

SYSTEM ARCHITECTURE SPECIFICATION AND EXPLORATION

USING A FAST FUNCTIONAL SIMULATOR

APRIL 2009

JAKOB ENGBLOM

WWW.VIRTUTECH.COM

INTRODUCTION

Virtualized Systems Development™ is a methodology where virtual platforms are used to enable process change in how you define, develop, and deploy integrated hardware-software systems. A key part of this development paradigm is to quickly get to running code, and to use real code for as much development as possible. The virtual platform is best used as an executable specification for the final system, and should come before the hardware design is finalized.

In this whitepaper, we provide an example of how a hardware accelerator can be specified and verified using a fast functional virtual platform, to the point that we know the main performance requirements on the accelerator and the characteristics of the software that will drive it.

We will show how you can use Simics and fast functional simulation to:

- 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 usefully 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) in order to increase system performance and reduce the processor computation load. The goal can be to either increase throughput, 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 thus to determine if and when a hardware accelerator makes sense as opposed to a pure software implementation on programmable processor core(s). The preferred modern 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.

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

This article describes how we use fast functional simulation to do such design exploration for a particular hardware accelerator candidate. We leverage the very fast simulation speed to do a broad evaluation of implementation alternatives, to cover a large number of different parameter values and software variations. We use the ability of Simics to 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 affects overall performance.

Instead of actually designing a hardware accelerator for our algorithm in a hardware design language like SystemC or Verilog, we incorporate the software algorithm written in plain C directly into a Simics device model at the functional level, written in Virtutech DML. We do not analyze the hardware design to determine how fast it would be in an actual circuit, but rather we assume a range of hardware latencies and test the performance of the combined system. In this way, we are creating an executable specification for the hardware accelerator and providing constraints for a later implementation step.

The next few sections will provide a fairly deep overview of the setup we are using, if you want to get to results, please jump ahead to Experimental Results.

SETUP OVERVIEW

In our scenario, we have a software program implementing a particular algorithm (a cellular automaton known as “rule30”, more details below), running on a dual-core Freescale MPC8641D Power Architecture-based SoC. We believe 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 affects the overall performance and how hard we need to optimize the hardware.

Note that if we can get along with a slower accelerator (higher latency), the hardware implementation can be smaller in terms of chip real-estate and use a slower clock frequency, reducing power.

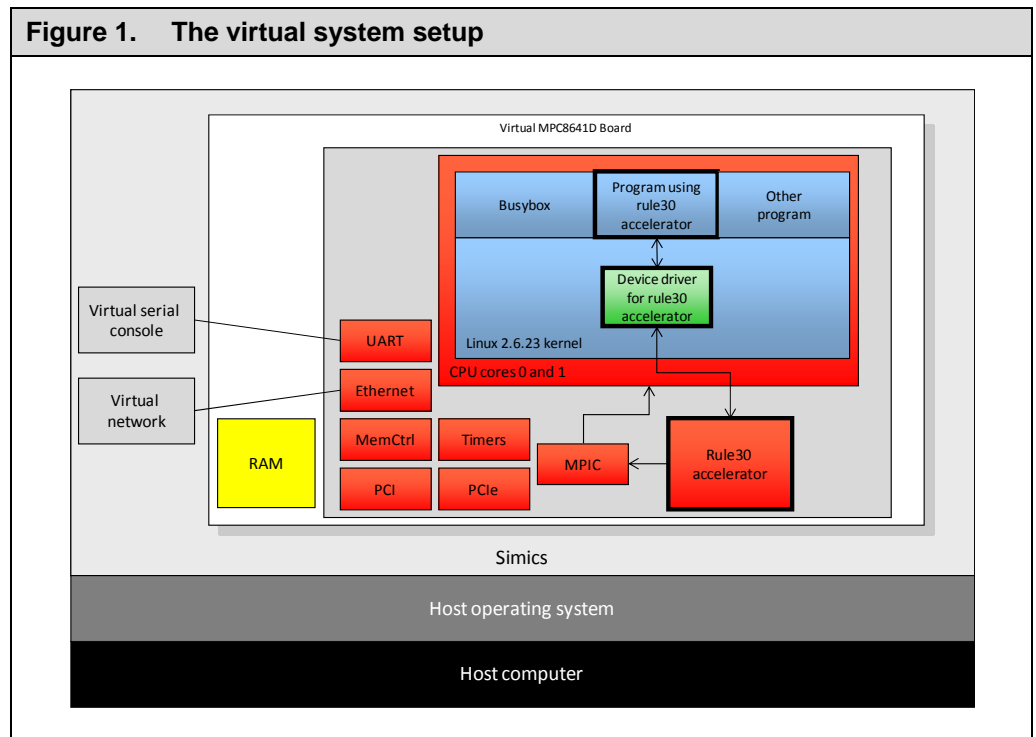
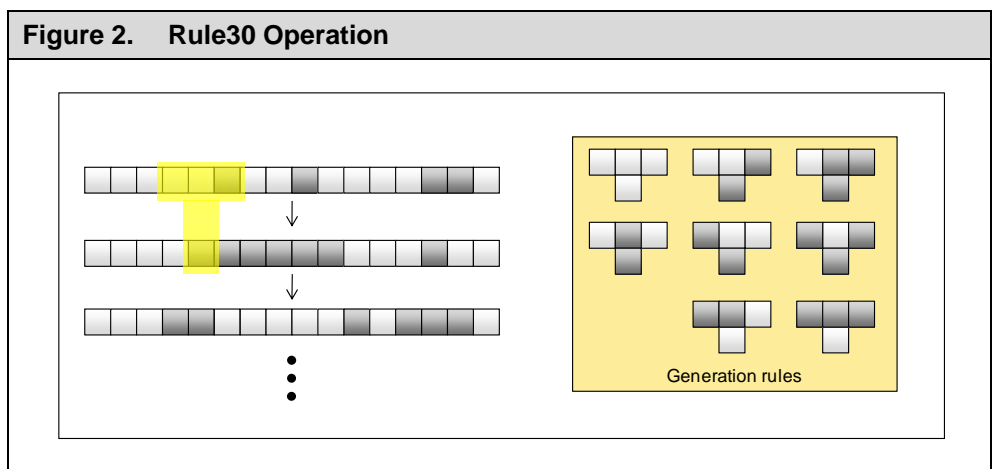


Figure 1 shows the overall system setup. Since we are in a virtual platform, we can add a custom accelerator into the MPC8641D SoC without much problem. We add the device to the virtual platform, and map into the memory map of the processor cores. We also connect it to the MPIC interrupt controller to enable it to interrupt the processor.

