Report Number: WUCSE-2014-52

2014

Performance Modeling of Virtualized Custom Logic Computations

Authors: Michael J. Hall and Roger D. Chamberlain

Virtualization of custom logic computations (i.e., by sharing a fixed function across distinct data streams) provides a means of reusing hardware resources, particularly when resources are limited. This is common practice in traditional processors where more than one user can share processor resources. In this paper, we virtualize a custom logic block using C-slow techniques to support fine-grain context-switching. We then develop and present an analytic model for several performance measures (throughput, latency, input queue occupancy) for both fine-grained and coarse-grained context switching (to a secondary memory). Next, we calibrate the analytic performance model with empirical measurements. We then validate the model via discrete-event simulation and use the model to predict the performance and develop optimal schedules for virtualized logic computations. We present results for a Taylor series expansion of a cosine function with added feedback and an AES encryption cipher.
... **Read complete abstract on page 2.**

Performance Modeling of Virtualized Custom Logic Computations

Complete Abstract:

Virtualization of custom logic computations (i.e., by sharing a fixed function across distinct data streams) provides a means of reusing hardware resources, particularly when resources are limited. This is common practice in traditional processors where more than one user can share processor resources. In this paper, we virtualize a custom logic block using C-slow techniques to support fine-grain context-switching. We then develop and present an analytic model for several performance measures (throughput, latency, input queue occupancy) for both fine-grained and coarse-grained context switching (to a secondary memory). Next, we calibrate the analytic performance model with empirical measurements. We then validate the model via discrete-event simulation and use the model to predict the performance and develop optimal schedules for virtualized logic computations. We present results for a Taylor series expansion of a cosine function with added feedback and an AES encryption cipher.

Performance modeling of virtualized custom logic computations

Michael J. Hall
Roger D. Chamberlain

Michael J. Hall and Roger D. Chamberlain. “Performance Modeling of Virtualized Custom Logic Computations,” Tech. Rpt. WUCSE-2014-52, Department of Computer Science and Engineering, Washington University in St. Louis, July 2014.

Dept. of Computer Science and Engineering
Washington University in St. Louis

Performance Modeling of Virtualized Custom Logic Computations

Michael J. Hall and Roger D. Chamberlain
Department of Computer Science & Engineering
Washington University in St. Louis
Email: {mhall24, roger}@wustl.edu

***Abstract*—Virtualization of custom logic computations (i.e., by sharing a fixed function across distinct data streams) provides a means of reusing hardware resources, particularly when resources are limited. This is common practice in traditional processors where more than one user can share processor resources. In this paper, we virtualize a custom logic block using C-slow techniques to support fine-grain context-switching. We then develop and present an analytic model for several performance measures (throughput, latency, input queue occupancy) for both fine-grained and coarse-grained context switching (to a secondary memory). Next, we calibrate the analytic performance model with empirical measurements. We then validate the model via discrete-event simulation and use the model to predict the performance and develop optimal schedules for virtualized logic computations. We present results for a Taylor series expansion of a cosine function with added feedback and an AES encryption cipher.**

I. Introduction

Virtualization of computational resources, primarily processor cores, has a long history. From early multitasking operating systems [1] that support context switching between otherwise unrelated processes, hypervisors [2], [3] that support context switching between operating system images, to simultaneous multithreading microarchitectures [4] that switch between threads effectively at each clock cycle, traditional computing paradigms (via fetch-execute engines) regularly share compute resources in such a way as to effectively provide a virtual copy of the compute resource to each user. Virtualization provides a way by which hardware resources can be reused, which is commonly known as resource sharing. Sharing is a common technique for utilizing available hardware resources for computing such as DSP blocks, memory controllers, memory bandwidth, and IP cores.

In this paper, we model the performance of a virtualized fixed logic computation and tune a schedule for optimal performance. Our interest is in supporting a set of distinct data streams that all wish to perform the same computation. Virtualizing the logic computation involves sharing hardware resources and context-switching each distinct data stream into the hardware. This context switch can be either fine-grained (in which context is changed each clock cycle) or coarse-grained (in which the state of the computation is swapped out to a secondary memory). Good candidates for virtualization are logic computations with long combinational paths (i.e., substantial computational requirements).

This research was supported by National Science Foundation (NSF) through grant CNS-0931693 and Exegy, Inc.

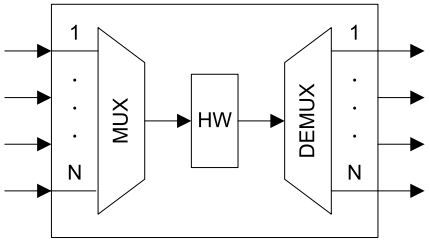

Fig. 1. Hardware virtualization for N distinct data streams that perform the same computation. The N streams are multiplexed into a shared hardware (HW) block, processed, and then demultiplexed back into N streams.

To virtualize a function, consider the hardware block (HW) shown in Figure 1 that has an Nx1 input multiplexer and 1xN output demultiplexer. This HW block is a function implemented in custom logic. N distinct data streams (each with a dedicated input and output port) share the single instance of the HW block. These data streams are then multiplexed into the custom logic block, processed, and then demultiplexed back into independent streams. For our purposes, we are not considering I/O bound computations, but rather assume there is sufficient bandwidth at the input and output ports of Figure 1. When the logic function is purely combinational (i.e., feed-forward), any input from any data stream can be presented to the HW block at any clock cycle, even if it is deeply pipelined. In this case, there are no constraints on scheduling. When the logic function is sequential (i.e., has feedback) and has been deeply pipelined, this imposes scheduling constraints. Once a data element from a particular stream has been delivered to the HW block, the stream has to wait a number of clock ticks equal to the pipeline depth before it can provide a subsequent data element from that same stream.

