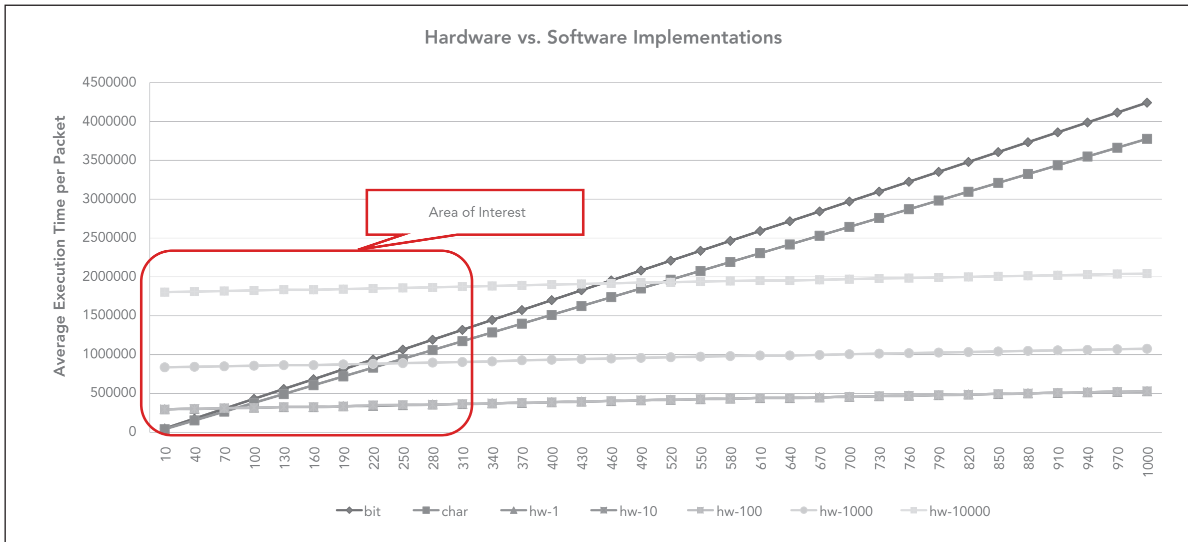

Figure 11: Initial performance measurements for hardware and software

Figure 11 shows the results: "bit" is the bit-based software implementation; "char" is the reference implementation; and "hw-x" is the hardware accelerator with a latency of x ns.

The times are the average computation times for each packet, over the 1,000 packets processed, for each packet length and operation mode, expressed as clock cycles at 1350MHz.

The area of real interest to compare hardware and software is below 256 bits of length. The hardware option with a latency of 10000 ns can be ignored because it does not win over software until a packet length of around 500 bits.

A new experiment was performed with packet lengths of 8 to 256, with a step of 8 bits. The results are presented in Figure 12. The following conclusions are drawn from this more detailed investigation:

- The hardware accelerators that are at 100 ns or faster outperform software at packet lengths above 88 bits.
- The hardware accelerator with latency 1000 ns is likely not an attractive option, but if packet lengths are long in certain use cases, it could allow for building a very simple and small hardware implementation with a slow clock speed.

Making a hardware accelerator faster than 100 ns appears to have no benefit at all.

