Using M/G/1 Queueing Models with Vacations to Analyze Virtualized Logic Computations

Michael J. Hall
Roger D. Chamberlain

Michael J. Hall and Roger D. Chamberlain, “Using M/G/1 Queueing Models with Vacations to Analyze Virtualized Logic Computations,” in *Proc. of 33rd IEEE International Conference on Computer Design* (ICCD), Oct. 2015, pp. 86-93.

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

Using M/G/1 Queueing Models with Vacations to Analyze Virtualized Logic Computations

Michael J. Hall
VelociData, Inc., St. Louis, MO, USA
Email: mhall@velocidata.com

Roger D. Chamberlain
Department of Computer Science & Engineering
Washington University in St. Louis, MO, USA
Email: roger@wustl.edu

***Abstract*—Virtualization of logic computations (i.e., by sharing a fixed function across distinct data streams) provides a means to effectively utilize hardware resources by context switching the logic to support multiple data streams of computation and to improve the total throughput of all streams. Context switching allows the pipeline stages of the logic to be fully utilized when feedback is present and to support additional contexts using secondary memory. In this paper, we analyze the performance of a virtualized hardware design and develop M/G/1 queueing model equations to predict circuit performance. The server is modeled using a general distribution that takes vacations during the computation of an individual data stream. Using the model, we predict circuit performance and tune a schedule for optimal performance.**

I. Introduction

Virtualization has become quite popular recently, from sharing microarchitectural components across virtual processors [1] to sharing complete computing platforms managed by hypervisors [2], [3]. Graphics processing units (GPUs) regularly use this technique, supporting many threads per core, with asymptotic models having been developed to better understand the performance implications of design decisions [4]. We are interested in applying virtualization techniques to custom logic [5], dedicated hardware that has been designed for a specific purpose. Here, a single physical copy of a hardware function is shared among N virtual "contexts" (see Figure 1). This allows the hardware function to be pipelined to improve the clock frequency and total throughput in the presence of feedback. Context switching can either be fine-grained (i.e., single cycle) or course-grained, with some higher overhead required for state replacement.

To virtualize a hardware function, consider the hardware (HW) block 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.

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

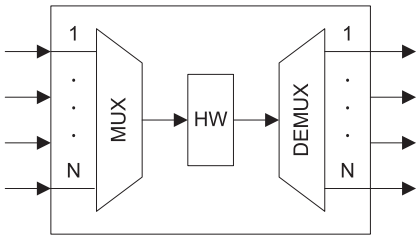


Fig. 1: Hardware virtualization for N distinct data streams that perform the same computation. The N streams are multiplexed into a shared hardware (HW) block, processed, and then demultiplexed back into N streams.

When virtualized in this style, the performance of each individual stream, and of the aggregation of all the streams, is a function of the underlying computation to be performed, the number of streams, the context switching mechanism(s), the schedule, etc. If the instantaneous arrival rate of data elements within a stream occasionally exceeds the service rate at which elements are processed, then queueing will occur at the input port. In systems where utilization (or occupancy) is high, queueing delays can be quite substantial (in many cases greatly exceeding service delays, or the time required to actually perform the computation of interest).

In this paper, we model the performance of a virtualized fixed logic computation using an M/G/1 queueing model with vacations and tune a schedule for optimal performance. The queueing model is used to understand system performance and the factors that effect it that may aid a hardware designer in system design. Our interest is in supporting a set of distinct data streams that all perform the same computation (or function). The performance metrics we model include throughput, latency, and queue occupancy. We illustrate the use of the model with an example circuit and design scenario in which the schedule period is adjusted to minimize latency.

II. Background and Related Work

A. Hardware virtualization

Plessl and Platzner [5] wrote a survey paper on hardware virtualization using FPGAs that describes three different approaches: temporal partitioning, virtualized execution, and

978-1-4673-7166-7/15/$31.00 ©2015 IEEE

virtual machine. Chuang [6] described another type of temporal partitioning whereby hardware logic can be reused by temporally shared state. A diagram illustrating the hardware virtualization approaches is shown in Figure 2.

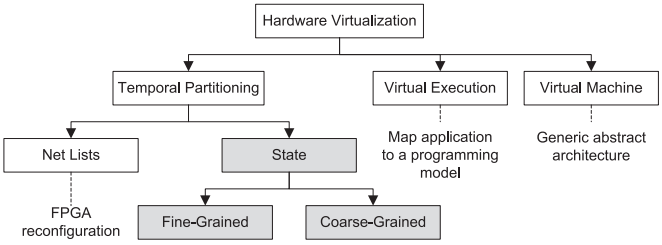


Fig. 2: Hardware virtualization approaches. White boxes are described by Plessl and Platzner [5]. Gray boxes are described by Chuang [6].