Pipelined logic circuits with feedback can be context switched to compute multiple data streams concurrently. Essentially, the circuit can be thought of as a sequential logic circuit with pipelined combinational logic. The pipelined combinational logic adds latency and decreases single stream throughput since it takes multiple clock cycles (corresponding to the number of pipeline stages) to compute a single result and feed it back to the input. If the number of pipeline stages

is C, then this circuit is said to be C-slowed since a single computation takes C times more clock cycles (often mitigated by a higher clock *rate*). C-slow is a technique described by Leiserson and Saxe [5] by which each register is replaced by C registers and then retimed to balance the registers throughout the combinational logic. Exploiting this characteristic allows processing multiple different contexts or data streams in a fine-grain way using the same hardware logic. The number of fine-grain contexts supported equals the pipeline depth.

When the number of contexts to be supported, N, is greater than the pipeline depth, C, coarse-grained context switching can be used, swapping out whatever state is stored in the circuit to a secondary memory. In general, this will incur some cost, representing the overhead of a context switch. While the fine-grained context switching of the C-slowed circuit naturally uses a round-robin schedule, there are a richer set of scheduling choices available when building a coarse-grain context-switched design. In this work, we constrain our consideration to round-robin schedules and explore the performance impact of the schedule period.

The contributions of this paper include the following: (1) development of an analytic performance model for hierarchically scheduled, virtualized, hardware logic; (2) calibration of the model using empirical measurements from C-slowed implementations of (a) a cosine function implemented via a Taylor series expansion with added feedback and (b) an AES encryption cipher operating in the CBC mode (also with feedback); (3) validation of the performance model using a discrete-event simulation; and (4) use of the model to both predict the performance of virtualized hardware logic and choose an optimal schedule period.

II. Related Work

Expanding our consideration of hardware virtualization beyond the fine-grained context switching that can be supported by C-slow techniques, Plessl and Platzner's survey paper [6] describes three different approaches to hardware virtualization on FPGAs: temporal partitioning of net lists, virtualized execution, and virtual machines. Chuang [7] describes another type of temporal partitioning whereby custom logic can be reused by temporally shared state.

Temporal partitioning of net lists [6] is a technique for virtualizing hardware described by a net list that would otherwise be too large to physically fit onto an FPGA. It is accomplished by partitioning the net list and swapping the partitions in and out like virtual memory blocks. This technique requires (at least partial) reconfiguration of the FPGA.

Temporal partitioning of state [7] is a way to share hardware by temporally swapping its state so as to compute multiple streams of computation on the same hardware (i.e., a single net list). The logic is fixed, and the state is swapped (context switched), allowing it to operate on independent streams. The context switch can be either fine- or coarse-grain.

Virtualized execution [6] is where an abstract programming model is defined and applications are developed targeting that model. Any application developed for the programming model can run on any hardware that supports the model for execution. One example programming model is the instruction set of a processor core. Code written for this instruction set can execute on any processor core that supports the instruction set. An alternative example is PipeRench [8], which has a pipelined streaming programming model.

A virtual machine [6] defines a generic abstract FPGA architecture that hardware can be deployed upon. Designs targeted to a generic FPGA architecture are then remapped to the actual architecture of a specific FPGA device for execution. This technique is different from the others in that it does not perform any kind of context switching.

Kudlur [9] describes a high-level synthesis methodology called Streamroller that employs resource sharing for designing a pipeline of accelerators for an application. Multifunction loop accelerators allow the hardware to be time-multiplexed through multiple pipeline stages. This sacrifices performance, but increases the ability to share hardware. Canis [10] uses resource sharing and multi-pumping where a hardware resource, such as a DSP block, is clocked at 2x rate and is shared by two concurrent data streams.

Within the domain of fine-grained context switching, several applications have been implemented using the C-slow technique. Weaver et al. [11] applied C-slow to three applications: AES encryption, Smith/Waterman sequence matching, and the LEON synthesized microprocessor core. They designed an automatic C-slow retiming tool that would replace every register in a synthesized design with C registers and retime the circuit. AES encryption achieved a speedup of 2.4 for a 5-slow by hand implementation. Smith/Waterman achieved a speedup of 2.2 for a 4-slow by hand implementation. And the LEON SPARC microprocessor core achieved a speedup of 2.0 for a 2-slow automatically C-slowed implementation. Su et al. [12] applied C-slow to an LDPC decoder to achieve a throughput-area efficient design.

III. Model Development

Figure 2(a) shows the general hardware configuration that we consider. An arbitrary sequential circuit with input x, state y, and output z has been C-slowed and augmented with a secondary memory that can load and unload copies of state y to/from the "active" state register. N FIFO buffers are present at the inputs to store data stream elements that are awaiting being scheduled. The circuit consumes one data element (from an individual input specified by the schedule) each clock tick.

Inputs to the performance model are described first. The number of input data streams is denoted by N. Each data stream, i, is assumed to provide elements with a known distribution and given mean arrival rates λ_i $^{\text{elements}}/_{\text{s}}$. In what follows, we will assume the input distribution is Poisson. The custom logic is characterized as follows. The total combinational propagation delay is given as t_{CL}. The pipelining depth is C (corresponding to a C-slowed design). We model the combinational propagation delay between the pipeline registers as a random variable X with mean $\mu_X = t_{CL}/C$ and standard deviation σ_X. We assume that both the secondary memory