In addition to the device, we also write a device driver so that the Linux kernel knows how to access the device, and a test program running in user space to test the performance of the device when used from a user-level program.

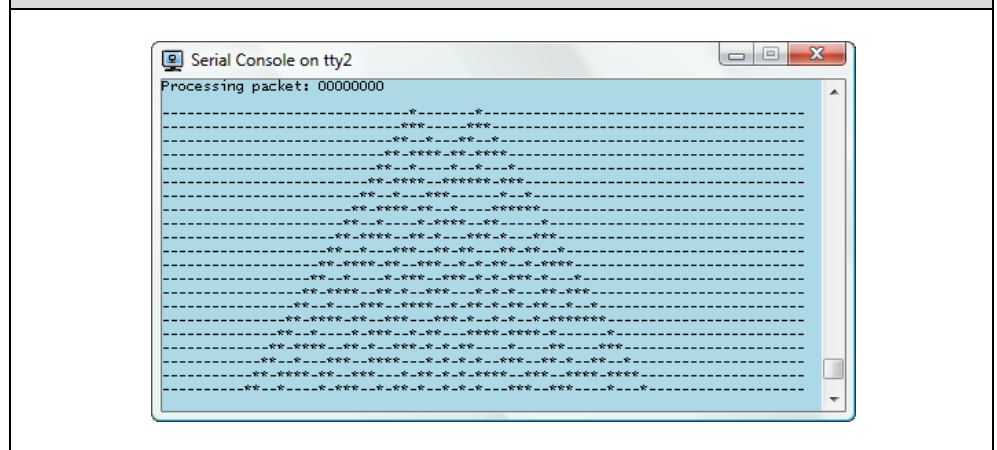
Software Program

The program we want to investigate acceleration for implements a cellular automaton known as **rule30**¹. 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.

¹ See <http://en.wikipedia.org/wiki/Rule30> for more on the algorithm. Also, note that a hardware implementation of rule30 is available as open source from InformAsic.

Figure 3. Rule30 Results


The core algorithm is very easy to describe in software, if you use a single C char to represent each bit. In this case, you can use very straightforward C code, as shown in Figure 4. The rules array corresponds to the rules illustrated in Figure 2.

Figure 4. Rule30 Char-Based Implementation

```

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];
}

```

This reference implementation is not very useful in a real setting, so we have also written an implementation using bit-based representation of a line. This required some more code, but was about as fast as the character-based implementation shown here. It also corresponded to the reasonable hardware

implementation. The rule set for the bit-based algorithm was expressed using eight bits in a word, rather than an array of eight bytes.

The software program runs the rule30 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, we compiled-in a set of test initial lines (called “packets” below) 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 store, and truncate it to the selected line length for this test run.
- For each packet, run the rule30 algorithm for a certain number of iterations (when the successive lines thus generated are plotted, we get the Christmas tree display).

Typically, we test each implementation variant by running it on 1000 packets for 99 iterations per packet (generating a pattern of 100 lines), which means that for each test case, we go through the core loop 99000 times.

Hardware Device

The hardware device is written in Virtutech DML², using DML to describe the programming register layout of the device, the interrupt and I/O handling towards 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.

² For more on DML, see http://www.virtutech.com/whitepapers/virtutech_dml.html.

Figure 5. DML Device Programming Interface

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

1. Set operation parameters into the `control`, `rule_set`, and `line_length` registers.
2. 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.
3. Start the computation by writing to `start_compute`.
4. Wait for an interrupt to signal completion, or spin on the value of the status register to set the operation complete flag.
5. 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 to give us an easy overview of the programming memory map.

The operation parameters that can be set are interrupt notification on, and whether the device should copy the output result to the input array on operation completion, in order to 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 experiments section below.

Figure 6. DML Device Latency Coding

```
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";  
}
```

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, we simply issue the Simics CLI command shown in Figure 7. This can be done at any point in time during a simulation run, and this is leveraged below.

Figure 7. Scripting Latency

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

Implementation Effort

Implementing the hardware accelerator model at this level of abstraction was fairly quick and took 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 testing that the complete software stack worked.

The complete DML 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 where it runs two different implementation variants together, and compare their outputs. We use the char-based implementation as the golden reference model, and validated the software bit-based implementation as well as the hardware implementation (in all its operation modes) against it. This methodology did find some bugs in

our initial bit-based implementation regarding certain corner cases, as well as an embarrassing bug where we forgot to read the output data of the computation in the hardware.

Device Driver

The device driver we 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, we also implemented a `mmap()` function 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 we had compiled it 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.

PERFORMANCE MEASUREMENTS

Once we had the basic infrastructure in place, we started collecting performance data. 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. We also used the Simics Linux process tracker 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
- Computer time per packet, start to end
- Compute time per packet, time spent in user mode

We used Simics magic instructions (NOPs in the code that Simics see but that do not affect the execution semantics) to delineate the start and end of each processing. In this way, we get very precise measurements without involving the software using the serial console to tell us when computations start and end. Note that diagnostic output from the test program is done outside of these kernels, so that they do not affect the execution time.