Temporal partitioning of net lists [5]: This is a technique for virtualizing a hardware design described by a net list that would otherwise be too large to physically fit onto an FPGA. This is done by partitioning the net list and swapping it like virtual memory, allowing only one part of the computation to run at a time. Each part must, therefore, run sequentially to perform the complete computation with logic reconfiguration done between parts.

Temporal partitioning of state [6]: This is a way to share hardware by temporally swapping its state so as to compute multiple streams of computations 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. Fine-grain context switching is done by applying a C-slow transformation (described below) on the hardware logic, allowing different contexts to be processed in each pipeline stage of the C-slowed hardware. Coarse-grain context switching is done by swapping the state infrequently to and from a memory. Temporal partitioning of state with both fine- and coarse-grain context switching is the type of virtualization that we use in this paper.

Virtualized execution [5]: This is where a programming model is used to specify applications. Any application developed in this programming model can run on any hardware that supports this model of execution. An example is the instruction set of a processor. Code written for this instruction set can execute on any processor that supports the instruction set. Another example is PipeRench [7], which has a pipelined streaming programming model where the application is decomposed into stripes and executed in a pipeline.

Virtual machine [5]: This defines a generic abstract architecture that hardware can be designed on. Designs targeted to a generic FPGA architecture are remapped to the actual architecture of a specific FPGA device.

B. C-slow transformation

C-slow is a transformation described by Leiserson and Saxe [8] whereby every register in a digital logic circuit is replaced by C registers. This allows sequential logic circuits, which have feedback paths, to be pipelined and retimed. Retiming is a technique for improving the clock frequency of a circuit by moving pipeline registers forward and backwards through the combinational logic to shorten the critical path of the circuit. Retiming a C-slowed circuit can theoretically give up to C times improvement in the clock frequency.

Several applications have been implemented using the C-slow technique. Weaver et al. applied C-slow to three applications: AES encryption, Smith/Waterman sequence matching, and LEON 1 synthesized microprocessor core [9]. 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 1 SPARC microprocessor core achieved a speedup of 2.0 for a 2-slow automatically C-slowed design implementation. Su et al. applied C-slow to an LDPC decoder for a throughput-area efficient design [10]. Akram et al. applied C-slow to a processor to execute multiple threads in parallel using a single datapath of an instruction set processing element. For a 3-slow microprogrammed finite-state machine, a speedup of 2.59 times in clock frequency was achieved [11].

C. Vacations in queueing models

Hall and Chamberlain [12], [13] describe an M/D/1 model for assessing the performance of hardware virtualized circuits. Here, we generalize that result to an M/G/1 model that explicitly includes the effects of the server not being available when it is processing other contexts. To the extent possible, we retain the same notation as the previous model.

A vacation model is an approach to analyzing queueing systems where the server is not continuously available (e.g., the server is executing other jobs). Bertsekas and Gallager [14] describe M/G/1 queues (*M*arkovian, or memoryless, arrival process; *G*eneral service process; 1 server) where the server can go on "vacation" for some random interval of time. This is illustrated in Figure 3. Here, X_j represents the service time of the jth job and V_k represents the vacation time of the kth vacation. The model is as follows. When a job is waiting in the queue, the server will begin servicing the job and enter a busy period represented by X_j. When the queue is empty, the server will go on vacation and enter a vacation period represented by V_k. If a new arrival enters an idle system, rather than going immediately into service, it waits for the end of a vacation period, and then enters service. If there is no new arrival into the system, then the server, after returning from a vacation, will immediately go into another vacation period.

III. Virtual Hardware Configuration

Returning to 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

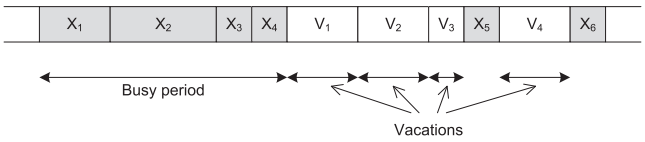


Fig. 3: M/G/1 queueing model with vacations. During busy periods, the server is servicing jobs. During vacation periods, the server is "away" and jobs may be waiting in the queue.

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*). 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.

We consider a general virtualized hardware configuration with fine- and coarse-grained context switching. An arbitrary sequential logic circuit (i.e., the HW block) is C-slowed and augmented with a secondary memory that can load and unload copies of the state to/from the "active" state register. N FIFO, or First In, First Out, 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.