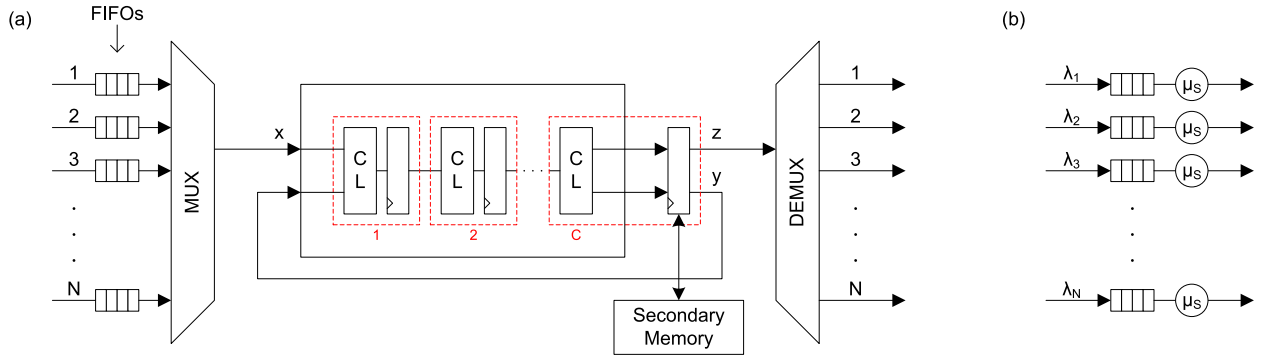

Fig. 2. (a) General hardware configuration of a C-slowed sequential logic circuit with secondary memory supporting N data input/output streams. (b) Queueing model of this circuit with N queueing stations (one for each data stream). Each queueing station consists of a FIFO queue and associated server representing one virtual copy of the hardware computation.

and the input buffers operate at the clock rate of the pipeline. State transfers to/from secondary memory take a given S clock cycles (enabling the model to support a range of context switch overheads), and the buffers are assumed to have single-cycle enqueue and dequeue capability (i.e, they do not limit the performance of the system).

With the above inputs available, we represent the performance of the context-switched hardware via an open queueing network model with effective service rates determined by the clock frequency achievable by the C-slowed circuit. This queueing model is illustrated in Figure 2(b). Each individual queueing station represents one virtual copy of the hardware computation. The arrival rate at each queue is λ_i $^{\text{element}}/_{\text{s}}$ and the model for the effective service rate is developed in the paragraphs below.

To model the effective service rate, we first consider the case where $N = C$ (i.e., only fine-grained context switching is being employed, and there is no secondary memory). The clock frequency, f_{CLK}, of the pipelined circuit will be limited by the maximum propagation delay between the pipeline registers. Order statistics establishes a bound on the clock period as follows:

$$t_{CLK} = E\left[\max\left(X\right)\right] \leq \mu_X + \sigma_X\sqrt{C-1}. \tag{1}$$

This represents a bound on the clock period which is equal to the expectation of the maximum of C samples drawn from random variable X. This bound does not depend upon the distribution of X, only its mean, standard deviation, and the number of samples drawn from the distribution. We know $\mu_X = t_{CL}/C$, but we do not know σ_X. Rather, we expect the clock period to trend with the $\sqrt{C-1}$ term and use the bound as a model for the clock period in the modified equation

$$t_{CLK} = t_{CL}/C + k \cdot \sqrt{C-1} \tag{2}$$

where σ_X is replaced by coefficient k which is fit to empirical data. As the notion of representing pipelined propagation delays via a random variable is a novel modeling approach, we assess the effectiveness of this model by comparing its predictions with empirical measurements below in Section IV.

Returning to Figure 2, if we employ a fixed, round-robin schedule (with no context switches to secondary memory), data elements will be dequeued from each input at a fixed rate of $\mu_s = f_{CLK}/C$. This corresponds to a deterministic service process for each server, with mean service rate μ_s $^{\text{elements}}/_{\text{s}}$. This implies we can treat each context as an independent $M/D/1$ queueing station (*M*arkovian, or memoryless, arrival process; *D*eterministic service process; 1 server) [13]. For this $M/D/1$ system, the maximum achievable throughput (per stream) is

$$\mu_s = \frac{1}{C \cdot t_{CLK}} \tag{3}$$

and the total achievable throughput is

$$T_{TOT} = C \cdot \mu_s. \tag{4}$$

The average (mean) waiting time of each data element in the queue is [13]

$$W_q = \frac{1}{\mu_s} \cdot \frac{\rho}{2\left(1-\rho\right)}, \tag{5}$$

where $\rho = \lambda/\mu_s$. The (deterministic) time in the pipelined circuit is

$$W_s = \frac{1}{\mu_s} = C \cdot t_{CLK}, \tag{6}$$

and therefore the average latency (elapsed time from arrival to completion of processing) for each data element is

$$W = W_q + W_s = \frac{1}{\mu_s} \cdot \frac{\rho}{2\left(1-\rho\right)} + C \cdot t_{CLK}. \tag{7}$$

The average (mean) occupancy of each queue is [13]

$$N_q = \lambda \cdot W_q = \frac{\rho^2}{2\left(1-\rho\right)}. \tag{8}$$

The above analysis holds for the circumstance where $N = C$ and we are only using fine-grained context switching. The second case to consider is when $N > C$ and we are exploiting both fine-grained and coarse-grained context switching. We will make the simplifying assumption that N is an integer multiple of C. The schedule is a fixed, hierarchical, round-robin schedule with period R_S (for each stream) that acts as follows: (1) a set of C input streams is chosen to share the

hardware resource and within that set a round-robin schedule is used; (2) after R_S rounds, the current set of input streams' state is swapped out to secondary memory and the next set of C input streams' state is swapped in (the time required to complete this operation is given as S clock cycles), this set is then scheduled to use the hardware in round-robin fashion; (3) the entire collection of N/C input stream sets is also chosen in round-robin fashion (hence the label hierarchical round-robin schedule), such that once every individual input stream has had R_S input elements processed the high-level schedule returns to the first set of C input streams.