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 we do not account for memory access latencies and use mandatory temporal decoupling to gain about an order of magnitude in simulation speed. However, it does model time, clock interrupts, and 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. For more on how hardware is modeled in Simics, please see our white paper on [system modeling with Simics](#).

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 checkpointing and scripting.

First, we booted the standard Linux 2.6.23 kernel for the MPC8641D on the virtual MPC8641D machine, and took a checkpoint after the boot had 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 made it possible to run several different simulation runs in parallel on multiple hosts.

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

- Load checkpoint
- Add the rule30 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 the ability of Simics to add objects to a simulation at arbitrary points in time. Figure 8 shows the Simics script operations to create a new accelerator and connect it to the target memory map and MPIC interrupt handler.³

Figure 8. Adding the Rule30 Hardware Accelerator to the Setup

```
@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
```

To load the software in an efficient way, we took advantage of the *simicsfs* virtual file system on the target. Simicsfs allows us to mount (any directory on) the host disk as a file system and access the host files as if they were local files on the target machine. The driver and test program were thus copied from the host to the target every time we do a run (they are cross-compiled on a Linux x86 host), which means that we can update the relevant parts of the target software stack without rebuilding a target disk image and rebooting the target.

We automate the target system operations 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 target serial console as we load the target software application, and after we have loaded the device driver and initialized it. Note that we mount and unmount the simicsfs file system (*/host*) for each file: we do not want to run a simulation with simicsfs mounted, necessarily, since that might impact repeatability and determinism.

³ 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.

Figure 9. Automated Loading of Software on the Target

```

Serial Console on uart0
~ # mount /host
[simicsfs] mounted
~ #
~ # cp /host/linux-kernel/linux-2.6.23/drivers/char/rule30_driver/rule30_driver.
ko .
~ # umount /host
insmod rule30_driver.ko phys_addr=0xf80f0000
~ # insmod rule30_driver.ko phys_addr=0xf80f0000
mknod /dev/rac c 240 100
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
~ # mount /host
[simicsfs] mounted
~ # cp /host/simics-workspace/targets/mpc8641-simple/target-code/rule30_cell_au
omaton/rule30.elf .
~ #

```

All the steps above were wrapped into a single Simics script, which when invoked brings the turn-around time to less than a minute for a recompile and test of the device model, device driver, and/or the test program. All the user needs to do is to make changes, and then start Simics with the script to setup 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

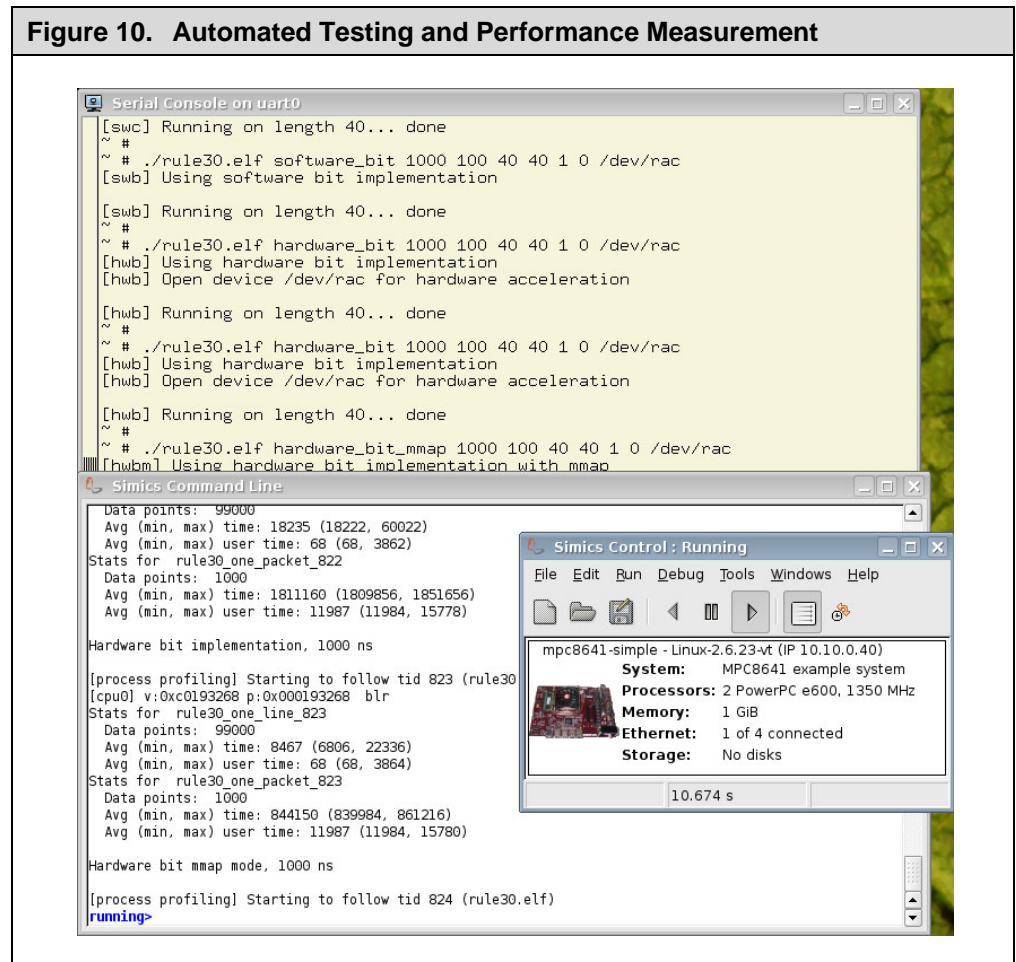
Based on the automated setup of the target machine for experiments, another script was built to perform the actual performance experiments. This script does the following:

- Configure experimental parameters as a set of Simics command-line variables
- Setup the target machine
- Loop over a range of packet sizes, and for each packet size run the test program in a variety of modes with a variety of hardware latency parameters.
- Collect timing data for each packet size and mode, and write 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 command-line parameters. The hardware is controlled by

changing the latency time before each run (note that this means that the latency of the accelerator is not constant through a single simulation run, but that it changes during the run, before each test program execution). We do not restart the target system between each run, as the variation caused by the execution history of the target Linux is insignificant.