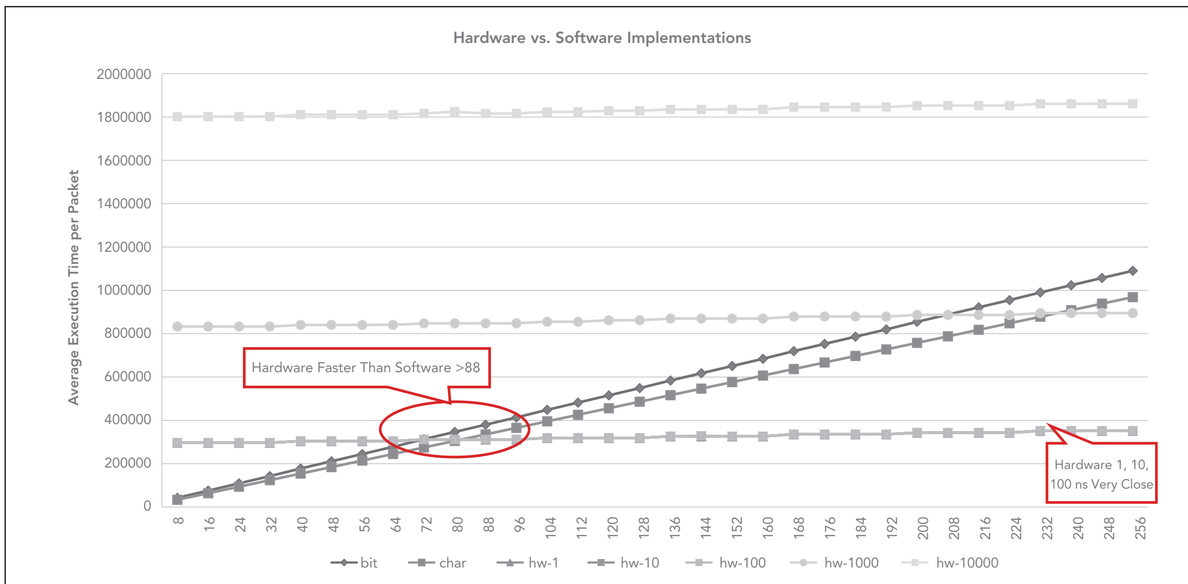

Figure 12: Zoom in at packet lengths of 256 bits or less

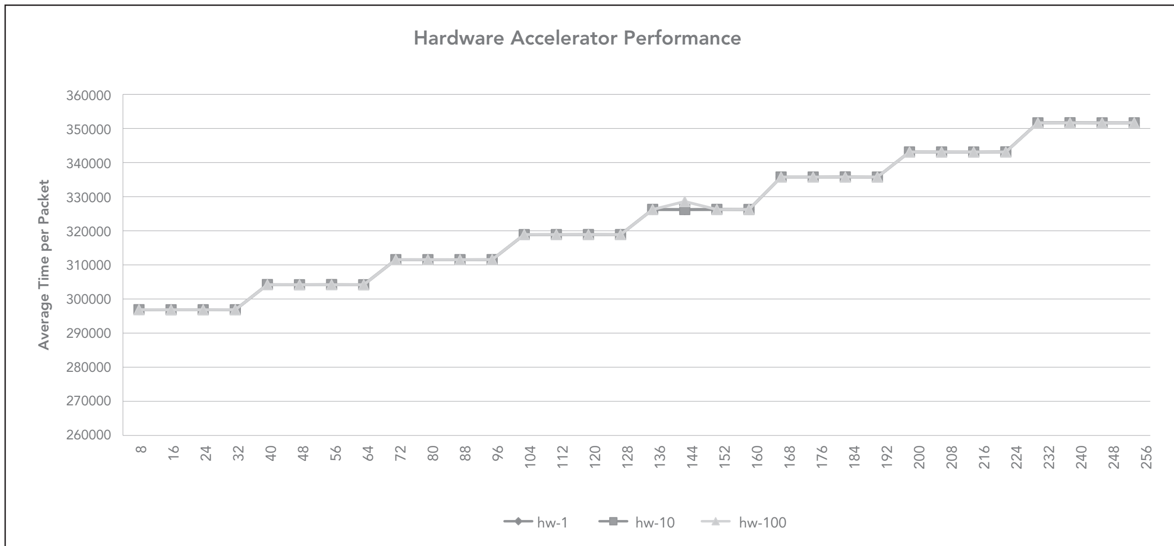

Figure 13: Hardware accelerator results

Hardware Accelerator Speed

The data shows that the execution time when using the hardware accelerator at 1, 10, and 100 ns is almost identical. To check this, the data is graphed in Figure 13. The conclusion from this is obvious: The driver overhead in the device driver negates any fundamental benefit from faster hardware. Essentially the software is not able to drive enough work through such a fast hardware accelerator to keep it busy. The fact that the execution time is independent of the speed of the accelerator (i.e., the lines for 1ns, 10ns, and 100ns overlap completely) and is only marginally dependent on the size of packets (note the scale in the graph) clearly shows there is a large fixed overhead associated with using the hardware accelerator.