The impact of $N > C$ on the queueing model starts with the effective service rate expression. The number of clock cycles to complete a full round of the hierarchical schedule (during which each server services R_S elements) is $R_S \cdot N + S \cdot N/C$. This implies the effective service rate is

$$\mu_s = \frac{R_S}{(R_S N + SN/C) \cdot t_{CLK}} \text{ elements/s}, \tag{9}$$

which simplifies to (3) when $S = 0$ (i.e., context switches are free) and $N = C$. The total achievable throughput, T_{TOT}, is now $N \cdot \mu_s$, or

$$\begin{aligned} T_{TOT} &= \frac{N \cdot R_S}{(R_S N + SN/C) \cdot t_{CLK}} \\ &= \frac{R_S}{(R_S + S/C) \cdot t_{CLK}} \text{ elements/s}. \end{aligned} \tag{10}$$

To develop an expression for the average (mean) time that each data element waits in one of the input buffers, we start by using (5), the $M/D/1$ expression for mean queue waiting time, and add a term, W_h, to reflect the additional time waiting in the queue due to the hierarchical round-robin schedule. The additional waiting time is experienced by the fraction of data elements that arrive when their input stream is swapped out (i.e., not receiving service). This fraction is $\frac{R_S(N-C)+SN/C}{R_S N+SN/C}$, or the number of clocks a stream is swapped out divided by the number of clocks in a full schedule round. The average additional time experienced by this fraction of input elements is one half of the time the stream is swapped out, $(R_S(N-C) + SN/C) \cdot t_{CLK}/2$. Multiplying the additional time by the fraction that experience the additional time yields

$$\begin{aligned} W_h &= \frac{(R_S(N-C) + SN/C) \cdot t_{CLK}}{2} \cdot \frac{R_S(N-C) + SN/C}{R_S N + SN/C} \\ &= \frac{(R_S(N-C) + SN/C)^2 \cdot t_{CLK}}{2(R_S N + SN/C)}. \end{aligned} \tag{11}$$

The time in the pipelined circuit does not change from $W_s = C \cdot t_{CLK}$, and therefore the average latency from arrival to completion of processing is

$$\begin{aligned} W &= W_q + W_h + W_s \\ &= \frac{1}{\mu_s} \cdot \frac{\rho}{2(1-\rho)} + \frac{(R_S(N-C) + SN/C)^2 \cdot t_{CLK}}{2(R_S N + SN/C)} \\ &\quad + C \cdot t_{CLK}. \end{aligned} \tag{12}$$

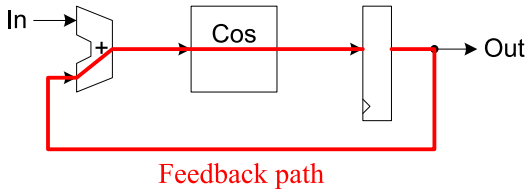

Fig. 3. Block diagram of Cosine function application with added feedback. The cosine function is approximated with a Taylor series expansion of N_t terms which form a long combinational path. With the added feedback path, this will determine the achievable clock rate.

Note that the expression for time in the server, W_s, no longer equals $1/\mu_s$, because the waiting time due to coarse-grained context switching is accounted for in the expression for W_h rather than W_s. The expression for the average (mean) queue occupancy, N_q, does not change from (8); however, the expression for μ_s, the effective service rate, has changed to (9). The mean number of elements waiting in the buffer due to the hierarchical round-robin scheduling is $N_h = \lambda W_h$.

The above discussion provides analytic performance expressions for total achievable throughput, T_{TOT}, and the average latency experienced by each data element, W, both in the buffer, $W_q + W_h$, and in computation, W_s. Also available is the average occupancy of the buffer, $N_q + N_h$.

IV. Calibration

We use two applications across two implementation technologies to validate and calibrate (2), the model for t_{CLK}. Once t_{CLK} is calibrated, it can be used as part of the model predictions presented in the results section below. The two applications constructed are (1) a cosine function implemented via a Taylor series expansion with added feedback, and (2) an AES encryption cipher. These applications were chosen because they both have long combinational paths with a feedback path from the output to the input, making simple pipelining insufficient to effectively utilize their logic blocks, and therefore requiring that they be scheduled with data from independent data streams.

Measurements are taken for both applications on FPGA and ASIC technologies. The logic is C-slowed to support C data streams of computation with $N = C$ data streams. For FPGA technology, we target a Xilinx Virtex-4 XC4VLX100 FPGA and use the Xilinx ISE 13.4 tools for synthesis, place, & route of the hardware designs. For ASIC technology, we target a 5M1P $0.18\,\mu$m process using the Virginia Tech VLSI for Telecommunications (VTVT) standard cell library. Cadence RTL Compiler and Encounter are likewise used to synthesize, place, & route the hardware designs. The clock period is unconstrained in the runs for both technologies.

The Cosine application is illustrated in Figure 3. It consists of a cosine function in the middle (representing the HW block), an output register, a feedback path, and an adder to mix the input with the output feedback. This is a synthetic application built to have a long combinational path through the cosine function via a configurable number of Taylor series terms, N_t, that approximate the cosine.

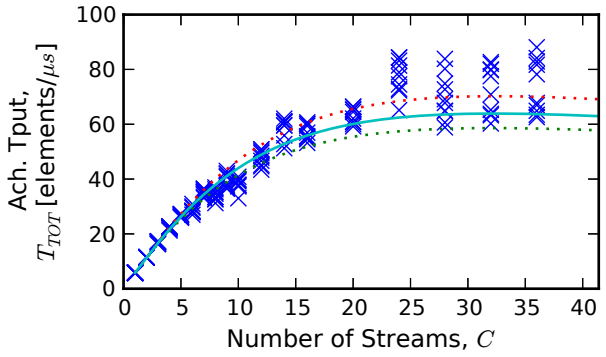

Fig. 4. Total achievable throughput plot for virtualized logic of the Cosine Taylor series expansion function with feedback on an FPGA with $N_t = 20$ terms. The number of streams refers to the logic pipeline depth, C, with $N = C$. For each value of C, data points were taken from 10 tool flow runs.