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. Thanks to the automation, it can be left to run completely unattended, including running on multiple host machines to cut time by running many experiment variants in parallel.

EXPERIMENTAL RESULTS

Over the course of the work on the software-hardware integration, we performed a series of experiments to check that we were on the right track. Here we present a selection of the most important steps in the process.

Initial Testing and Basic Optimization of Driver

In very early testing of the device driver, we realized 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 the same for read, using a single `read()` call to retrieve all results. Writing input a word at a time was many times slower, since the overhead of a Linux kernel call is fairly significant, as we will show below. This also simplified the driver, since 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%⁴.

Next, we noted some odd variations in the measured end-to-end execution times, and realized that 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 we wanted a clean comparison of implementation alternatives with as many variables under control as possible, we used processor affinity to tie the compute process to core 0 for all tests.

So far, this told us quite a bit about how to do Linux device drivers and that we can indeed evaluate their performance in a fast functional simulator. Thanks to the volume of test cases executed, we also realized that we had to concern ourselves with the scheduler: only about one run in 30 was affected, and it required quite a few program runs to make the issue stand out.

Map out the Landscape

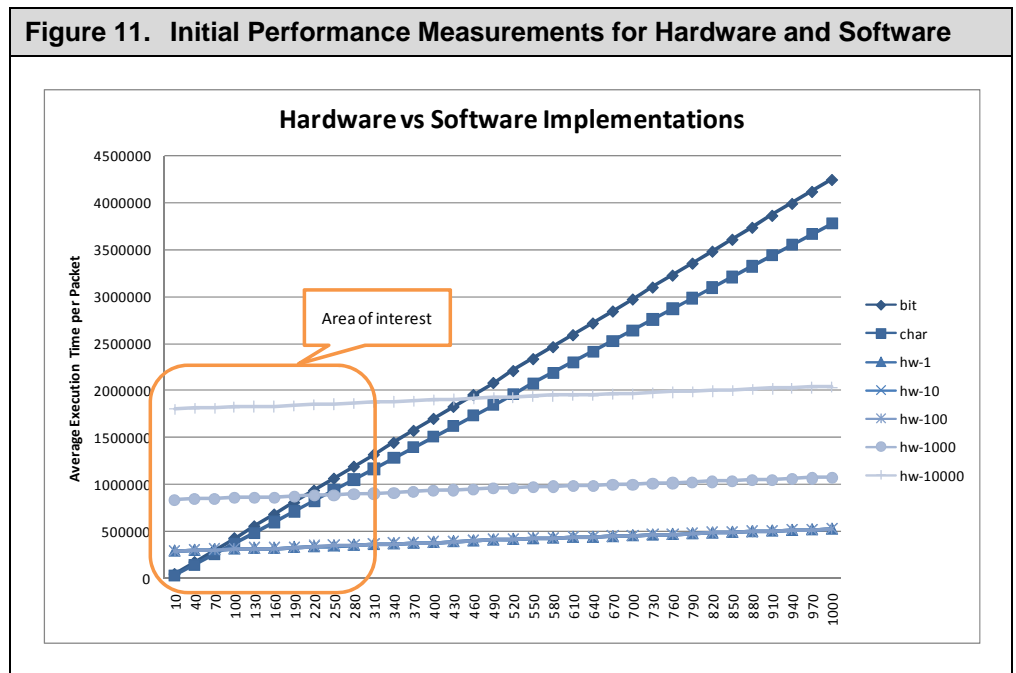
To get an overview of the performance landscape with the hardware accelerator and the basic complete device driver, we ran a set of experiments where we varied the length of the packet from 10 to 1020 bits, in increments of 30 bits. The hardware latency was varied from 1 ns to 10000 ns,

⁴ The initial version of the driver used `kmalloc()` and `kfree()` to create and delete a kernel buffer for the data from user space each time `read()` or `write()` was called.

logarithmically. In this way, we hoped to determine the order of magnitude of speed where the hardware accelerator would become relevant.

Figure 11 shows the results obtained:

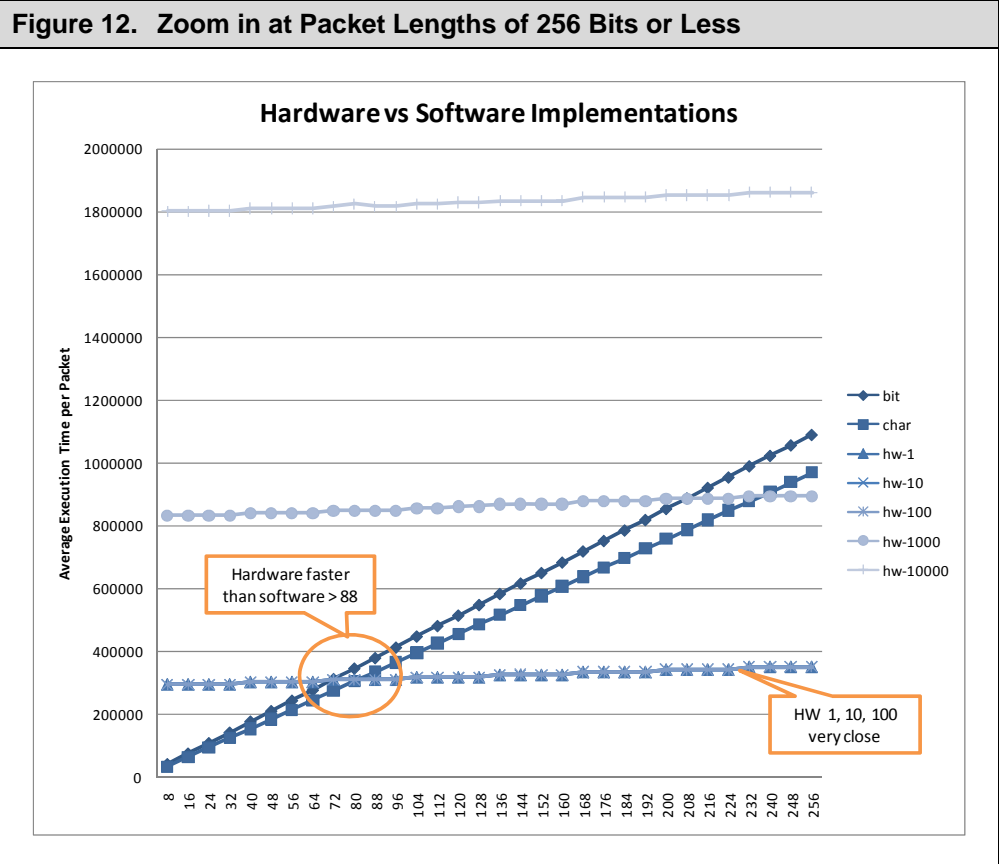
- “bit” is the bit-based software implementation
- “char” is the reference implementation
- “hw- x ” is the hardware accelerator with a latency of x ns.
- The times are the average computation times for each packet, over the 1000 packets processed for each packet length and operation mode, expressed as clock cycles at 1350 MHz.



As we can see, the area of real interest to compare hardware and software is below 256 bits of length. We can also probably ignore the hardware option with a latency of 10000 ns, as that does not win over software until a packet length of around 500 bits.

Thus, we redid our experiments with lengths of 8 to 256, with a step of 8 bits. The results are presented in Figure 12. From this more detailed investigation, we draw the following conclusion:

- 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 also not an attractive option, but if packet lengths will be long in certain use case, it could allow us to build 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.

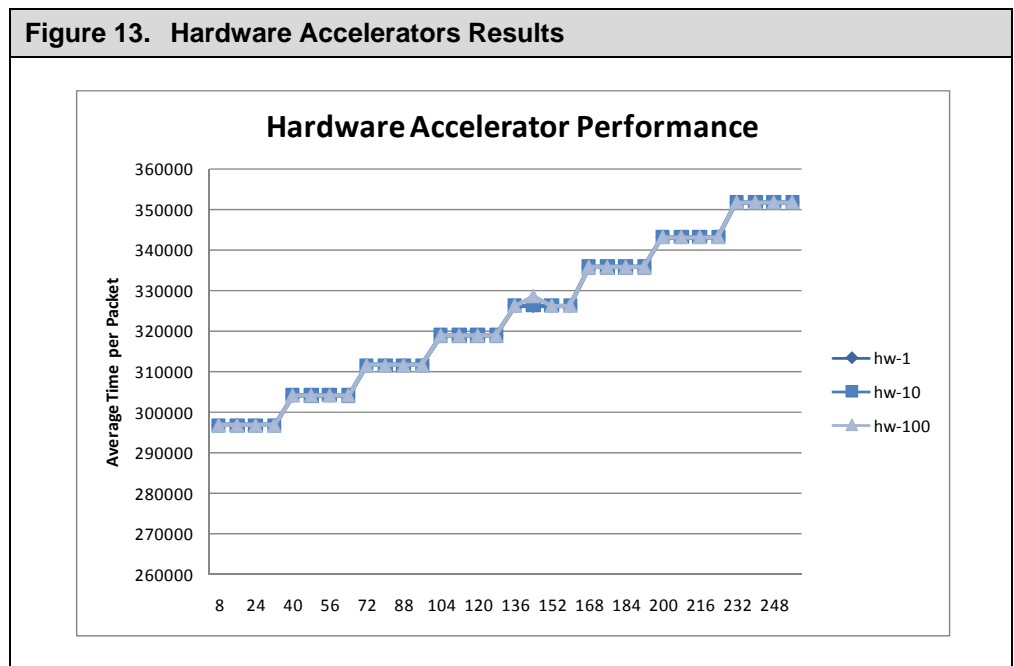


Hardware Accelerator Speed Irrelevant?

The data thus far shows that the execution time when using the hardware accelerator at 1, 10, and 100 ns was almost identical. To check this, we graphed the data shown 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 (as seen from the fact that the lines for 1ns, 10ns, and 100ns overlap completely) and only marginally dependent on the size of packets (note the scale in the graph) clearly shows that we have a large fixed overhead attached to doing a single computation in the hardware.

Figure 13. Hardware Accelerators Results



If we were happy with the performance we are seeing from the hardware-accelerated option at this point, we could now leave the specification stage and ask the hardware team to create an accelerator with a target latency of 100 ns, or at most 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 that make sense and which do not. Without committing to any kind of detailed hardware implementation, we have determined the required performance when using a realistic software stack. Had we performed these experiments using a bare-metal setup, we had surely drawn quite different conclusions since a bare-

metal setup removes the operating system and driver model overhead from the equation.

MMAP Driver

The results can also be taken to indicate that the software stack needs to be improved to make the overall system maximally efficient. For example, if we assume that the latency of the accelerator was 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.