If the performance of the hardware-accelerated option at this point is satisfactory, the specification stage could end and the hardware team could create an accelerator with a target latency of 100 ns, or maybe 10 ns to have some room to absorb later software improvements.

This demonstrates the power of fast functional simulations to map out the performance landscape and determine which optimizations make sense and which do not. Without committing to any kind of detailed hardware implementation, the required

```
// Input data
for(i=0;i<words_to_process;i++) {
  write_rule30_register(rule30_hw_acc_va,
                        (RULE30_INPUT_BASE + (4*i)),
                        in_line[i]);
}

// Kick compute
write_rule30_register(rule30_hw_acc_va, RULE30_START_COMPUTE, 1);

// Wait for complete
while( ((read_rule30_register(rule30_hw_acc_va, RULE30_STATUS))
      & RULE30_STATUS_OC_MASK) == 0) {
   // Busy wait loop
}
// Clear the completion bit in HW
write_rule30_register(rule30_hw_acc_va,
                        RULE30_STATUS,
                        RULE30_STATUS_OC_MASK);

// Read output
for(i=0;i<words_to_process;i++) {
  out_line[i] = read_rule30_register(rule30_hw_acc_va,
                        (RULE30_OUTPUT_BASE + (4*i)));
}
```

Figure 14: Using mmap() in the test program

performance is determined when using a realistic software stack. These experiments if done with a bare-metal setup would have drawn quite different conclusions because it removes the operating system and driver model overhead from the equation.

mmap Driver

It can also be interpreted that the software stack needs to be improved to make the overall system really efficient. For example, if it is assumed that the latency of the accelerator is known to be 10 ns in the physical hardware, the data indicates that the software group needs to do some optimization to their part of the system because there is no benefit from fast hardware compared to slower 100 ns latency hardware.

There are two obvious alternatives to reducing the overhead of the driver model. One is to put the core loop of the application into the device driver, which is not considered elegant but is a solution used in practice in high-performance Linux systems. The other is to offer user-space programs direct access to the control registers of the hardware, using the mmap() function in Linux. The mmap option was explored. The core of the code in the test program is shown in Figure 14. It is very similar to bare-metal code. Note that it uses a busy loop to wait for the hardware to complete an operation.

Final Evaluation

With this optimization in place, the final evaluation run was performed. This comprised a total of 352 complete end-to-end runs, each processing 1,000 packets for 99 generations (generating a picture 100 lines long). Thus, at each data point, 99,000 iterations of the core loop are executed. We tried 32 different lengths between 8 and 256, and in each length 11 different variants were tried. The following cases were tested:

- Software char and bit
- Hardware accelerator with the normal driver, with latencies of 10000, 1000, 100, 10, and 1 ns
- Hardware accelerator with mmap optimization, latencies of 1000, 100, 10, and 1 ns
- Hardware accelerator with register input-output latching optimization, and a latency of 10 ns

The total run time on the target was 134 seconds, and some 200 billion target instructions were executed in this time frame (a combined count on the two cores). This took about two and half hours to run on a contemporary PC. Most of the simulator overhead came from the very detailed performance measurements, using Python scripting. Running without that scripting approximately doubled the speed. Still, the simulation ran usefully fast to provide a lot of good data in a relatively short amount of time.

Figure 15 shows the results for all modes and a range of hardware latencies. The following conclusions can be drawn:

- The data points labeled "hw-mmap-x" show mmap-optimized hardware access is well worth the implementation complexity, providing the fastest implementation for all packet lengths.
- Even with mmap optimization, the difference between 1, 10, and100 ns hardware latencies is insignificant. The hardware team can produce a 100 ns-latency hardware block without risking losing any significant performance.