A queueing model of this circuit can be developed to predict system performance that can then be used by a hardware designer to tune the performance. Inputs to the model and model assumptions 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 rate $\lambda_i \, {}^{\text{elements}}/_{\text{s}}$. We will assume the input distribution is Poisson. We will also assume that both the secondary memory and input buffers operate at the clock rate of the pipeline, and that the input buffers are sized sufficiently large to assume infinite capacity. 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 illustrated in Figure 4. Each individual queueing station represents one virtual copy of the hardware computation. The arrival rate at each queue is $\lambda_i \, {}^{\text{elements}}/_{\text{s}}$.

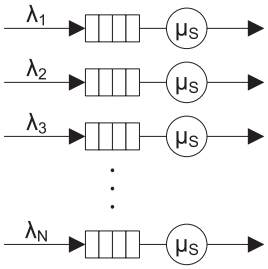


Fig. 4: Queueing model of virtualized hardware with N queueing stations (one for each data stream). Each queueing station consists of a FIFO queue and associated server representing one virtual copy of the hardware computation.

IV. M/G/1 Model Development

We now consider a model of the system with an independent M/G/1 queueing station where the server can go on vacation for some time. In this model, the service distribution is general and, during a vacation, the server is not servicing any of its queued data elements.

Referring to Figure 4, the system consists of N queueing stations which all share a single physical server which is represented as being N "virtual" servers. We employ a fixed, hierarchical, round-robin schedule. A set of C input streams share the hardware resource and execute in a round-robin fashion in the server (in the computation pipeline). After R_S rounds, the current set of C input streams' state is swapped out (context switched) to secondary memory with cost S and the next set is swapped in. The collection of input stream sets are also context switched in a round-robin fashion.

The server can take short or long vacations. Short vacations are due to fine-grain context switching and are only taken when the server is idle (i.e., the FIFO queue is empty). They occur because arrivals to a non-empty queue are aligned with the fine-grained schedule by the current entry in service. They last for C clock cycles. Long vacations are due to coarse-grain context switching and are taken when the schedule has completed R_S rounds of the current set of C input streams

and context switches to execute the other $N-C$ input streams, according to the fixed schedule. They occur for the duration of the vacation period. During these times, no new elements are processed until the vacation completes.

A single queueing station of the system, shown in Figure 5, consists of a FIFO queue and "virtual" server. W_q is the mean waiting time in the queue and W_s is the time spent in the server. When the server takes a vacation, this produces an additional waiting time at the head of the queue. We can derive a vacation waiting time, V, that is used in the calculation of W_q. It is modeled with a distribution determined by the fixed, hierarchical, round-robin schedule and is defined next. W_s is modeled by a fixed service time X.

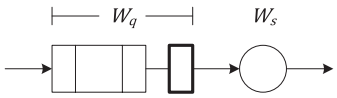
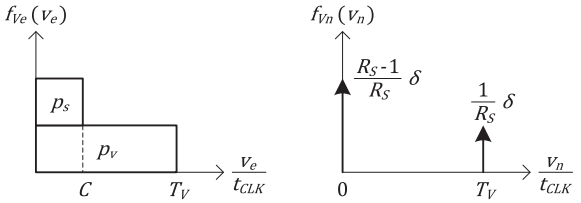


Fig. 5: Single queueing station of system.

A. Vacation waiting time model

To model the vacation waiting time, V, we define two sub-model distributions: V_e for an empty queue shown in Figure 6a, and V_n for a non-empty queue shown in Figure 6b. For the case of an empty queue, the probability density function, $f_{V_e}(v_e)$ shows the distribution of the vacation waiting time for continuous random variable V_e. T_V is the time in a long vacation period (i.e., during a coarse-grain context switch), p_s is the fraction of time in a service period, and p_v is the fraction of time in the long vacation period. Along the x-axis, v_e is measured in time and normalized to clocks.



(a) Empty queue (b) Non-empty queue

Fig. 6: Sub-model distributions used in the derivation of the vacation waiting time model.

We start the model by defining the total number of clocks to complete a full round of the hierarchical schedule as

$$T_T = R_S N + SN/C. \tag{1}$$

The number of clocks in a long vacation period, which is the time during which the server has context switched to process other contexts, is

$$T_V = R_S (N - C) + SN/C. \tag{2}$$

The fraction of time in a service period is

$$p_s = R_S C / T_T, \tag{3}$$

and the fraction of time in a long vacation period is

$$p_v = 1 - R_S C / T_T. \tag{4}$$

The probability density function, $f_{V_e}(v_e)$, consists of two stacked rectangles with areas p_s and p_v whose sums must equal 1. Solving for this, $f_{V_e}(v_e)$ is