Figure 14. Using mmap() in the Test Program

```
// 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)));
}
```

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. We explored the `mmap` option. The core of the code in the test program is shown in Figure 14. It is very similar to bare-metal code.

Final Evaluation

With this optimization in place, we performed our final evaluation run. This comprised a total of 352 complete end to end runs, each run processing 1000 packets for 99 generation (generating a picture 100 lines long). Thus, at each data point, we execute 99000 iterations of the core loop. We tried 32 different lengths between 8 and 256, and in each length we tried 11 different variants. 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 of this on the target was 134 seconds, and we executed some 200 billion target instructions in this time frame (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 we did using Python scripting, running without that scripting approximately doubled the speed. Still, the simulation ran usefully fast to provide us with lots of good data over a long lunch.

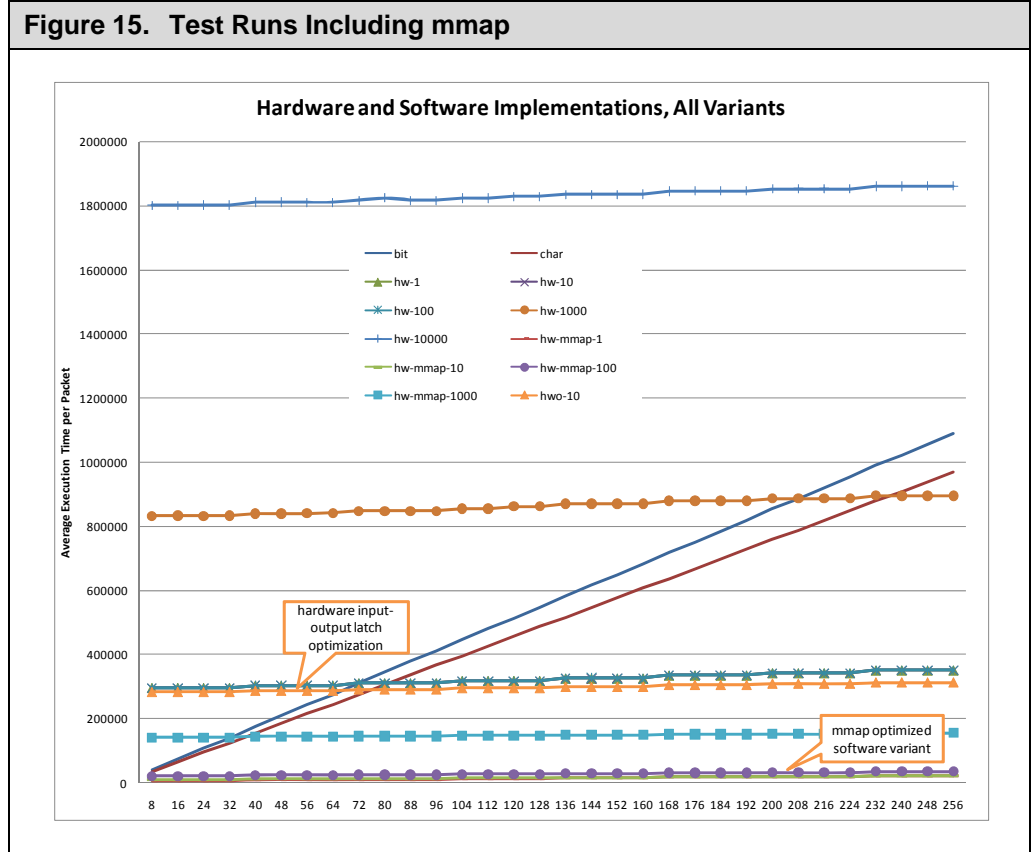
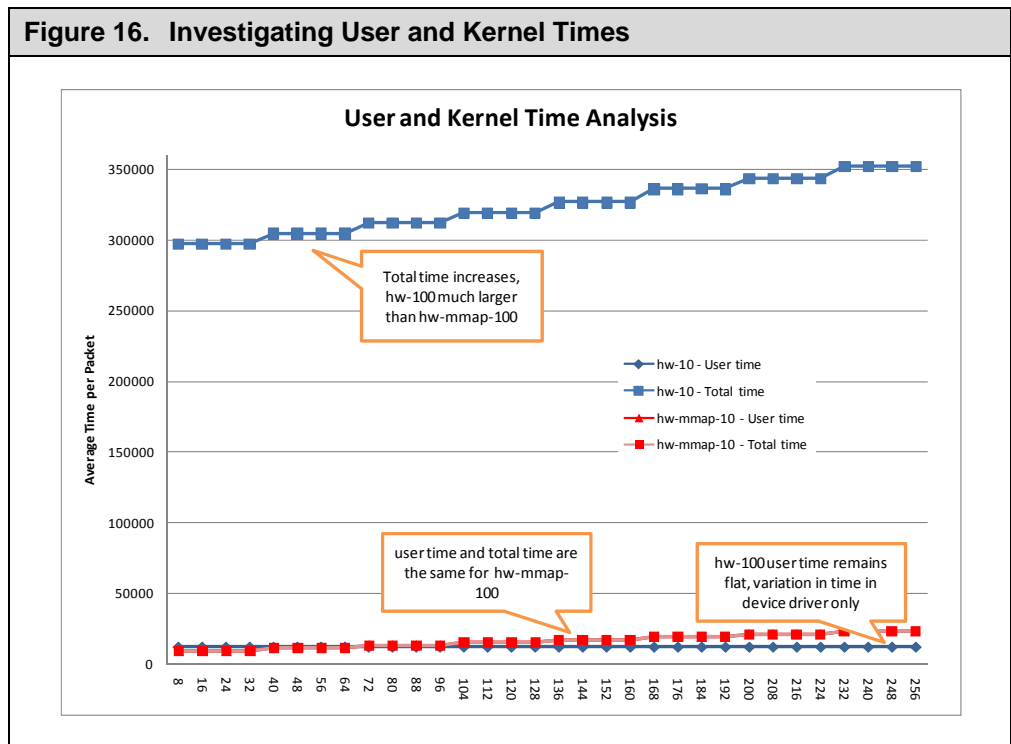