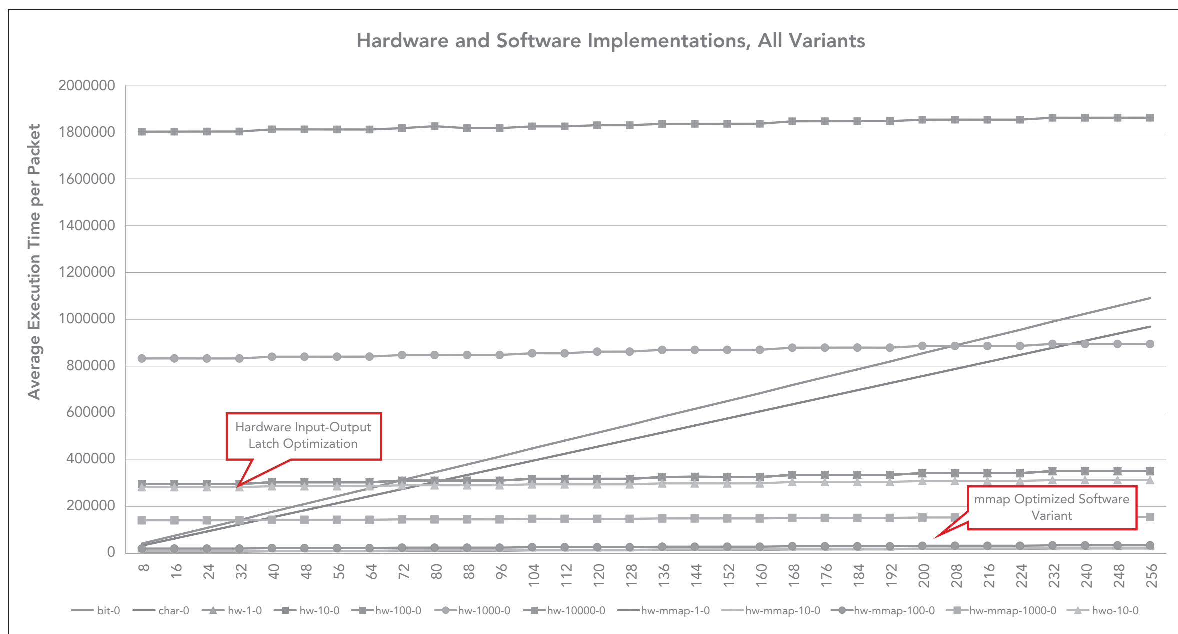

Figure 15: Test runs including mmap

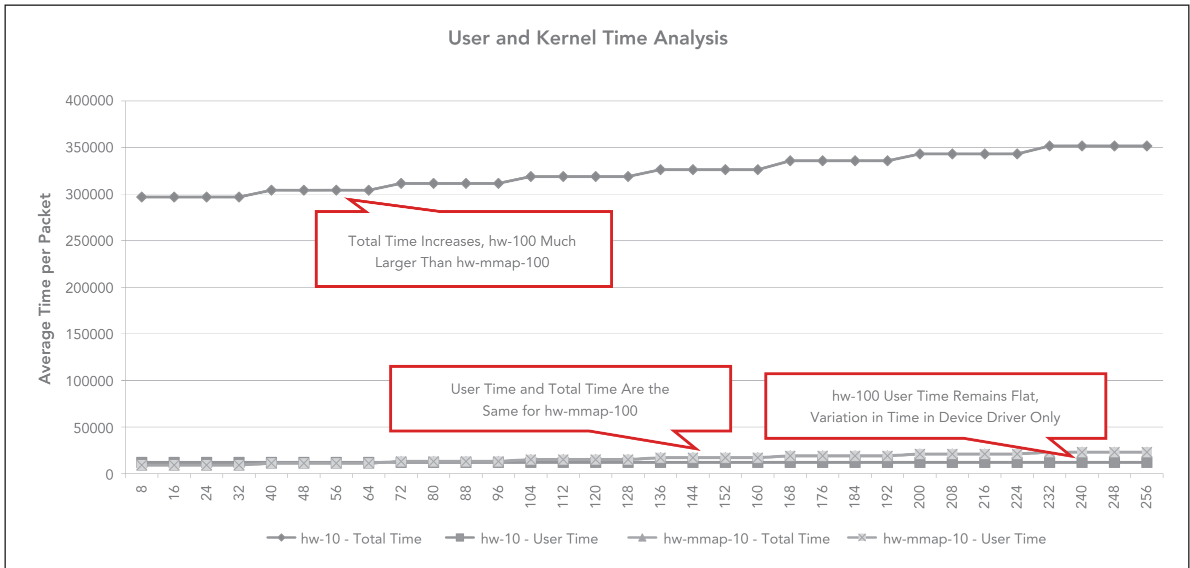

Figure 16: Investigating user and kernel times

- The latched optimization (hwo-10) performs slightly better than the default driver. This optimization is therefore not meaningful to require it from an actual hardware implementation. Using the fast functional analysis, the requirements on the hardware can be simplified to remove useless features that look good on paper.
- Changing the software architecture and driver structure has a greater impact than optimizing the hardware performance. From a system design perspective, this would indicate that it makes sense to use automated tools that deliver fairly simple hardware with a low investment in development cost and spend more effort on getting the software optimized.

Figure 16 shows that less time is spent in the kernel to get the good performance of mmap-optimized software.

The total execution time and the time spent in user mode (which is a component of the total time and never bigger than the total time) are plotted for two selected execution variants: hw-100 and hw-mmap-100. The results are striking:

- The regular device driver spends very little time in user mode but much more time overall. The time in user mode is also flat, as all variability comes from the device driver, which corresponds to the gradual increase in total time.
- The mmap-optimized driver spends all its time in user mode, and the execution in user mode therefore increases slowly as it goes to longer packets.