$$\begin{aligned} f_{V_e}(v_e) = &\left(\frac{1-p_s}{T_V \cdot t_{CLK}}\right) \cdot [u(v_e) - u(v_e - T_V \cdot t_{CLK})] \\ &+ \left(\frac{p_s}{C \cdot t_{CLK}}\right) [u(v_e) - u(v_e - C \cdot t_{CLK})] \end{aligned} \tag{5}$$

where

$$u(x) = \begin{cases} 0 & x < 0, \\ 1 & x \geq 0. \end{cases}$$

Next, for the case of a non-empty queue, the probability density function, $f_{V_n}(v_n)$, in Figure 6b, shows the distribution of the vacation waiting time for continuous random variable V_n. In this figure, the impulses defined at 0 and $T_V \cdot t_{CLK}$ are Dirac delta functions ($\delta(x)$ notation) where

$$d_\epsilon(x) = \begin{cases} 1/\epsilon & -\epsilon/2 \leq x \leq \epsilon/2, \\ 0 & otherwise, \end{cases}$$
$$\delta(x) = \lim_{\epsilon \to 0} d_\epsilon(x),$$

and $\int_{-\infty}^{\infty} \delta(x)\, dx = 1$. Along the x-axis, v_n is measured in time and normalized to clocks.

Here, the two impulses define the vacation waiting time (and are normalized to the total number of jobs, R_S, in a full round of the hierarchical schedule). The first impulse, at $v_n = 0$ (i.e., no wait time), defines a group of $R_S - 1$ jobs being serviced consecutively. When these jobs complete, the server goes on a long vacation, and the next job waits for the vacation to complete. The wait time of the one job is then modeled by the second impulse, at $v_n = T_V \cdot t_{CLK}$.

The probability density function, $f_{V_n}(v_n)$, is then

$$f_{V_n}(v_n) = \frac{R_S - 1}{R_S} \delta(v_n) + \frac{1}{R_S} \delta(v_n - T_V \cdot t_{CLK}). \tag{6}$$

Since the queueing station is using the fixed, hierarchical schedule, the probability of the queue being empty changes depending on whether the server is in a service or vacation period. Although it might change, we will assume that this probability is fixed for the purposes of combining the probability density functions, $f_{V_e}(v_e)$ and $f_{V_n}(v_n)$, into a single approximate function, $f_V(v)$. When we do this, we get

$$\begin{aligned} f_V(v) &= p_0 \cdot f_{V_e}(v) + (1 - p_0) \cdot f_{V_n}(v) \\ &= p_0 \cdot \left[\frac{1-p_s}{T_V \cdot t_{CLK}} [u(v) - u(v - T_V \cdot t_{CLK})] \right. \\ &\quad \left. + \frac{p_s}{C \cdot t_{CLK}} [u(v) - u(v - C \cdot t_{CLK})] \right] \\ &\quad + (1 - p_0) \cdot \left[\frac{R_S - 1}{R_S} \delta(v) + \frac{1}{R_S} \delta(v - T_V \cdot t_{CLK}) \right] \end{aligned} \tag{7}$$

where $p_0 = 1 - \rho$, the probability of an empty queue.

Calculating the expected value, $E[V]$, of the vacation waiting time to get the mean vacation waiting time, we get the following:

$$\begin{aligned} E[V] &= \int_{-\infty}^{\infty} v \cdot f_V(v)\, dv \\ &= \left[\frac{1}{2} p_0 \cdot [(1-p_s) \cdot T_V + p_s \cdot C] \right. \\ &\quad \left. + (1-p_0) \cdot \left[\frac{T_V}{R_S}\right]\right] \cdot t_{CLK}. \end{aligned} \tag{8}$$

The mean vacation waiting time will be denoted as $\overline{V}$ when used in the derivation of the queueing model equations.

B. Service time model

The model for the service time, X, is fixed and deterministic. For every job that enters the computational pipeline, it always takes exactly C clock cycles to complete service of that job. Therefore, the service time can be modeled as a single impulse function at $x = C \cdot t_{CLK}$. The resulting probability density function, $f_X(x)$, is

$$f_X(x) = \delta(x - C \cdot t_{CLK}). \tag{9}$$

Next, we can calculate $E[X]$, the expected service time, as

$$E[X] = \int_{-\infty}^{\infty} x \cdot f_X(x)\, dx = C \cdot t_{CLK}. \tag{10}$$

As part of the queueing model, we also need to know the second moment of the service time, $E\left[X^2\right]$. This is

$$E\left[X^2\right] = \int_{-\infty}^{\infty} x^2 f_X(x)\, dx = C^2 \cdot t_{CLK}^2. \tag{11}$$

We will denote $E[X]$ as $\overline{X}$ and $E\left[X^2\right]$ as $\overline{X^2}$ in the development of the queueing model equations that follow.