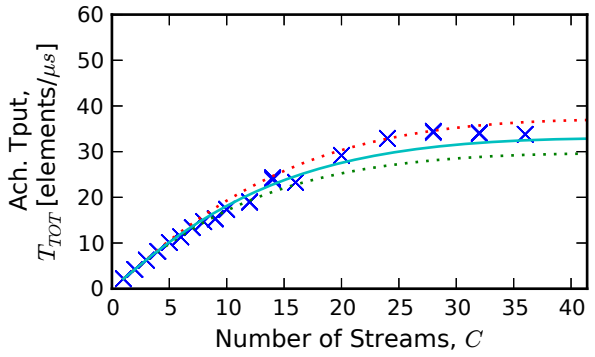

Fig. 5. Total achievable throughput plot for virtualized logic of the Cosine Taylor series expansion function with feedback on an ASIC with $N_t = 20$ terms. The number of streams refers to the logic pipeline depth, C, with $N = C$. For each value of C, data points were taken from 10 tool flow runs.

In the experiment, we measure t_{CLK} on 10 independent runs first for an FPGA with the number of terms, N_t, ranging from 2 to 24 terms, and the number of streams, C, ranging from 1 to 44 streams. Curve fitting the model in (2) across data sets concurrently yields:

$$t_{CLK} = 8.6\,\tfrac{\text{ns}}{\text{term}} \cdot \frac{N_t}{C} + \left(1.9\,\text{ns} - 4.7\,\tfrac{\text{ps}}{\text{term}} \cdot N_t\right) \cdot \sqrt{C-1}. \tag{13}$$

Comparing this expression to (2) shows that we are modeling total combinational delay, t_{CL}, as $8.6\,^{\text{ns}}/_{\text{term}} \cdot N_t$ and the coefficient, k, as $1.9\,\text{ns} - 4.7\,^{\text{ps}}/_{\text{term}} \cdot N_t$.

To validate this model, we compute the total achievable throughput, T_{TOT}, from equations (3, 4) as $1/t_{CLK}$, and plot the observed data values from the synthesis, place, & route runs and the model prediction in Figure 4. This shows the achievable throughput for the virtualized design with $N_t = 20$ terms of the Taylor series. The solid line represents the model prediction. The dotted lines represent confidence intervals on the model [14] (computed as a 95% confidence for 10 future observations).

We make several observations about the total achievable throughput of the virtualized designs. First, the model does a reasonably good job of characterizing the shape of the curve. Many of the measured data points are within the confidence intervals, although not all. Second, throughput initially increases linearly (at low stream counts) but eventually levels off and adding additional streams does not provide any significant throughput gains. Essentially, the clock rate gains (due to deeper pipelining) have provided their maximum benefit. Finally, the maximum throughput achievable is present at 24 streams and higher.

Next, we curve fit t_{CLK} for the ASIC technology. The resulting model yields:

$$t_{CLK} = 2.3\,\tfrac{\text{ns}}{\text{term}} \cdot \frac{N_t}{C} + \left(3.2\,\text{ns} - 9.8\,\tfrac{\text{ps}}{\text{term}} \cdot N_t\right) \cdot \sqrt{C-1}. \tag{14}$$

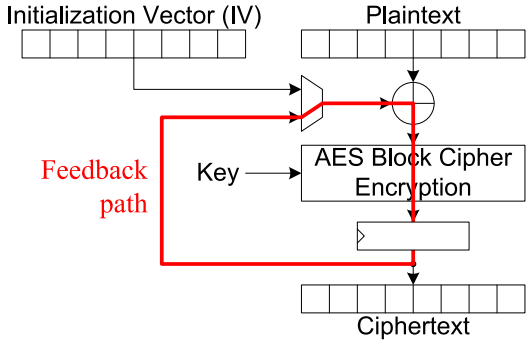

Fig. 6. Block diagram of AES encryption cipher (CBC block mode) application. With a fully unrolled block cipher and no pipelining (initially), the highlighted feedback path will determine the achievable clock rate.

Comparing this expression to (2) shows that we are modeling total combinational delay, t_{CL}, as $2.3\,^{\text{ns}}/_{\text{term}} \cdot N_t$, and the coefficient, k, as $3.2\,\text{ns} - 9.8\,^{\text{ps}}/_{\text{term}} \cdot N_t$. The plot of the empirical performance results of the total achievable throughput for the ASIC design is shown in Figure 5. The observations made with FPGA technology clearly still hold with the ASIC technology.

Next, we use an AES encryption cipher [15] in CBC block mode (that has a feedback path) illustrated in Figure 6 to calibrate t_{CLK} in the model. In our implementation, the individual block cipher is fully unrolled forming a long combinational function up to 14 rounds (the AES 256-bit standard), enabling us to investigate the impact of short vs. deep combinational logic functions. The AES block cipher is shown in the middle of the figure. Operating in cipher-block chaining (CBC) mode, an initialization vector (IV) is XOR'd with a plaintext block to produce the input to the cipher. The output is then registered which becomes the ciphertext output. The ciphertext is then fed back, through a multiplexer to be mixed again with the next block of plaintext.

In the experiment, we measure t_{CLK} on 10 independent runs for an FPGA for design parameters $N_r = 4$ and $N_r = 14$, and the number of streams, C, ranging from 1 to 28 (i.e., up

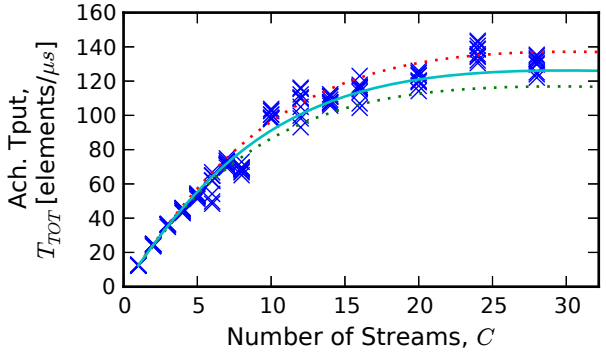

Fig. 7. Total achievable throughput plot for virtualized logic of the AES encryption cipher on an FPGA with $N_r = 14$ rounds. The number of streams refers to the logic pipeline depth, C, with $N = C$. For each value of C, data points were taken from 10 tool flow runs.