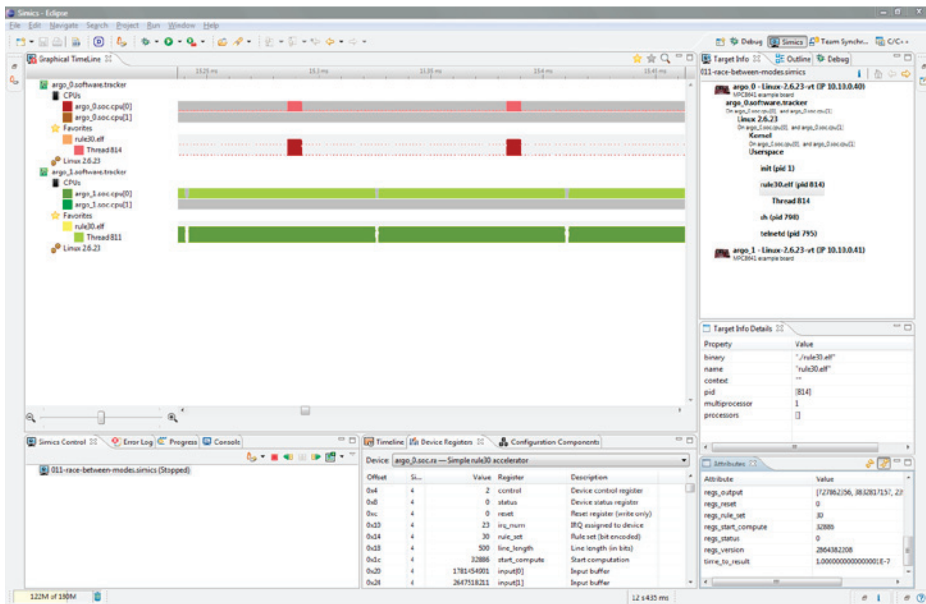

Figure 17: Wind River Simics Analyzer on mmap and default driver modes

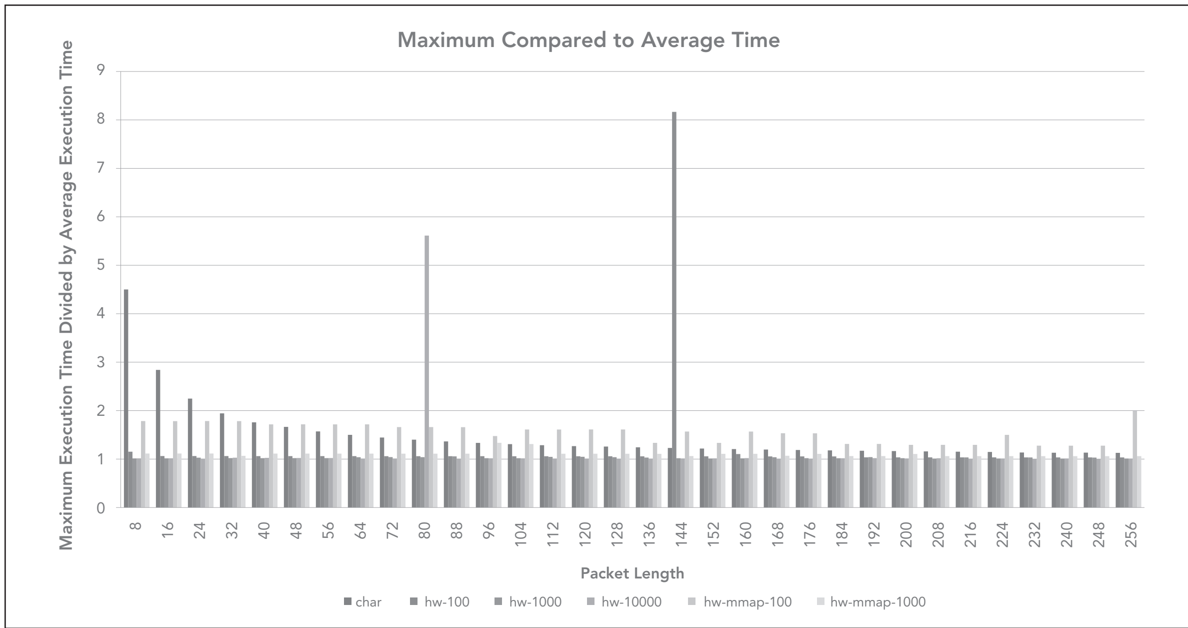

Figure 18: Investigating jitter

Applying Simics Analyzer

Using Wind River Simics Analyzer, some additional insight is gained in the execution patterns of the various driver modes. Figure 17 shows the execution history timeline. The argo0 machine is running the driver in its default mode (which is known to be slow), and the argo1 machine is running it in mmap mode. Note how on argo0 the kernel (gray time on the processor cores) dominates the execution. On argo1, in contrast, the program is almost always running in user mode. The only exception is the short burst of kernel activity that results from a completion interrupt from the hardware accelerator.

With this additional insight, you might consider adding a new mode to the accelerator, where it does not actually send completion interrupts but relies entirely on polling to signal completion to software.

Jitter

Using an operating system affects the execution time of software, making it more variable as interrupts and task switches interfere with the execution. This effect is also captured in the Simics virtual platform, as shown in the graph in Figure 18. It compares the maximum observed execution time with the average observed execution time for some execution modes and packet lengths.

This diagram has some interesting information in it:

- For the char mode, the jitter gets lower as the packet lengths increase. For a compute-intense workload like this on a lightly loaded system, the jitter should decrease as the overall execution time gets longer.
- For the hardware-based modes, jitter is higher for short packets but not with as pronounced an effect.

There are some occasional spikes of very high jitter for the hw-x modes. This is likely due to the fact that they give up the CPU, waiting for a completion interrupt, giving the OS a chance to swap the process out. It also indicates that for reliable real-time end-to-end latencies for processing, it is necessary to do some more work on the software stack (but hw-mmap seems more stable because it never gives up the CPU in the same way).

Sanity Checks

The previous investigation was performed using a fast functional model of the processor and memory system, where the caches were not modeled, which is a necessary optimization to run large-scale software loads on a virtual platform. To check that this kind of optimization does not skew the results, some overnight runs were done with an added cache model.