C. Queueing model

First, we start by defining the (deterministic) time in the pipelined circuit as

$$W_s = \overline{X} = C \cdot t_{CLK}. \tag{12}$$

The effective service rate, μ_s, of a "virtual" server is not equal to $1/W_s$, but has to take into account the long vacation time due to a coarse-grain context switch. We can do this by defining the service rate as the number of jobs processed by a stream in one full period of the hierarchical schedule. Therefore, for this M/G/1 system, the service rate, which is also the maximum achievable throughput (per stream), is

$$\mu_s = \frac{R_S}{(R_S N + SN/C) \cdot t_{CLK}}. \tag{13}$$

The total achievable throughput is then $T_{TOT} = N \cdot \mu_s$, or

$$T_{TOT} = \frac{R_S}{(R_S + S/C) \cdot t_{CLK}}. \tag{14}$$

The average (mean) waiting time of each data element for an M/G/1 system (without vacations) is determined by the Pollaczek-Khinchin (P-K) formula as $W_q = \frac{\lambda \overline{X^2}}{2(1-\rho)}$ where $\rho = \lambda/\mu_s$, λ is the arrival rate, and $\overline{X^2}$ is the second moment of the service time [14]. To account for vacations in this formula, we need to add an additional waiting time at the head of the queue. For this, we need to use equation 3.46 in [14] of the P-K formula derivation which uses a mean residual time, R, to solve for the wait time in the queue. The wait time formula then becomes $W_q = R + \frac{1}{\mu_s} N_q$ where $R = \frac{1}{2}\lambda\overline{X^2}$ and $N_q = \lambda W_q$ (Little's Law [15]). Now, we can add the mean vacation waiting time, $\overline{V}$, to W_q and apply Little's Law:

$$W_q = R + \frac{1}{\mu_s}\lambda W_q + \overline{V}.$$

Solving for W_q then gives

$$W_q = \frac{R + \overline{V}}{1-\rho} = \frac{\lambda\overline{X^2}}{2(1-\rho)} + \frac{\overline{V}}{1-\rho}. \tag{15}$$

Next, the average latency (elapsed time from arrival to completion of processing; we assume one output is generated per input) for each data element is

$$\begin{aligned} W_T = W_q + W_s &= \frac{\lambda\overline{X^2}}{2(1-\rho)} + \frac{\overline{V}}{1-\rho} + \overline{X} \\ &= \left[\frac{C^2(\lambda \cdot t_{CLK})}{2(1-\rho)} + \frac{1}{2}[(1-p_s) \cdot T_V + p_s \cdot C] \right. \\ &\quad \left. + \frac{\rho}{1-\rho} \cdot \left[\frac{T_V}{R_S}\right] + C\right] \cdot t_{CLK}. \end{aligned} \tag{16}$$

The average (mean) occupancy of each queue is $N_q = \lambda W_q$.

The expression for the average latency consists of 5 terms that model the delay through the system. The first term ($\frac{\lambda C^2 t_{CLK}^2}{2(1-\rho)}$) models service queueing delay. The second term ($\frac{1}{2}(1-p_s) T_V \cdot t_{CLK}$) models long vacation delay during a vacation period. The third term ($\frac{1}{2} p_s \cdot C \cdot t_{CLK}$) models short vacation delay during a service period for an empty queue. The fourth term ($\frac{\rho}{1-\rho}\frac{T_V \cdot t_{CLK}}{R_S}$) models vacation queueing delay. And the fifth term ($C \cdot t_{CLK}$) models service time delay. Of these delays, those due to the long vacation period (the second and fourth terms) have the greatest impact on the average latency. The vacation queueing delay (fourth term) accounts for large initial latency at low R_S due to the effect of S, but decreases when R_S increases (by amortizing the effect of S). The long vacation delay (second term) accounts for a gradual increase in latency with increasing R_S since T_V increases with R_S.

To summarize the queueing model notation used in the derivation above, Table I lists the notation in abbreviated form.

D. Validation

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. The workload is generated probabilistically and the performance is measured at steady state. Consider a candidate design with 10

TABLE I: Queueing model notation.