Figure 15 shows the results for all modes and a range of hardware latencies. The conclusions we can draw from this is that:

- As seen from the data points labeled “hw-mmap-x”,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. We can still ask the hardware team to produce a 100 ns-latency hardware block without risking losing any significant performance.
- The latched optimization (*hwo-10*) does perform slightly better than the default driver, but not significantly more. This optimization is thus not meaningful to require from an actual hardware implementation. Using the fast functional analysis, we can simplify the requirements on the hardware to remove useless features that did 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 but with a low investment in development cost, and spend more effort on getting the software optimized.

That the reason for the good performance of mmap-optimized software is less time spent in the kernel is easy to see from Figure 16.



Here, we plot 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) for two selected execution variants: hw-100 and hw-mmap-100. The results are striking:

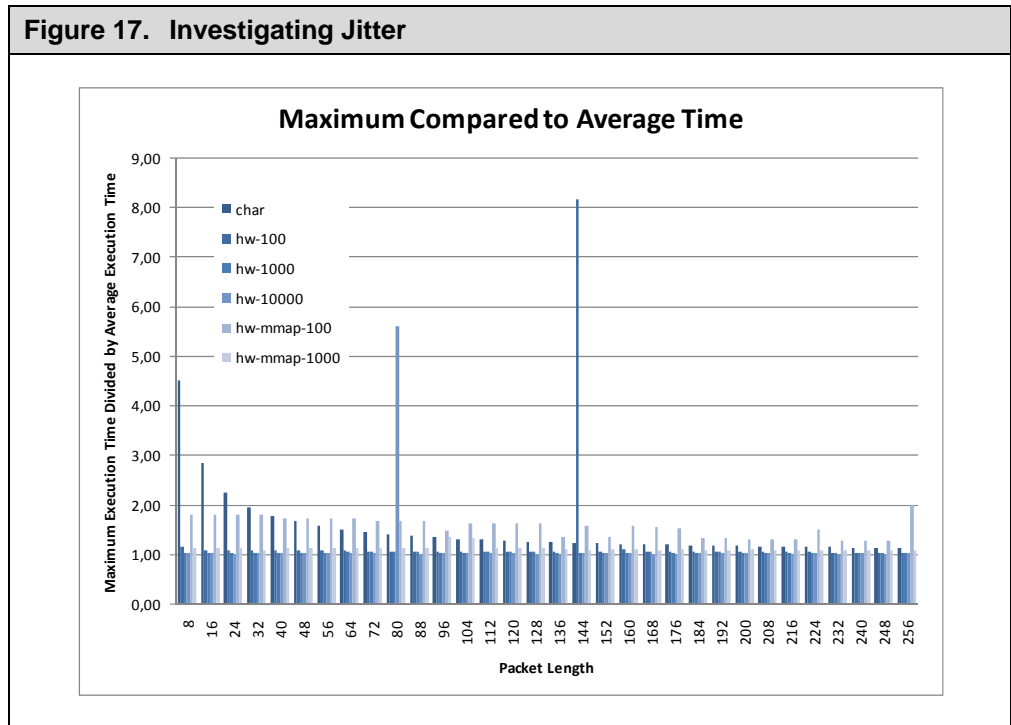
- The regular device driver spends very little time in user mode, but much more time overall. The time in user mode is also flat, as all

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 we go to longer packets.

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 17. 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 also 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 actually give up the CPU, waiting for a completion interrupt, giving the OS a chance to swap the process out. It also indicates that if we want 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, since it never gives up the CPU in the same way).

SANITY CHECKS

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

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

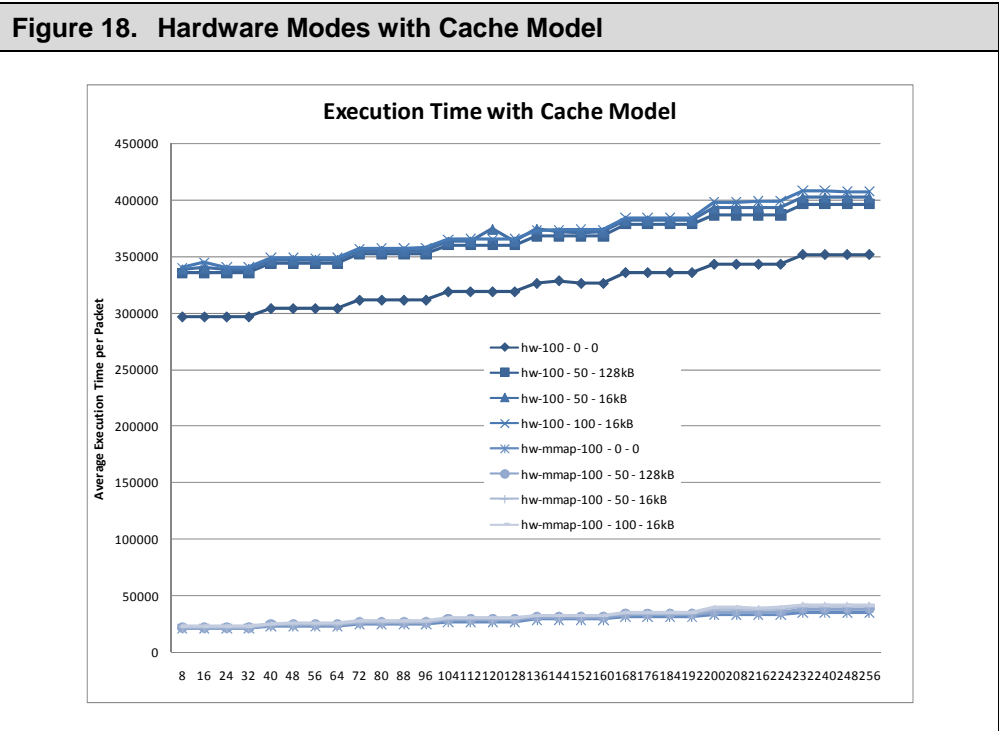
- 16 kB cache, latency 100 cycles to main memory
- 16 kB cache, latency 50 cycles to main memory
- 128 kB cache, latency 100 cycles to main memory