The cache parameters were chosen not to reflect any particular machine but to clarify the performance impact of caches on the workload in general. Three configurations were tested:

- 16KB cache, latency 100 cycles to main memory
- 16KB cache, latency 50 cycles to main memory
- 128KB cache, latency 100 cycles to main memory
- Hardware vs. hardware 2

The first check is to compare the main hardware accelerator modes with the cache modes (and without cache).

As shown in Figure 19, adding a cache model does provide some new insights, even though it does nothing to impact the superiority of hw-mmap over hw. The cache model increases the execution time of the hw mode by about 10%. The same is true for hw-mmap. Therefore, caches have no impact on the previous results, and a fast functional simulator can safely be used for this type of executable specification and early performance requirement work.

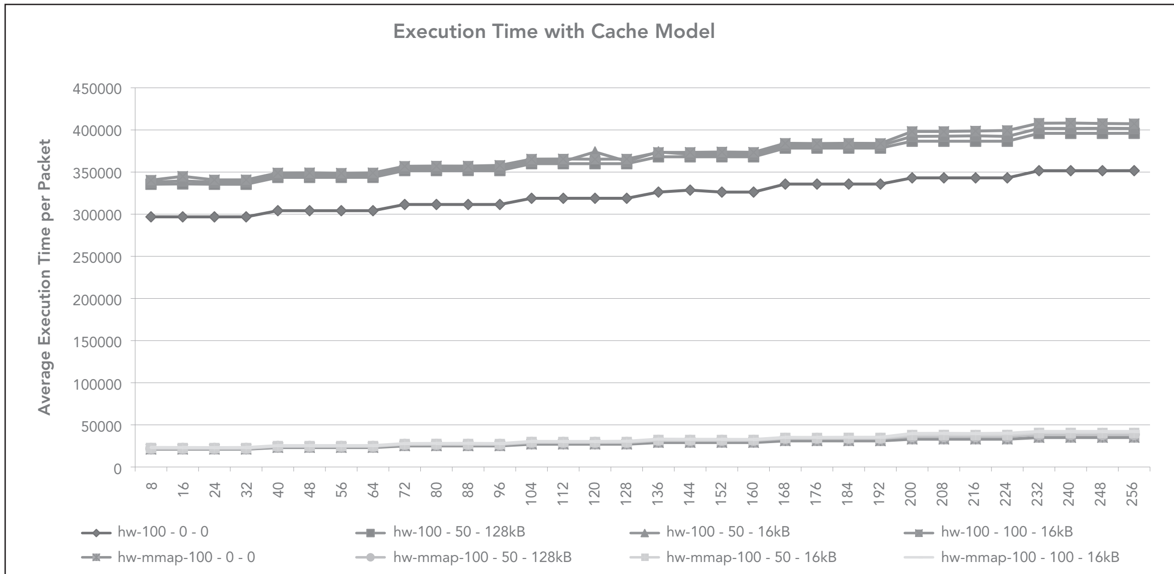

Figure 19: Hardware modes with cache model

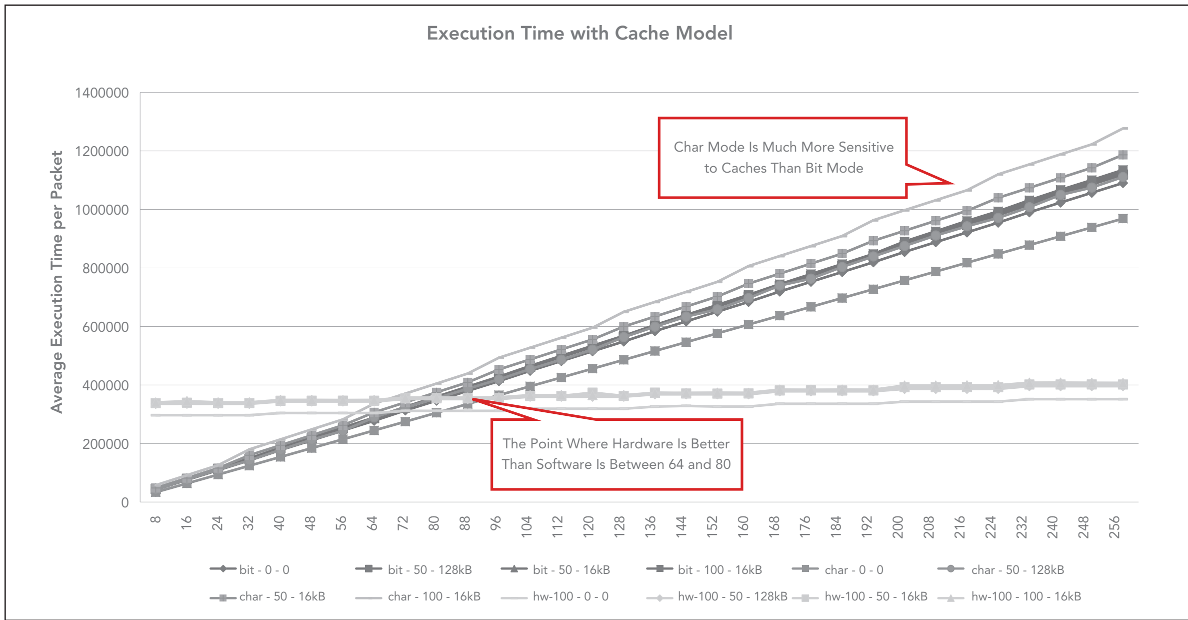

Figure 20: Software modes with cache model

Software and Hardware

In Figure 20 the performance of the two software modes is plotted with caches as well as one hardware mode for reference. It shows some interesting results:

- Without a cache, the char-based implementation is faster because it executes fewer instructions per iteration. However, most of these are memory accesses to various arrays.
- With a cache, the bit-based implementation is faster because it performs much fewer memory accesses per iteration. This implementation is not shown here, but it essentially only accesses memory twice for each 32-bit segment of a line processed.

This indicates that if the hardware accelerator is skipped altogether and a software-based implementation is used, it would make sense to use the bit-based implementation if the memory system of the target machine is expected to be slow or caches to be small.

The point where hardware is faster than software moves to short packet lengths as memory gets slower. Compared to the char-based mode, the crossover moves from 88 to 64, while the change is much less pronounced for the bit-based software implementation. However, the impact does not really change the high-level view of the accelerator design. The conclusion that hardware acceleration is worthwhile at packet lengths around 90 or so for the standard driver model and always for the mmap-based model remains true.