Term	Label
N	Number of data streams
C	Pipeline depth
S	Context switch cost
R_S	Schedule period
t_{CLK}	Clock period
T_T	Total schedule time
T_V	Vacation time
p_s	Service time fraction
$\overline{V}$	Mean vacation waiting time
$\overline{X}$	Mean service time
$\overline{X^2}$	Service time second moment
T_{TOT}	Total achievable throughput

fine-grained contexts ($C = 10$), 100 total contexts ($N = 100$), and a 100 clock overhead to perform a coarse-grained context switch ($S = 100$). Figure 7 plots the total latency, W_T, predicted by (16) vs. the schedule period, R_S, for an offered load of 0.08. The points on the graph correspond to empirical results from the discrete-event simulation run with the same parameters. The offered load is defined as 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 and the empirical simulation results. This bolsters our confidence that the analytic model is reasonable. Second, there is a schedule period that optimally minimizes latency for a given offered load. At an offered load of 0.08, minimum latency is experienced with schedule period $R_S = 3$.

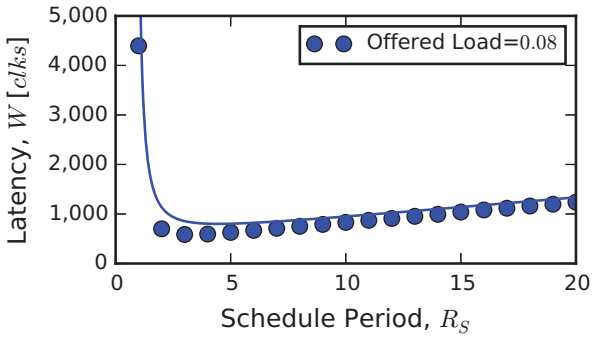


Fig. 7: Discrete-event simulation of latency vs. schedule period for an example circuit with parameters $C = 10$, $N = 100$, $S = 100$. The curves are analytically generated from (16) and the points are empirically measured via discrete-event simulation.

Additional validation was performed with input values $C = 4$, $N = 8$, $S = 4$, and $OL = 0.16$ and 0.48. Similar results were obtained having good correspondence between the analytical model and simulation and the existence of a schedule period that minimizes latency.

V. Experimental Setup

We use the Advanced Encryption Standard (AES) cipher operating in cipher-block chaining (CBC) mode for encryption [16] (that has a feedback path) shown in Figure 8 to illustrate the use of the model. In our implementation, the AES encryption cipher is fully unrolled forming a long combinational logic path with 14 rounds (the AES 256-bit standard) and without secondary memory. 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 registered which contains the ciphertext output. The ciphertext is then fed back, through a multiplexer, to be mixed again with the next block of plaintext. The AES encryption cipher has a configurable number of pipeline stages, C, to improve the clock period (up to 2 pipeline stages per round). This is implemented via outer loop pipelining [17]. Increasing C beyond 2 times the number of rounds subsequently adds pipeline stages to the end of the computation which adds latency and does not further improve the clock period.

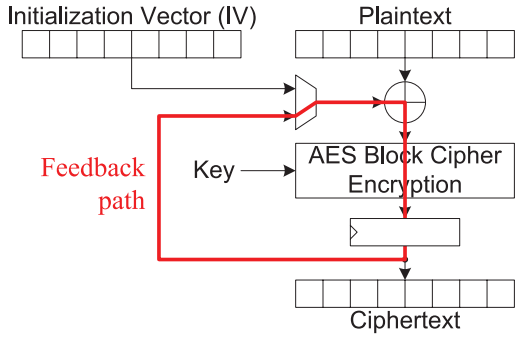


Fig. 8: Block diagram of AES encryption cipher application in the CBC block mode. With a fully unrolled block cipher and no pipelining (initially), the highlighted feedback path will determine the achievable clock rate.

This application is implemented on a field-programmable gate array (FPGA). We target a Xilinx Virtex-4 XC4VLX100 FPGA and use the Xilinx ISE 13.4 tools for synthesis, place, & route of the hardware design. This implementation allows us to calibrate t_{CLK} which is valid for designs both with or without secondary memory.

VI. Results

We have application and technology independent M/G/1 queueing model equations that can be used to predict the performance of virtualized logic computations for fine- and coarse-grain contexts. In this section, we show prediction results for the AES encryption application on FPGA technology.

The queueing model consists of inputs and outputs that make it flexible in predicting performance. For these results, we focus on a specific design scenario. We are given a circuit, technology, the total number of contexts (N), the pipeline depth (C), and the cost of a context switch (S), and have a varying offered load (OL). We can tune the

schedule period (R_S). We first present prediction results for total achievable throughput and latency. Then we optimize for minimum latency (W_T).

Total achievable throughput is shown in Figure 9 for the AES application. In this application, one output is generated per input (i.e., a block output is computed directly from each block input into AES). As the schedule period (R_S) increases, the cost of a context switch (S) is amortized, resulting in higher total achievable throughput.