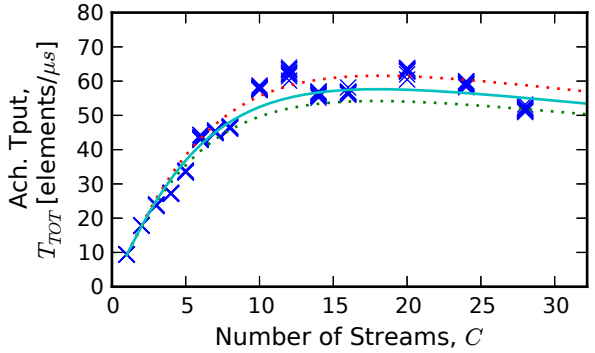

Fig. 8. Total achievable throughput plot for virtualized logic of the AES encryption cipher on an ASIC with $N_r = 14$ rounds. The number of streams refers to the logic pipeline depth, C, with $N = C$. For each value of C, data points were taken from 10 tool flow runs.

to 2 pipeline registers are included per round). Curve fitting the model in (2) across data sets concurrently yields:

$$t_{CLK} = 5.7\,\tfrac{\text{ns}}{\text{rnd}} \cdot \frac{N_r}{C} + \left(1.7\,\text{ns} - 55\,\tfrac{\text{ps}}{\text{rnd}} \cdot N_r\right) \cdot \sqrt{C-1}. \quad (15)$$

Comparing this expression to (2) shows that we are modeling total combinational delay, t_{CL}, as $5.7\,\text{ns}/\text{rnd} \cdot N_r$ and the coefficient, k, as $1.7\,\text{ns} - 55\,\text{ps}/\text{rnd} \cdot N_r$. The plot of the empirical performance of the total achievable throughput for this FPGA design is shown in Figure 7. We can observe that the modeled total achievable throughput matches closely to the observed data values.

Next, we curve fit t_{CLK} for the ASIC technology. The resulting model yields:

$$t_{CLK} = 7.7\,\tfrac{\text{ns}}{\text{rnd}} \cdot \frac{N_r}{C} + \left(4.3\,\text{ns} - 112\,\tfrac{\text{ps}}{\text{rnd}} \cdot N_r\right) \cdot \sqrt{C-1}. \quad (16)$$

Comparing this expression to (2) shows that we are modeling total combinational delay, t_{CL}, as $7.7\,\text{ns}/\text{rnd} \cdot N_r$, and the coefficient, k, as $4.3\,\text{ns} - 112\,\text{ps}/\text{rnd} \cdot N_r$. The plot of the empirical performance results of the total achievable throughput for the ASIC design is shown in Figure 8. The observations made with FPGA technology again still hold with the ASIC technology.

V. Results

With the modeling expressions from Section III, we can explore the performance implications of some of the design parameters. For example, consider the impact of schedule period, R_S, on throughput and latency. For throughput, (10) implies that whenever there is a non-zero cost for context switching, $S > 0$, increasing the schedule period, R_S, always leads to higher achievable throughput, T_{TOT}. Essentially, the cost of doing a context switch is being amortized over a larger number of executions. The circumstances get a little more cloudy; however, when we consider the average latency experienced by a data element. For small schedule periods (i.e., small values of R_S), the total achievable throughput is low which has a negative impact on latency. On the other hand, small schedule periods diminish the time that a data element waits at the front of a queue for the hierarchical schedule to get around to serving its group, resulting in better (lower) latency. Large schedule periods give the exact opposite effect, greater achievable throughput (with its inherent positive impact on latency) but longer times waiting to be scheduled.

To validate these modeling expressions, we developed a cycle-accurate discrete-event simulation of the system and measured the average latency of data elements from when they enter the input queue to when they exit the system. Consider a candidate design with 4 fine-grained contexts ($C = 4$), 8 total contexts ($N = 8$), a 100 MHz clock rate, and a 4 clock overhead to perform a coarse-grained context switch ($S = 4$). Figure 9 plots the total latency, W, as predicted by (12) vs. the schedule period, R_S, for two different offered loads. The points on the graph correspond to empirical results from a discrete-event simulation run with the same parameters. The offered load is the ratio of the aggregate arrival rate (of all streams) to the peak service rate of the system (i.e., when $S = 0$). Offered load then evaluates to $N \cdot \lambda \cdot t_{CLK}$. We draw two conclusions from this figure. First, there is good correspondence between the analytical model of (12) and the empirical simulation results. This bolsters our confidence that the analytic model is reasonable. Second, as is readily apparent in the graph, the schedule period that optimally minimizes latency is different for different offered loads. At low offered load (0.16) minimum latency is experienced with schedule period $R_S = 2$; and at higher offered load (0.48) one must increase the schedule period to $R_S = 4$ to achieve minimum latency.