Hardware vs Hardware

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

As seen in Figure 18, adding a cache model does give us 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, even if it is hard to see in the graph due to the scale. Thus, caches have no impact on our previous results, and we can safely

use a fast functional simulator for this type of executable specification and early performance requirement work.



Software and Hardware

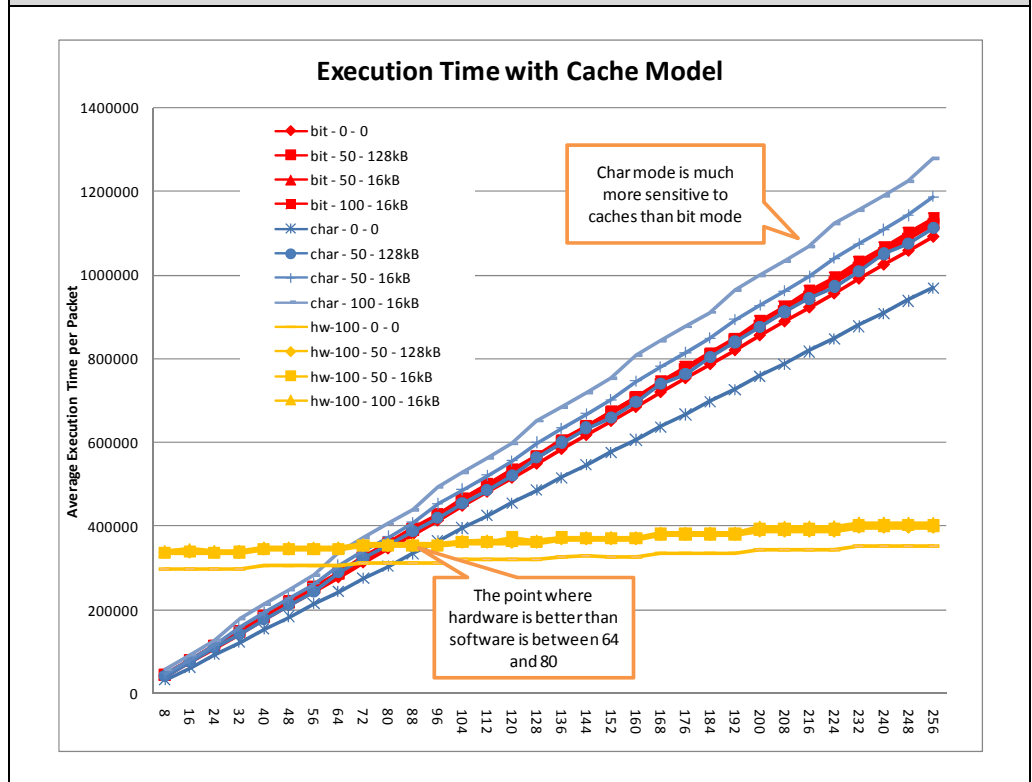
In Figure 19 we plot the performance of the two software modes with caches, as well as one hardware mode for reference. It shows some interesting results:

- Without a cache, the char-based implementation is faster since it executes fewer instructions per iteration. However, as can be seen in Figure 4, most of these are memory accesses to various arrays.
- With a cache, the bit-based implementation is faster since it performs much fewer memory accesses per iteration. We did not show the implementation in this white paper, but it essentially only accesses memory twice for each 32-bit segment of a line processed.

This indicates that if we were to skip the hardware accelerator altogether and go for a software-based implementation, it would make sense to use the bit-

based implementation if we expect the memory system of our target machine to be slow or caches small.

Figure 19. Software Modes with Cache Model



Furthermore, the point where hardware is faster than software moves to short packet lengths as memory gets slower. Compared to the char-based mode, the cross-over 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 our high-level view of our 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 that it does not have a significant impact on our high-level results. The differences between different software stack variants and hardware accelerator latencies are much larger than the effect caused by

caches. From a system-design perspective, we learn most of what we need to learn from a pure fast functional simulation.

CONCLUSIONS

In this white paper, we have shown how a Simics-based fast functional simulation can be used to perform system design and specification with respect to hardware offloading of critical software functions. Working at the ST level of abstraction, we get a complete view of the performance landscape, since we can execute large-scale workloads involving hundreds of billions of target instructions in a reasonable time frame. Overall, the measurements presented in this white paper required more than a trillion target instructions to be executed.

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

The result of the work is both significant insights into the important performance parameters for our design, and 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.

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

CONTACT INFORMATION

North and South America sales_americas@virtutech.com	Europe, Middle-East, Africa sales_emea@virtutech.com
Asia-Pacific sales_apac@virtutech.com	Japan sales_japan@virtutech.com
http://www.virtutech.com	

© Copyright 2009 Virtutech, Inc. All Rights Reserved.

TRADEMARKS. Virtutech, Simics, Hindsight, and the logos thereof, are trademarks or registered trademarks of Virtutech, Inc. and/or its subsidiaries, in the United States and/or other countries.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. VIRTUTECH MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Virtutech, Inc., 2001 Gateway Place, Suite 201E, San Jose, CA 95110, USA