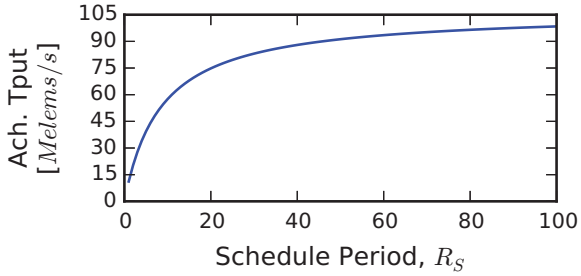


Fig. 9: Total achievable throughput prediction plot vs. schedule period. $C = 14$, $N = 8C$, and $S = 120$.

The results for latency are shown in Figure 10. The latency we are considering is the delay from an input block arriving at a stream input to the corresponding encrypted block being available at the output stream. In the figure, there is both a latency and optimization plot. In the latency plot, the queueing model predicts mean latency (W_T) across a range of schedule periods (R_S) and three offered loads (OL). Offered load, as was previously defined, is the ratio of the aggregate arrival rate (of all streams) to the peak service rate of the system (i.e., when $S = 0$). It evaluates to $N \cdot \lambda \cdot t_{CLK}$.

Returning to 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 traditional queueing delay, a normal consequence of high effective server utilization. The beginning of this region marks the minimum R_S needed for a given offered load. Below this value, the context switch overhead is high enough that the provided offered load cannot be serviced, and the queueing system is unstable (i.e., grows to infinite queue length). As R_S increases, total achievable throughput increases, causing queueing to decrease. The second region is at the knee of the curve. In this region, R_S is optimal and gives minimum latency performance. The third region is to the right where latency gradually increases. This increase is due to the wait time incurred during a vacation period by the hierarchical, round-robin schedule as R_S increases. It now takes longer for a data stream to be serviced. We can observe for an $OL = 0.7$, the minimum latency is about 30 μs.

In the optimization plot, latency is optimized across a range of offered loads swept from 0 to 1. There are two curves. The solid blue curve shows the minimum R_S needed to service a given offered load (that is, $\rho < 1$). The dashed green curve shows the optimal R_S that minimizes latency at the given offered load. We can see that at low offered load, the optimal

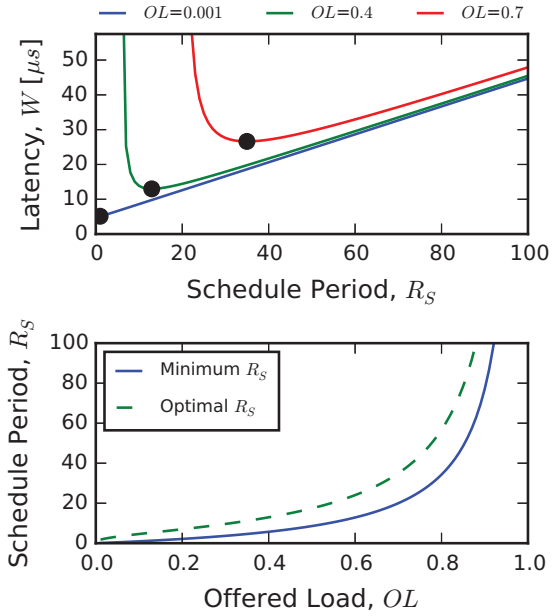


Fig. 10: Latency prediction (top) and optimization (bottom) plots. In the latency plot, the optimal R_S for minimum latency is indicated by a dot. In these plots, $C = 14$, $N = 8C$, and $S = 120$.

R_S is small, then gradually increases until about 0.8, and finally increases steeply.

Suppose we choose a value of R_S from the optimization plot. We can then plot the latency for this fixed R_S versus the offered load to show a performance curve across all loads. This is shown in Figure 11 for three values of R_S. We can see that the latency is initially flat at low offered loads, but then increases steeply at some point. The "knee" of these curves, indicated by a black dot, is approximated to be 3 dB above the minimum latency of each curve. Beyond the knee, the latency increases sharply. As R_S varies, the latency curve changes in two ways. First, it changes in the value of the latency in the flat part of the curve (low R_S gives low latency). Second, it changes the position of the knee of the curve (high R_S pushes the knee out further, allowing a large range of loads to be handled). This indicates a tradeoff. If we expect the load on the system to be small, we can sacrifice range to get low latency. Otherwise, we can sacrifice low latency for a large range. Optimizing R_S continuously across all offered loads, we can attain the minimum latency for each offered load. This is shown as the dashed black curve, which establishes a lower bound on the latency.