Now that the model has been validated, we can use it with the calibrated t_{CLK} sub-model from (13) for the Cosine application on an FPGA with $N_t = 20$ to predict the latency and total achievable throughput, T_{TOT}, performance (shown in Figure 10) for a different set of parameters. We then optimize the value of the schedule period, R_S, using a computed figure of merit (described below). Depending on the offered load, a minimum R_S is needed in order to supply enough total

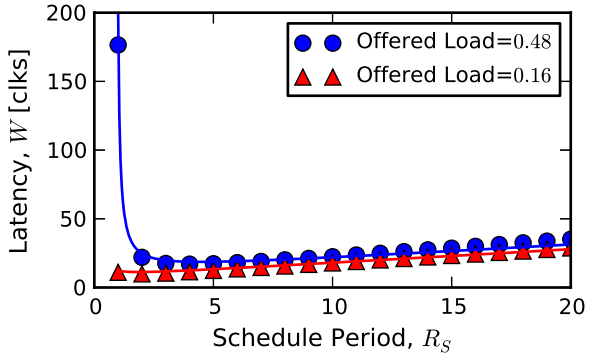

Fig. 9. Latency vs. schedule period at two offered loads (0.48 and 0.16). $C = 4$, $N = 8$, and $S = 4$. The curves are from (12) and the points are empirically measured via a discrete-event simulation.

throughput to meet demand. As offered load increases, the minimum R_S needed also increases. On the latency plot, there are three regions of interest. The first region is the high latency on the left which drops steeply. This high latency is due to queueing delay. As R_S increases, total achievable throughput increases, causing queueing to decrease. The second region is at the knee in the curve. This region gives optimal latency performance. The third region is to the right where the latency gradually increases. This increase is due to the wait time incurred by the round robin schedule as R_S increases. It now takes longer for a data stream to be serviced.

The total achievable throughput plot increases as R_S increases because the cost of a context switch, S, is amortized by the higher schedule period. To optimize for both latency and throughput, we computed a figure of merit (FoM) defined as T_{TOT}/W (total achievable throughput divided by latency). A plot of the figure of merit shows that it is initially low (due to low throughput and high latency), increases to a peak (the optimal value), and then decreases gradually (since latency is increasing faster than throughput).

Solving for the maximum figure of merit optimizes the schedule period, R_S, concurrently for high throughput and low latency. A plot of the optimal R_S is shown across a range of offered loads (from 0.0 to 1.0) as the dashed curve in Figure 11. Also shown as a solid curve is the minimum feasible R_S required for a stable system. Both of these curves follow the same general shape, with schedule period increasing at higher offered loads.

Next, we look at the model results with t_{CLK} calibrated to (16) for the AES encryption cipher application on an ASIC. Instead of showing different values of the offered load, we will instead show different values of the number of pipeline stages, C while keeping the total number of contexts, N, and the arrival rate, λ, constant. The performance plot in Figure 12 shows the latency, W, again, having an initial steep drop, followed by a gradual climb as the schedule period, R_S, increases. As C increases, the overall latency improves. This is due to more contexts being able to execute on the hardware

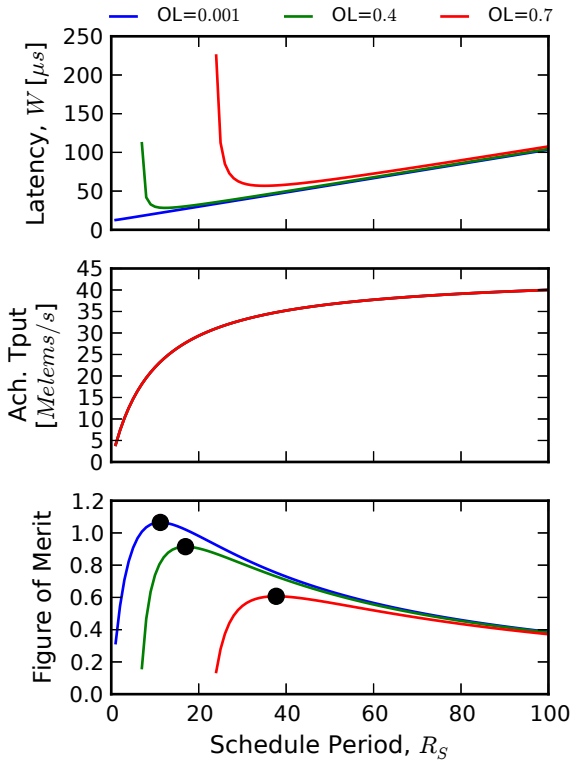

Fig. 10. Latency (top), total achievable throughput (middle), and figure of merit (bottom) at three different offered loads (OL) versus the schedule period, R_S. C is 10, N is 100, S is 100, and t_{CLK} is calibrated using (13) for the Cosine application on an FPGA with $N_t = 20$.

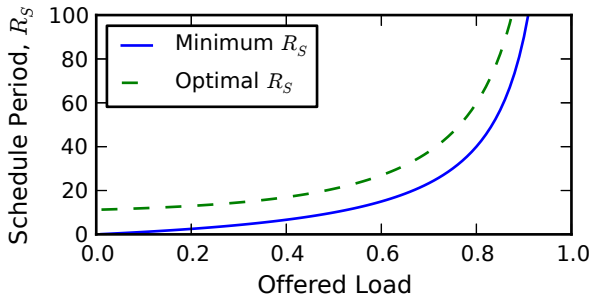

Fig. 11. Schedule period, R_S, versus the offered load (using the same parameters as in Figure 10).

without needing to be swapped out. We also see that the total achievable throughput increases with C which is due to both an improved clock frequency and less context-switching required to execute all N contexts. (Only N/C contexts-switches are required in a full hierarchical scheduling round.) The figure of merit, likewise, is higher for higher C because total achievable throughput is higher and latency is lower.

Optimizing on the figure of merit for the best schedule period, R_S, versus the number of pipeline stages, C, gives the plot in Figure 13. This shows that as C increases, the optimal R_S decreases.