The conclusion is that investigating cache behavior is a good way to check the overall sanity of results, but it does not have a significant impact on high-level results. The differences between software stack variants and hardware accelerator latencies are much larger than the effect caused by caches. From a system-design perspective, most everything can be learned from a pure fast functional simulation.

Conclusion

Wind River Simics can be used to perform system design and specification with respect to hardware offloading of critical software functions. Working at the software timing level of abstraction, a complete view of the performance landscape is obtained because large-scale workloads involving hundreds of billions of target instructions can be executed in a reasonable time frame. Overall the measurements presented here required more than a trillion target instructions to be executed.

Because of the ease of modeling offered by fast functional models, the Simics DML modeling tools, and the reuse of an existing compute kernel, a complete hardware model could be quickly created and tests performed using a complete real software stack, including an operating system and device drivers.

The result of the work is significant insights into the important performance parameters for the design as well as an executable specification of a hardware accelerator. The hardware accelerator specification includes the complete software-facing programming interface as well as requirements on operation latencies and golden reference for the results calculated.

Simics features were also leveraged to completely automate the evaluation of performance (and checks of correctness) after each change to the software stack or hardware accelerator.

WIND RIVER

Wind River is a leader in embedded and mobile software. We enable companies to develop, run, and manage device software faster, better, at lower cost, and more reliably. www.windriver.com

© 2010 Wind River Systems, Inc. The Wind River logo is a trademark of Wind River Systems, Inc., and Wind River and VxWorks are registered trademarks of Wind River Systems, Inc. Other marks used herein are the property of their respective owners. For more information, see www.windriver.com/company/terms/trademark.html. Rev. 08/2010