The above results enable a user to dynamically tune the performance of a virtualized custom logic application in response to varying input load conditions. By adjusting the schedule period based on offered load, the system can be continuously operated in its minimum latency configuration.

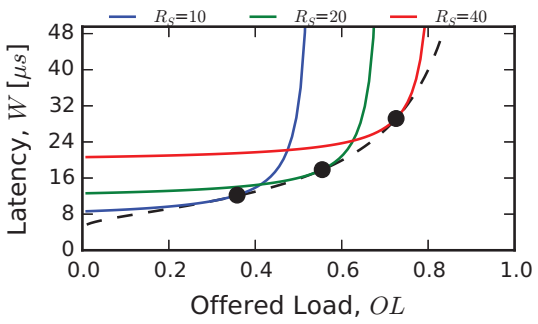


Fig. 11: Latency prediction plots vs. offered load. $C = 14$, $N = 8C$, and $S = 120$. A 3 dB point is drawn for each curve above the minimum latency to approximate the knee of the curve. The dashed black curve is a lower bound of the optimal latency for R_S optimized at each offered load.

VII. Conclusions and Future Work

We have developed a vacation-based M/G/1 queueing model for virtualized custom logic functions. The model predicts throughput, latency, and queue occupancy when provided with a circuit, an implementation technology, clock period, pipeline depth, number of contexts, schedule period, input arrival rate, and overhead of a context switch. The model has been empirically validated by comparing its performance predictions with those of a cycle-accurate discrete-event simulation.

We illustrated the use of the model with an example circuit (AES encryption cipher operating in CBC mode) that has a deep combinatorial logic path and therefore benefits from deep pipelining. Using the model, we fixed the pipeline depth, number of contexts, and context switch overhead, optimizing the mean latency for various arrival rates by tuning the schedule period.

The optimization use case described here is one of many possible uses for the model. Rather than optimizing for latency alone, a designer might wish to co-optimize over multiple objectives, by combining those objectives into a single figure of merit. For example, maximizing throughput divided by latency is a reasonable way to combine these two objectives.

Data buffers at the inputs to the virtualized custom logic functions allow elements to queue up while waiting for service. They are often expensive in dedicated hardware designs (in contrast to software-based systems where large main memories are the norm in modern processors). On-chip memory is almost always expensive (whether in an FPGA or an ASIC technology) and off-chip memory might or might not even exist. The ability of the model to predict queue occupancy can be quite helpful to designers in estimating the buffering requirements for a candidate design.

Future directions for this work primarily involve identifying and generalizing one or more aspects of the present model. Two immediately relevant candidates are (1) the distribution of the input arrival process, and (2) the hierarchical round-robin schedule.

The assumption that the input process is Poisson supports a wide range of analytical results in the queueing literature; however, a real system may act quite differently by, as an example, buffering up data and sending it in bursts (this is more efficient to do, especially in software systems). Generalizing the input arrival process would then allow more accurate modeling. There are many results available for phase-type distributions and state-space solution techniques that can be used with fully general input distributions.

Extending the model to support additional scheduling algorithms is an area of particular interest. The current scheduling algorithm is not work-conserving, meaning that an empty input queue will still get scheduled even when other input queues have data to be processed. This can result in missed opportunities for improved performance where empty input queues can be skipped by the scheduler. There is a clear opportunity to both develop richer scheduling algorithms as well as model their performance.

References

[1] D. M. Tullsen, S. Eggers, J. Emer, and H. Levy, "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.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003.

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

[4] L. Ma, K. Agrawal, and R. D. Chamberlain, "A memory access model for highly-threaded many-core architectures," *Future Generation Computer Systems*, vol. 30, pp. 202–215, Jan. 2014.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] M. A. Akram, A. Khan, and M. M. Sarfaraz, "C-slow technique vs multiprocessor in designing low area customized instruction set processor for embedded applications," *International Journal of Computer Applications*, vol. 36, no. 7, Dec. 2011.

[12] M. J. Hall and R. D. Chamberlain, "Performance modeling of virtualized custom logic computations," in *Proc. of 24th ACM Int'l Great Lakes Symposium on VLSI*, 2014.

[13] ——, "Performance modeling of virtualized custom logic computations," in *Proc. of 25th IEEE Int'l Conf. on Application-specific Systems, Architectures and Processors*, Jun. 2014, pp. 72–73.

[14] D. Bertsekas and R. Gallager, *Data Networks*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1992.

[15] J. D. C. Little, "A proof for the queuing formula: $L = \lambda W$," *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961.

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

[17] P. R. Chodowiec, "Comparison of the hardware performance of the AES candidates using reconfigurable hardware," Master's thesis, George Mason University, 2002.