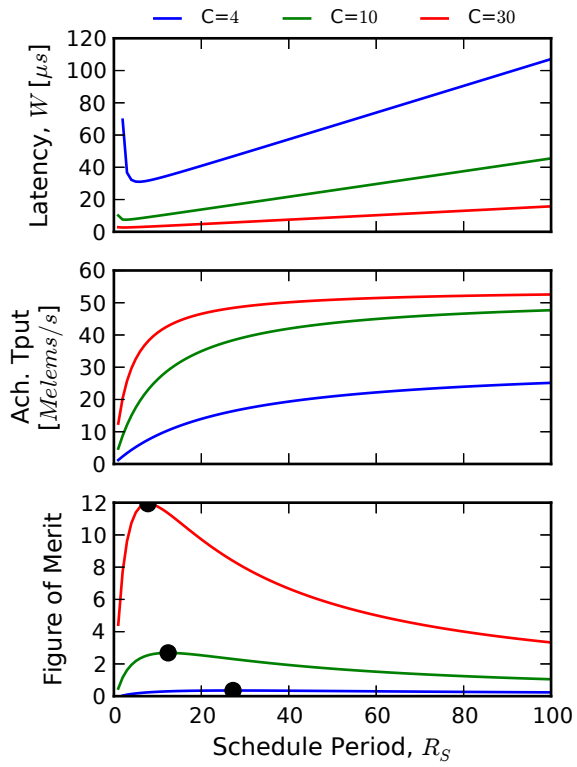

Fig. 12. Latency (top), total achievable throughput (middle), and figure of merit (bottom) at three different number of pipeline stages, C, versus the schedule period, R_S. λ is 30 $\frac{\text{Kelems}}{\text{s}}$, N is 60, S is 100, and t_{CLK} is calibrated using (16) for the AES application on an ASIC with $N_r = 14$.

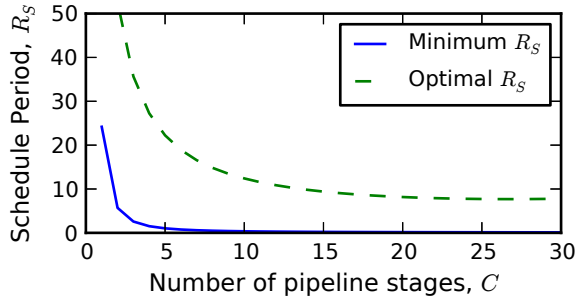

Fig. 13. Schedule period, R_S, versus the number of pipeline stages, C (using the same parameters as in Figure 12).

VI. Conclusions and Future Work

Virtualization of compute resources has a long history, yet there isn't yet good guidance for designers as to how to best provision pipelined sequential circuits for sharing across input data streams. This paper provides an initial effort at providing that guidance. Analytic models are developed for achievable throughput, latency (including traditional queueing, increased queueing due to hierarchical scheduling, and compute time), and occupancy of the input buffers. The novel portions of these models have been empirically validated, including: (1) the expression for increased queueing delay due to the hierarchical schedule, and (2) the modeling of combinational logic delay between pipeline stages as a random variable. Using the models, we demonstrate the ability to choose an optimal schedule period as a function of one (or more) input parameters. This is illustrated by optimizing schedule period as the offered load varies or as the number of pipeline stages vary.

Future work is to extend the modeling results to include analytic predictions of resource usage, which will enable quantitative time-area tradeoffs to be understood. The hierarchical schedule could be broadened to incorporate dynamic scheduling decisions. For example, a schedule that gives more time to highly backlogged input streams (i.e., with deep input buffers) has the potential to significantly improve latency. The performance model in this paper only considers fully unrolled logic computations, however, it would be relatively straightforward to expand the model to support partially rolled implementations as well. Lastly, the case study can be expanded along two dimensions, both adding a coarse-grained context-switch implementation, possibly resulting in a model for the cost of context switching, S, and also a larger set of functions.

References

[1] E. I. Organick, *The Multics System: An Examination of Its Structure*. Cambridge, MA: MIT Press, 1972.

[2] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[3] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*. Upper Saddle River, NJ: Prentice Hall, 2007.

[4] D. M. Tullsen *et al.*, "Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proc. of 23rd Int'l Symp. on Computer Architecture*, May 1996, pp. 191–202.

[5] C. Leiserson and J. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, pp. 5–35, Jun. 1991.

[6] C. Plessl and M. Platzner, "Virtualization of hardware – Introduction and survey," in *Proc. of 4th Int'l Conf. on Engineering of Reconfigurable Systems and Algortihms*, 2004, pp. 63–69.

[7] K. K. Chuang, "A virtualized quality of service packet scheduler accelerator," Master's thesis, Georgia Institute of Technology, Aug. 2008.

[8] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, Apr. 2000.

[9] M. Kudlur, K. Fan, and S. Mahlke, "Streamroller: Automatic synthesis of prescribed throughput accelerator pipelines," in *Proc. of 4th Int'l Conf. on Hardware/Software Codesign and System Synthesis*, 2006, pp. 270–275. [Online]. Available: http://doi.acm.org/10.1145/1176254.1176321

[10] A. Canis, J. H. Anderson, and S. D. Brown, "Multi-pumping for resource reduction in FPGA high-level synthesis," in *Proc. of Conf. on Design, Automation and Test in Europe*, 2013, pp. 194–197. [Online]. Available: http://dl.acm.org/citation.cfm?id=2485288.2485338

[11] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek, "Post-placement C-slow retiming for the Xilinx Virtex FPGA," in *Proc. of 11th Int'l Symp. on Field Programmable Gate Arrays*, 2003, pp. 185–194.

[12] M. Su, L. Zhou, and C.-J. Shi, "Maximizing the throughput-area efficiency of fully-parallel low-density parity-check decoding with C-slow retiming and asynchronous deep pipelining," in *Proc. of 25th Int'l Conf. on Computer Design*, Oct. 2007, pp. 636–643.

[13] L. Kleinrock, *Queueing Theory, Volume 1*. Wiley-Interscience, 1975.

[14] R. Jain, *The Art of Computer System Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. New York: John Wiley & Sons, Inc., 1991.

[15] NIST, "FIPS-197: Advanced Encryption Standard," *Federal Information Processing Standards (FIPS) Publications*, Nov. 2001.